



Quick answers to common problems

SQL Server 2012 with PowerShell V3 Cookbook

Increase your productivity as a DBA, developer, or IT Pro, by using PowerShell with SQL Server to simplify database management and automate repetitive, mundane tasks

Donabel Santos

[PACKT] enterprise 
PUBLISHING professional expertise distilled

SQL Server 2012 with PowerShell V3 Cookbook

Increase your productivity as a DBA, developer, or IT Pro, by using PowerShell with SQL Server to simplify database management and automate repetitive, mundane tasks.

Donabel Santos



BIRMINGHAM - MUMBAI

SQL Server 2012 with PowerShell V3 Cookbook

Copyright © 2012 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2012

Production Reference: 1151012

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-646-4

www.packtpub.com

Cover Image by Artie Ng (artherng@yahoo.com.au)

Credits

Author

Donabel Santos

Project Coordinator

Yashodhan Dere

Reviewers

Edwin Sarmiento

Laerte Poltronieri Junior

Proofreader

Chris Smith

Acquisition Editor

Rukhsana Khambatta

Indexer

Tejal R. Soni

Lead Technical Editor

Azharuddin Sheikh

Production Coordinator

Manu Joseph

Technical Editors

Charmaine Pereira

Sharvari Baet

Jalasha D'costa

Cover Work

Manu Joseph

Copy Editors

Alfida Paiva

Brandt D'Mello

Insiya Morbiwala

Aditya Nair

About the Author

Donabel Santos is a SQL Server MVP and is the senior SQL Server Developer/DBA/Trainer at QueryWorks Solutions, a consulting and training company in Vancouver, BC. She has worked with SQL Server since version 2000 in numerous development, tuning, reporting, and integration projects with ERPs, CRMs, SharePoint, and other custom applications. She holds MCITP certifications for SQL Server 2005/2008, and an MCTS for SharePoint. She is a Microsoft Certified Trainer (MCT), and is also the lead instructor for SQL Server Administration, Development, and SSIS courses at British Columbia Institute of Technology (BCIT).

Donabel is a proud member of PASS (Professional Association of SQL Server), and a proud BCIT alumna (CST diploma and degree). She blogs (www.sqlmusings.com), tweets (@sqlbelle), speaks and presents (SQLSaturday, VANPASS, Vancouver TechFest, and so on), trains (BCIT, QueryWorks Solutions), and writes (Packt, Idera, SSWUG, and so on).

Acknowledgement

Writing a book would not be possible without the unwavering support of family, friends, colleagues, mentors, acquaintances, and an awesome community. This is my first book, a dream come true, so please forgive me if I go overboard with my thanks.

To Eric, thank you... for finding me. Despite long days, sleepless nights, lengthy writing marathons, one smile from you never fails to wipe away my tiredness. Thank you for always supporting me, for believing in me, for helping me reach whichever dreams I dare to chase. I look forward to our journey together—a lifelong of hopes, dreams, and happiness.

To Mama and Papa, I am the luckiest daughter to have you as my parents. Thank you for all the sacrifices you made for me and my brothers. Words are not enough to express how much we love you, and how grateful we will always be.

To JR and RR—you will always be my baby brothers, and I am so proud to be your big sis. To Lisa, my dear sis-in-law, thank you for being part of our family. The whole family adores you. To Veronica, thanks for keeping up with the Santos' quirks. You're cool, girl! Now that the book is done, we can all play more Kinect, Acquire, and Ticket to Ride.

To my in laws—Mom Lisa, Dad Richard, Ama, Aunt Rose, Catherine, David, and Jayden—thank you for always making me feel welcome, for never making me feel I am different from your family. And to my unborn niece Kristina, auntie will teach you and Jayden SQL Server... one of these years.

To Edwin Sarmiento and Laerte Junior—my utmost and sincerest thanks for all the advice and constructive feedback. I have the highest respect for both of you. It is very humbling to work with both of you, and I learned so much from all the corrections and suggestions. Thank you for bearing with me through the revisions, despite your respective hectic schedules and numerous other commitments. I am very grateful.

To Elsie Au, thank you for introducing me to databases. I cannot imagine doing anything else. Thank you for the friendship all these years. To Kevin Cudihee, thank you for all the support all these years, for letting me do two things that I love the most—teaching and SQL Server. To Anne Marie Johnston and Alan Marchant, thank you for giving me fun work with databases. To my students, thank you for learning, sharing, and growing with me.

To BCIT—my second home. To me, BCIT was my place of refuge. When I was at a low point in my life, feeling down and out, and without direction (and afraid of computers!), BCIT provided me a place to learn, grow, and dream again. Now as an instructor, I hope I can help give back to students what BCIT gave me when I was one.

To the SQL community, the SQL family, and the SQL Server MVPs—I am so proud to be part of this group. There are so many smart SQL rockstars that I admire (Brent Ozar, Glenn Berry, Kevin Kline, Brian Knight, Grant Fritchey, Jorge Serragarrá, Jeremiah Peschka, Jen Stirrup, and so many others I would love to mention and thank), who are way up there, yet who are always ready to help and inspire anyone who asks. "Community" for this group is not just lip service. It's *the* SQL way of life. I have learned so much from this community, and I would not be anywhere near where I am today if not for the selfless way this community shares and helps.

To the PowerShell community, thank you to the awesome authors, bloggers, and tweeps. Your articles, blogs, and books have immensely helped folks like me to learn, understand, and get excited about PowerShell. To Microsoft and the SQL Server and PowerShell respective Product Teams—thanks for creating these two amazing products. It doubles the fun for SQL geeks like me!

To the Packt team—Dhwani Dewater, Yashodhan Dere, Azharuddin Sheikh, Charmaine Pereira, Sharvari Baet and the rest of the editors and technical reviewers—thank you for giving me the chance to write this book and helping me as the book writing progressed. It is one of the most humbling, but also one of the most rewarding experiences.

To numerous friends (Shereen Qumsieh, Matthew Carriere, Grace Dimaculangan, Ben Peach, Yaroslav Pentsarsky, Joe Xing, Min Zhu, Mary Mootatamby, Blake Wiggs, and many others), to all of my mentors and students, acquaintances via twitter (such as @pinaldave, @dsfnet, @StangSCT, @retracement, @NikoNeugebauer, @TimCost), and so many others who have helped, inspired, and encouraged me along the way—thank you.

And most importantly, thank you Lord, for all the miracles and blessings in my life.

About the Reviewers

Edwin Sarmiento is a Microsoft SQL Server MVP from Ottawa, Canada specializing in high availability, disaster recovery, and system infrastructures running on the Microsoft server technology stack. He is very passionate about technology but has interests in music, professional and organizational development, leadership, and management matters when not working with databases. He lives up to his primary mission statement—*To help people and organizations grow and develop their full potential as God has planned for them.*

He wants the whole world to know that the FILIPINO is a world-class citizen and brings *Jesus Christ* to the world.

Laerte Poltronieri Junior started in the IT world early, at the age of 12. When 16, he was developing software using Clipper Summer 85 and he used almost all versions. Then in 1998 he was introduced to SQL Server 6.5; since then it was love at first sight and marriage. In 2008, he met PowerShell and as he is an aficionado for automated, smart, and flexible solutions in SQL Server, from this marriage was born a son. And today they are a happy family.

Currently, he is writing a book for Manning Publications.

First of all, I would like to thank God. I have not always been a guy next to him, but I'm learning to give back all the love and affection that he has given me.

My family—my father, an unforgettable super-hero, my beloved mother and grandma, and my dear sister and nephews.

Also, a special thanks to some exceptional professionals and friends who are teaching and mentoring me from the beginning: Buck Woody, Chad Miller, Shay Levy, and Ravikanth Chaganti.

And last but not the least, all the #sqlfamily , #powershell and Simple-Talk friends, you guys simply rock. I owe you all the good things that happened and are happening to me.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following [@PacktEnterprise](https://twitter.com/PacktEnterprise) on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface	1
Chapter 1: Getting Started with SQL Server and PowerShell	7
Introduction	7
Before you start: Working with SQL Server and PowerShell	10
Working with the sample code	12
Exploring the SQL Server PowerShell hierarchy	14
Installing SMO	18
Loading SMO assemblies	20
Discovering SQL-related cmdlets and modules	22
Creating a SQL Server instance object	29
Exploring SMO server objects	32
Chapter 2: SQL Server and PowerShell Basic Tasks	35
Introduction	36
Listing SQL Server instances	39
Discovering SQL Server services	43
Starting/stopping SQL Server services	45
Listing SQL Server configuration settings	51
Changing SQL Server instance configurations	55
Searching for database objects	60
Creating a database	67
Altering database properties	68
Dropping a database	72
Changing a database owner	73
Creating a table	75
Creating a view	81
Creating a stored procedure	85
Creating a trigger	90
Creating an index	95

Executing a query / SQL script	99
Performing bulk export using Invoke-Sqlcmd	100
Performing bulk export using bcp	102
Performing bulk import using BULK INSERT	105
Performing bulk import using bcp	110
Chapter 3: Basic Administration	115
Introduction	116
Creating a SQL Server instance inventory	116
Creating a SQL Server database inventory	120
Listing installed hotfixes and service packs	124
Listing running/blocking processes	128
Killing a blocking process	131
Checking disk space usage	133
Setting up WMI Server event alerts	136
Detaching a database	143
Attaching a database	145
Copying a database	149
Executing a SQL query to multiple servers	152
Creating a filegroup	153
Adding secondary data files to a filegroup	156
Moving an index to a different filegroup	158
Checking index fragmentation	162
Reorganizing/rebuilding an index	164
Running DBCC commands	167
Setting up Database Mail	168
Listing SQL Server jobs	178
Adding a SQL Server operator	181
Creating a SQL Server job	183
Adding a SQL Server event alert	187
Running a SQL Server job	190
Scheduling a SQL Server job	192
Chapter 4: Security	203
Introduction	203
Listing SQL Server service accounts	204
Changing SQL Server service account	206
Listing authentication modes	210
Changing authentication mode	211
Listing SQL Server log errors	215
Listing failed login attempts	220
Listing logins, users, and database mappings	222

Listing login/user roles and permissions	225
Creating a login	227
Assigning permissions and roles to a login	229
Creating a database user	232
Assigning permissions to a database user	234
Creating a database role	237
Fixing orphaned users	241
Creating a credential	244
Creating a proxy	246
Chapter 5: Advanced Administration	251
<hr/>	
Introduction	252
Listing facets and facet properties	252
Listing policies	254
Exporting a policy	257
Importing a policy	261
Creating a condition	264
Creating a policy	268
Evaluating a policy	272
Enabling/disabling change tracking	275
Running and saving a profiler trace event	276
Extracting the contents of a trace file	284
Creating a database master key	289
Creating a certificate	291
Creating symmetric and asymmetric keys	293
Setting up Transparent Data Encryption (TDE)	299
Chapter 6: Backup and Restore	305
<hr/>	
Introduction	305
Changing database recovery model	306
Listing backup history	309
Creating a backup device	310
Listing backup header and file list information	312
Creating a full backup	316
Creating a backup on mirrored media sets	321
Creating a differential backup	324
Creating a transaction log backup	327
Creating a filegroup backup	329
Restoring a database to a point in time	332
Performing an online piecemeal restore	342

Chapter 7: SQL Server Development	351
Introduction	351
Inserting XML into SQL Server	352
Extracting XML from SQL Server	355
Creating an RSS feed from SQL Server content	358
Applying XSL to an RSS feed	363
Storing binary data into SQL Server	366
Extracting binary data from SQL Server	370
Creating a new assembly	374
Listing user-defined assemblies	378
Extracting user-defined assemblies	379
Chapter 8: Business Intelligence	385
Introduction	386
Listing items in your SSRS Report Server	386
Listing SSRS report properties	388
Using ReportViewer to view your SSRS report	391
Downloading an SSRS report in Excel and PDF	396
Creating an SSRS folder	400
Creating an SSRS data source	404
Changing an SSRS report's data source reference	409
Uploading an SSRS report to Report Manager	412
Downloading all SSRS report RDL files	416
Adding a user with a role to an SSRS report	421
Creating folders in an SSIS package store and MSDB	425
Deploying an SSIS package to the package store	428
Executing an SSIS package stored in the package store or File System	430
Downloading an SSIS package to a file	433
Creating an SSISDB catalog	435
Creating an SSISDB folder	439
Deploying an ISPAC file to SSISDB	441
Executing an SSIS package stored in SSISDB	444
Listing SSAS cmdlets	447
Listing SSAS instance properties	448
Backing up an SSAS database	450
Restoring an SSAS database	451
Processing an SSAS cube	452
Chapter 9: Helpful PowerShell Snippets	455
Introduction	456
Documenting PowerShell script for Get-Help	456
Getting a timestamp	459

Getting additional error messages	461
Listing processes	462
Getting aliases	466
Exporting to CSV and XML	467
Using Invoke-Expression	468
Testing regular expressions	470
Managing folders	474
Manipulating files	476
Searching for files	478
Reading an event log	481
Sending e-mail	482
Embedding C# code	484
Creating an HTML report	486
Parsing XML	488
Extracting data from a web service	490
Using PowerShell Remoting	492
Appendix A: SQL Server and PowerShell CheatSheet	497
Learning PowerShell	497
PowerShell V2 versus V3 Where-Object syntax	498
Changing execution policy	498
Running a script	499
Common aliases	499
Displaying output	500
Special characters	500
Special variables	501
Common operators	502
Common date-time format strings	502
Comment based help	503
Here-string	504
Common regex characters and patterns	504
Arrays and hash tables	505
Arrays and loops	506
Logic	506
Functions	507
Common Cmdlets	508
Import SQLPS module	509
Add SQL Server Snapins	509
Add SQL Server Assemblies	509
Getting credentials	510
Running and blocking SQL Server processes	510
Read file into an array	510

SQL Server-Specific Cmdlets	510
Invoke-SqlCmd	512
Create SMO Server Object	512
Create SSRS Proxy Object	512
Create SSIS Object (SQL Server 2005/2008/2008R2)	513
Create an SSIS Object (SQL Server 2012)	513
Create SSAS Object	513
Appendix B: PowerShell Primer	515
Introduction	515
What is PowerShell, and why learn another language	515
Setting up the Environment	516
Running PowerShell scripts	517
Basics—points to remember	520
Scripting syntax	527
Converting script into functions	539
More about PowerShell	542
Appendix C: Resources	543
Resources	543
Appendix D: Creating a SQL Server VM	549
Introduction	549
Terminology	550
Downloading software	551
VM details and accounts	552
Creating an empty virtual machine	553
Installing Windows Server 2008 R2 as	
Guest OS	556
Installing VMWare tools	567
Configuring a domain controller	569
Creating domain accounts	577
Installing SQL Server 2012 on a VM	580
Installing sample databases	598
Installing PowerShell V3	598
Index	601

Preface

PowerShell is Microsoft's new command-line shell and scripting language that promises to simplify automation and integration across different Microsoft applications and components. Database professionals can leverage PowerShell by utilizing its numerous built-in cmdlets, or using any of the readily available .NET classes, to automate database tasks, simplify integration, or just discover new ways to accomplish the job at hand.

SQL Server 2012 with PowerShell V3 Cookbook provides easy-to-follow, practical examples for the busy database professional. Whether you're auditing your servers, or exporting data, or deploying reports, there is a recipe that you can use right away!

You start off with basic topics to get you going with SQL Server and PowerShell scripts and progress into more advanced topics to help you manage and administer your SQL Server databases.

The first few chapters demonstrate how to work with SQL Server settings and objects, including exploring objects, creating databases, configuring server settings, and performing inventories. The book then dives deeply into more administration topics such as backup and restore, credentials, policies, and jobs.

Additional development and BI-specific topics are also explored, including deploying and downloading assemblies, BLOB data, SSIS packages, and SSRS reports.

A short PowerShell primer is also provided as a supplement in the Appendix, which the database professional can use as a refresher or occasional reference material. Packed with more than 100 practical, ready-to-use scripts, *SQL Server 2012 with PowerShell V3 Cookbook* will be your go-to reference in automating and managing SQL Server.

What this book covers

Chapter 1, Getting Started with SQL Server and PowerShell explains what PowerShell is, and why you should consider learning PowerShell. It also introduces PowerShell V3 new features, and explains what needs to be in place when working with SQL Server 2012 and PowerShell.

Chapter 2, SQL Server and PowerShell Basic Tasks demonstrates scripts and snippets of code that accomplish some basic SQL Server tasks using PowerShell. We 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 while working with SQL Server programmatically.

Chapter 3, Basic Administration tackles more administrative tasks that can be accomplished using PowerShell, and provides recipes that can help automate a lot of repetitive tasks. Some recipes deal with instance and database properties; others provide ways of checking disk space, creating WMI alerts, setting up Database Mail, and creating and maintaining SQL Server Jobs.

Chapter 4, Security provides snippets that simplify security monitoring, including how to check failed login attempts by parsing out event logs, or how to administer roles and permissions.

Chapter 5, Advanced Administration shows how PowerShell can help you leverage features such as Policy Based Management (PBM) and encryption using PowerShell. This chapter also explores working with SQL Server Profiler trace files and events programmatically.

Chapter 6, Backup and Restore looks into different ways of backing up and restoring SQL Server databases programmatically using PowerShell.

Chapter 7, SQL Server Development provides snippets and guidance on how you can work with XML, XSL, binary data, and CLR assemblies with SQL Server and PowerShell.

Chapter 8, Business Intelligence covers how PowerShell can help automate and manage any BI-related tasks—from rendering SQL Server Reporting Services (SSRS) reports, to deploying the new SQL Server Integration Services (SSIS) 2012 ISPAC files, to backing up and restoring SQL Server Analysis Services (SSAS) cubes.

Chapter 9, Helpful PowerShell Snippets tackles a variety of recipes that are not SQL Server specific, but you may find them useful as you work with PowerShell. Recipes include snippets for creating files that use timestamps, analyzing event logs for recent system errors, and exporting a list of processes to CSV or XML.

Appendix A, SQL Server and PowerShell CheatSheet provides a concise cheatsheet of commonly used terms and snippets when working with SQL Server and PowerShell.

Appendix B, PowerShell Primer offers a brief primer on PowerShell fundamentals.

Appendix C, Resources lists additional PowerShell and SQL Server books, blogs and links.

Appendix D, Creating a SQL Server VM provides a step-by-step tutorial on how to create and configure the virtual machine that was used for this book.

What you need for this book

Windows Server 2008 R2

SQL Server 2012 Developer

Visual Studio 2010 Professional

Windows Management Framework 3.0 (includes PowerShell 3.0, WMI, and WinRM)

Who this book is for

This book is written for the SQL Server database professional (DBA, developer, BI developer) who wants to use PowerShell to automate, integrate, and simplify database tasks. A little bit of scripting background is helpful, but not necessary.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

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

#list server instances
$managedComputer.ServerInstances
```



When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:



```
$instanceName = "KERRIGAN"
$managedComputer = New-Object 'Microsoft.SqlServer.Management.Smo.Wmi.
ManagedComputer' $instanceName
#list server instances
$managedComputer.ServerInstances
```

Any command-line input or output is written as follows:

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

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

 Warnings or important notes appear in a box like this. 

 Tips and tricks appear like this. 

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

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 `PScheduledJob` 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.

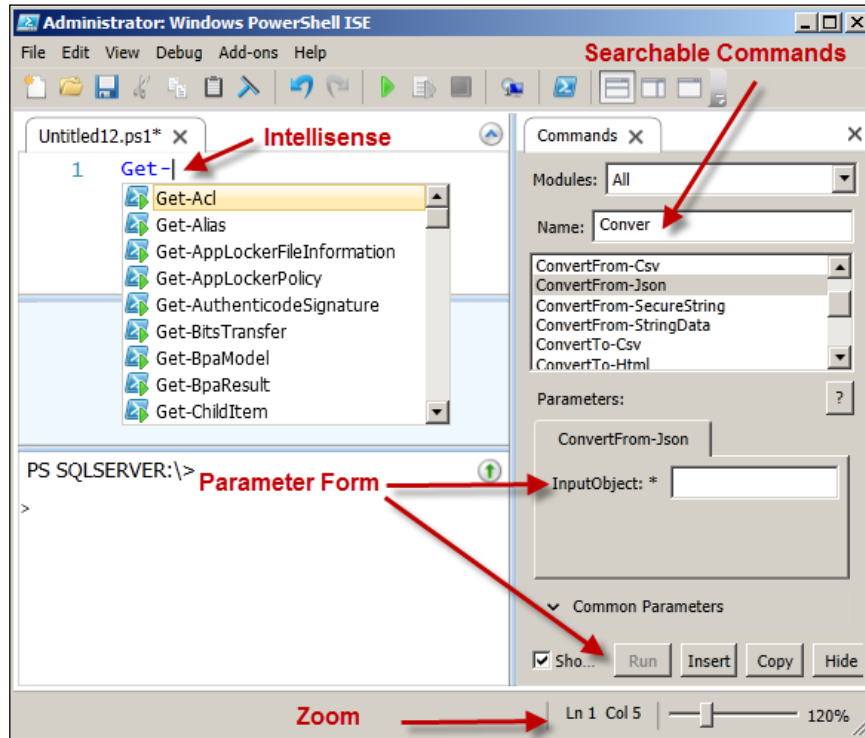
What you used to write in V1 and V2 with curly braces and `$_` as follows:

```
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](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.

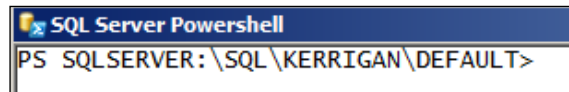
Getting ready

Log in to SQL Server 2012 Management Studio.

How to do it...

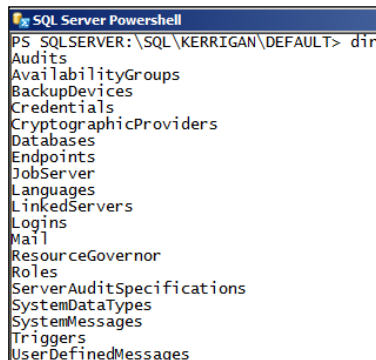
In this recipe, we will navigate the SQL Server PowerShell hierarchy by launching a PowerShell session from SQL Server Management Studio:

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:

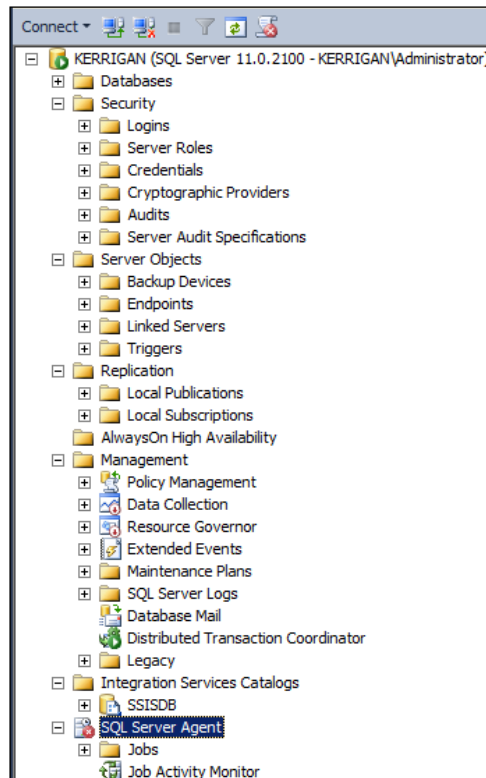


Note the starting path in this window.

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

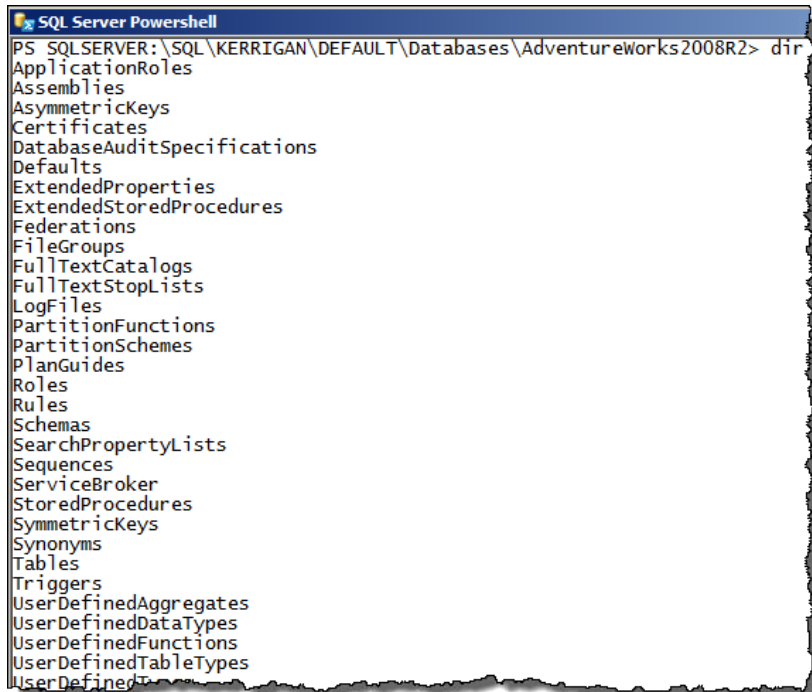
Name            Root                Description
-----
SQL             SQLSERVER:\SQL      SQL Server Database Engine
SQLPolicy       SQLSERVER:\SQLPolicy SQL Server Policy Management
SQLRegistration SQLSERVER:\SQLRegistration SQL Server Registrations
DataCollection SQLSERVER:\DataCollection SQL Server Data Collection
XEvent          SQLSERVER:\XEvent   SQL Server Extended Events
Utility         SQLSERVER:\Utility  SQL Server Utility
DAC            SQLSERVER:\DAC      SQL Server Data-Tier Application Component
IntegrationServices SQLSERVER:\IntegrationServices SQL Server Integration Services
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:

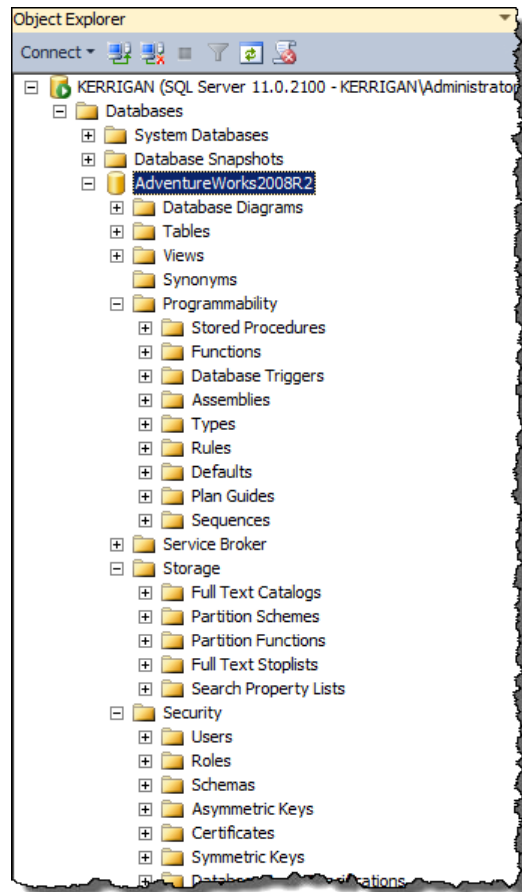


```
SQL Server Powershell
PS SQLSERVER:\SQL\KERRIGAN\DEFAULT\Databases\Adventureworks2008R2>
```



```
SQL Server Powershell
PS SQLSERVER:\SQL\KERRIGAN\DEFAULT\Databases\Adventureworks2008R2> dir
ApplicationRoles
Assemblies
AsymmetricKeys
Certificates
DatabaseAuditSpecifications
Defaults
ExtendedProperties
ExtendedStoredProcedures
Federations
FileGroups
FullTextCatalogs
FullTextStopLists
LogFiles
PartitionFunctions
PartitionSchemes
PlanGuides
Roles
Rules
Schemas
SearchPropertyLists
Sequences
ServiceBroker
StoredProcedures
SymmetricKeys
Synonyms
Tables
Triggers
UserDefinedAggregates
UserDefinedDataTypes
UserDefinedFunctions
UserDefinedTableTypes
UserDefinedTypes
```

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 Powershell
PS SQLSERVER:\SQL\KERRIGAN\DEFAULT\Databases\Adventureworks2008R2>
```


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

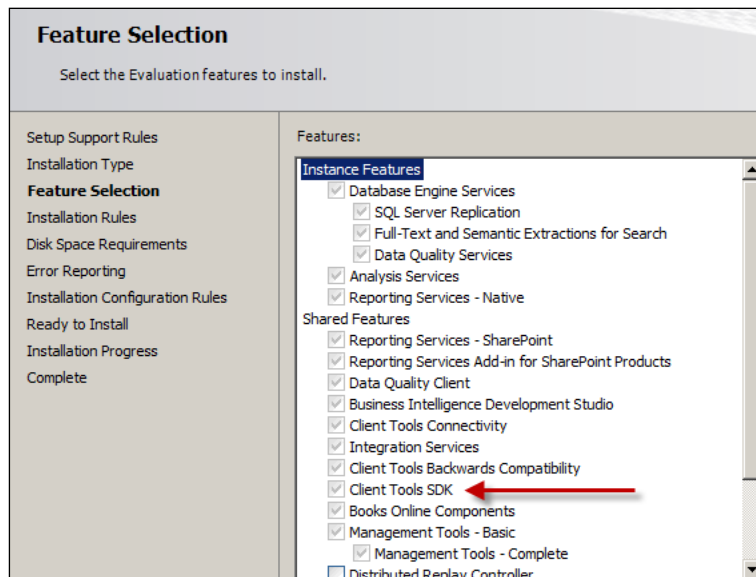
There are a few ways to get SMO installed:

- ▶ 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.

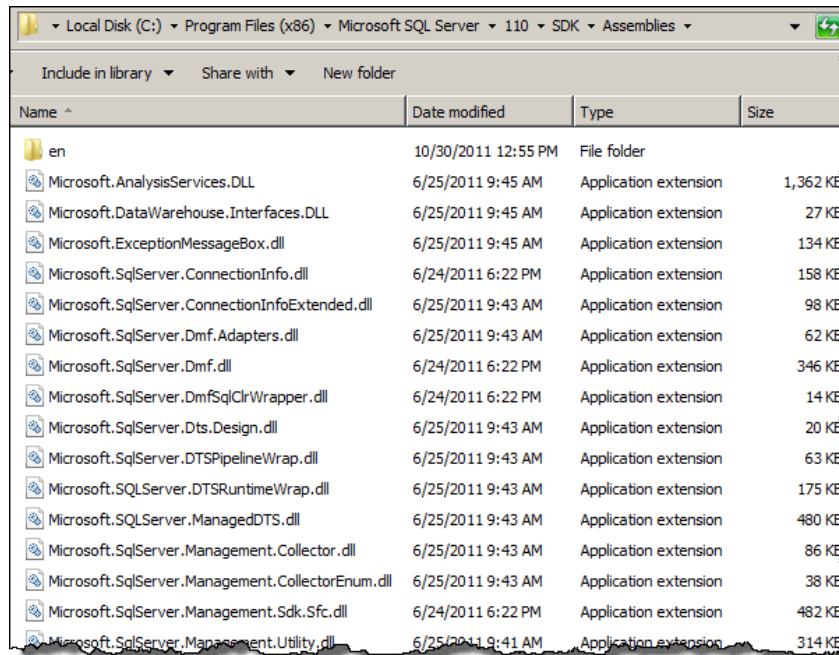
After this, you should already have all the binaries needed to use SMO.

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**.



Name	Date modified	Type	Size
en	10/30/2011 12:55 PM	File folder	
Microsoft.AnalysisServices.DLL	6/25/2011 9:45 AM	Application extension	1,362 KB
Microsoft.DataWarehouse.Interfaces.DLL	6/25/2011 9:45 AM	Application extension	27 KB
Microsoft.ExceptionMessageBox.dll	6/25/2011 9:45 AM	Application extension	134 KB
Microsoft.SqlServer.ConnectionInfo.dll	6/24/2011 6:22 PM	Application extension	158 KB
Microsoft.SqlServer.ConnectionInfoExtended.dll	6/25/2011 9:43 AM	Application extension	98 KB
Microsoft.SqlServer.Dmf.Adapters.dll	6/25/2011 9:43 AM	Application extension	62 KB
Microsoft.SqlServer.Dmf.dll	6/24/2011 6:22 PM	Application extension	346 KB
Microsoft.SqlServer.DmfSqlClrWrapper.dll	6/24/2011 6:22 PM	Application extension	14 KB
Microsoft.SqlServer.Dts.Design.dll	6/25/2011 9:43 AM	Application extension	20 KB
Microsoft.SqlServer.DTSPipelineWrap.dll	6/25/2011 9:43 AM	Application extension	63 KB
Microsoft.SQLServer.DTSRuntimeWrap.dll	6/25/2011 9:43 AM	Application extension	175 KB
Microsoft.SQLServer.ManagedDTS.dll	6/25/2011 9:43 AM	Application extension	480 KB
Microsoft.SqlServer.Management.Collector.dll	6/25/2011 9:43 AM	Application extension	86 KB
Microsoft.SqlServer.Management.CollectorEnum.dll	6/25/2011 9:43 AM	Application extension	38 KB
Microsoft.SqlServer.Management.Sdk.Sfc.dll	6/24/2011 6:22 PM	Application extension	482 KB
Microsoft.SqlServer.Management.Utility.dll	6/25/2011 9:41 AM	Application extension	314 KB

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"
```

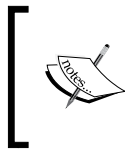
In PowerShell V3, loading these assemblies one by one is no longer necessary as long as the `SQLPS` module is loaded:

```
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.

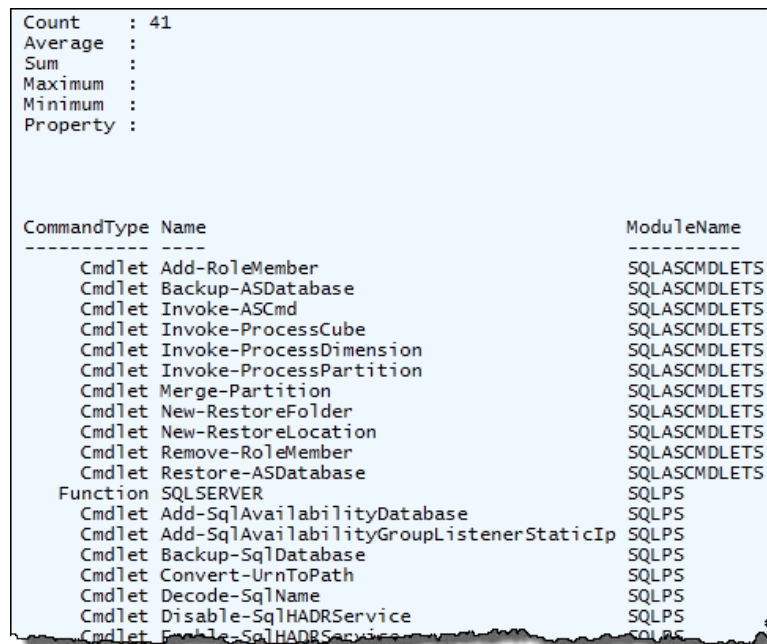
1. To discover SQL-related cmdlets, type the following in your PowerShell editor and run:

```
#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
```

- 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.

- 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	<code>Get-PSSnapin</code>
List installed snap-ins	<code>Get-PSSnapin -Registered</code>
Show commands in a snap-in	<code>Get-Command -Module "SnapinName"</code>
Load a specific snap-in	<code>Add-PSSnapin "SnapinName"</code>

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

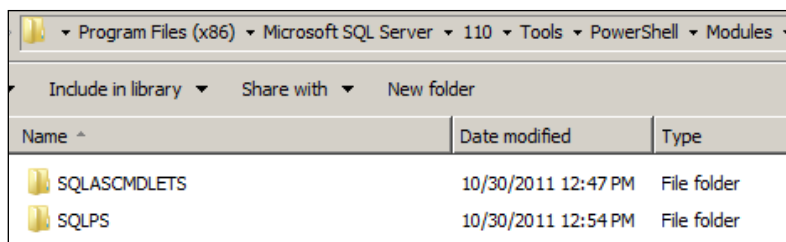
A module is a package that can contain cmdlets, providers, functions, variables, and aliases. In PowerShell V2, modules are not loaded by default, so required modules need to be explicitly imported.

Common tasks with modules are summarized in the following table:

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

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:

```

FullName
-----
Microsoft.SqlServer.Smo, Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845d...
Microsoft.SqlServer.Dmf, Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845d...
Microsoft.SqlServer.SqlWmiManagement, Version=11.0.0.0, Culture=neutral, PublicKey...
Microsoft.SqlServer.ConnectionInfo, Version=11.0.0.0, Culture=neutral, PublicKeyT...
Microsoft.SqlServer.SmoExtended, Version=11.0.0.0, Culture=neutral, PublicKeyToken...
Microsoft.SqlServer.Management.RegisteredServers, Version=11.0.0.0, Culture=neutra...
Microsoft.SqlServer.Management.Sdk.Sfc, Version=11.0.0.0, Culture=neutral, Public...
Microsoft.SqlServer.SqlEnum, Version=11.0.0.0, Culture=neutral, PublicKeyToken=898...
Microsoft.SqlServer.RegSvrEnum, Version=11.0.0.0, Culture=neutral, PublicKeyToken...
Microsoft.SqlServer.WmiEnum, Version=11.0.0.0, Culture=neutral, PublicKeyToken=898...
Microsoft.SqlServer.ServiceBrokerEnum, Version=11.0.0.0, Culture=neutral, Publick...
Microsoft.SqlServer.Management.Collector, Version=11.0.0.0, Culture=neutral, Publ...
Microsoft.SqlServer.Management.CollectorEnum, Version=11.0.0.0, Culture=neutral, ...
Microsoft.SqlServer.Management.Utility, Version=11.0.0.0, Culture=neutral, Public...
Microsoft.SqlServer.Management.UtilityEnum, Version=11.0.0.0, Culture=neutral, Pub...
Microsoft.SqlServer.Management.HadrDMF, Version=11.0.0.0, Culture=neutral, Public...
Microsoft.SqlServer.Management.PSSnapins, Version=11.0.0.0, Culture=neutral, Publ...
Microsoft.SqlServer.Management.PSProvider, Version=11.0.0.0, Culture=neutral, Pub...
Microsoft.SqlServer.SqlClrProvider, Version=11.0.0.0, Culture=neutral, PublicKeyT...

```

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](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

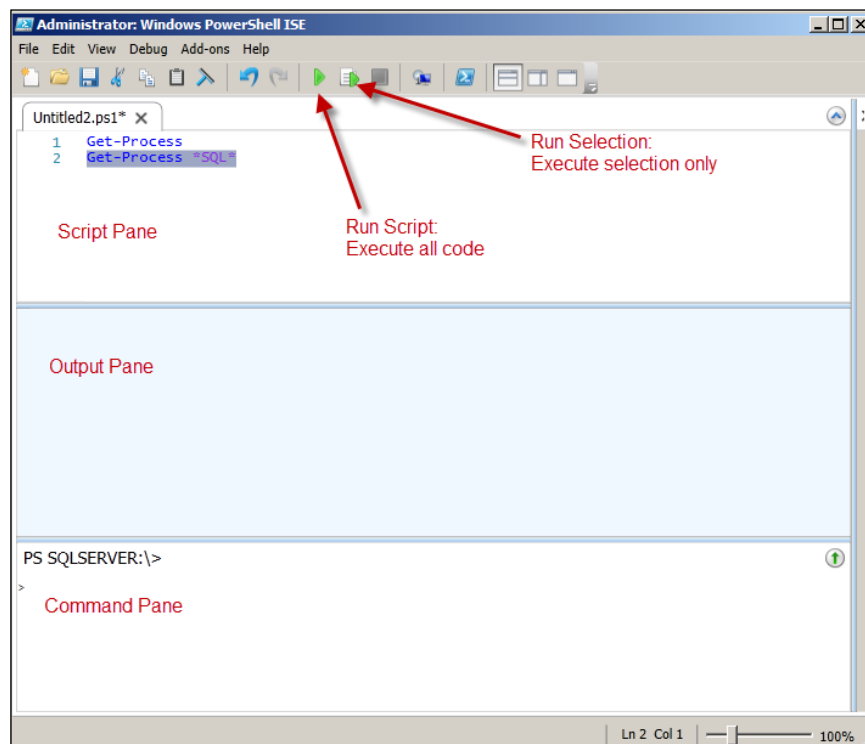
We will be using the PowerShell ISE for all the scripts in this task. These are some things you need to remember.

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
```

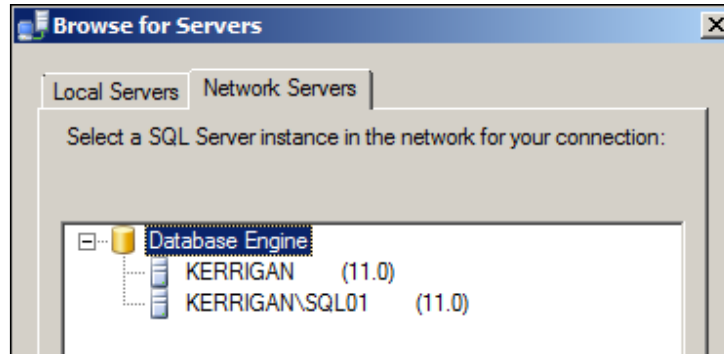
Your result should look similar to the one shown in the following screenshot:

```
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:

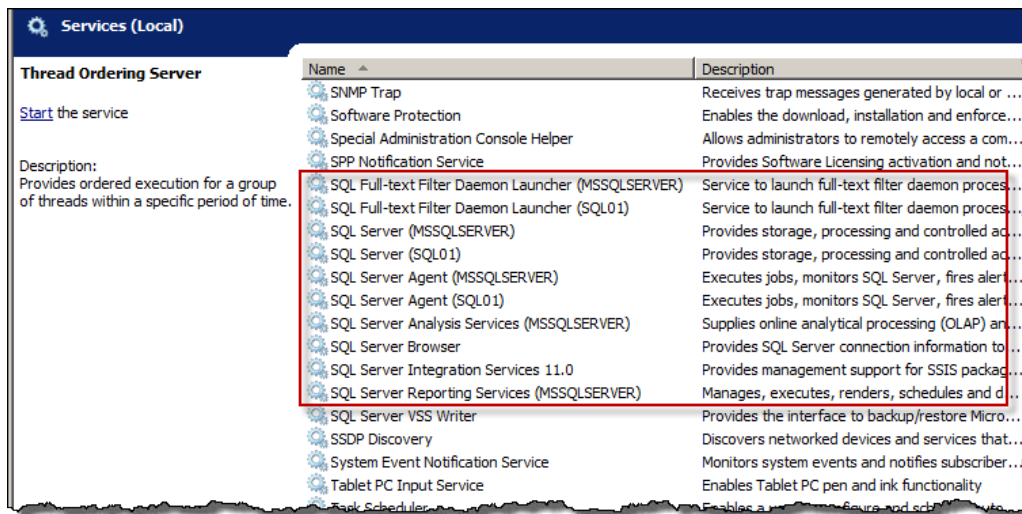
ServiceName	DisplayName	SQLServiceType
MSSQL\$SQL01	SQL Server (SQL01)	1
MSSQLSERVER	SQL Server (MSSQLSERVER)	1

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

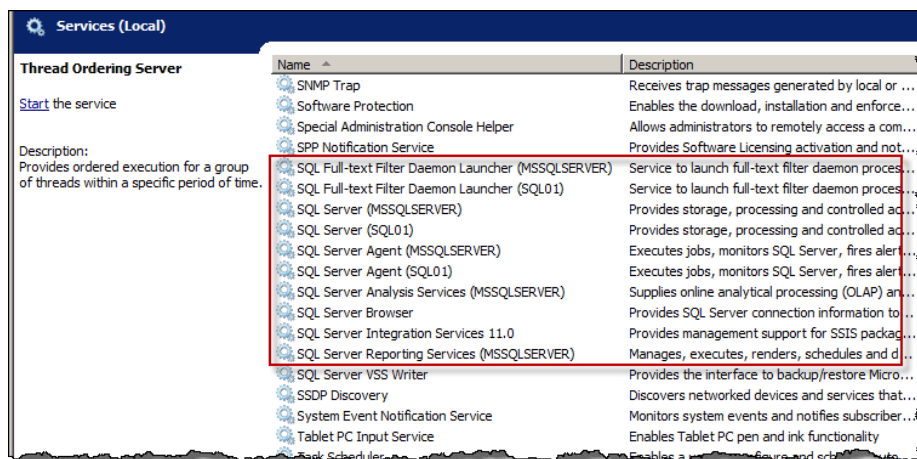
- ▶ The *Listing SQL Server instances* recipe

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...

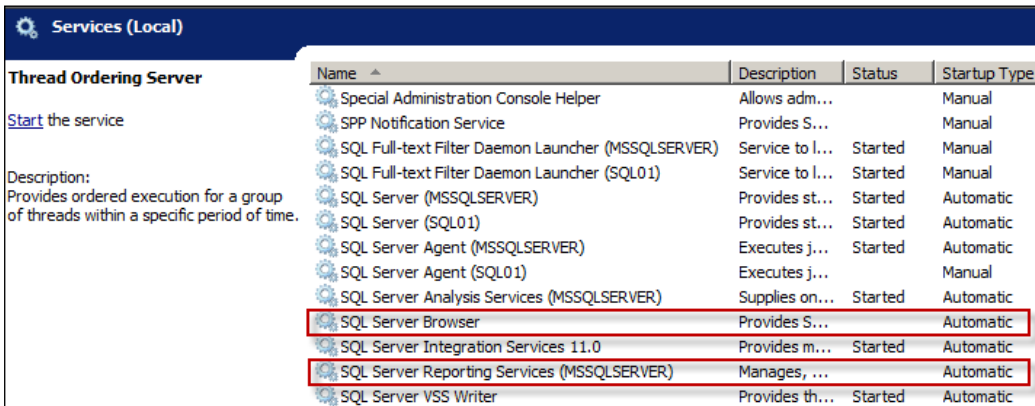
Let's look at the steps to toggle states for your SQL Server services:

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`.



Name	Description	Status	Startup Type
Special Administration Console Helper	Allows adm...		Manual
SPP Notification Service	Provides S...		Manual
SQL Full-text Filter Daemon Launcher (MSSQLSERVER)	Service to l...	Started	Manual
SQL Full-text Filter Daemon Launcher (SQL01)	Service to l...	Started	Manual
SQL Server (MSSQLSERVER)	Provides st...	Started	Automatic
SQL Server (SQL01)	Provides st...	Started	Automatic
SQL Server Agent (MSSQLSERVER)	Executes j...	Started	Automatic
SQL Server Agent (SQL01)	Executes j...		Manual
SQL Server Analysis Services (MSSQLSERVER)	Supplies on...	Started	Automatic
SQL Server Browser	Provides S...		Automatic
SQL Server Integration Services 11.0	Provides m...	Started	Automatic
SQL Server Reporting Services (MSSQLSERVER)	Manages, ...	Automatic	
SQL Server VSS Writer	Provides th...	Started	Automatic

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(strin
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
<code>Start()</code>	<code>Start-Service</code>
<code>Stop()</code>	<code>Stop-Service</code>
<code>Continue()</code>	<code>Resume-Service</code>
<code>Pause()</code>	<code>Suspend-Service</code>
<code>Refresh()</code>	<code>Restart-Service</code>

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"}
```

SQL Server and PowerShell Basic Tasks

```
#The Information class lists nonconfigurable 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 minutes
allow updates	Allow updates to system tables
user connections	Number of user connections allowed
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 triggers
nested triggers	Allow triggers to be invoked with recursion
server trigger recursion	Allow recursion for server level triggers
remote access	Allow remote access
default language	default language
cross db ownership chaining	Allow cross db ownership chaining
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 remote procedure calls
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 (KB)
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 <code>BuildNumber</code> , <code>Edition</code> , <code>OSVersion</code> , and <code>ProductLevel</code> . It also includes settings specified during install, for example <code>Collation</code> , <code>MasterDBPath</code> , and <code>MasterDBLogPath</code> .
Settings	Lists some instance-level configurable settings, such as <code>LoginMode</code> and <code>BackupDirectory</code> .
UserOptions	Contain options that can be set for user connections, such as <code>AnsiWarnings</code> , <code>AnsiNulls</code> , <code>AnsiPadding</code> , and <code>NoCount</code> .
Configuration	Instance-specific settings, such as <code>AgentXPs</code> , <code>remote access</code> , <code>clr enabled</code> , and <code>xp_cmdshell</code> , which you will normally see and set when you use the <code>sp_configure</code> 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
```

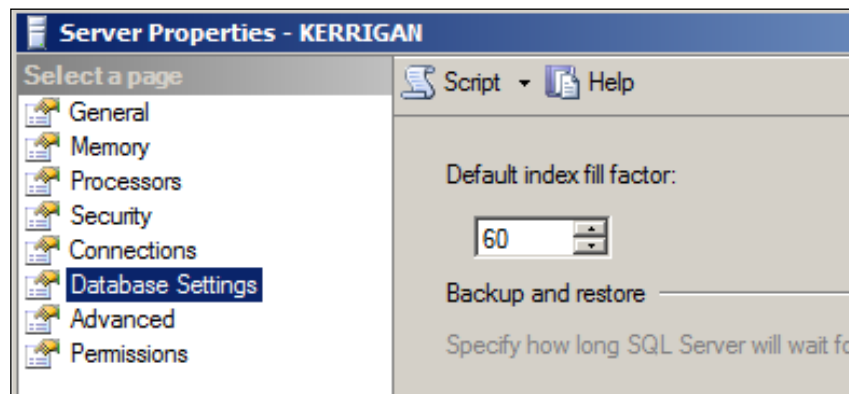
```
#change authentication mode
$server.Settings.LoginMode = [Microsoft.SqlServer.Management.Smo.
ServerLoginMode]::Mixed
$server.Alter()

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

4. Confirm the changes.

To confirm **fill factor**:

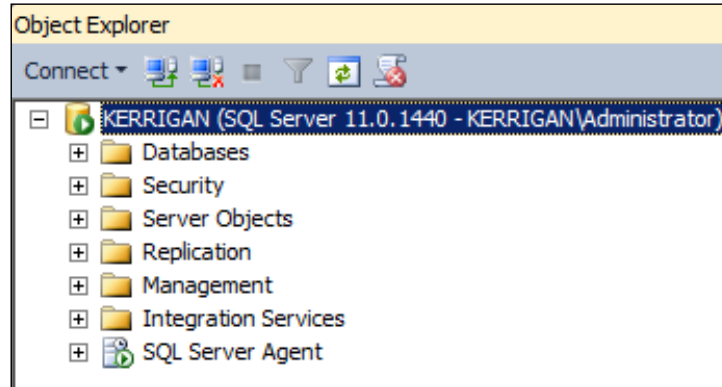
1. Go to **Management Studio**.
2. Connect to your instance.
3. Right-click on your instance and select **Properties**.
4. Go to **Database Settings**, and check whether your **fill factor** value has changed.



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

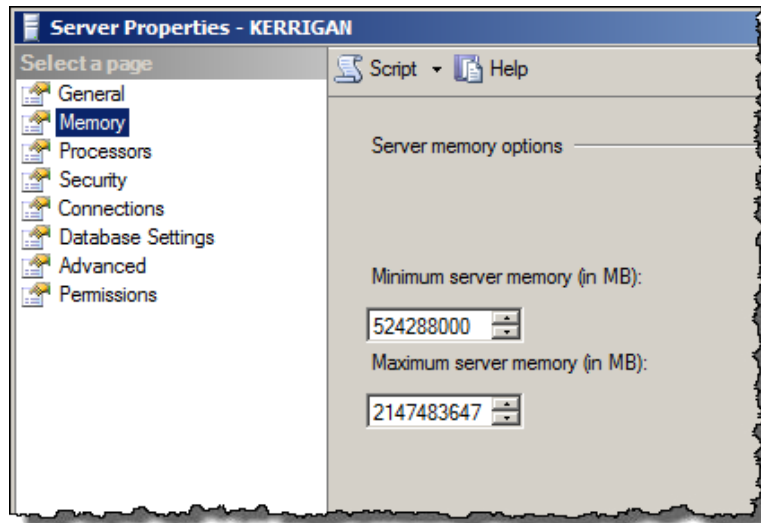
1. Go to **Management Studio**.
2. Connect to your instance.

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



To confirm **Minimum server memory**:

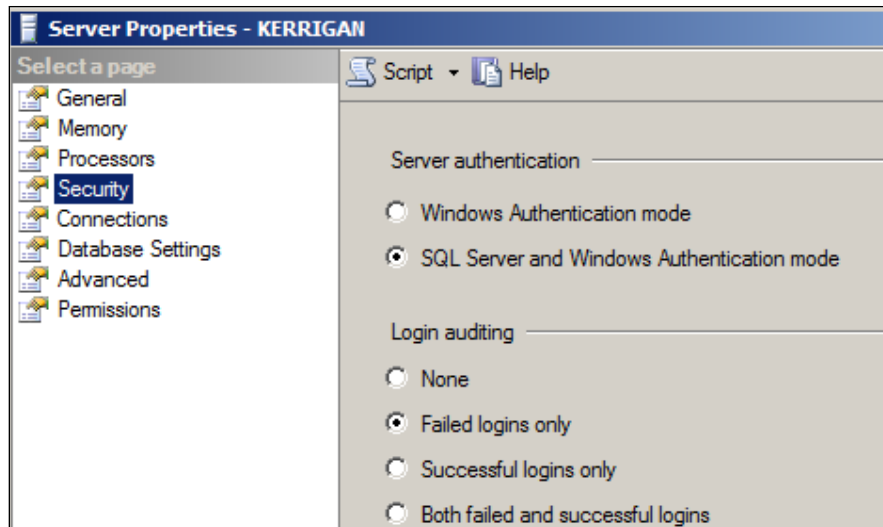
1. Go to **Management Studio**.
2. Right-click on your instance and select **Properties**.
3. Go to **Memory** and check that the value has changed to what you set it to.



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 compatibility 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()

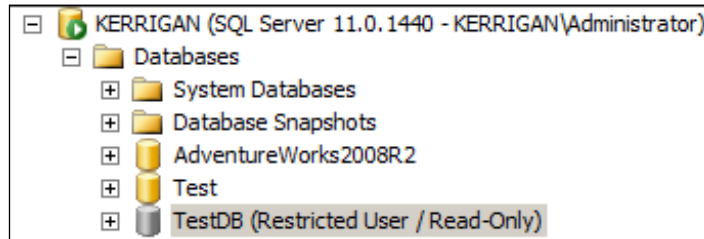
#set to readonly
$db.DatabaseOptions.ReadOnly = $true
$db.Alter()
```

4. Confirm the changes.

To start confirming:

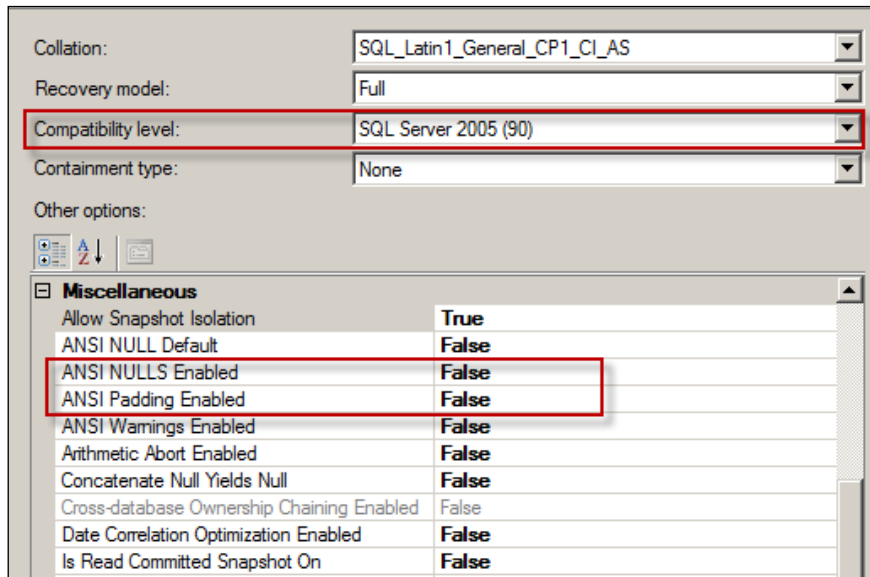
1. Go to **Management Studio**.
2. Connect to your instance.

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
```

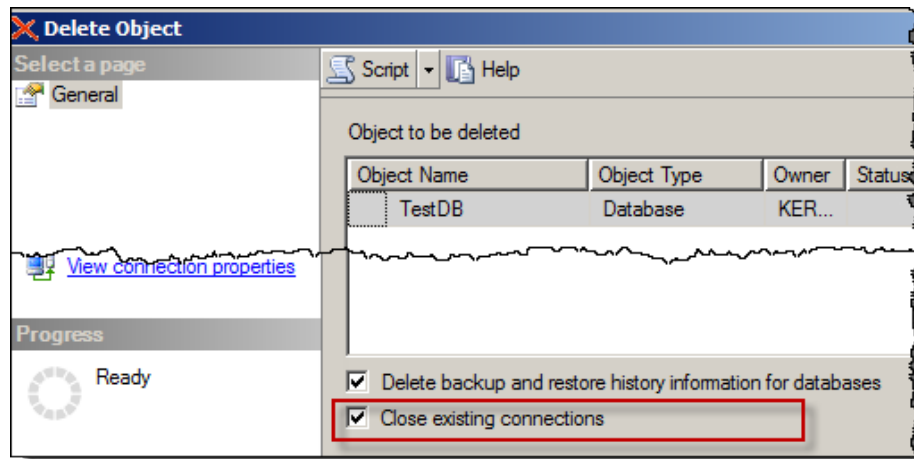
3. Add the following script and run it:

```
$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]

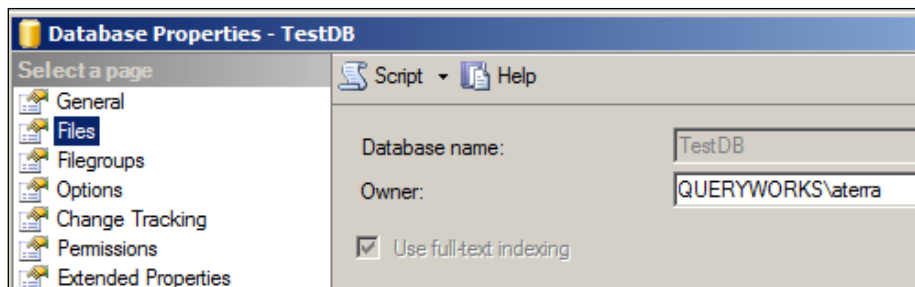
#display current owner
$db.Owner

#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()
```

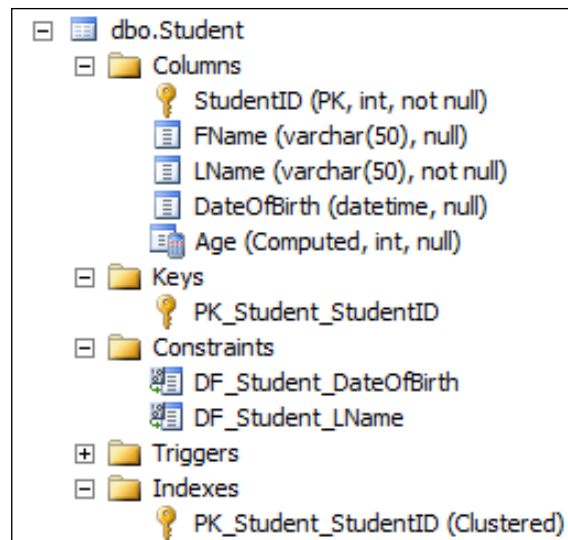
5. Make StudentID the primary key, as follows:

```
#####
#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:

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

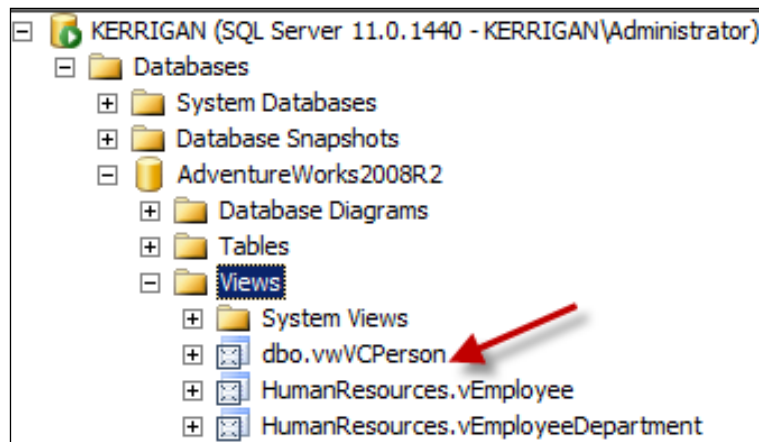
```
$view.Create()
```

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]

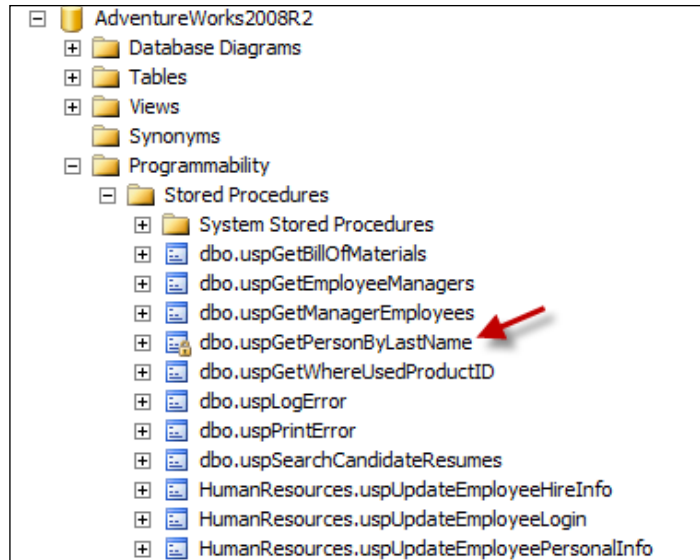
#storedProcedure class on MSDN:
#http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.
management.smo.storedprocedure.aspx

$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);
$params = 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,
```

```

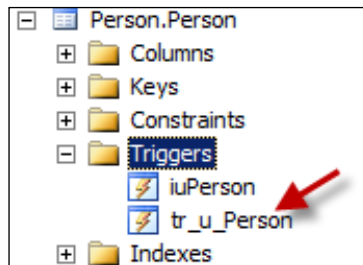
i.LastName AS NewLastName,
  i.FirstName AS NewFirstName,
  d.LastName AS OldLastName,
  d.FirstName AS OldFirstName
FROM
  inserted i
  INNER JOIN deleted d
  ON i.BusinessEntityID = d.BusinessEntityID

```

"@

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

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



- 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)

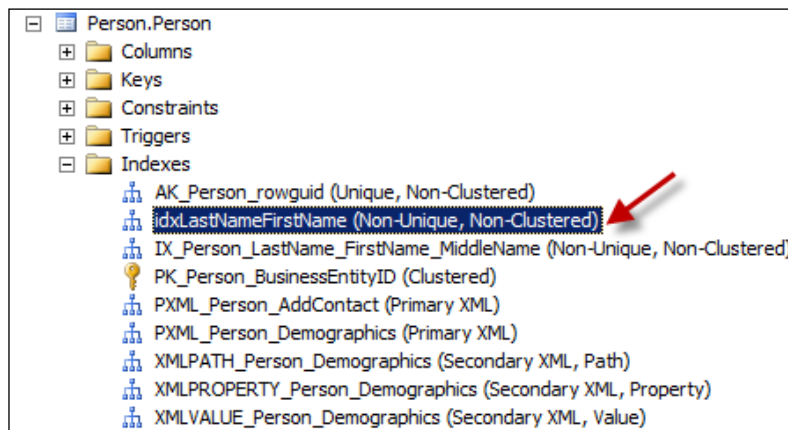
#second index column, by default sorted ascending
$idxCol2 = New-Object -TypeName Microsoft.SqlServer.Management.
SMO.IndexedColumn -ArgumentList $index, "FirstName", $false
$index.IndexedColumns.Add($idxCol2)

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

```
#second index column
$idxColl2 = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
IndexedColumn -ArgumentList $index, "FirstName", $false; #sort asc
$index.IndexedColumns.Add($idxColl2)
```

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


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	<code>HasFilter</code> <code>FilterDefinition</code>
FullText	<code>IsFullTextKey = \$true</code>
XML	<code>IsXMLIndex = \$true</code>
Spatial	<code>IsSpatialIndex = \$true</code>

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" `
-NoTypeInfoation

#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" `
-NoTypeInfoation
```

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 `-NoTypeInfo`, 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 -NoTypeInfo
```

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
}

```

- 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*

3

Basic Administration

In this chapter, we will cover:

- ▶ Creating a SQL Server instance inventory
- ▶ Creating a SQL Server database inventory
- ▶ Listing installed hotfixes and service packs
- ▶ Listing running/blocking processes
- ▶ Killing a blocking process
- ▶ Checking disk space usage
- ▶ Setting up WMI Server event alerts
- ▶ Detaching a database
- ▶ Attaching a database
- ▶ Copying a database
- ▶ Executing a SQL query to multiple servers
- ▶ Creating a filegroup
- ▶ Adding secondary data files to a filegroup
- ▶ Moving an index to a different filegroup
- ▶ Checking index fragmentation
- ▶ Reorganizing/rebuilding an index
- ▶ Running DBCC commands
- ▶ Setting up Database Mail
- ▶ Listing SQL Server jobs
- ▶ Adding a SQL Server operator
- ▶ Creating a SQL Server job

- ▶ Adding a SQL Server event alert
- ▶ Running a SQL Server job
- ▶ Scheduling a SQL Server job

Introduction

In this chapter, we will tackle some more administrative tasks that can be accomplished using PowerShell. PowerShell can help automate a lot of the repetitive, tedious, and mundane tasks that take many clicks to accomplish. We will look at ways to get SQL Server instance and database properties and log them to a file. We are also going to explore tasks such as checking disk space, creating WMI alerts, setting up Database Mail, and creating and maintaining SQL Server jobs.

Check out the *Introduction* section in *Chapter 2, SQL Server and PowerShell Basic Tasks*, for the development environment settings needed for the recipes in this chapter.

Creating a SQL Server instance inventory

In this recipe, we will export SQL Server instance properties to a text file.

How to do it...

4. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
5. Import the SQLPS module, and create a new SMO Server object as follows:

```
#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
```

6. Add the following script and run:

```
#specify folder and filename to be produced
$folder = "C:\Temp"
$currdate = Get-Date -Format "yyyy-MM-dd_hmmtt"
$filename = "$($instanceName)_InstanceInventory_$( $currdate ).csv"
$fullpath = Join-Path $folder $filename

#export all "server" object properties
#note we are using V3 simplified Where-Object syntax
```

```

$server |
Get-Member |
Where-Object Name -ne "SystemMessages" |
Where-Object MemberType -eq "Property" |
Select Name, @{Name="Value";Expression={$server.($_.Name)}} |
Export-Csv -Path $fullpath -NoTypeInfoation

#jobs are also extremely important to monitor, archive
#export all job names + last run date and result
$server.JobServer.Jobs |
Select @{Name="Name";Expression={"Job: $($_.Name)"}} ,
      @{Name="Value";Expression={"Last run: $($_.LastRunDate)
($($_.LastRunOutcome))" }} |
Export-Csv -Path $fullpath -NoTypeInfoation -Append

#show file in explorer
explorer $folder

```

How it works...

It is important to regularly take an inventory of your SQL Server instances, in other words, get a list of the instances and their properties, for auditing and archiving purposes. It will be easier to detect changes if you know what your baseline properties are.

There are different ways of extracting different SQL Server settings using PowerShell. What we will be using in this recipe is a fairly simple script, but exhaustive.

Let's dissect the first part first. Note that this block of code will work only in PowerShell V3 because of the simplified `Where-Object` syntax:

```

$server |
Get-Member |
Where-Object Name -ne "SystemMessages" |
Where-Object MemberType -eq "Property" |
Select Name, @{Name="Value";Expression={$server.($_.Name)}} |
Export-Csv -Path $fullpath -NoTypeInfoation

```

If you want to do this in V2, this is the equivalent block of code:

```

#export all "server" object properties
$server |
Get-Member |
Where {$_.MemberType -eq "Property" -and $_.Name -ne
"SystemMessages"} |
| Select Name, @{Name="Value";Expression={$server.($_.Name)}}
| Export-Csv -path $fullpath -noTypeInfoation

```

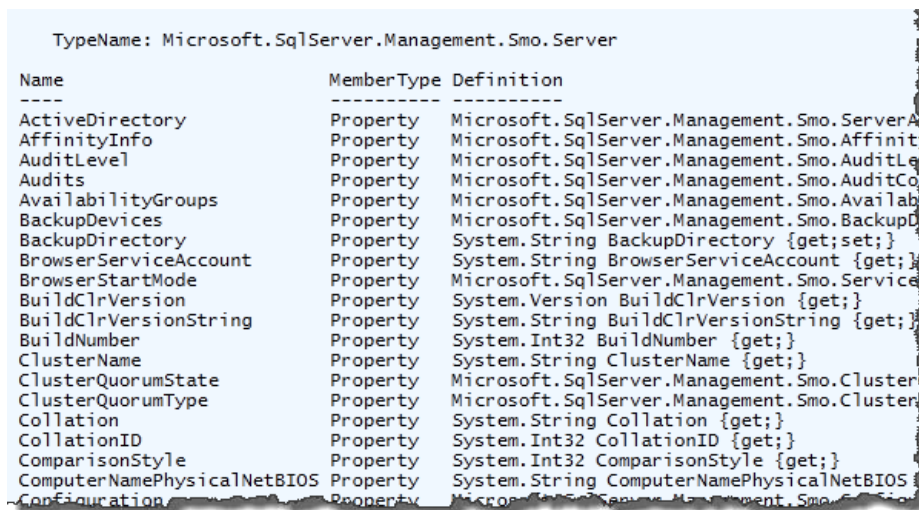
The first couple of lines retrieve all the properties and methods of the server object:

```
$server |
Get-Member |
```

The next part retrieves all the non-system message properties:

```
Where-Object Name -ne "SystemMessages" |
Where-Object MemberType -eq "Property" |
```

We filter out all system messages because these would clutter our inventory. This filter would normally lead to a result that looks similar to the one shown in the following screenshot:



Instead of displaying the results, we pipe the results to the line:

```
Select Name, @{Name="Value";Expression={$server.($_.Name)}} |
```

This is at the core of retrieving the inventory. The interim results containing the properties are piped to this line, and `$server.($_.Name)` retrieves the current property in the pipe. For example, if the current property in the pipeline is `Collation`, then this would be translated to `$server.Collation`.

The last part of this line exports the results to a text, **Comma-Separated Value (CSV)** file:

```
Export-Csv -Path $fullpath -NoTypeInfoation
```

This is not where we stop our script though. We append the job names of a server, including the last run date and last run result, to this file:

```

$server.JobServer.Jobs |
Select @{Name="Name";Expression={"Job: $($_.Name)"}} ,
       @{Name="Value";Expression={"Last run: $($_.LastRunDate) ($($_.
LastRunOutcome))" }} |
Export-Csv -Path $fullpath -NoTypeInformation -Append

```

For this line, we have to use `$server.JobServer.Jobs` instead of `$server` only. We take the Job's Name, LastRunDate, and LastRunOutcome properties.

Your resulting Excel file should look similar to this:

A	
Name	Value
ComputerNamePhysicalNetBIOS	KERRIGAN
Configuration	Microsoft.SqlServer.Management.Smo.Configuration
ConnectionContext	server='KERRIGAN';Trusted_Connection=true;Application Name='SQL
Credentials	[E23B854E-C27B-4ADC-A68A-7573C3C767FE]
CryptographicProviders	
Databases	[AdventureWorks2008R2] [master] [model] [msdb] [ReportServer] [R
DefaultFile	
DefaultLog	
DefaultTextMode	
Edition	Enterprise Evaluation Edition (64-bit)
Endpoints	[Dedicated Admin Connection] [TSQL Default TCP] [TSQL Default VIA]
EngineEdition	EnterpriseOrDeveloper
ErrorLogPath	C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSS
Events	Microsoft.SqlServer.Management.Smo.ServerEvents
FileStreamLevel	TSqlFullFileSystemAccess
FileStreamShareName	MSSQLSERVER
FullTextService	[KERRIGAN]
HadrManagerStatus	Failed
Information	Microsoft.SqlServer.Management.Smo.Information
InstallDataDirectory	C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSS
InstallSharedDirectory	C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSS
InstanceName	

There's more...

There are different ways to extract inventory information. The recipe just loops through all properties exposed with SMO and exports them to our CSV file. However, you may prefer to extract *specific properties* and eliminate ones that are not applicable to your inventory. This will entail exploring the SMO object model and working with `Get-Member` to nail down exactly which properties you want exported. With this approach, the resulting CSV is going to be more concise and relevant to your needs.

These are examples of other explicitly defined properties:

```
$server.Information.EngineEdition
$server.Information.Collation
$server.Settings.LoginMode
$server.Settings.MailProfile

$server.Configuration.AgentXPsEnabled
$server.Configuration.DatabaseMailEnabled
```

To export to CSV, you can store these properties into a hash and create a `PSObject` class from the hash. The `PSObject` class can be piped to the `Export-Csv` cmdlet:

```
#export some "server" object properties
#capture info you want to capture into a hash
#the hash will make it easier to export to CSV
$hash = @{
    "EngineEdition"      = $server.Information.EngineEdition
    "Collation"          = $server.Information.Collation
    "LoginMode"          = $server.Settings.LoginMode
    "MailProfile"        = $server.Settings.MailProfile
    "AgentXPsEnabled"    = $server.Configuration.AgentXPsEnabled
    "DatabaseMailEnabled" = $server.Configuration.DatabaseMailEnabled
}
#create a new "row" and add to the results array
$item = New-Object PSObject -Property $hash

$item |
Export-Csv -Path $fullpath -NoTypeInfo
```

See also

- ▶ The *Creating a SQL Server database inventory* recipe

Creating a SQL Server database inventory

This recipe walks you through the process of retrieving database properties and saving them to a file for inventorying purposes.

Getting ready

Log in to your SQL Server instance. Check which user databases are available for you to investigate. These same databases should appear in your resulting file after you run the PowerShell script.

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, as follows:

```
#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:

```
#specify folder and filename to be produced
$folder = "C:\Temp"
$currdate = Get-Date -Format "yyyy-MM-dd_hmmtt"
$filename = "$($instanceName)_DatabaseInventory_{$currdate}.csv"
$fullpath = Join-Path $folder $filename

$result = @()

#get properties of all databases in instance
foreach ($db in $server.Databases)
{
    $item = $null

    #capture info you want to capture into a hash
    #the hash will make it easier to export to CSV
    $hash = @{
        "DatabaseName"      = $db.Name
        "CreateDate"       = $db.CreateDate
        "Owner"             = $db.Owner
        "RecoveryModel"    = $db.RecoveryModel
        "SizeMB"            = $db.Size
        "DataSpaceUsage"   = ($db.DataSpaceUsage/1MB) .
ToString("0.00")
        "IndexSpaceUsage"  = ($db.IndexSpaceUsage/1MB) .
ToString("0.00")
        "Collation"        = $db.Collation
        "Users"             = (($db.Users | Foreach {$_.Name}) -join
",")
        "UserCount"        = $db.Users.Count
        "TableCount"       = $db.Tables.Count
        "SPCount"          = $db.StoredProcedures.Count
        "UDFCount"         = $db.UserDefinedFunctions.Count
```



```
"ViewCount"          = $db.Views.Count
"TriggerCount"       = $db.Triggers.Count
"LastBackupDate"     = $db.LastBackupDate
"LastDiffBackupDate" = $db.LastDifferentialBackupDate
"LastLogBackupDate" = $db.LastBackupDate
}
#create a new "row" and add to the results array
$item = New-Object PSObject -Property $hash
$result += $item
}

#export result to CSV
#note CSV can be opened in Excel, which is handy
$result |
Select DatabaseName, CreateDate, Owner, RecoveryModel,
SizeMB, DataSpaceUsage, IndexSpaceUsage, Collation, UserCount,
TableCount, SPCount, ViewCount, TriggerCount, LastBackupDate,
LastDiffBackupDate, LastLogBackupDate, Users |
Export-Csv -Path $fullpath -NoTypeInformation

#view folder
explorer $folder
```

How it works...

We have taken a slightly different approach with the database inventory compared to the previous server inventory.

In this recipe, we first constructed our timestamped filename.

```
#specify folder and filename to be produced
$folder = "C:\Temp"
$currdate = Get-Date -Format "yyyy-MM-dd_hmmtt"
$filename = "$($instanceName)_DatabaseInventory_{$currdate}.csv"
$fullpath = Join-Path $folder $filename
```

We then created an empty array where we can store our data:

```
$result = @()
```

In the next step, we created a hash of values that we then stored back to our `$result` array. The hash helps us create a nice tabular result that we can easily export into our CSV file.

```
foreach ($db in $server.Databases)
{
    $item = $null

    #capture info you want to capture into a hash
    #the hash will make it easier to export to CSV
    $hash = @{
        "DatabaseName"      = $db.Name
        "CreateDate"       = $db.CreateDate
        #other properties
        "Users"             = (($db.Users | Foreach {$_.Name}) -join ",")
        "LastLogBackupDate" = $db.LastBackupDate
    }
    #create a new "row" and add to the results array
    $item = New-Object PSObject -Property $hash
    $result += $item
}
```

We have explicitly identified the properties we want to record. Once done, your result should look similar to this:

A	B	C	D	E	F	G	H	I	J	K	L
DatabaseName	CreateDate	Owner	RecoveryM	SizeMB	DataSp	IndexSp	Collation	UserCount	TableCount	SPCount	ViewCount
AdventureWorks	11/27/2011 11:14	KERRIGAN	Simple	266.9375	0.11	0.08	Latin1_General	5	74	1373	398
master	4/8/2003 9:13	sa	Simple	9.5625	0	0	SQL_Latin1_Ger	7	6	1364	396
model	4/8/2003 9:13	sa	Full	3.8125	0	0	SQL_Latin1_Ger	4	0	1362	395
msdb	6/24/2011 18:48	sa	Simple	14.4375	0.01	0	SQL_Latin1_Ger	8	137	1775	470
ReportServer	1/8/2012 14:02	KERRIGAN	Full	5.125	0	0	Latin1_General	5	34	1598	400
ReportServerTempdb	1/8/2012 14:02	KERRIGAN	Simple	5.125	0	0	Latin1_General	5	13	1363	395
temp2	1/22/2012 21:14	KERRIGAN	Full	3.828125	0	0	SQL_Latin1_Ger	4	0	1362	395
tempdb	1/16/2012 7:06	sa	Simple	9	0	0	SQL_Latin1_Ger	4	0	1362	395
tempdb	1/9/2012 11:43	QUERYWOR	Full	13	0	0	SQL_Latin1_Ger	0	0	1362	395

A database has many properties that you may, or may not, want to capture in an inventory file. We have picked a few properties here, but your situation and needs may be different, so adjust this script as necessary.

See also

- ▶ [The Creating a SQL Server instance inventory recipe](#)

Listing installed hotfixes and service packs

In this recipe, we will check which service pack and hotfixes/patches are installed on our server.

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 as follows:

```
#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. To list the version of SQL Server and Service Pack level, add the following script and run:

```
#to get the version
#major.minor.build.buildminor
#this should tell you collectively at what
#level your install is
$server.Information.VersionString

#from MSDN
#version of SQL Server
#RTM = Original release version
#SPn = Service pack version
#CTP, = Community Technology Preview version
$server.Information.ProductLevel

#to get hotfixes/updates/patches, we can use
#the Get-Hotfix cmdlet
#Get-Hotfix wraps the WMI class Win32_QuickFixEngineering
#but this may miss some updates or properties,
#depending on your OS
#this also does not include updates that are supplied by
#Microsoft Windows Installer (MSI)
```

```
#get all hotfixes
#note the Get-Hotfix cmdlet does not list updates
#applied by MSI (Microsoft Installer)
Get-Hotfix

#check if a specific hotfix is installed
Get-Hotfix -Id "KB2620704"
```

How it works...

The script for this task can be divided into two separate parts. The first part is a SQL Server script that specifically allows us to check which version and service pack has been installed in our instance.

The bit that gives us the service pack level is straightforward:

```
#version of SQL Server
#RTM = Original release version
#SPn = Service pack version
#CTP, = Community Technology Preview version
$server.Information.ProductLevel
```

The block that gives us the version string provides a bit more information than you might guess:

```
#to get the version
#major.minor.build.buildminor
#this should tell you collectively at what
#level your install is
$server.Information.VersionString
```

You may get a version such as **10.50.2796.0**, which is SQL Server 2008 R2 (major and minor version 10.50) with Service Pack 1 and Cumulative Update 4 (build number 2796.0). When you install a hotfix or service pack, it should tell you what build your instance is going to be:

Article ID: 2633146 - Last Review: January 9, 2012 - Revision: 3.0

Cumulative update package 4 for SQL Server 2008 R2 Service Pack 1



Hotfix Download Available
[View and request hotfix downloads](#)

This article describes cumulative update package 4 for Microsoft SQL Server 2008 R2 Service Pack 1 (SP1). This update contains hotfixes for issues that were fixed after the release of SQL Server 2008 R2 SP1.

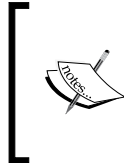
Note The build of this cumulative update package is known as build 10.50.2796.0. 

The second part of the script is not SQL Server-specific. PowerShell has a cmdlet called `Get-Hotfix`, which can query either the local or a remote machine for installed hotfixes. Simply calling `Get-Hotfix` will list all installed hotfixes, or you can also pass a specific hotfix number (or KB Number) and it will query that specific item for you:

```
#check if a specific hotfix is installed
Get-Hotfix -Id "KB2620704"
```

Be aware that there is a documented limitation of `Get-Hotfix`. It is documented in MSDN (<http://msdn.microsoft.com/en-us/library/dd315358.aspx>) as follows:

This cmdlet uses the Win32_QuickFixEngineering WMI class, which represents small system-wide updates of the operating system. Starting with Windows Vista, this class returns only the updates supplied by Component Based Servicing (CBS). It does not include updates that are supplied by Microsoft Windows Installer (MSI) or the Windows update site.



To get a complete picture of all updates, see Laerte Junior's Simple-Talk article *List updates, hotfixes, and Service Packs with Simple Commands* (<http://www.simple-talk.com/blogs/2011/09/08/list-updates-hotfixes-and-service-packs-with-simple-commands/>).

There's more...

Some of the terms used in this recipe may be familiar to you, but only vaguely. In case they are, let's define some of these terms. After all, you may hear them again and again in your dealings with your network admin, system admin, or your DBA.

Terminology	Description	Cycle
RTM	<ul style="list-style-type: none">▶ Release to Manufacturing▶ Version of the product that is released to the market	N/A

Terminology	Description	Cycle
Hotfix	<ul style="list-style-type: none"> ▶ Also referred to as Quick Fix Engineering (QFE) ▶ Designed to address single or isolated issues; usually on a per-client basis ▶ Has to be specifically requested from Microsoft, either through a support call or from their site (https://support.microsoft.com/contactus/emailcontact.aspx?scid=sw;%5BLN%5D;1422&WS=hotfix) ▶ Distributed by Microsoft Customer Service and Support (CSS) and cannot be redistributed by clients 	N/A
Cumulative Update (CU)	<p>A package that contains a bundle of hotfixes that have passed an <i>acceptance criteria</i></p> <p>Full regression test still not performed, and should not be applied by all customers</p>	Every 2 months
Service pack	<p>According to Microsoft's official terminology guide, it is defined as follows:</p> <p style="text-align: center;"><i>a tested, cumulative set of all hotfixes, security updates, critical updates, and update</i></p>	Every 12 to 18 months

See also

- ▶ Check out Microsoft's best practice guide on applying hotfixes and service packs here:
<http://technet.microsoft.com/en-us/library/cc750077.aspx#XSLTsection127121120120>
- ▶ Additional terminologies are explained here:
<http://support.microsoft.com/kb/824684>
- ▶ There is also an unofficial guide to the SQL Server builds, which is quite comprehensive. You can check it out at:
<http://sqlserverbuilds.blogspot.com/>

Listing running/blocking processes

This recipe lists processes in your SQL Server instance and their status.

Getting ready

In order to see blocking processes in your list, we will have to force some blocking queries.

Open SQL Server Management Studio. Connect to the instance you want to test. We will assume you have AdventureWorks2008R2. If not, you can use a different database and table altogether.

Open two new query windows for that connection. Type and run the following in the two query windows:

```
USE AdventureWorks2008R2
GO

BEGIN TRAN
SELECT *
FROM dbo.ErrorLog
WITH (TABLOCKX)
```

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, as follows:

```
#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. Run the following script to see all processes:

```
#List all processes
$server.EnumProcesses() |
Select Name, Spid, Command, Status, Login, Database, BlockingSpid
|
Format-Table -AutoSize
```

You should see something similar to this:

Name	Spid	Command	Status	Login	Database	BlockingSpid
1	1	LOG WRITER	background	sa	master	0
2	2	RECOVERY WRITER	background	sa	master	0
3	3	RESOURCE MONITOR	background	sa	master	0
4	4	XE TIMER	background	sa	master	0
5	5	XE DISPATCHER	background	sa	master	0
6	6	LAZY WRITER	background	sa	master	0
7	7	LOCK MONITOR	background	sa	master	0
8	8	SIGNAL HANDLER	background	sa	master	0
9	9	TASK MANAGER	sleeping	sa	master	0
10	10	TASK MANAGER	sleeping	sa	master	0
11	11	BRKR EVENT HNDLR	background	sa	master	0
12	12	TASK MANAGER	sleeping	sa	master	0
13	13	UNKNOWN TOKEN	background	sa	master	0
14	14	BRKR TASK	background	sa	master	0
15	15	BRKR TASK	background	sa	master	0
16	16	TRACE QUEUE TASK	background	sa	master	0
17	17	RECEIVE	background	sa	master	0

4. To list blocking processes, run the following code:

```
#List blocking Processes
#This assumes you already ran the SQL Script in the
#prep section to create the blocking processes
#Otherwise you may not see any results
#Note this is a V3 syntax because of the simplified
#Where syntax
$server.EnumProcesses() |
Where-Object BlockingSpid -ne 0 |
Select Name, Spid, Command, Status, Login, Database, BlockingSpid
|
Format-Table -AutoSize
```

Your result should show the blocking process you produced in the prep section:

Name	Spid	Command	Status	Login	Database	BlockingSpid
63	63	SELECT	suspended	KERRIGAN\Administrator	AdventureWorks2008R2	62

How it works...

The SMO server object has a method named `EnumProcesses` that simplifies the listing of running processes in an instance. Once the SMO server object is instantiated, all you need to invoke is the `EnumProcesses` method:

```
$server.EnumProcesses() |
Select Name, Spid, Command, Status, Login, Database, BlockingSpid |
Format-Table -AutoSize
```


If you wish to display processes that are blocked, this command can be filtered to show processes where the `BlockingSpid` is not zero, that is, blocked:

```
Where-Object BlockingSpid -ne 0 |
```

Note that this is a PowerShell V3 syntax because of the simplified use of the `Where-Object` cmdlet. To use this in PowerShell V2, simply modify this line to use the curly braces `{}` and `$_` special variable:

```
Where-Object {$_ .BlockingSpid -ne 0} |
```

There are a number of overloads for the `EnumProcesses` method. Without any parameter, it returns all processes. Other overloads allow you to:

- ▶ List processes excluding system processes
- ▶ List information for a specific process ID
- ▶ List processes for a specific login

The result returned by `EnumProcesses` is similar to the information you get from the system stored procedure `sp_who2`. The information includes the following:

- ▶ Name
- ▶ Login
- ▶ Host
- ▶ Status
- ▶ Command
- ▶ Database
- ▶ Blocking SPID

See also

- ▶ *The Killing a blocking process recipe*
- ▶ Learn more about the `EnumProcesses` method here:
[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.server.enumprocesses\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.server.enumprocesses(v=sql.110).aspx)

Killing a blocking process

This recipe illustrates how you can kill a blocking process in SQL Server.

Getting ready

In order to see blocking processes in your list, we will have to force some blocking queries. If you have already done the prep work in the *List running/blocking processes* recipe, you do not need to do this prep section. If you haven't, go ahead and perform this section:

Open SQL Server Management Studio. Connect to the instance you want to test. We will assume you have AdventureWorks2008R2. If not, you can use a different database and table altogether.

Open two new query windows for that connection. Type and run the following in the two query windows:

```
USE AdventureWorks2008R2
GO
```

```
BEGIN TRAN
SELECT *
FROM dbo.ErrorLog
WITH (TABLOCKX)
```

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 as follows:

```
#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:

```
$VerbosePreference = "Continue"

#This assumes you already ran the SQL Script in the
#prep section to create the blocking processes
```

```
#Otherwise you may not see any results
$server.EnumProcesses() |
Where-Object BlockingSpid -ne 0 |
ForEach-Object {
    Write-Verbose "Killing SPID $($_.BlockingSpid)"
    $server.KillProcess($_.BlockingSpid)
}

$VerbosePreference = "SilentlyContinue"
```

How it works...

To kill a blocking process in PowerShell using SMO simply requires the invocation of the `KillProcess` method of the SMO Server class:

```
$server.KillProcess($_.BlockingSpid)
```

However, this entails knowing which Process ID needs to be killed. In this recipe, we've also identified—via scripting—which processes are blocking, and then killed them. Thus, we need to identify all blocking processes:

```
$server.EnumProcesses() |
Where-Object BlockingSpid -ne 0 |
ForEach-Object {
    Write-Verbose "Killing SPID $($_.BlockingSpid)"
    $server.KillProcess($_.BlockingSpid)
}
```

Once we've identified all blocking processes, we can kill the processes. In our recipe we also display which process ID we are killing:

```
$server.EnumProcesses() |
Where-Object BlockingSpid -ne 0 |
ForEach-Object {
    Write-Verbose "Killing SPID $($_.BlockingSpid)"
    $server.KillProcess($_.BlockingSpid)
}
```

There's more...

We have all run into a situation where SQL Server is running a process that is out of control. Perhaps it is a query missing a join or a process that is taking up too much memory. Using scripting can reduce manual errors of accidentally killing a process that wasn't blocking, and help with automating this task.



Killing a process is a drastic measure. Use this script with caution.

See also

- ▶ The *Listing running/blocking processes* recipe

Checking disk space usage

This recipe shows how to list disks available for your SQL Server instance, how much is used, and how much is available.

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:

```
#get server list
$servers = @"KERRIGAN"

#this can come from a file instead of hardcoding
#the servers
#servers = Get-Content <filename>

Get-WmiObject -ComputerName $servers -Class Win32_Volume |
Select @{N="Name";E={$_.Name}},
       @{N="DriveLetter";E={$_.DriveLetter}},
       @{N="DeviceType";
        E={switch ($_.DriveType)
            {
                0 {"Unknown"}
                1 {"No Root Directory"}
                2 {"Removable Disk"}
                3 {"Local Disk"}
                4 {"Network Drive"}
                5 {"Compact Disk"}
                6 {"RAM"}
            }
        }};
},
```

```

@{N="Size (GB) ";E="{0:N1}" -f($_.Capacity/1GB)}},
@{N="FreeSpace (GB) ";E="{0:N1}" -f($_.FreeSpace/1GB)}},
@{N="FreeSpacePercent";E={
    if ($_.Capacity -gt 0)
    {
        "{0:P0}" -f($_.FreeSpace/$_.Capacity)
    }
    else
    {
        0
    }
}
} |
Format-Table -AutoSize

```

The result should look similar to the following screenshot:

Name	DriveLetter	DeviceType	SizeGB	FreeSpaceGB	FreeSpacePercent
\\?\Volume{fa3bc650-0326-11e1-ab55-806e6f6e6963}\		Local Disk	0.10	0.07	71.87
C:\	C:	Local Disk	79.90	33.55	41.98
D:\	D:	Compact Disk	0.00	0.00	0

How it works...

An essential task for a database administrator is to know how much disk the database server is consuming. An automated script can help the administrator create an accurate profile of the database server storage, and allows for scaling the system too.

For this recipe, we enlist the help of the **Windows Management Instrumentation (WMI)** Win32_Volume class.

```
Get-WmiObject -ComputerName $servers -Class Win32_Volume
```



WMI is further discussed in the *Listing SQL Server instances* recipe in *Chapter 2, SQL Server and PowerShell Basic Tasks*.

Using WMI, we can list all the drives recognized on the target machine, including removable drives, local hard drives, network disks, compact disks, and RAM disks.

The `Win32_Volume` WMI class, according to MSDN ([http://msdn.microsoft.com/en-us/library/windows/desktop/aa394515\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa394515(v=vs.85).aspx)), represents an area of storage on a hard disk. The class returns local volumes that are either formatted, unformatted, mounted, or offline.

We use `Win32_Volume` instead of `Win32_LogicalDisk` because:

- ▶ `Win32_Volume` does not manage floppy disk drives, and `Win32_LogicalDisk` does. Since we're dealing with databases, we do not need the floppy disk drives. Databases will not be stored in floppy disks.
- ▶ `Win32_Volume` enumerates all volumes, even those that do not have drive letters. This is useful for databases that are stored in volume mount points.

For purposes of this recipe, we list all disks. In reality, you will most likely always filter the results to show just the local and networked hard drives. In the script, once we capture the disks using the `Win32_Volume` class, we pipe the information to a `Select` or `Select-Object` cmdlet, where we format our output. Note that formatting the output in the `Select` cmdlet will require that we specify the hash, the `Name`, and the `Expression`:

```
Select @{Name="Name";Expression={$_.Name}},
```

We can also shorten this by using `N` for `Name` and `E` for `Expression`:

```
Select @{N="Name";E={$_.Name}},
```

Expressions can also accept some format specifiers, and we have used `{0:N1}` for single decimal numeric values and `{0:P0}` for 0 decimal percent.

In the recipe we display each disk name, drive letter, device type, drive type, size in GB, free space in GB, and percent free space.

```
Get-WmiObject -ComputerName $servers -Class Win32_Volume |
Select @{N="Name";E={$_.Name}},
       @{N="DriveLetter";E={$_.DriveLetter}},
       @{N="DeviceType";
        E={switch ($_.DriveType)
          {
            0 {"Unknown"}
            1 {"No Root Directory"}
            2 {"Removable Disk"}
            3 {"Local Disk"}
            4 {"Network Drive"}
            5 {"Compact Disk"}
            6 {"RAM"}
          }
        }};
},
```

```
@{N="Size (GB)";E="{0:N1}" -f($_.Capacity/1GB)},
@{N="FreeSpace (GB)";E="{0:N1}" -f($_.FreeSpace/1GB)},
@{N="FreeSpacePercent";E={
    if ($_.Capacity -gt 0)
    {
        "{0:P0}" -f($_.FreeSpace/$_.Capacity)
    }
    else
    {
        0
    }
}
} |
```

```
Format-Table -AutoSize
```

See also

- ▶ Learn more about the `Win32_Volume` class from here:
[http://msdn.microsoft.com/en-us/library/windows/desktop/aa394515\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa394515(v=vs.85).aspx)
- ▶ Learn more about `Win32_LogicalDisk` from here:
[http://msdn.microsoft.com/en-us/library/windows/desktop/aa394173\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa394173(v=vs.85).aspx)
- ▶ You can also check out the standard .NET format specifiers here:
[http://msdn.microsoft.com/en-us/library/dwhawy9k\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/dwhawy9k(v=vs.110).aspx)

Setting up WMI Server event alerts

In this recipe, we will set up a simple WMI Server event alert for a DDL event.

Getting ready

We will set up an alert that creates a timestamped text file every time there is a DDL Login event (CREATE, ALTER, or DROP). We will utilize the WMI provider for server events in this exercise.

These are the values you will need to know:

Item	Value
Namespace (if using the default instance)	root\Microsoft\SqlServer\ServerEvents\MSSQLServer
Namespace (if using a named instance)	root\Microsoft\SqlServer\ServerEvents\SQL01
WMI query	SELECT * FROM DDL_LOGIN_EVENTS
DDL_LOGIN_EVENTS properties (partial list)	SQLInstance LoginName PostTime SPID ComputerName LoginType

For WMI events hitting SQL Server, you will also need to ensure that SQL Server Broker is running on your target database. In our case, we need to ensure that the Broker is running on the `msdb` database.

```
SELECT
    is_broker_enabled, *
FROM
    sys.databases
ORDER BY
    name
```

Check the `msdb` database's `is_broker_enabled` field in the result.

	is_broker_enabled	name	database_id
1	0	AdventureWorks2008R2	6
2	0	master	1
3	0	model	3
4	1	msdb	4
5	1	ReportServer	7
6	1	ReportServerTempDB	8
7	1	tempdb	2
8	0	TestDB	5
9	1	TestDB_copy	9

If service broker is not running on msdb, run the following T-SQL statement from SQL Server Management Studio:

```
ALTER DATABASE msdb
SET ENABLE_BROKER
```

Alternatively, you can do this using PowerShell:

```
$database.BrokerEnabled = $true
$database.Alter()
```

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:

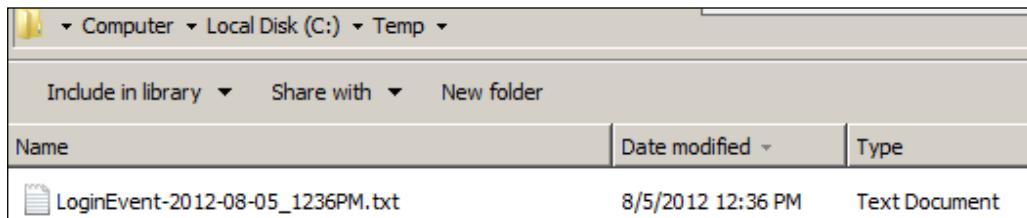
```
$namespace = "root\Microsoft\SqlServer\ServerEvents\MSSQLSERVER"

#WQL for Login Events
#note we will capture CREATE, DROP, ALTER
#if you want to more specific, use these events
#DROP_LOGIN, CREATE_LOGIN, ALTER_LOGIN
$query = "SELECT * FROM DDL_LOGIN_EVENTS"

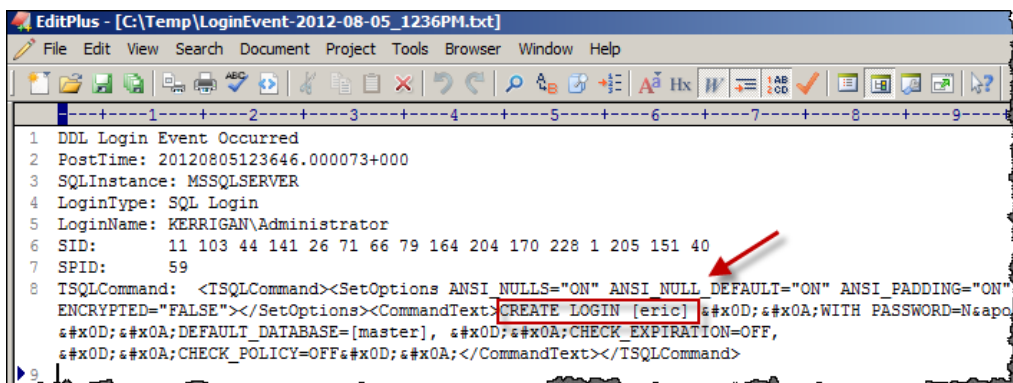
#register the event
#if the event is triggered, it will respond by
#creating a timestamped file containing event
#details
Register-WMIEvent `
-Namespace $namespace `
-Query $query -SourceIdentifier "SQLLoginEvent" `
-Action {
    $date = Get-Date -Format "yyyy-MM-dd_hmmst"
    $filename = "C:\Temp\LoginEvent-$(($date)).txt"
    New-Item -ItemType file $filename
}
$msg = @"
DDL Login Event Occurred`n
PostTime: $($event.SourceEventArgs.NewEvent.PostTime)
SQLInstance: $($event.SourceEventArgs.NewEvent.SQLInstance)
LoginType: $($event.SourceEventArgs.NewEvent.LoginType)
LoginName: $($event.SourceEventArgs.NewEvent.LoginName)
SID:      $($event.SourceEventArgs.NewEvent.SID)
SPID:     $($event.SourceEventArgs.NewEvent.SPID)
TSQLCommand: $($event.SourceEventArgs.NewEvent.TSQLCommand)
"@
$msg | Out-File -FilePath $filename -Append
}
```

3. Test fire a DDL event and check if the file gets created in response to the event:
 1. Open SQL Server Management Studio.
 2. In a new query window, execute the following code. This will trigger the DDL Login WMI event:


```
USE [master]
GO
CREATE LOGIN [eric]
WITH PASSWORD=N'P@ssword',
DEFAULT_DATABASE=[master],
CHECK_EXPIRATION=OFF,
CHECK_POLICY=OFF
GO
```
 3. Go to your **Temp** folder and check if there is a file created for the **LoginEvent**:



4. Open the LoginEvent file to see the entries. Note that the T-SQL statement we used to create the new login has been captured in this file.



Note that this is a fairly generic log. If you want to narrow it down to exactly which login event has occurred, you can attach this to more specific events, such as DROP_LOGIN, CREATE_LOGIN, and ALTER_LOGIN.

How it works...

We are utilizing **Windows Management Instrumentation (WMI)** and **WMI Query Language (WQL)** in this recipe. However, before we can put this into place, Service Broker has to be enabled in your instance, as specified in the *Getting Ready* section. The Service Broker is what the WMI provider uses to send the SQL Server instance events.



WMI is further discussed in the *Listing SQL Server Instances* recipe in *Chapter 2, SQL Server and PowerShell Basic Tasks*.

The first thing to identify is which namespace to use. For our purposes, because we want to capture the SQL Server events from the default instance, our namespace will be:

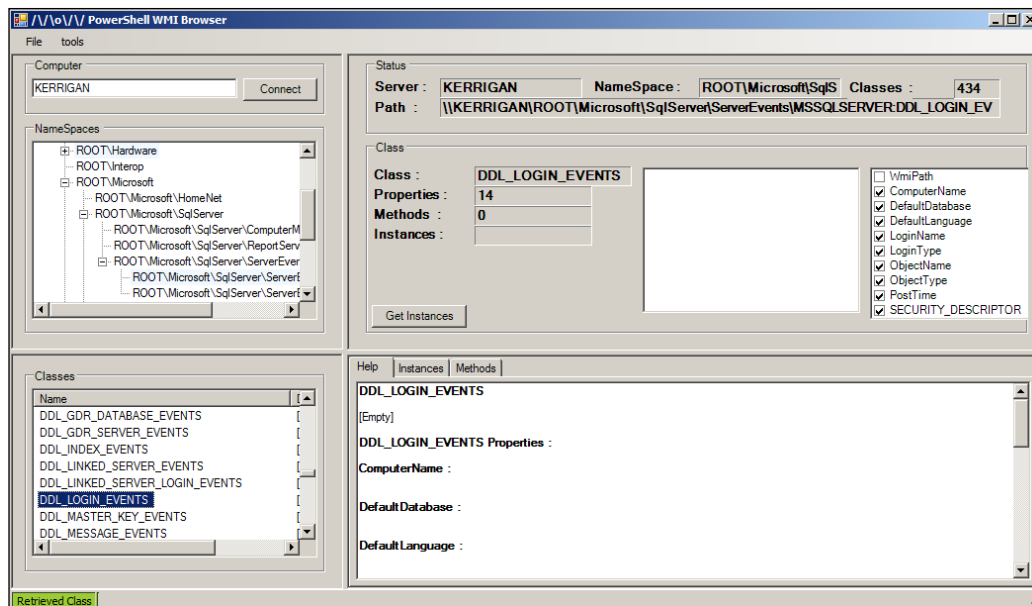
```
$namespace = "root\Microsoft\SqlServer\ServerEvents\MSSQLSERVER"
```

If you have a named instance, you simply have to replace MSSQLSERVER with the instance name.

The next step is to identify which WQL query we need to capture the events on which we want to be alerted. In our case, it is just DDL_LOGIN_EVENTS. The other available events that you can query are listed in MSDN's *WMI Provider for Server Events Classes and Properties* article.

```
#WQL for Login Events
#note we will capture CREATE, DROP, ALTER
#if you want to more specific, use these events
#ROP_LOGIN, CREATE_LOGIN, ALTER_LOGIN
$query = "SELECT * FROM DDL_LOGIN_EVENTS"
```

Another way to explore the SQL Server WMI events is to use a tool similar to Marc van Orsouw's (also known as The PowerShell Guy) PowerShell WMI Explorer (<http://thepowershellguy.com/blogs/posh/archive/2007/03/22/powershell-wmi-explorer-part-1.aspx>):



Marc has provided instructions on his blog on how to use this tool, which is pretty straightforward. Once you navigate to the **ROOT\Microsoft\SqlServer\ServerEvents\MSSQLSERVER** namespace and the **DDL_LOGIN_EVENTS** class, the supported properties and methods will be displayed on the right-hand pane.

After you finalize the namespace and WQL query, you need to register this as a WMI event. When registering this event, we will specify an action section to create a timestamped log file each time the event is triggered. This log file will contain event properties such as `PostTime`, `LoginType`, `LoginName`, `SID`, `SPID`, and the T-SQL command that caused the event trigger to fire.

```
Register-WMIEvent `
-Namespace $namespace `
-Query $query -SourceIdentifier "SQLLoginEvent" `
-Action {
    $date = Get-Date -Format "yyyy-MM-dd_hmmst"
    $filename = "C:\Temp>LoginEvent-$( $date ).txt"
    New-Item -ItemType file $filename

    $msg = @"
DDL Login Event Occurred`n
PostTime: $($event.SourceEventArgs.NewEvent.PostTime)
SQLInstance: $($event.SourceEventArgs.NewEvent.SQLInstance)
LoginType: $($event.SourceEventArgs.NewEvent.LoginType)
LoginName: $($event.SourceEventArgs.NewEvent.LoginName)
```

```
SID:          $($event.SourceEventArgs.NewEvent.SID)
SPID:         $($event.SourceEventArgs.NewEvent.SPID)
TSQLCommand: $($event.SourceEventArgs.NewEvent.TSQLCommand)
"@
$msg | Out-File -FilePath $filename -Append
}
```

The `Register-WmiEvent` cmdlet translates the query into SQL Server event notifications, which are handled by the Service Broker.

To unregister the event, use the `Unregister-Event` cmdlet:

```
Unregister-Event "SQLLoginEvent"
```

One caveat about the `Register-WmiEvent` cmdlet is that it's a temporarily registered event. This means that it will go away if the program hosting it stops or the server gets restarted.

There's more...

The *WMI Provider for Server Events Classes and Properties* article can be found here:

[http://msdn.microsoft.com/en-us/library/ms186449\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms186449(v=sql.110).aspx)

To learn more about DDL event groups, check out MSDN:

[http://msdn.microsoft.com/en-us/library/bb510452\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/bb510452(v=sql.110).aspx)

Also check out the MSDN article on *Understanding the WMI Provider for Server Events*:

[http://msdn.microsoft.com/en-us/library/ms181893\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms181893(v=sql.110).aspx)

WMI Query Language (WQL) will become more and more important as you work with more WMI events. There is an excellent free e-book provided by one of the prominent bloggers in the PowerShell community, Ravikanth Chaganti. You can download his WQL e-book from:

<http://www.ravichaganti.com/blog/?p=1979>

One tool that can help you explore the WMI properties and events is Marc van Orsouw's PowerShell WMI Explorer:

<http://thepowershellguy.com/blogs/posh/archive/2007/03/22/powershell-wmi-explorer-part-1.aspx>

Detaching a database

In this recipe we will detach a database programmatically.

Getting ready

For purposes of this recipe, let's create a database called `TestDB`. Open up SQL Server Management Studio and run the following code:

```
CREATE DATABASE [TestDB]
  CONTAINMENT = NONE
  ON PRIMARY
  ( NAME = N'TestDB', FILENAME = N'C:\Program Files\Microsoft SQL
  Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\TestDB.mdf' , SIZE = 4096KB ,
  FILEGROWTH = 1024KB ) ,
  FILEGROUP [FG1]
  ( NAME = N'data1', FILENAME = N'C:\Program Files\Microsoft SQL Server\
  MSSQL11.MSSQLSERVER\MSSQL\DATA\data1.ndf' , SIZE = 4096KB , FILEGROWTH
  = 1024KB ) ,
  FILEGROUP [FG2]
  ( NAME = N'data2', FILENAME = N'C:\Program Files\Microsoft SQL Server\
  MSSQL11.MSSQLSERVER\MSSQL\DATA\data2.ndf' , SIZE = 4096KB , FILEGROWTH
  = 1024KB )
  LOG ON
  ( NAME = N'TestDB_log', FILENAME = N'C:\Program Files\Microsoft SQL
  Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\TestDB_log.ldf' , SIZE = 1024KB
  , FILEGROWTH = 10%)
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, as follows:

```
#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:

```
$databasename = "TestDB"

#parameters accepted are databasename, boolean
#flag for updatestatistics, and boolean flag
#for removeFulltextIndexFile
$server.DetachDatabase($databasename, $false, $false)
```

How it works...

Detaching a database programmatically is fairly straightforward. The `DetachDatabase` method of the `$server` object accepts three parameters: database name and the `updateStatistics` and `removeFulltextIndexFile` Boolean flags.

```
$server.DetachDatabase($databasename, $false, $false)
```

There is another overload of the `DetachDatabase` method that accepts only two parameters: database name and the `updateStatistics` flag.

Also note that there are settings that may prevent you from detaching your databases, such as:

- ▶ Insufficient privileges on the instance
- ▶ Database is being replicated
- ▶ Database has a snapshot

You can read the full documentation from MSDN:

<http://msdn.microsoft.com/en-us/library/ms190794.aspx>

There's more...

Capturing the `mdf`, `ndf`, and `ldf` information can be useful, especially if you plan to detach the database and re-attach it right away to a different instance.

One way to get this information is by using the `mdf` file to extract all the other data and log files that the detached database uses. You can supply the full `mdf` file path to two methods to get all the information about the data and log files:

```
$server.EnumDetachedDatabaseFiles($mdfname)
$server.EnumDetachedLogFiles($mdfname)
```

From the script, you can easily pass this information to your `Attach Database` script or code block.

See also

- ▶ The *Attaching a database* recipe

Attaching a database

In this recipe, we will programmatically attach a database with a primary data file (.mdf), log file (.ldf), and multiple secondary data files (.ndf).

Getting ready

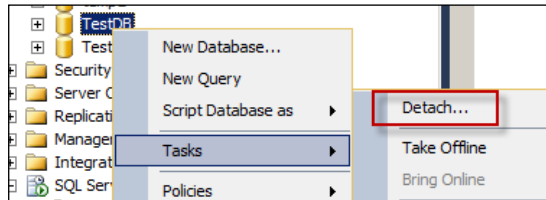
Before we can attach a database, we must have the data files and optional log files attached. If you have not completed the *Detaching a database* recipe, perform the following steps:

1. When we attach the database, we will set `QUERYWORKS\jraynor` as the owner. This principal has been created with our development VM. Feel free to replace the appropriate code with a login available with your system.
2. We will create a database called `TestDB`. Open up SQL Server Management Studio and run the following code:

```
IF DB_ID('TestDB') IS NOT NULL
DROP DATABASE TestDB
GO

CREATE DATABASE [TestDB]
CONTAINMENT = NONE
ON PRIMARY
( NAME = N'TestDB', FILENAME = N'C:\Program Files\Microsoft SQL
Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\TestDB.mdf' , SIZE = 4096KB
, FILEGROWTH = 1024KB ),
FILEGROUP [FG1]
( NAME = N'data1', FILENAME = N'C:\Program Files\Microsoft SQL
Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\data1.ndf' , SIZE = 4096KB ,
FILEGROWTH = 1024KB ),
FILEGROUP [FG2]
( NAME = N'data2', FILENAME = N'C:\Program Files\Microsoft SQL
Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\data2.ndf' , SIZE = 4096KB ,
FILEGROWTH = 1024KB )
LOG ON
( NAME = N'TestDB_log', FILENAME = N'C:\Program Files\Microsoft
SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\TestDB_log.ldf' , SIZE =
1024KB , FILEGROWTH = 10%)
GO
```


3. Once you have this, go to your SSMS Object Explorer, right-click on **TestDB** and go to **Tasks | Detach**:



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, as follows:

```
#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:

```
$databasename = "TestDB"

#identify the primary data file
#this typically has the .mdf extension
$mdfname = "C:\Program Files\Microsoft SQL Server\MSSQL11.
MSSQLSERVER\MSSQL\DATA\TestDB.mdf"

#FYI only
#view detached database info
$server.DetachedDatabaseInfo($mdfname) | Format-Table

#attachdatabase accepts a StringCollection, so we need
#to add our files in this collection
$filecoll = New-Object System.Collections.Specialized.
StringCollection

#add all data files
#this includes the primary data file
$server.EnumDetachedDatabaseFiles($mdfname) |
Foreach-Object {
```

```

    $filecoll.Add($_)
}

#add all log files

$server.EnumDetachedLogFiles($mdfname) |
ForEach-Object {
    $filecoll.Add($_)
}

$owner = "QUERYWORKS\jraynor"

<#
http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.
management.smo.attachoptions.aspx
None      There are no attach options. Value = 0.
EnableBroker Enables Service Broker . Value = 1.
NewBroker  Creates a new Service Broker . Value = 2.
ErrorBrokerConversations Stops all current active Service Broker
conversations at the save point and issues
an error message. Value = 3.
RebuildLog Rebuilds the log. Value = 4.
#>

$server.AttachDatabase($databasename, $filecoll, $owner,
[Microsoft.SqlServer.Management.Smo.AttachOptions]::None)

```

How it works...

Attaching a database requires a little bit more work compared to detaching a database. With detaching a database, all you really need to know and supply is the instance details and the database name.

With attaching a database, you will also need to supply, at minimum, all the files (primary data, secondary data, and log) that the database used to use. You can attach a database without supplying log files. SQL Server will recreate new log files for you. While log files are technically "optional", it is best if you have preserved the log files in case this will be needed later on for any point-in-time restore (applicable only to Bulk Logged and Full Recovery Model).



Backup and Restore are covered in *Chapter 5, Advanced Administration*. Recovery models Simple, Bulk Logged and Full are discussed in this chapter.

Before we can attach the database, we need to identify the primary data file.

```
#identify the primary data file
#this typically has the .mdf extension
$mdfname = "C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\
MSSQL\DATA\TestDB.mdf"
```

Note that primary data files *do not* have to have the .mdf extension, although it is very typical to preserve this extension.

We also need to create a `StringCollection` object that we will pass as parameter to the `AttachDatabase` method of the SMO server object:

```
#attachdatabase accepts a StringCollection, so we need
#to add our files in this collection
$filecoll = New-Object System.Collections.Specialized.StringCollection
```

Once we have our primary data file path and our `StringCollection` object, we can start adding all the files listed in the mdf header into our collection:

```
#add all data files
$server.EnumDetachedDatabaseFiles($mdfname) |
Foreach-Object {
    $filecoll.Add($_)
}
```

If you need to change the location of the files, you will need to replace the path before you add the filename to the collection. For example:

```
$newpath = "C:\Temp"
$server.EnumDetachedDatabaseFiles($mdfname) |
Foreach-Object {
    $newfile = Join-Path $newpath (Split-Path $_ -Leaf)
    $filecoll.Add($newfile)
}
```

Ideally, you will also add all the logfile information:

```
$server.EnumDetachedLogFiles($mdfname) |
ForEach-Object {
    $filecoll.Add($_)
}
```

You can also reset a few additional properties, including database owner:

```
$owner = "QueryWorks\jraynor"
```

When ready, you can invoke the `AttachDatabase` method:

```
$server.AttachDatabase($databasename, $filecoll, $owner, [Microsoft.
SqlServer.Management.Smo.AttachOptions]::None)
```

There are 5 attach options: `None`, `EnableBroker`, `NewBrooker`, `ErrorBrokerConversations`, and `RebuildLog`. If you do not have the logfiles handy, make sure to choose `RebuildLog`.

See also

- ▶ The *Detaching a database* recipe
- ▶ Read more about the `AttachDatabase` options here:

```
http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.
management.smo.attachoptions\(v=sql.110\).aspx
```

Copying a database

In this recipe, we will look at how to copy a database using PowerShell and SMO.

Getting ready

In this recipe, we will assume you have the `TestDB` database already created from previous recipes. If you do not have it, you can also substitute this with any database you already have in your instance.

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, as follows:

```
#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:

```
$databasename = "TestDB"
$sourcedatabase = $server.Databases[$databasename]
```

```
#Create a database to hold the copy of your database
$dbnamecopy = "$($databasename)_copy"

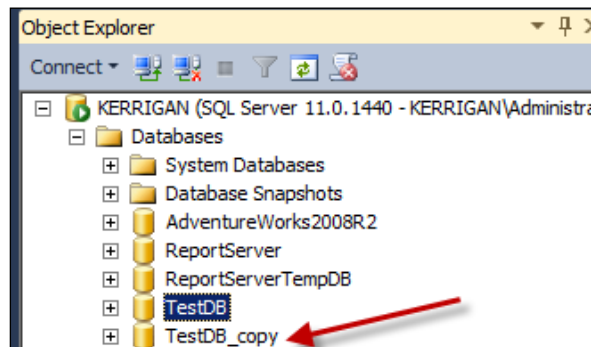
$dbcopy = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Database -ArgumentList $server, $dbnamecopy
$dbcopy.Create()

#need to specify source database
#Use SMO Transfer Class
$transfer = New-Object -TypeName Microsoft.SqlServer.Management.
SMO.Transfer -ArgumentList $sourcedatabase
$transfer.CopyAllTables = $true
$transfer.Options.WithDependencies = $true
$transfer.Options.ContinueScriptingOnError = $true
$transfer.DestinationDatabase = $dbnamecopy
$transfer.DestinationServer = $server.Name
$transfer.DestinationLoginSecure = $true
$transfer.CopySchema = $true

#if you want to only produce a script that will
#"copy" your database, use the ScriptTransfer method
$transfer.ScriptTransfer()

#if you want to perform the actual transfer
#you should use the TransferData method
$transfer.TransferData()
```

4. Check that the database has been created. Go to SQL Server Management Studio and inspect the user databases in **Object Explorer**. You may need to refresh **Object Explorer**:



How it works...

Copying a database using SMO is made a lot simpler by the `Microsoft.SqlServer.Management.SMO.Transfer` class. To create a database copy, we first need to create an empty database that will eventually hold the copied database:

```
#Create a database to hold the copy of your database
$dbnamecopy = "$($databasename)_copy"

$dbcopy = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Database -ArgumentList $server, $dbnamecopy
$dbcopy.Create()
```

We will then need to create an SMO Transfer class, which accepts the source database as a parameter:

```
$transfer = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Transfer -ArgumentList $sourcedatabase
```

In the transfer object, you can specify properties you want either brought over or excluded, when the copy happens:

```
$transfer.CopyAllTables = $true
$transfer.Options.WithDependencies = $true
$transfer.Options.ContinueScriptingOnError = $true
$transfer.DestinationDatabase = $dbnamecopy
$transfer.DestinationServer = $server.Name
$transfer.DestinationLoginSecure = $true
$transfer.CopySchema = $true
```

There is an option to just script out the transfer, if you wish to just generate the copy script. You achieve this using the ScriptTransfer method:

```
#if you want to only produce a script that will
#"copy" your database, use the ScriptTransfer method
$transfer.ScriptTransfer()
```

When you are ready to bring the data and schema over, you can use the TransferData method:

```
#if you want to perform the actual transfer
#you should use the TransferData method
$transfer.TransferData()
```

See also

- ▶ To learn more about the SMO Transfer class, check out the MSDN documentation here:

[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.transfer\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.transfer(v=sql.110).aspx)

Executing a SQL query to multiple servers

This recipe executes a pre-defined SQL query to multiple SQL Server instances specified in a text file.

Getting ready

In this recipe, we will connect to multiple SQL Server instances and execute a SQL command against all of them.

Identify the available instances for you to run your query on. Once you have identified all the instances you want to execute the command to, create a text file in `C:\Temp` called `sqlinstances.txt` and put each instance name line by line into that file. For example:

```
KERRIGAN
KERRIGAN\SQL01
```

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the `SQLPS` module:

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

3. Add the following script and run:

```
$instances = Get-content "C:\Temp\sqlinstances.txt"
$query = "SELECT @@SERVERNAME 'SERVERNAME', @@VERSION 'VERSION'"
$databasename = "master"
$instances |
ForEach-Object {
    $server = New-Object -TypeName Microsoft.SqlServer.Management.
Smo.Server -ArgumentList $_
    Invoke-Sqlcmd -ServerInstance $_ -Database $databasename -Query
$query
}
```

How it works...

In this script, we are leveraging the `Invoke-Sqlcmd` cmdlet to accomplish our task.

We first get all the instances and temporarily store them in a variable. Note that you can alternatively just pipe the results of the `Get-Content` cmdlet to the succeeding cmdlets in the pipeline.

```
$instances = Get-content "C:\Temp\sqlinstances.txt"
```

Next we just define the global query we want to execute and the database we want to execute it against, regardless of the instance.

```
$query = "SELECT @@SERVERNAME 'SERVERNAME', @@VERSION 'VERSION'"
$databasename = "master"
```

The core of the recipe is iterating through all instances. For each instance, we create a new SMO server object and use the `Invoke-Sqlcmd` cmdlet to execute the query. Note that what we are passing in the pipeline is the instance name, thus we need to refer to it as `$_` when we create the SMO server object.

```
$instances |
ForEach-Object {
    $server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
    Server -ArgumentList $_
    Invoke-Sqlcmd -ServerInstance $_ -Database $databasename -Query
    $query
}
```

See also

- ▶ The *Executing a Query/SQL script* recipe in Chapter 2

Creating a filegroup

This recipe describes how to create a filegroup programmatically, using PowerShell and SMO.

Getting ready

We will add a filegroup called `FGActive` to your `TestDB` database.

In this recipe, this is the T-SQL equivalent of what we are trying to accomplish:

```
ALTER DATABASE [TestDB]
ADD FILEGROUP [FGActive]
GO
```


How to do it...

These are the steps to add a filegroup to your 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, as follows:

```
#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
```

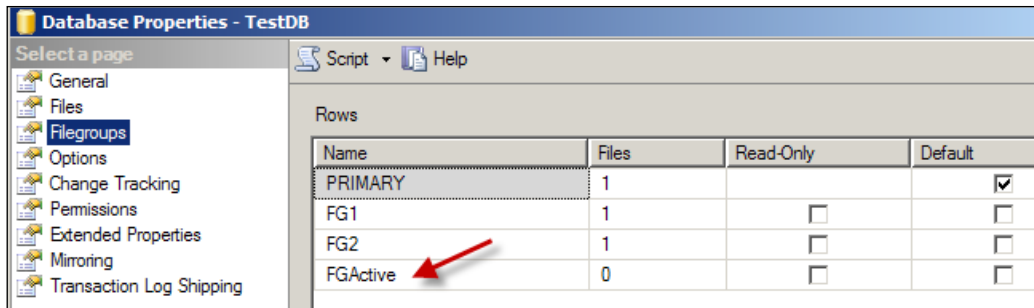
3. Add the following script and run:

```
$databasename = "TestDB"
$database = $server.Databases[$databasename]
$fgroupName = "FGActive"

#For purposes of this test, we are going to drop this
#filegroup if it exists, so we can recreate it without
#any issues
if ($database.FileGroups[$fgname])
{
    $database.FileGroups[$fgname].Drop()
}

#create the filegroup
$fg = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Filegroup -ArgumentList $database, $fgname
$fg.Create()
```

4. Log in to Management Studio and confirm that the filegroup has been added:
 - a. Right-click on **TestDB database** and go to **Properties**.
 - b. On the left-hand pane, click on **Filegroups**. Check if the **FGActive** filegroup is there.



How it works...

Adding a filegroup can be accomplished with very little code, in PowerShell. This task entails creating a `Microsoft.SqlServer.Management.SMO.Filegroup` object and invoking its `Create` method.

```
$fg = New-Object -TypeName Microsoft.SqlServer.Management.SMO.  
Filegroup -ArgumentList $database, $fgname  
$fg.Create()
```

If you want to make this filegroup the default filegroup, it will require adding data files to this filegroup first.

Once data files are added, you can use the following block to make a filegroup default:

```
#make sure there's a data file before you set a  
#filegroup default  
#otherwise you will get an error  
$fg = $database.FileGroups[$fgname]  
$fg.IsDefault = $true  
$fg.Alter()
```

See also

- ▶ [The Adding secondary data files to a filegroup recipe](#)

Adding secondary data files to a filegroup

This recipe walks you through adding secondary data files to a filegroup using PowerShell and SMO.

Getting ready

In this recipe, we will add data files to the `FGActive` filegroup we created for the `TestDB` database in the previous recipe. If you don't have this filegroup yet, execute the following T-SQL statement in Management Studio to create the filegroup:

```
ALTER DATABASE [TestDB]
ADD FILEGROUP [FGActive]
GO
```

In this recipe, we will accomplish this T-SQL equivalent:

```
ALTER DATABASE [TestDB]
ADD FILE (
NAME = N'datafile1',
FILENAME = N'C:\Temp\datafile1.ndf')
TO FILEGROUP [FGActive]
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, as follows:

```
#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:

```
$databasename = "TestDB"
$fgname = "FGActive"
```

```

$fg = $database.FileGroups[$fgname]

#Define a DataFile object on the file group and set the logical
#file name.
$df = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
DataFile -ArgumentList $fg, "datafile1"

#Make sure to have a directory created to hold the designated data
#file
$df.FileName = "c:\\Temp\\datafile1.ndf"

#Call the Create method to create the data file on the instance of
#SQL Server.
$df.Create()

```

How it works...

You will first need to get a handle to the filegroup to which you want to add the secondary file:

```
$fg = $database.FileGroups[$fgname]
```

Once the filegroup handle is in place, you can create a `Microsoft.SqlServer.Management.SMO.DataFile` object and specify the logical filename:

```

#Define a DataFile object on the file group and set the logical file
#name
$df = New-Object -TypeName Microsoft.SqlServer.Management.SMO.DataFile
-ArgumentList $fg, "datafile1"

#Make sure to have a directory created to hold the designated data
#file
$df.FileName = "c:\\Temp\\datafile1.ndf"

```

The last step is to invoke the `Create` method of the `DataFile` object:

```

#Call the Create method to create the data file on the instance of SQL
#Server.
$df.Create()

```

See also

- ▶ *The Creating a filegroup recipe*

Moving an index to a different filegroup

This recipe illustrates how to move indexes to a different filegroup.

Getting ready

Using the `TestDB` database, or any database of your choice, let's create a table called `Student` with a clustered primary key.

Open SQL Server Management Studio, and execute the following code:

```
USE TestDB
GO
-- this is going to be stored to the default filegroup
IF OBJECT_ID('Student') IS NOT NULL
DROP TABLE Student
GO
CREATE TABLE Student
(
  ID INT IDENTITY(1,1) NOT NULL,
  FName VARCHAR(50),
  CONSTRAINT [PK_Student] PRIMARY KEY CLUSTERED
  ([ID] ASC)
)
GO
-- insert some sample data
-- nothing fancy, every student will be called Joe for now :)
INSERT INTO Student(FName)
VALUES('Joe')
GO 20
INSERT INTO Student(FName)
SELECT FName FROM Student
GO 10

-- check how many records are inserted
-- this should give 20480
SELECT COUNT(*) FROM Student
```

The T-SQL equivalent of what we are trying to accomplish in this recipe is as follows:

```
CREATE UNIQUE CLUSTERED INDEX PK_Student
ON dbo.Student
(
  ID ASC
```

```
)
WITH (DROP_EXISTING=ON, ONLINE=ON)
ON FGStudent
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, as follows:

```
#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:

```
$databasename = "TestDB"
$database = $server.Databases[$databasename]
$tablename = "Student"
$table = $database.Tables[$tablename]

#now move to a different filegroup
$fgroupName = "FGStudent"

if ($database.FileGroups[$fggroupName])
{
    $database.FileGroups[$fggroupName].Drop()
}

$fg = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Filegroup -ArgumentList $database, $fggroupName
$fg.Create()

$fg = $database.FileGroups[$fggroupName]

#create a datafile and specify the filename
$df = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
DataFile -ArgumentList $fg, "studentdata"

$df.FileName = "c:\\Temp\\studentdata.ndf"
```

```
#create the datafile
$df.Create()

#now let's recreate the clustered index
#(Microsoft.SqlServer.Management.Smo.Index)
#onto the new filegroup
#note this is V3 syntax because of simplified Where-Object
$clusteredindex = $table.Indexes |
Where-Object IsClustered -eq $true

$clusteredindex.FileGroup = $fgname
$clusteredindex.Recreate()

#display which filegroup the table is on now
$table.Refresh()
$table.FileGroup
```

How it works...

Your indexes might outgrow your initial space allocation for them, or you may want to place them into a different disk purely for performance reasons. There will be a number of reasons to move your indexes to a different filegroup, and the good news is that PowerShell and SMO can accomplish this task.

For purposes of our exercise, the first few steps are creating a filegroup called `FGStudent` and adding a secondary data file into the new filegroup.



See the *Creating a filegroup* and *Adding secondary data files to a filegroup* recipes for additional information.

For this recipe, we will be moving our clustered index into a different filegroup. We need to capture the clustered index. The following code implicitly creates a `Microsoft.SqlServer.Management.Smo.Index` object. Here we use the V3 syntax:

```
$clusteredindex = $table.Indexes |
Where-Object IsClustered -eq $true
```

If you want to do this in a V2 environment, you have to change the `Where-Object` clause:

```
$clusteredindex = $table.Indexes |
Where-Object {$_.IsClustered -eq $true}
```

After you get a handle to the clustered index, you will need to specify the new filegroup this clustered index should belong to:

```
$clusteredindex.FileGroup = $fgname
```

Once you've specified the filegroup, you can invoke the `Recreate` method of the `Microsoft.SqlServer.Management.Smo.Index` object. Note that we are recreating the index—not simply creating it—because the index already exists. The `Recreate` method is equivalent to `CREATE...WITH DROP EXISTING`.

```
$clusteredindex.Recreate()
```

To check, you can refresh the table and see which filegroup the index is attached to:

```
#display which filegroup the table is on now
$table.Refresh()
$table.FileGroup
```

There's more...

To move nonclustered indexes to a different filegroup, you will follow the same method described in the previous recipe. Here's an example:

```
$idxname = $table.Indexes["idxname"]
$idxname.FileGroup = $fgname
$idxname.Recreate()
$idxname.Refresh()
$idxname.FileGroup
```

If you are dealing with a clustered index that is not a primary key, you can also consider the `DropAndMove` method of the `Microsoft.SqlServer.Management.Smo.Index` object. This method drops the clustered index and recreates it in the specified filegroup.

```
$idxname.DropAndMove($fgname)
```

See also

- ▶ *The Creating a filegroup* recipe
- ▶ *The Adding secondary data files to a filegroup* recipe
- ▶ *The Creating an index* recipe

Checking index fragmentation

In this recipe, we will look at the steps to display index fragmentation using SMO and PowerShell.

Getting ready

We will investigate the index fragmentation of the `Person.Person` table in the `AdventureWorks2008R2` database.

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, as follows:

```
#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:

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

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

$table = $database.Tables |
    Where Schema -Like $schemaName |
    Where Name -Like $tableName

#From MSDN:
#EnumFragmentation enumerates a list of
#fragmentation information for the index
#using the default fast fragmentation option.
$table.Indexes |
Foreach {
    $_.EnumFragmentation() |
```

```

Select Index_Name, @{Name="Value";Expression={($_.
AverageFragmentation).ToString("0.0000")}}
} |
Format-Table -AutoSize

```

The result you see should look similar to the one shown in the following screenshot:

Index_Name	Value
AK_Person_rowguid	0.0000
idxLastNameFirstName	0.0000
IX_Person_LastName_FirstName_MiddleName	2.8571
PK_Person_BusinessEntityID	0.0263
PK_Person_BusinessEntityID	0.0000
PK_Person_BusinessEntityID	0.0000
PXML_Person_AddContact	0.0000
PXML_Person_Demographics	0.0000
XMLPATH_Person_Demographics	0.2165
XMLPATH_Person_Demographics	0.0000
XMLPROPERTY_Person_Demographics	0.2168
XMLPROPERTY_Person_Demographics	0.0000
XMLVALUE_Person_Demographics	0.1443
XMLVALUE_Person_Demographics	0.0000

How it works...

The SMO `Index` class contains the `EnumFragmentation` method for the `Microsoft.SqlServer.Management.Smo.Index` object. This object can enumerate fragmentation of indexes in a table.

You can invoke the `EnumFragmentation` method against all indexes in a table. This method provides the following information:

```

Index_Name
Index_ID
Depth
Pages
Rows
MinimumRecordSize
MaximumRecordSize
AverageRecordSize
ForwardedRecords
AveragePageDensity
IndexType
PartitionNumber
GhostRows
VersionGhostRows
AverageFragmentation

```

In the script, we looped through all the indexes and invoked `EnumFragmentation`. We are displaying only the index name and `AverageFragmentation` property (formatted to display four decimal places):

```
$table.Indexes |  
Foreach {  
  $_.EnumFragmentation() |  
  select Index_Name, @{Name="Value";Expression={($_.  
AverageFragmentation).ToString("0.0000")}}  
} |  
Format-Table -AutoSize
```

See also

- ▶ The *Reorganizing/rebuilding an index* recipe
- ▶ You can read more on the `EnumFragmentation` method of the `Microsoft.SqlServer.Management.Smo.Index` object from MSDN:
[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.index.enumfragmentation\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.index.enumfragmentation(v=sql.110).aspx)

Reorganizing/rebuilding an index

This recipe demonstrates how to reorganize or rebuild an index.

Getting ready

We will iterate through all the indexes in the `Person.Person` table in the `AdventureWorks2008R2` database, for this exercise.

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, as follows:

```
#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:

```
$VerbosePreference = "Continue"
$databasename = "AdventureWorks2008R2"
$database = $server.Databases[$databasename]

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

$table = $database.Tables |
    Where Schema -Like $schemaName |
    Where Name -Like $tableName

#From MSDN:
#EnumFragmentation enumerates a list of
#fragmentation information
#for the index using the default fast fragmentation option.
$table.Indexes |
ForEach-Object {
    $_.EnumFragmentation() |
    ForEach-Object {
        $item = $_
        #reorganize if 10 and 30% fragmentation
        if($item.AverageFragmentation -ge 10 -and `
            $item.AverageFragmentation -le 30 -and `
            $item.Pages -ge 1000)
        {
            Write-Verbose "Reorganizing $index.Name ... "
            $index.Reorganize()
        }
        #rebuild if more than 30%
        elseif ($item.AverageFragmentation -gt 30 -and `
            $item.Pages -ge 1000)
        {
            Write-Verbose "Rebuilding $index.Name ... "
            $index.Rebuild()
        }
    }
}

$VerbosePreference = "SilentlyContinue"
```

How it works...

The `EnumFragmentation` method allows additional information about indexes to be extracted—like average fragmentation and number of pages. Instead of just blindly rebuilding or reorganizing all indexes, we can check these properties and put more smarts as to when the indexes need to be reorganized or rebuilt, if at all.

These are the rules of thumb:

- ▶ If fragmentation > 30 percent and pages >= 1000, rebuild
- ▶ If fragmentation is between 10 percent and 30 percent and pages >= 1000, reorganize

1,000 pages for the index page count is more of a guideline (documented in articles and discussed in conferences; check out an old Index **Defragmentation Best Practices White Paper** that discusses this <http://technet.microsoft.com/library/Cc966523>). I personally have used this number in a benchmarking exercise and it worked well in that environment. Test this on your system; you may find that the number of pages that work for you are a little bit higher or a little bit lower.

To do this conditional rebuild/reorganize strategy in PowerShell, you can use an `if/else` statement to divert the action to the correct code block depending on the fragmentation and page values:

```
#reorganize if 10 and 30% fragmentation
if($item.AverageFragmentation -ge 10 -and `
    $item.AverageFragmentation -le 30 -and `
    $item.Pages -ge 1000)
{
    Write-Verbose "Reorganizing $index.Name ... "
    $index.Reorganize()
}
#rebuild if more than 30%
elseif ($item.AverageFragmentation -gt 30 -and `
    $item.Pages -ge 1000)
{
    Write-Verbose "Rebuilding $index.Name ... "
    $index.Rebuild()
}
```

See also

- ▶ The *Checking index fragmentation* recipe

Running DBCC commands

This recipe shows you some of the DBCC commands that can be run using PowerShell.

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, as follows:

```
#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. Some DBCC commands are built into SMO, so you can just call the methods:

```
$databasename = "AdventureWorks2008R2"
$database = $server.Databases[$databasename]
#RepairType Values: AllowDataLost, Fast, None, Rebuild
$database.CheckTables([Microsoft.SqlServer.Management.Smo.
RepairType]::None)
```

How it works...

Not all DBCC commands are wrapped in SMO methods. Some of the available methods on a database level are:

- ▶ CheckAllocations
- ▶ CheckCatalog
- ▶ CheckTables

To invoke the SMO DBCC methods, you need to get a handle to the database.

The `CheckTables` method requires a parameter for `RepairType`:

```
#RepairType Values: AllowDataLost, Fast, None, Rebuild
$database.CheckTables([Microsoft.SqlServer.Management.Smo.
RepairType]::None)
```

For other DBCC commands that are not nicely wrapped in methods, you can still execute them using the `Invoke-Sqlcmd` cmdlet. For example:

```
$query = "DBCC SHRINKFILE(TestDB_Log) "
Invoke-Sqlcmd -ServerInstance $instanceName -Query $query
```

Setting up Database Mail

This recipe demonstrates how to set up Database Mail programmatically, using PowerShell.

Getting ready

The assumption in this recipe is that database mail is not yet configured on your instance.

These are the settings we will use for this recipe:

Setting	Value
Mail Server	mail.queryworks.local
Mail Server Port	25
Email Address for Database Mail Profile	dbmail@queryworks.local
SMTP Authentication	Basic authentication
Credentials for Email Address	Username: dbmail@queryworks.local Password: <somepassword>

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, as follows:

```
#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
```

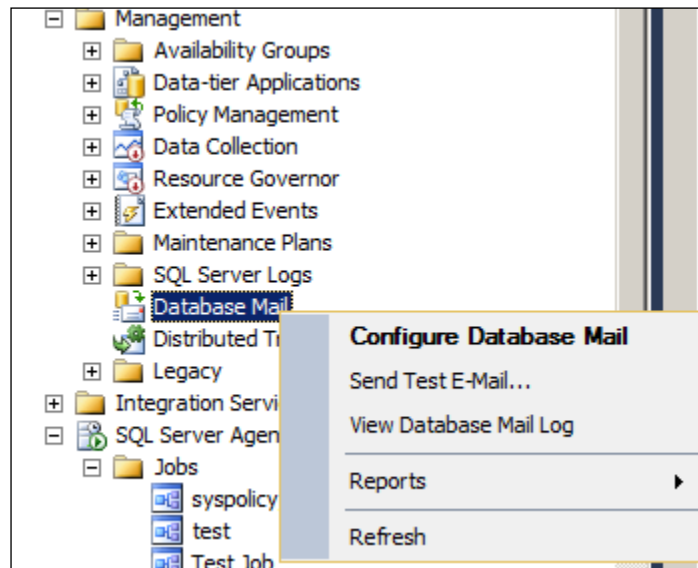
3. Add the following script and run:

```
#enable DatabaseMail
#this is similar to an sp_configure TSQL command
$server.Configuration.DatabaseMailEnabled.ConfigValue = 1
$server.Configuration.Alter()
$server.Refresh()
```

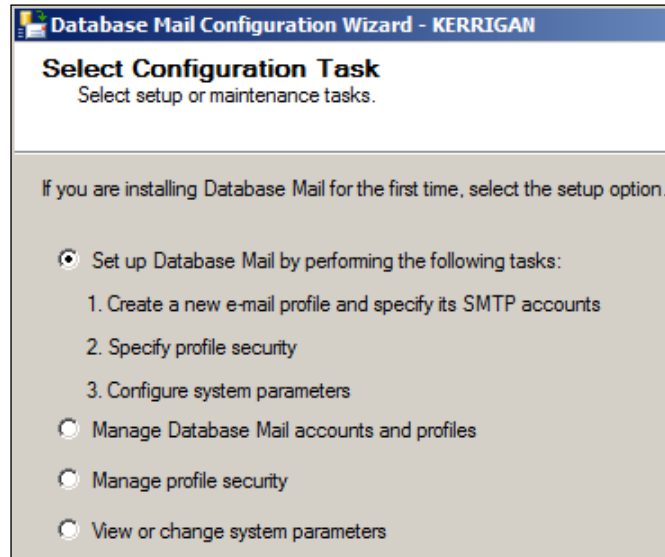
```
#set up account
$accountName = "DBMail"
$accountDescription = "QUERYWORKS Database Mail"
$displayName = "QUERYWORKS mail"
$emailAddress = "dbmail@queryworks.local"
$replyToAddress = "dbmail@queryworks.local"
$mailServerAddress = "mail.queryworks.local"

$account = New-Object -TypeName Microsoft.SqlServer.Management.
SMO.Mail.MailAccount -ArgumentList $server.Mail, $accountName,
$accountDescription, $displayName, $emailAddress
$account.ReplyToAddress = $replyToAddress
$account.Create()
```

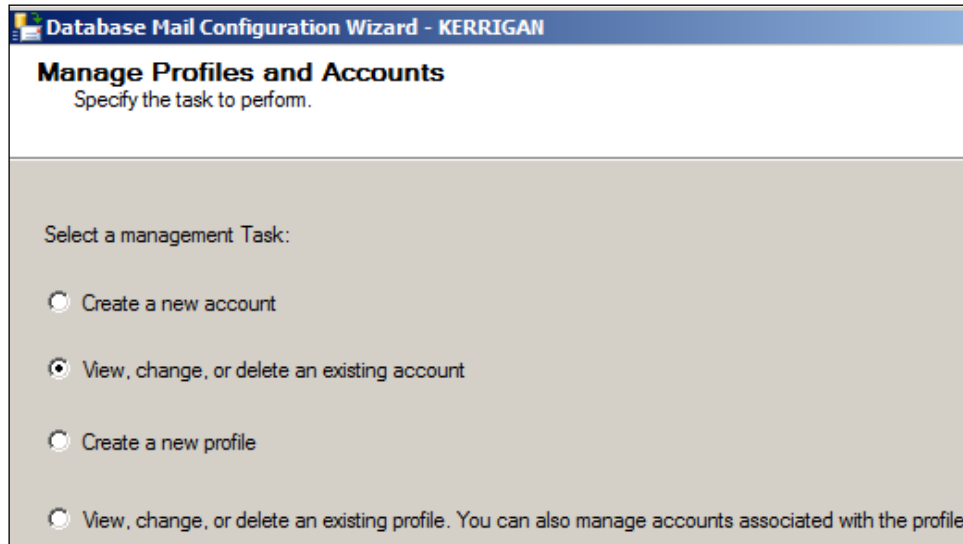
4. Check the settings that have been changed from SQL Server Management Studio:
 1. Open SQL Server Management Studio.
 2. Expand the **Management** node.
 3. Right-click on **Database Mail** and choose **Configure Database Mail**.



4. In the **Select Configuration Task** window, select the **Manage Database Mail accounts and profiles** radio button.



5. In the **Manage Profiles and Accounts** window, select the **View, change, or delete an existing account** option.



6. Visually check the **Manage Existing Account** page. See what settings have been saved from executing your PowerShell script. Notice that in the **SMTP authentication** section, **Anonymous Authentication** has been selected by default.

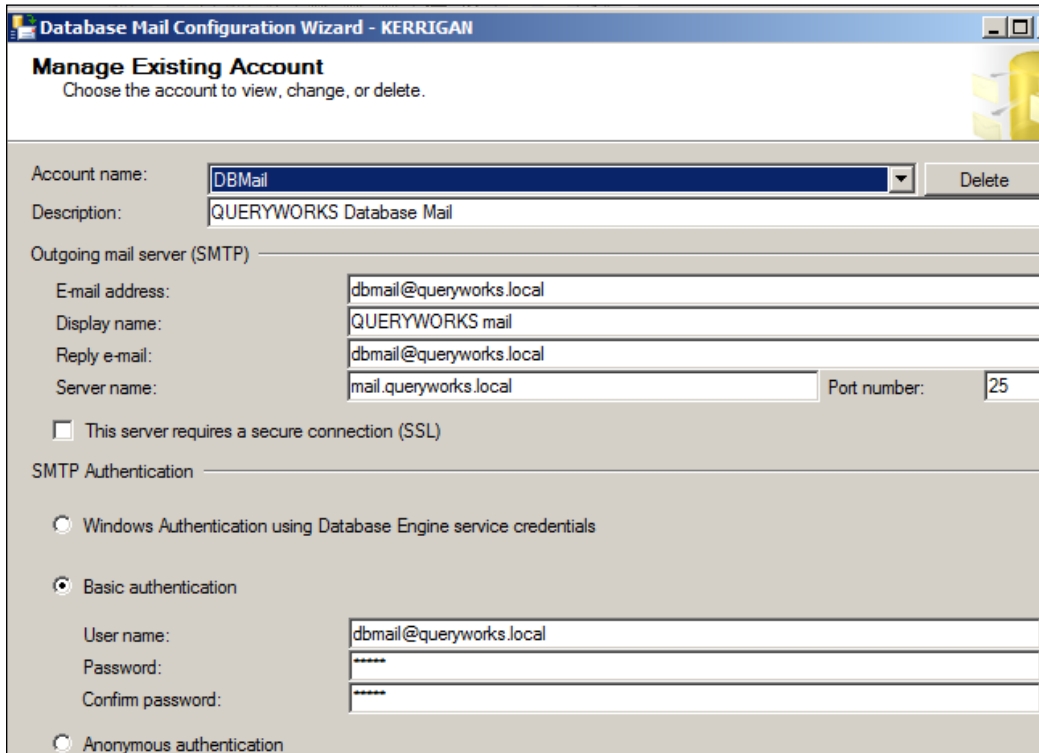
The screenshot shows the 'Manage Existing Account' window in the Database Mail Configuration Wizard. The account name is 'DBMail' and the description is 'QUERYWORKS Database Mail'. The outgoing mail server (SMTP) settings are: E-mail address: dbmail@queryworks.local, Display name: QUERYWORKS mail, Reply e-mail: dbmail@queryworks.local, Server name: KERRIGAN, and Port number: 25. The 'SMTP Authentication' section has three options: 'Windows Authentication using Database Engine service credentials' (unselected), 'Basic authentication' (unselected), and 'Anonymous authentication' (selected).

5. Click on **Cancel** to exit the wizard and go back to **PowerShell ISE**.
6. Add the following script and run:

```
#default mail server that was saved in the previous script
#was the server name, we need to change this to the
#appropriate mail server
$mailserver = $account.MailServers[$instanceName]
$mailserver.Rename($mailServerAddress)
$mailserver.Alter()

#default SMTP authentication is Anonymous Authentication
#set propert authentication
$mailserver.SetAccount("dbmail@queryworks.local", "some password")
$mailserver.Port = 25
$mailserver.Alter()
```

7. Check the **Manage Existing Account** window from **Management Studio** again. Check if these new settings have been saved.



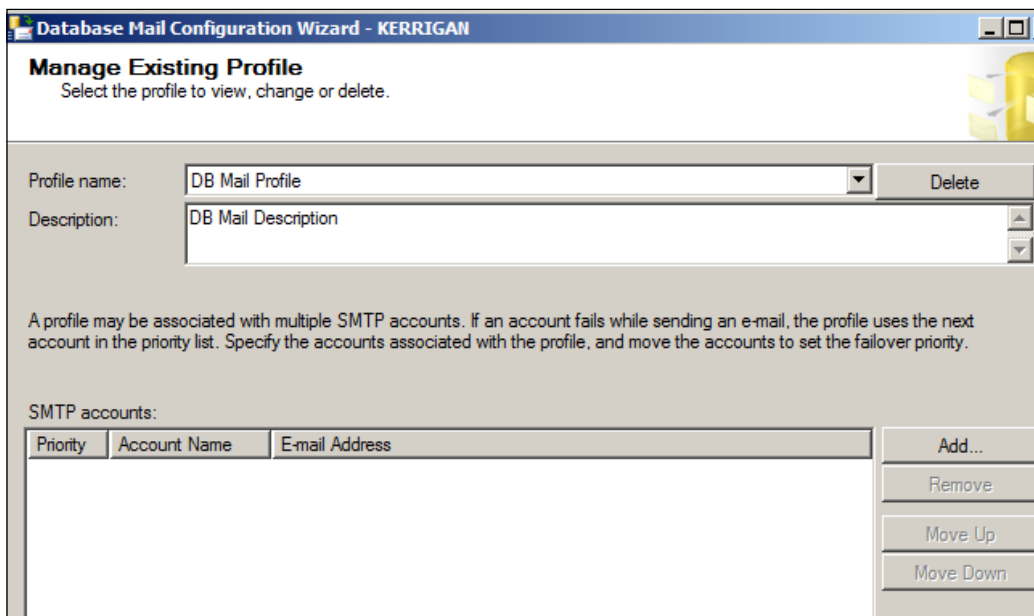
8. Click on **Cancel** to exit the wizard and go back to **PowerShell ISE**.
9. Add the following script and run:

```
#create a profile
$profileName = "DB Mail Profile"
$profileDescription= "DB Mail Description"

if($mailProfile)
{
    $mailProfile.Drop()
}
```

```
$mailProfile = New-Object -TypeName Microsoft.SqlServer.
Management.SMO.Mail.MailProfile -ArgumentList $server.Mail,
$profileName, $profileDescription
$mailProfile.Create()
$mailProfile.Refresh()
```

10. Check the settings from SQL Server Management Studio.
 1. Go back to the **Manage Profiles and Accounts** window, but this time select **View, change, or delete an existing profile**.
 2. Visually check the **Manage Existing Profile** page. Notice that, apart from the name and description, the window is still fairly empty.



11. Click on **Cancel** to exit the wizard and go back to your **PowerShell ISE**.
12. Add the following script and run:

```
#add account to the profile
$mailProfile.AddAccount($accountName, 0)
$mailProfile.AddPrincipal('public', 1)
$mailProfile.Alter()
```

13. Check the settings from SQL Server Management Studio:
 1. Go back to the **Manage Profiles and Accounts** window, but this time select **View, change, or delete an existing profile**.
 2. Visually check the **Manage Profile Security** page. Notice the default profile that has been saved.



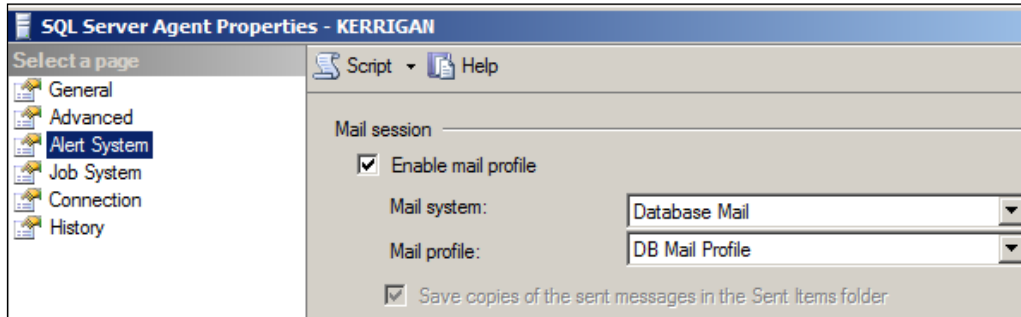
14. Click on **Cancel** to exit the wizard and go back to your **PowerShell ISE**.
15. Add the following script and run:

```
#link this mail profile to SQL Server Agent
$server.JobServer.AgentMailType = 'DatabaseMail'
$server.JobServer.DatabaseMailProfile = $profileName
$server.JobServer.Alter()

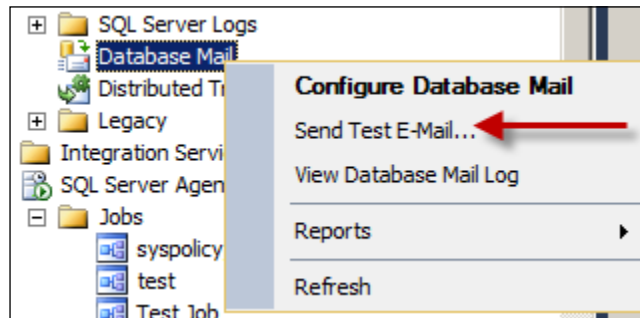
#restart SQL Server Agent
$managedComputer = New-Object 'Microsoft.SqlServer.Management.Smo.Wmi.ManagedComputer' $instanceName
$servicename = "SQLSERVERAGENT"
$service = $managedComputer.Services[$servicename]
$service.Stop()
$service.Start()
```

16. Check settings from **Management Studio**:
 1. Right-click on **SQL Server Agent** and go to **Properties**.
 2. Click on **Alert System** from the left-hand pane.

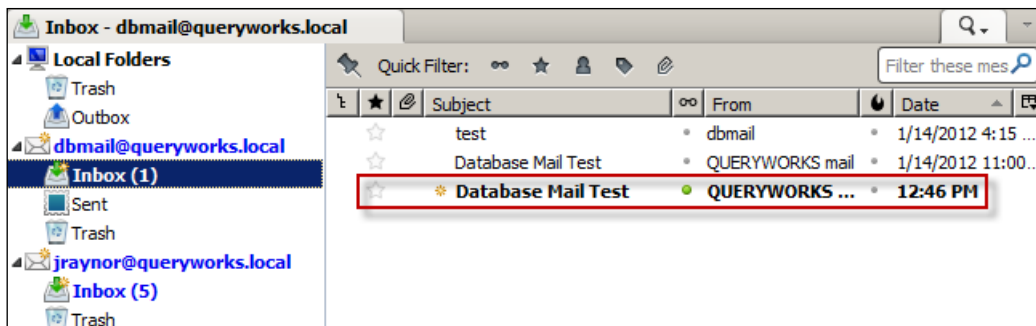
3. Check the settings. The **Enable mail profile** option should be checked.



17. Manually test sending an e-mail. Right-click on **Database Mail** and choose **Send Test E-mail**, as shown in the following screenshot:



18. Check your mail client to see if you have received the e-mail.



How it works...

Database Mail is a feature introduced in SQL Server 2005 that simplifies the sending of e-mails from your SQL Server instance. With Database Mail, you can set up:

- ▶ **Accounts:** These store the e-mail accounts and associated credentials that Database Mail can use to send e-mails.
- ▶ **Profiles:** These can store multiple accounts. If an account in a profile fails, the next one in the queue will be used.

Database Mail is a disabled service by default. To start using it, you first need to enable it.

```
#enable DatabaseMail
#this is similar to an sp_configure TSQL command
$server.Configuration.DatabaseMailEnabled.ConfigValue = 1
$server.Configuration.Alter()
```

The previous statement is equivalent to the following T-SQL statement:

```
EXEC sp_configure 'Database Mail XPs', 1
GO
RECONFIGURE
GO
```

To continue with setting up Database Mail, you need to set up an account first:

```
#set up account
$accountName = "DBMail"
$accountDescription = "QUERYWORKS Database Mail"
$displayName = "QUERYWORKS mail"
$emailAddress = "dbmail@queryworks.local"
$replyToAddress = "dbmail@queryworks.local"
$mailServerAddress = "mail.queryworks.local"

$account = New-Object -TypeName Microsoft.SqlServer.Management.
SMO.Mail.MailAccount -ArgumentList $server.Mail, $accountName,
$accountDescription, $displayName, $emailAddress
$account.ReplyToAddress = $replyToAddress
$account.Create()
```

The next step is to create a profile:

```
$mailProfile = New-Object -TypeName Microsoft.SqlServer.Management.  
SMO.Mail.MailProfile -ArgumentList $server.Mail, $profileName,  
$profileDescription;  
$mailProfile.Create()  
$mailProfile.Refresh()
```

Once both the account(s) and profile are set up, you need to add the accounts to the mail profile:

```
#add account to the profile  
$mailProfile.AddAccount($accountName, 0)  
$mailProfile.AddPrincipal('public', 1)  
$mailProfile.Alter()
```

A big reason to set up Database Mail is to use this with SQL Server Agent. If this is not set up, SQL Server Agent will not be able to alert operators for a job via e-mail. Setting up the alert for SQL Server Agent is a key step and is often missed.

```
#link this mail profile to SQL Server Agent  
$server.JobServer.AgentMailType = 'DatabaseMail'  
$server.JobServer.DatabaseMailProfile = $profileName  
$server.JobServer.Alter()
```

Once the Database Mail profile is hooked to SQL Server Agent, you also need to restart the server before you can start using it.

There's more...

In the development server, I've used hMailServer as my mail server. hMailServer (<http://www.hmailserver.com/>) is a free e-mail server for machines running on Windows operating systems. hMailServer supports IMAP, SMTP and POP3.

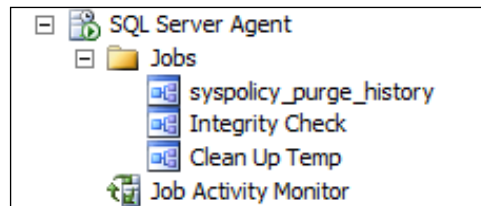
I needed to install a mail server in my Windows Server 2008 R2 VM because Windows Server 2008 and 2008 R2 no longer come with a POP3 server, which can be used to demonstrate or test e-mail capabilities. Windows Server 2003 used to come with this service.

Listing SQL Server jobs

This recipe illustrates how to list SQL Server jobs using PowerShell.

Getting ready

Do a visual check of the SQL Server jobs in your instance. These should be the jobs you will see after you run the script in this 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, as follows:

```
#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:

```
$jobs=$server.JobServer.Jobs
$jobs |
Select Name, OwnerLoginName, LastRunDate, LastRunOutcome |
Sort -Property Name |
Format-Table -AutoSize
```

How it works...

Listing SQL Server jobs is a short, simple task in PowerShell. To list the jobs, you first need to get a handle to the `JobServer.Jobs` object:

```
$jobs=$server.JobServer.Jobs
```

Once you have the jobs, you can query the properties you are interested in:

```
$jobs |
Select Name, OwnerLoginName, LastRunDate, LastRunOutcome |
Sort -Property Name |
Format-Table -AutoSize
```

Each `Job` object exposes a variety of information about the job. Here is a sample of the complete output from a single job:

```
Parent           : [KERRIGAN]
Category         : [Uncategorized (Local)]
CategoryType     : 1
CurrentRunRetryAttempt : 0
CurrentRunStatus  : Idle
CurrentRunStep    : 0 (unknown)
DateCreated      : 1/15/2012 4:30:58 PM
DateLastModified : 1/26/2012 8:58:30 PM
DeleteLevel      : Never
Description       : No description available.
EmailLevel       : OnFailure
EventLogLevel    : OnFailure
HasSchedule      : True
HasServer        : True
HasStep          : True
IsEnabled        : True
JobID            : 908881ad-ad98-42f7-813a-52b93853b1d2
JobType          : Local
LastRunDate      : 1/27/2012 11:30:00 PM
LastRunOutcome   : Succeeded
NetSendLevel     : Never
NextRunDate      : 1/30/2012 12:00:00 AM
NextRunScheduleID : 37
```

```
OperatorToEmail      : jraynor
OperatorToNetSend    :
OperatorToPage       :
OriginatingServer    : KERRIGAN
OwnerLoginName       : KERRIGAN\Administrator
PageLevel            : Never
StartStepID          : 1
VersionNumber        : 23
Name                 : Sample Job
CategoryID           : 0
JobSteps             : {Step 1}
JobSchedules         : {Every 3rd Friday 6AM, Every Monthend 1130PM,
Every Monthend 1130PM 2, Every Monthend 1130PM 2...}
Urn                  : Server[@Name='KERRIGAN']/JobServer/Job[@
Name='Test Job' and @CategoryID='0']
Properties           : {Name=Category/Type=System.String/Writable=True/
Value=[Uncategorized (Local)], Name=CategoryID/Type=System.Int32/
Writable=True/Value=0,
                        Name=CategoryType/Type=System.Byte/
Writable=True/Value=1, Name=CurrentRunRetryAttempt/Type=System.Int32/
Writable=False/Value=0...}
UserData            :
```

If you want to list only the failed jobs using PowerShell V3, pipe the results and filter for LastRunOutcome of Failed:

```
$jobs=$server.JobServer.Jobs
$jobs |
Where LastRunOutcome -Like "Failed" |
Select Name, OwnerLoginName, LastRunDate, LastRunOutcome |
Format-Table -AutoSize
```

On a V2 environment, you can use the following Where-Object syntax:

```
Where {$_.LastRunOutcome -Like "Failed"} |
```

See also

- ▶ The *Creating a SQL Server job* recipe
- ▶ You can learn more about the SQL Server Job class from here:
[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.agent.job\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.agent.job(v=sql.110).aspx)

Adding a SQL Server operator

This recipe shows how you can create a SQL Server operator using SMO and PowerShell

Getting ready

For this recipe, we will create an operator with the following settings:

Setting	Value
Operator name	jraynor
Operator e-mail	jraynor@queryworks.local

If you do not have this account set up in your system, you can substitute this with another available account in your environment.

To set up an operator, you must be a sysadmin in your instance.

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, as follows:

```
#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
```

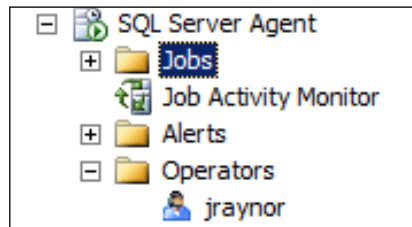
3. Add the following script and run:

```
$jobserver      = $server.JobServer
$operatorName   = "jraynor"
$operatorEmail  = "jraynor@queryworks.local"

$operator = New-Object Microsoft.SqlServer.Management.Smo.Agent.
Operator -ArgumentList $jobserver, $operatorName
$operator.EmailAddress = $operatorEmail
$operator.Create()

#verify by listing operators
$jobserver.Operators
```

4. Open SQL Server Management Studio and check if the operator has been created. Expand **SQL Server Agent | Operators**.



How it works...

To create an operator, you must first get a handle to the `JobServer` object of your instance:

```
$jobserver = $server.JobServer
```

An operator will require a name, and a method to be contacted. We are going to use e-mail in this case, but you can also specify the `NetSendAddress` and `PagerAddress` properties of the `Microsoft.SqlServer.Management.Smo.Agent.Operator` object:

```
$operatorName = "jraynor"
$operatorEmail = "jraynor@queryworks.local"

$operator = New-Object Microsoft.SqlServer.Management.Smo.Agent.
Operator -ArgumentList $jobserver, $operatorName
$operator.EmailAddress = $operatorEmail
```

Once these settings are in place, you can just invoke the `Create` method of the `Microsoft.SqlServer.Management.Smo.Agent.Operator` object to persist the operator in the instance:

```
$operator.Create()
```

See also

- ▶ The *Creating a SQL Server job* recipe
- ▶ The *Adding a SQL Server agent alert* recipe
- ▶ To learn more about the SQL Server `Operator` class, check out the MSDN entry:
[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.agent.operator\(v=SQL.110\)](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.agent.operator(v=SQL.110))

Creating a SQL Server job

In this recipe, we will create a simple SQL Server job programmatically.

Getting ready

We are going to create a simple job called `Test Job`, and set up `jraynor` as our operator. If you don't have `jraynor`, choose another SQL Server operator that's available in your instance.

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, as follows:

```
#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:

```
$jobName = "Test Job"

if($server.JobServer.Jobs[$jobName])
{
    $server.JobServer.Jobs[$jobName].Drop()
}

$job = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Agent.Job -ArgumentList $server.JobServer, $jobName

#Specify which operator to inform and the completion action.
$operatorName = "jraynor"
$operator = $server.JobServer.Operators[$operatorName]
$job.OperatorToEmail = $operator.Name

#CompletionAction can be Never, OnSuccess, OnFailure, Always
$job.EmailLevel = [Microsoft.SqlServer.Management.SMO.Agent.
CompletionAction]::OnFailure
```

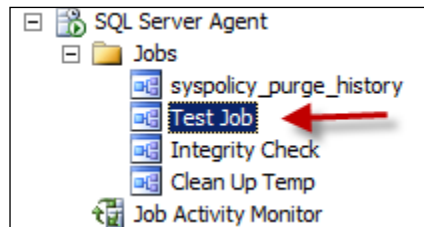
Basic Administration

```
#create
$job.Create()

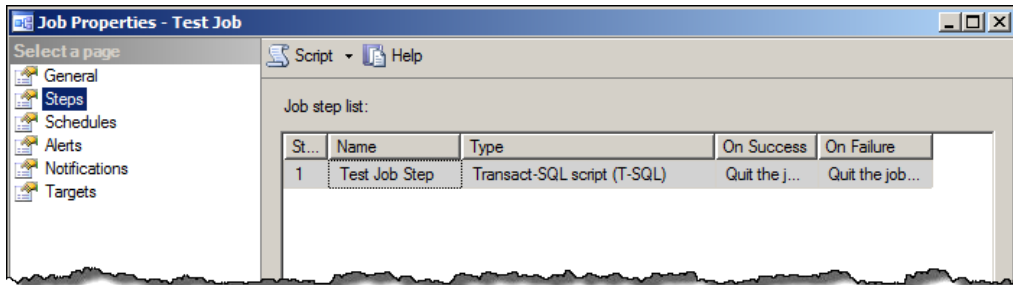
#apply to local instance of SQL Server
$job.ApplyToTargetServer($instanceName)

#now let's add a simple T-SQL Job Step
$jobStep = New-Object Microsoft.SqlServer.Management.Smo.Agent.
JobStep($job, "Test Job Step")
$jobStep.Subsystem = [Microsoft.SqlServer.Management.Smo.Agent.
AgentSubSystem]::TransactSql
$jobStep.Command = "SELECT GETDATE()"
$jobStep.OnSuccessAction = [Microsoft.SqlServer.Management.Smo.
Agent.StepCompletionAction]::QuitWithSuccess
$jobStep.OnFailAction = [Microsoft.SqlServer.Management.Smo.Agent.
StepCompletionAction]::QuitWithFailure
$jobStep.Create()
```

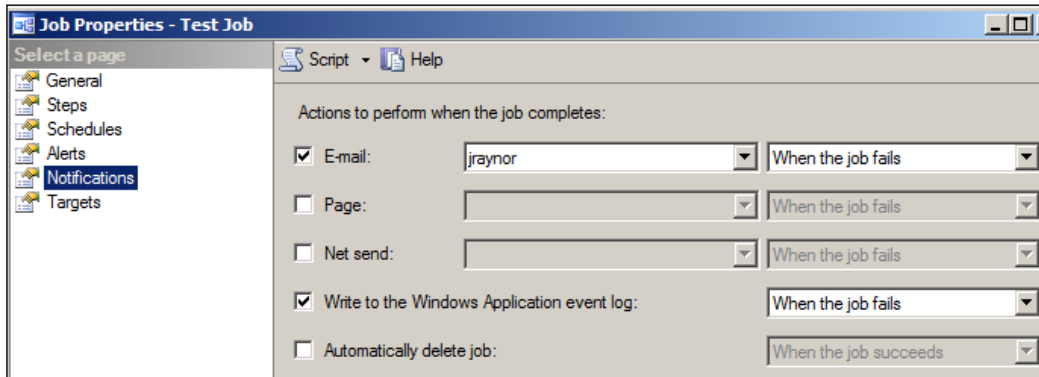
4. Use SQL Server Management Studio to check if this job has been created:



5. Go to **Steps** and you should see the T-SQL step we added:



- Go to **Notifications** and you should see that **jraynor** was added as the operator to receive the e-mail notifications:



How it works...

To create a Job programmatically, first create a `Microsoft.SqlServer.Management.SMO.Agent.Job` object:

```
$job = New-Object -TypeName Microsoft.SqlServer.Management.SMO.Agent.Job -ArgumentList $server.JobServer, "Test Job"
```

Next, specify the operator. This is an optional step.

```
#Specify which operator to inform and the completion action.
$operatorName = "jraynor"
$operator = $server.JobServer.Operators[$operatorName]
$job.OperatorToEmail = $operator.Name
```

For the notification, you can select either by e-mail, net send, or pager. You will also need to specify when the alert should happen. This can be either `Never`, `OnSuccess`, `OnFailure`, or `Always`.

```
$job.EmailLevel = [Microsoft.SqlServer.Management.SMO.Agent.CompletionAction]::OnFailure
```

When ready, invoke the `Create` method of the `Microsoft.SqlServer.Management.SMO.Agent.Job` object.

We also need to specify the target server; in our case, just the local instance of SQL Server:

```
$job.ApplyToTargetServer($instanceName)
```


In this recipe we also add a simple job step:

```
#now let's add a simple T-SQL Job Step
$jobStep = New-Object Microsoft.SqlServer.Management.Smo.Agent.
JobStep($job, "Test Job Step")
```

We can create different types of job steps in SQL Server, and these are defined in PowerShell as an `AgentSubSystem` enumeration. The possible values for this `Microsoft.SqlServer.Management.Smo.Agent.AgentSubSystem` enumeration are:

- ▶ `TransactSql`
- ▶ `ActiveScripting`
- ▶ `CmdExec`
- ▶ `Snapshot`
- ▶ `LogReader`
- ▶ `Distribution`
- ▶ `Merge`
- ▶ `QueueReader`
- ▶ `AnalysisQuery`
- ▶ `AnalysisCommand`
- ▶ `Ssis`
- ▶ `PowerShell`

For our simple step, we will use a T-SQL subsystem. We will also attach a simple T-SQL statement to this step to retrieve the current system date as a command:

```
$jobStep.Subsystem = [Microsoft.SqlServer.Management.Smo.Agent.
AgentSubSystem]::TransactSql
$jobStep.Command = "SELECT GETDATE() "
```

We can also define the failure and completion actions:

```
$jobStep.OnSuccessAction = [Microsoft.SqlServer.Management.Smo.Agent.
StepCompletionAction]::QuitWithSuccess
$jobStep.OnFailAction = [Microsoft.SqlServer.Management.Smo.Agent.
StepCompletionAction]::QuitWithFailure
```

When ready, we can create the job step by invoking the `Create` method of the `Microsoft.SqlServer.Management.Smo.Agent.JobStep` object:

```
$jobStep.Create()
```

See also

- ▶ The *Listing SQL Server jobs* recipe
- ▶ The *Creating a SQL Server operator* recipe
- ▶ Check out MSDN for the `Microsoft.SqlServer.Management.Smo.Agent.AgentSubSystem` enumeration:
<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.agent.agentsubsystem.aspx>

Adding a SQL Server event alert

This recipe walks you through the steps in adding a SQL Server event alert.

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, as follows:

```
#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:

```
$jobserver = $server.JobServer
#for purposes of our exercise, we will drop this
#alert if it already exists
$alertname = "Test Alert"
$alert = $jobserver.Alerts[$alertname]

if($alert)
{
    $alert.Drop()
}

#accepts a JobServer and an alert name
$alert = New-Object Microsoft.SqlServer.Management.Smo.Agent.
Alert $jobserver, "Test Alert"
$alert.Severity = 10
```

```
#Raise Alert when Message contains
$alert.EventDescriptionKeyword = "failed"

#Set notification message
$alert.NotificationMessage = "This is a test alert, dont worry"

$alert.Create()
```

How it works...

To create an alert, you will first need to create a `Microsoft.SqlServer.Management.Smo.Agent.Alert` object:

```
$alert = New-Object Microsoft.SqlServer.Management.Smo.Agent.Alert
$jobserver, "Test Alert"
```

This alert, by default, is a `SQLServerEvent` alert type. For this alert event type, we will need to specify either error number or severity. You can also optionally specify a keyword that can trigger this notification:

```
$alert.Severity = 10

#Raise Alert when Message contains
$alert.EventDescriptionKeyword = "failed"
```

Other options for the alert type are: `SqlServerPerformanceCondition`, `NonSqlServerEvent`, and `WmiEvent`. The `AlertType` property is a read-only property. To choose an event alert type, you will need to set the properties required for that alert type. For example, if you want to create a `WmiEvent` alert, you will need to set the values for `WmiEventNamespace` and `WmiEventQuery`.

Once the alert settings have been provided, you can also add a notification message:

```
$alert.NotificationMessage = "This is a test alert, dont worry"
```

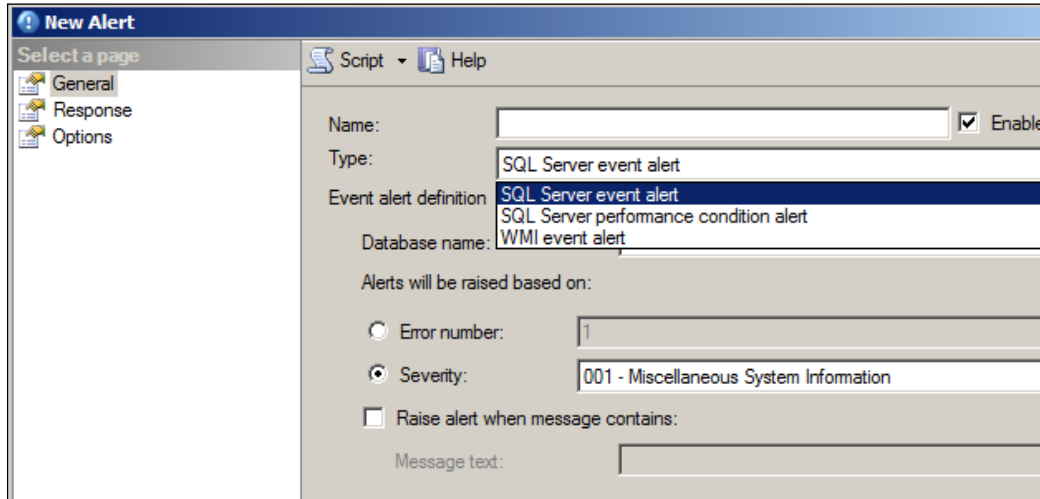
To create the alert, just invoke the `Create` method of the `Microsoft.SqlServer.Management.Smo.Agent.Alert` object:

```
$alert.Create()
```

There's more...

SQL Server provides a mechanism to alert DBAs and other database staff of possible issues or thresholds reached by the instances. If you navigate to **SQL Server Agent** and expand **Alerts**, you should see all the alerts set up in your instance.

When you first set up a SQL Server Agent alert, you will be shown the **New Alert** window:



This table summarizes the types of alerts you can set up in SQL Server:

Alert type	Description
SQL Server event alert	Typically used for specific error numbers, severity, or keywords that exist in the error message.
SQL Server performance condition alert	Typically set up if a performance threshold is reached. For example, if data file size exceeds 100 GB.
WMI event alert	Used for WMI events that you want to flag within SQL Server. For example, if you want to monitor if a file gets created, or a deadlock is detected in one of the instances.

To learn more about the `Alert` class, check out the MSDN documentation here:

[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.agent.alert\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.agent.alert(v=sql.110).aspx)

See also

- ▶ The *Setting up WMI events* recipe

Running a SQL Server job

This recipe demonstrates how you can run a SQL Server job programmatically.

Getting ready

In this recipe, we assume you have a job in your development environment called `Test Job` that you can run. If not, pick another job in your system that you can run.

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, as follows:

```
#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:

```
$jobserver = $server.JobServer
$jobname = "Test Job"

$job = $jobserver.Jobs[$jobname]
$job.Start()

#sleep to wait for job to finish
#check last run date
Start-Sleep -s 1
$job.Refresh()
$job.LastRunDate
```

How it works...

The first step is to get a handle to your instance's `JobServer` object.

```
$jobserver = $server.JobServer
$jobname = "Test Job"
```

You also need to specify the name of the job you want to run. Once you get a handle to the name, you can just invoke the `Start` method of the `JobServer.Job` object:

```
$job = $jobserver.Jobs[$jobname]
$job.Start()
```

If you want to start your job at a specified step, you can pass in the job step name to the `Start` method.

To check if it recently ran, you can check the `LastRunDate`:

```
#sleep to wait for job to finish
#check last run date
Start-Sleep -s 1
$job.Refresh()
$job.LastRunDate
```

An alternative way to check is to go to SQL Server Management Studio. Go to that job, right-click on it, and select **View Job History**. The window that will appear should show a history of the times this job has been run, including the job run status.

The screenshot shows the 'Log File Viewer - KERRIGAN' window. On the left, under 'Select logs', 'Job History' is selected. The main pane shows a table of job history entries. A red arrow points to the first entry, which has a green checkmark in the status column.

Date	Step ID	Server	Job Name
1/29/2012 3:25:11 PM		KERRIGAN	Test Job
1/27/2012 11:30:00 PM		KERRIGAN	Test Job
1/27/2012 11:00:00 PM		KERRIGAN	Test Job
1/27/2012 10:30:00 PM		KERRIGAN	Test Job

See also

- ▶ The *Scheduling a SQL Server job* recipe

Scheduling a SQL Server job

In this recipe, we will demonstrate how to schedule a SQL Server job using PowerShell and SMO.

Getting ready

In this recipe, we assume you have a job in your development environment called `Test Job` that you can run. If not, pick another job in your system that you can run.

We will schedule this job to run every weekend night at 10 P.M.

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, as follows:

```
#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:

```
$jobserver = $server.JobServer
$jobname = "Test Job"

$job = $jobserver.Jobs[$jobname]
$jobschedule = New-Object -TypeName Microsoft.SqlServer.
Management.SMO.Agent.JobSchedule -ArgumentList $job, "Every
Weekend Night 10PM"

#Values for FrequencyTypes are:
#AutoStart, Daily, Monthly, MonthlyRelative, OneTime,
#OnIdle, Unknown, Weekly
$jobschedule.FrequencyTypes = [Microsoft.SqlServer.Management.
SMO.Agent.FrequencyTypes]::Weekly

<#
#from MSDN:
#These are the list of FrequencyInterval values
```

```

WeekDays.Sunday = 1
WeekDays.Monday = 2
WeekDays.Tuesday = 4
WeekDays.Wednesday = 8
WeekDays.Thursday = 16
WeekDays.Friday = 32
WeekDays.Saturday = 64
WeekDays.WeekDays = 62
WeekDays.WeekEnds = 65
WeekDays.EveryDay = 127

```

Combine values using an OR logical operator to set more than a single day.

For example, combine `WeekDays.Monday` and `WeekDays.Friday` (`FrequencyInterval = 2 + 32 = 34`) to schedule an activity for Monday and Friday.

```
#>
```

```
#every Saturday and Sunday
```

```
#can also use 65
```

```
$jobschedule.FrequencyInterval = [Microsoft.SqlServer.Management.SMO.Agent.WeekDays]::WeekEnds
```

```
#set time
```

```
#3 parameters - hours, mins, days
```

```
#if we don't specify time, it will start at midnight
```

```
$starttime = New-Object -TypeName TimeSpan -ArgumentList 22, 0, 0
```

```
$jobschedule.ActiveStartTimeOfDay = $starttime
```

```
#frequency of recurrence
```

```
$jobschedule.FrequencyRecurrenceFactor = 1
```

```
$jobschedule.ActiveStartDate = "01/01/2012"
```

```
#Create the job schedule on the instance of SQL Agent.
```

```
$jobschedule.Create()
```

4. Check the schedule from SQL Server Management Studio:
 1. Go to **Management Studio**.
 2. Under **SQL Server Agent**, double-click on **Test Job**.

3. Click on **Schedules** on the left-hand side pane. Confirm that the schedule has been created.

Schedule list:			
ID	Name	Enabled	Description
39	Every Weekend Night 10PM	Yes	Occurs every week on Saturday, Sunday at 10:00:00 PM.

How it works...

To schedule a job, you first need to get a handle to the job you are scheduling:

```
$job = $jobserver.Jobs[$jobname]
```

The next step is to create a `Microsoft.SqlServer.Management.SMO.Agent.JobSchedule` object. You need to pass the job object and the name of the schedule.

```
$jobschedule = New-Object -TypeName Microsoft.SqlServer.Management.SMO.Agent.JobSchedule -ArgumentList $job, "Every Weekend Night 10PM"
```

For this recipe, we wanted to schedule it every Saturday and Sunday at 10 P.M. The settings that need to be set are:

- ▶ **FrequencyTypes**
- ▶ **FrequencyInterval**
- ▶ **ActiveStartTimeOfDay**
- ▶ **FrequencyRecurrenceFactor**
- ▶ **ActiveStartDate**

You will notice that depending on the schedule you want to set, you may need to skip some of these settings, but need to set different properties altogether.



More scheduling examples are provided in the *There's more...* section.

Because the schedule happens every week, we need to set the `FrequencyType` to `Weekly`:

```
$jobschedule.FrequencyTypes = [Microsoft.SqlServer.Management.SMO.Agent.FrequencyTypes]::Weekly
```

Different values available for `FrequencyTypes` are: `AutoStart`, `Daily`, `Monthly`, `MonthlyRelative`, `OneTime`, `OnIdle`, `Unknown`, and `Weekly`.

For `FrequencyInterval`, we need to set the value to every weekend:

```
#every Saturday and Sunday
#can also use 65
$jobschedule.FrequencyInterval = [Microsoft.SqlServer.Management.SMO.
Agent.WeekDays]::WeekEnds
```

Note that valid `FrequencyInterval` values are as follows:

FrequencyInterval	Value	Notes
<code>WeekDays.Sunday</code>	1	2^0
<code>WeekDays.Monday</code>	2	2^1
<code>WeekDays.Tuesday</code>	4	2^2
<code>WeekDays.Wednesday</code>	8	2^3
<code>WeekDays.Thursday</code>	16	2^4
<code>WeekDays.Friday</code>	32	2^5
<code>WeekDays.Saturday</code>	64	2^6
<code>WeekDays.WeekDays</code>	62	Monday + Tuesday + ... + Friday
<code>WeekDays.WeekEnds</code>	65	Saturday + Sunday
<code>WeekDays.EveryDay</code>	127	Sunday + Monday + ... + Saturday

As documented in MSDN, should you decide to mix and match the days, you will have to use a logical OR to get the value. For example, if you want to schedule a job for Wednesday (8) and Thursday (16), the value you assign to `FrequencyInterval` should be $8+16 = 24$.

To specify that the job needs to run at 10 P.M., we need to use a `TimeSpan` object, which accepts three parameters for hour, minute, and second:

```
$starttime = New-Object -TypeName TimeSpan -ArgumentList 22, 0, 0
$jobschedule.ActiveStartTimeOfDay = $starttime
```

To set the start date, we need to set the `ActiveStartDate` property of the `JobSchedule` object:

```
$jobschedule.ActiveStartDate = "01/01/2012"
```

The `FrequencyRecurrenceFactor` property specifies how often in this time period should the job run. In this case, only once:

```
#frequency of recurrence
$jobschedule.FrequencyRecurrenceFactor = 1
```

The last piece is to invoke the `Create` method:

```
#Create the job schedule on the instance of SQL Agent.
$jobschedule.Create()
```

There's more...

There are various possible schedules that you may need to set up for jobs in your instance. Here are a few more samples, with different variations, to get you started:

Schedule	PowerShell code to set up a schedule
Every weekend at 10 P.M.	<pre>\$jobschedule.FrequencyTypes = [Microsoft.SqlServer.Management.SMO.Agent.FrequencyTypes]::Weekly #every Saturday and Sunday \$jobschedule.FrequencyInterval = [Microsoft.SqlServer.Management.SMO.Agent.WeekDays]::WeekEnds #10PM \$starttime = New-Object -TypeName TimeSpan -ArgumentList 22, 0, 0 \$jobschedule.ActiveStartTimeOfDay = \$starttime \$jobschedule.FrequencyRecurrenceFactor = 1</pre>

Schedule	PowerShell code to set up a schedule
Every half hour between 8 A.M. and 4 P.M. on each weekday	<pre> \$jobschedule.FrequencyTypes = [Microsoft. SqlServer.Management.SMO.Agent. FrequencyTypes]::Weekly <# WeekDays.Sunday = 1 WeekDays.Monday = 2 WeekDays.Tuesday = 4 WeekDays.Wednesday = 8 WeekDays.Thursday = 16 WeekDays.Friday = 32 WeekDays.Saturday = 64 WeekDays.WeekDays = 62 WeekDays.WeekEnds = 65 WeekDays.EveryDay = 127 #> #every weekday \$jobschedule.FrequencyInterval = 62 #every half hour \$jobschedule.FrequencySubDayTypes = [Microsoft.SqlServer.Management.SMO.Agent. FrequencySubDayTypes]::Minute \$jobschedule.FrequencySubDayInterval = 30 \$jobschedule.ActiveStartDate = "01/01/2012" #from 8-4 \$starttime = New-Object -TypeName TimeSpan -ArgumentList 8, 0, 0 \$jobschedule.ActiveStartTimeOfDay = \$starttime \$endtime = New-Object -TypeName TimeSpan -ArgumentList 16, 0, 0 \$jobschedule.ActiveEndTimeOfDay = \$endtime </pre>

Schedule	PowerShell code to set up a schedule
At 11:30 P.M. on the last day of every month	<pre>\$jobschedule.FrequencyTypes = [Microsoft. SqlServer.Management.SMO.Agent.FrequencyTypes]:: MonthlyRelative \$jobschedule.FrequencyRelativeIntervals = [Microsoft.SqlServer.Management.SMO.Agent.Freque ncyRelativeIntervals]::Last #month end can fall any day, so we'll have #to set interval to everyday \$jobschedule.FrequencyInterval = [Microsoft. SqlServer.Management.SMO.Agent.MonthlyRelativeWeek Days]::EveryDay \$jobschedule.FrequencyRecurrenceFactor = 1 \$jobschedule.ActiveStartDate = "01/01/2012" #start at 11:30 PM #3 params - hours, mins, days \$starttime = New-Object -TypeName TimeSpan -ArgumentList 23, 30, 0 \$jobschedule.ActiveStartTimeOfDay = \$starttime</pre>

Schedule	PowerShell code to set up a schedule
At noon on every Tuesday and Thursday	<pre>\$jobschedule.FrequencyTypes = [Microsoft.SqlServer.Management.SMO.Agent.FrequencyTypes]::Weekly <# WeekDays.Sunday = 1 WeekDays.Monday = 2 WeekDays.Tuesday = 4 WeekDays.Wednesday = 8 WeekDays.Thursday = 16 WeekDays.Friday = 32 WeekDays.Saturday = 64 WeekDays.WeekDays = 62 WeekDays.WeekEnds = 65 WeekDays.EveryDay = 127 #> #every Tuesday and Thursday #Tuesday = 4, Thursday = 16 so 20 \$jobschedule.FrequencyInterval = 20 \$jobschedule.FrequencyRecurrenceFactor = 1 \$jobschedule.ActiveStartDate = "01/01/2012" #noon #3 params - hours, mins, days \$starttime = New-Object -TypeName TimeSpan -ArgumentList 12, 00, 0 \$jobschedule.ActiveStartTimeOfDay = \$ starttime</pre>

Schedule	PowerShell code to set up a schedule
At 6 A.M. on every 3rd Friday of the month	<pre>\$jobschedule.FrequencyTypes = [Microsoft.SqlServer.Management.SMO.Agent.FrequencyTypes]::MonthlyRelative \$jobschedule.FrequencyRelativeIntervals = [Microsoft.SqlServer.Management.SMO.Agent.FrequencyRelativeIntervals]::Third \$jobschedule.FrequencyInterval = [Microsoft.SqlServer.Management.SMO.Agent.MonthlyRelativeWeekDays]::Friday \$jobschedule.FrequencyRecurrenceFactor = 1 \$jobschedule.ActiveStartDate = "01/01/2012" #start at 10:30 PM \$starttime = New-Object -TypeName TimeSpan -ArgumentList 6, 00, 0 \$jobschedule.ActiveStartTimeOfDay = \$starttime</pre>
At 11 P.M. on every last Thursday of the month	<pre>\$jobschedule.FrequencyTypes = [Microsoft.SqlServer.Management.SMO.Agent.FrequencyTypes]::MonthlyRelative \$jobschedule.FrequencyRelativeIntervals = [Microsoft.SqlServer.Management.SMO.Agent.FrequencyRelativeIntervals]::Last \$jobschedule.FrequencyInterval = [Microsoft.SqlServer.Management.SMO.Agent.MonthlyRelativeWeekDays]::Thursday \$jobschedule.FrequencyRecurrenceFactor = 1 \$jobschedule.ActiveStartDate = "01/01/2012" #11PM \$starttime = New-Object -TypeName TimeSpan -ArgumentList 23, 00, 0 \$jobschedule.ActiveStartTimeOfDay = \$starttime</pre>

Check out `FrequencyTypes` from the following URL:

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

Frequency interval documentation can be found here:

`http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.agent.jobschedule.frequencyinterval(v=sql.110).aspx`

See also

- ▶ The *Listing SQL Server jobs* recipe
- ▶ The *Creating a SQL Server job* recipe
- ▶ The *Running a SQL Server job* recipe

4

Security

In this chapter, we will cover:

- ▶ Listing SQL Server service accounts
- ▶ Changing SQL Server service account
- ▶ Listing authentication modes
- ▶ Changing authentication mode
- ▶ Listing SQL Server log errors
- ▶ Listing failed login attempts
- ▶ Listing logins, users, and database mappings
- ▶ Listing login/user roles and permissions
- ▶ Creating a login
- ▶ Assigning permissions and roles to a login
- ▶ Creating a database user
- ▶ Assigning permissions to a database user
- ▶ Creating a database role
- ▶ Fixing orphaned users
- ▶ Creating a credential
- ▶ Creating a proxy

Introduction

PowerShell can help database administrators and developers to automate security tasks. Whether you need to monitor repeated failed login attempts by parsing out event logs, or manage roles and permissions, especially if the number of users in the system is very large, PowerShell can help you deliver. This chapter will show you the classes and snippets of scripts that will help you manage your SQL Server logins and database users programmatically.

Listing SQL Server service accounts

We will list service accounts in this recipe.

How to do it...

These are the steps to listing SQL Server service accounts:

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the SQLPS module as follows:

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

3. Add the following script and run:

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

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

Name	ServiceAccount	DisplayName
MsDtsServer110	QUERYWORKS\sqlservice	SQL Server Integration Services 11
MSSQL\$SQL01	NT Service\MSSQL\$SQL01	SQL Server (SQL01)
MSSQLFDLauncher	NT Service\MSSQLFDLauncher	SQL Full-text Filter Daemon Launch
MSSQLFDLauncher\$SQL01	NT Service\MSSQLFDLauncher\$SQL01	SQL Full-text Filter Daemon Launch
MSSQLSERVER	QUERYWORKS\sqlservice	SQL Server (MSSQLSERVER)
MSSQLServerOLAPService	QUERYWORKS\sqlservice	SQL Server Analysis Services (MSS
ReportServer	QUERYWORKS\sqlservice	SQL Server Reporting Services (MSS
SQLAgent\$SQL01	NT Service\SQLAgent\$SQL01	SQL Server Agent (SQL01)
SQLBrowser	NT AUTHORITY\LOCALSERVICE	SQL Server Browser
SQLSERVERAGENT	QUERYWORKS\sqlagent	SQL Server Agent (MSSQLSERVER)

How it works...

A service account is an account created for the exclusive purpose of running a service. To list service accounts, we can use the `Wmi.ManagedComputer` object:

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

The `managedComputer` instance has a property called `ServiceAccount`, which is what we want to list:

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

We can also alternatively use the `Get-WmiObject` cmdlet to list the service accounts.

To use `Get-WmiObject`, we must first identify the hostname and the SQL Server namespace:

```
$hostname = "KERRIGAN"

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

For SQL Server 2012, this value is `ROOT\Microsoft\SQLServer\ComputerManagement11`

We can then use `Get-WmiObject` to list all the SQL Server services and service accounts. The service account is stored in the property `StartName`:

```
Get-WmiObject -ComputerName $hostname `
-NameSpace "$($namespace.__NAMESPACE)\$($namespace.Name)" `
-Class SqlService |
Select ServiceName,
       DisplayName,
       @{N="ServiceAccount";E={$_.StartName}} |
Format-Table -AutoSize
```

See also

- ▶ [The Changing SQL Server service account recipe](#)
- ▶ [The Listing SQL Server instances recipe in Chapter 2, SQL Server and PowerShell Basic Tasks](#)

Changing SQL Server service account

We will see how to change SQL Server accounts in this recipe.

Getting ready

To perform this recipe, you will need to create another Windows/Domain account that you can use to change the service account to.

In this recipe, we will change the service account for `SQLSERVERAGENT` from `QUERYWORKS\sqlagent` to `QUERYWORKS\sqlagent01`. Feel free to substitute these with accounts that already exist in your system.

How to do it...

Let's explore the code required to change a SQL Server service account:

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

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

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

3. Add the following script and run:

```
#get handle to service
#note we are using V3 simplified Where-Object syntax
$servicename = "SQLSERVERAGENT"
$sqlservice = $managedComputer.Services |
Where-Object Name -eq $servicename

#=====
#Option 1: change account using bare text
#=====
#might be ok as long as no one is looking over
#your shoulder, especially if you need to
```

```
#set password for many servers
$username = "QUERYWORKS\sqlagent01"
$password = "P@ssword"
$sqlservice.SetServiceAccount($username, $password)

#sleep to wait for account change to finish
Start-Sleep -s 1
#display new service account
$sqlservice.ServiceAccount

#=====
#Option 2: change account using GetNetworkCredentials
#=====
$username = "QUERYWORKS\sqlagent01"
$credential = Get-Credential -credential $username

#problem here: SetServiceAccount accepts two strings
#Get-Credential provides the password as securestring
#by default if you pass this to SetServiceAccount,
#you will get an error to pass, you need to use $credential.
GetNetworkCredential().password to

#get text equivalent
$sqlservice.SetServiceAccount($credential.UserName, $credential.
GetNetworkCredential().Password)

#sleep to wait for account change to finish
Start-Sleep -s 1
#display new service account
$sqlservice.ServiceAccount
```


4. Confirm that the service account has changed:

```
#list services
$managedComputer.Services |
Where Name -eq $servicename |
Select Name, ServiceAccount, DisplayName, ServiceState |
Format-Table -AutoSize
```

How it works...

To change the service account, the first step is to get a handle to the service that you want to change. In this recipe, we get a handle to `SQLSERVERAGENT`:

```
#get handle to service
$servername = "SQLSERVERAGENT"
$sqlservice = $wmiinstance.Services |
Where Name -eq $servername
```

 If you are using PowerShell V2, you will have to change the `Where`, or `Where-Object`, cmdlet usage to use the curly braces `{}` and the `$_` variable:
`Where { $_.Name -eq $serviceName }`

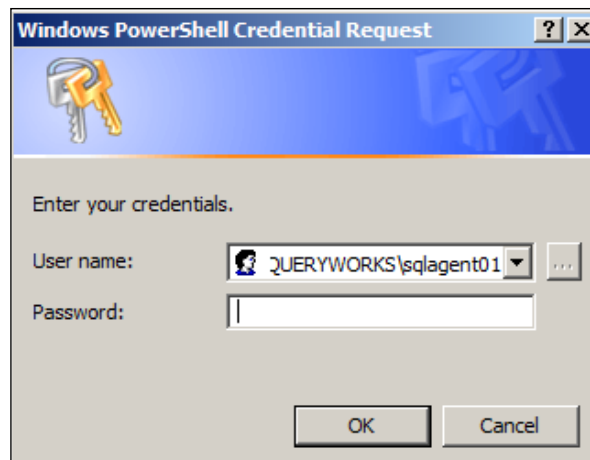
In this recipe, we looked at two alternatives. The first alternative is using a variable for the username, and another one to store bare, clear text password, which we pass to the `SetServiceAccount` method of the `Microsoft.SqlServer.Management.Smo.Wmi.ManagedComputer` class.

```
#####
#Option 1: change account using bare text
#####
#might be ok as long as no one is looking over your shoulder, esp if
you need to set password for many servers
$username = "QUERYWORKS\sqlagent01"
$password = "P@ssword"
$sqlservice.SetServiceAccount($username, $password)
```

This is not the ideal way to pass a password. Therefore, in the second alternative, we do pretty much the same steps, but replace the password variable assignment with the `Get-Credential` cmdlet:

```
$username = "QUERYWORKS\sqlagent01"
$credential = Get-Credential -credential $username
```

With the `Get-Credential` cmdlet, you will be prompted for the password, and the password will be stored as a `SecureString`. A `SecureString` is text that is encrypted using the Windows Data Protection API (<http://msdn.microsoft.com/en-us/library/ms995355.aspx>).



It's good news because now our password is secure, isn't it? There's a caveat though. The `SetServiceAccount` method accepts a string password, not a `SecureString` password. This means that to set the new service account's password, we need to convert the password back to a readable string that `SetServiceAccount` method can accept:

```
$sqlservice.SetServiceAccount($credential.UserName, $credential.  
GetNetworkCredential().Password)
```

This is still a better approach than the first alternative. However, you need to take care that nobody gets a handle to this script before you end your session. Otherwise, they will still see the password in clear text if they invoke the following command:

```
$credential.GetNetworkCredential().Password
```

See also

- ▶ The *Listing SQL Server service accounts* recipe
- ▶ Check out these MSDN articles related to service accounts:
 - Service Accounts Step-by-Step Guide ([http://msdn.microsoft.com/en-us/library/dd548356\(WS.10\).aspx](http://msdn.microsoft.com/en-us/library/dd548356(WS.10).aspx))
 - Create Windows PowerShell Scripts that Accept Credentials (<http://msdn.microsoft.com/en-us/magazine/ff714574.aspx>)

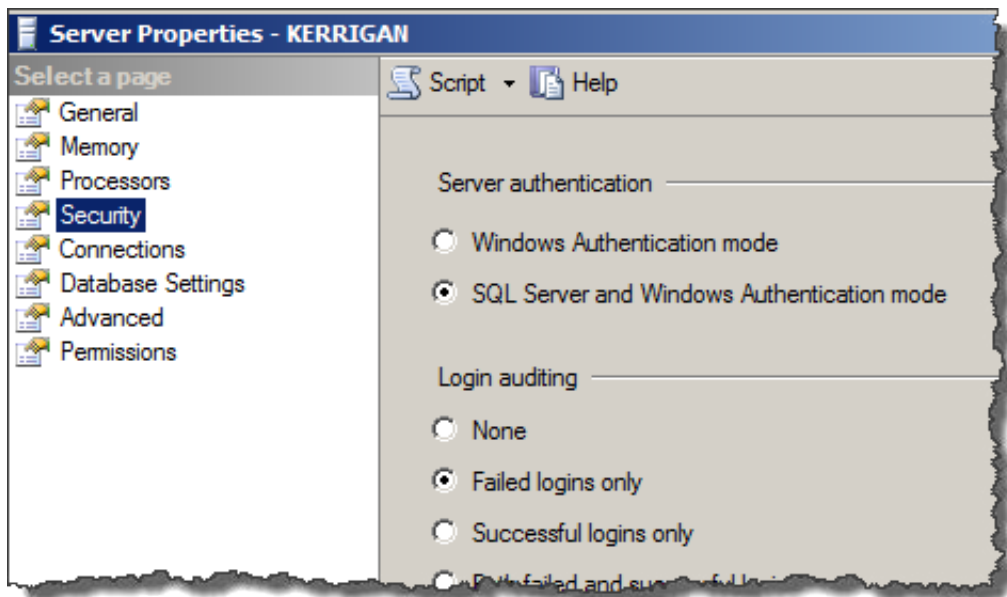
Listing authentication modes

In this recipe, we will list authentication modes using PowerShell and SMO.

Getting ready

Confirm which authentication mode your instance is running.

Go to SQL Server Management Studio, and log in to your instance. Once logged in, right-click on the instance and go to **Properties**, and then to **Security**:



How to do it...

Let's list the steps required to display your instance's current authentication mode:

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 as follows:

```
#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:

```
#display login mode
$server.settings.LoginMode
```

How it works...

This is a very short recipe. To display the login mode, you need to have a handle to the instance first. Once the server handle is established, you need to access the server object's `Settings.LoginMode` property:

```
#display login mode
$server.settings.LoginMode
```



Authentication modes are discussed in more detail in the *Changing authentication mode* recipe.

See also

- ▶ The *Changing authentication mode* recipe

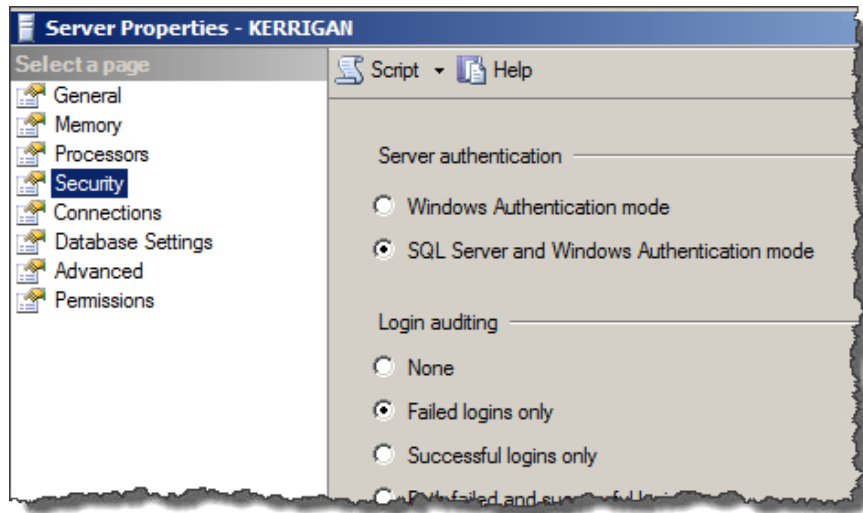
Changing authentication mode

In this recipe, we will change the SQL Server authentication mode.

Getting ready

Confirm which authentication mode your instance is running.

Go to SQL Server Management Studio, and log in to your instance. Once logged in, right-click on the instance and go to **Properties**, and to **Security**, similar to what we did in the previous recipe:



In this recipe, we will change the authentication mode from **Mixed** to **Integrated**.

How to do it...

Let's explore the steps required to complete the task:

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 as follows:

```
#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:

```
#according to MSDN, there are four (4) possible
#values for LoginMode:
#Normal, Integrated, Mixed and Unknown
#let's change ours to Integrated
```

```

$server.settings.LoginMode = [Microsoft.SqlServer.Management.Smo.
ServerLoginMode]::Integrated
$server.Alter()
$server.Refresh()

#display login mode
$server.settings.LoginMode

```

How it works...

To change the authentication mode, you first need to get a handle to the server instance. Once you have the handle, you can assign a valid `LoginMode` enumeration value to the `LoginMode` property:

```

$server.settings.LoginMode = [Microsoft.SqlServer.Management.Smo.
ServerLoginMode]::Integrated

```

There are four possible values: `Normal`, `Integrated`, `Mixed`, and `Unknown`. Once the new authentication mode is assigned, you can invoke the `Alter` method of the SMO server object. Optionally, you can also call the `Refresh` method if you want to display the new value right away:

```

$server.Alter()
$server.Refresh()

```

Note however, that while the GUI may reflect the change in authentication mode, the actual change will not take effect until the SQL Server service is restarted.

There's more...

Authentication mode in SQL Server identifies how login accounts can connect to an instance. There are two well-known modes: `Mixed` and `Integrated`.

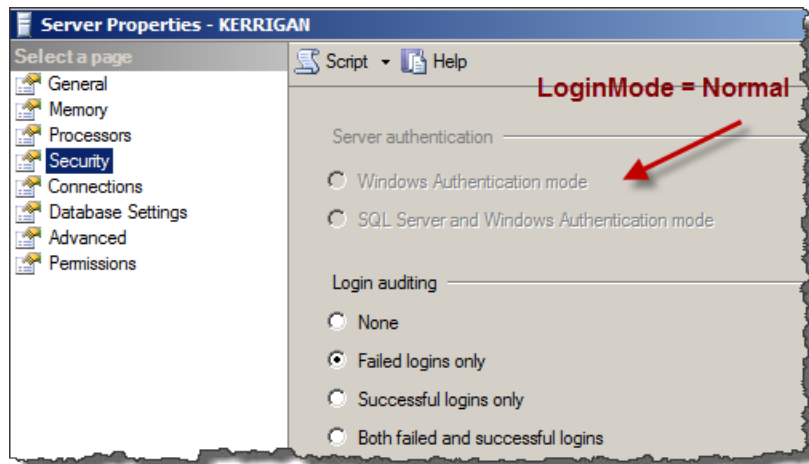
However, if you check out the valid enumeration values for `LoginMode` on MSDN, there are four:

LoginMode	Description
Normal	SQL Authentication only
Integrated	Windows Authentication only
Mixed	SQL and Windows Authentication
Unknown	Unknown

It is interesting to note that the two lesser-known modes are not accessible using SQL Server Management Studio. If you do try to set these values using PowerShell and SMO, it will disable the **Authentication Mode** in **Management Studio**:

```
$server.settings.LoginMode = [Microsoft.SqlServer.Management.Smo.  
ServerLoginMode]::Normal  
$server.Alter()  
$server.Refresh()
```

In Management Studio, this is what you will see.



For our example, we only need to be concerned with Mixed and Integrated. Normal and Unknown are legacy values, and should not be used in today's production environments.

Check out the MSDN article on different `ServerLoginMode` enumeration values:

<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.serverloginmode.aspx>

More on legacy LoginMode values

Tibor Karaszi wrote a blog post called *Watch out for Old Stuff* that explains the four `ServerLoginMode` values and where we might encounter them:

http://sqlblog.com/blogs/tibor_karaszi/archive/2010/09/15/watch-out-for-old-stuff.aspx

See also

- ▶ The *Listing authentication mode* recipe

Listing SQL Server log errors

In this recipe, we will list SQL Server log errors.

Getting ready

Check your SQL Server log in Management Studio. This should be what our PowerShell script should report.

How to do it...

Let's check how we can list SQL Server errors 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 as follows:

```
#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:

```
#According to MSDN:
#ReadErrorLog: returns A StringCollection system object
#value that contains an enumerated list of errors from
#the SQL Server error log.

#Note we are using PowerShell V3 because of simplified
#Where-Object syntax
[datetime]$date = "2011-11-01"

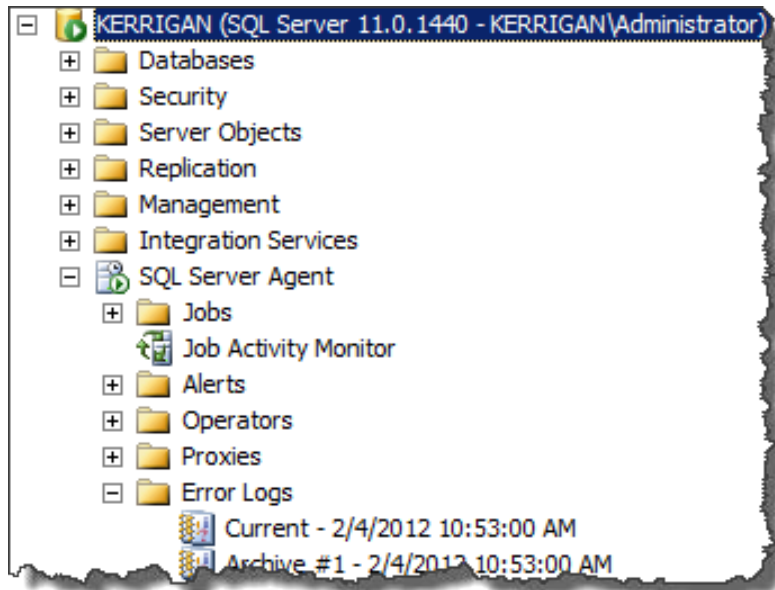
$server.ReadErrorLog() |
Where-Object Text -Like "*failed*" |
Where-Object LogDate -ge $date |
Format-Table -AutoSize
```

Your result should look similar to the following screenshot:

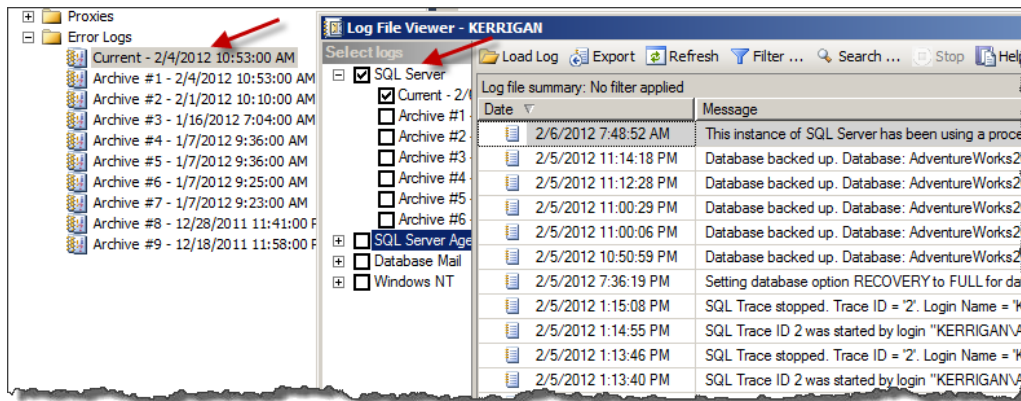
LogDate	ProcessInfo	Text
2/1/2012 10:12:18 AM	Server	Logging SQL Server messages in file 'C:\Pr
2/1/2012 10:12:18 AM	Server	Registry startup parameters: ...
2/1/2012 10:25:16 AM	Logon	Error: 18456, Severity: 14, State: 38.
2/1/2012 10:25:23 AM	Logon	Error: 18456, Severity: 14, State: 38.
2/1/2012 10:25:31 AM	Logon	Error: 18456, Severity: 14, State: 38.
2/1/2012 10:25:38 AM	spid23s	CHECKDB for database 'AdventureWorks2008R2
2/4/2012 2:17:40 PM	Logon	Error: 18456, Severity: 14, State: 8.
2/4/2012 2:17:48 PM	Logon	Error: 18456, Severity: 14, State: 8.

4. Confirm that these entries exist in the **Error Logs** via Management Studio.

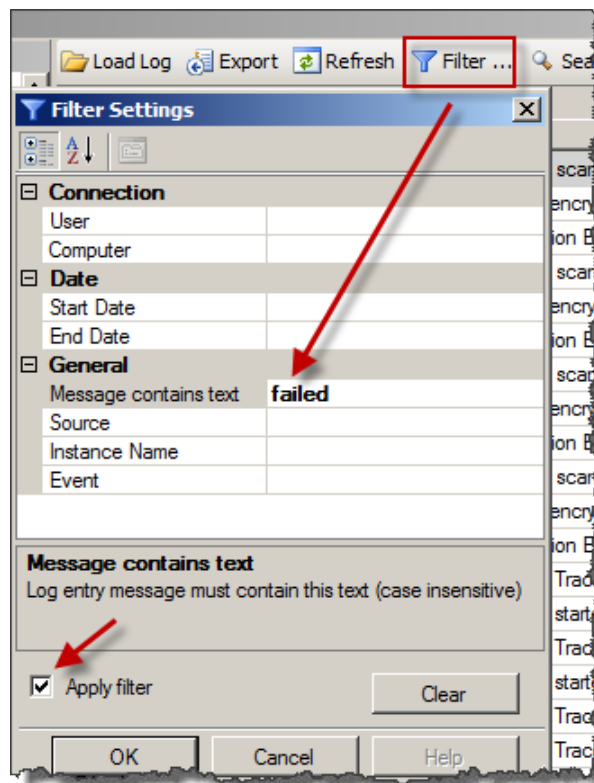
Open SQL Server Management Studio, and connect to your instance. Expand **SQL Server Agent | Error Logs**:



5. Double-click on the **Current** error log. By default, this opens to **SQL Server Agent** logs. Change the selected log to be **SQL Server**.



6. To filter, click on the **Filter...** icon. Add the string `failed` in the **Message contains text** field, and check the **Apply filter** checkbox:



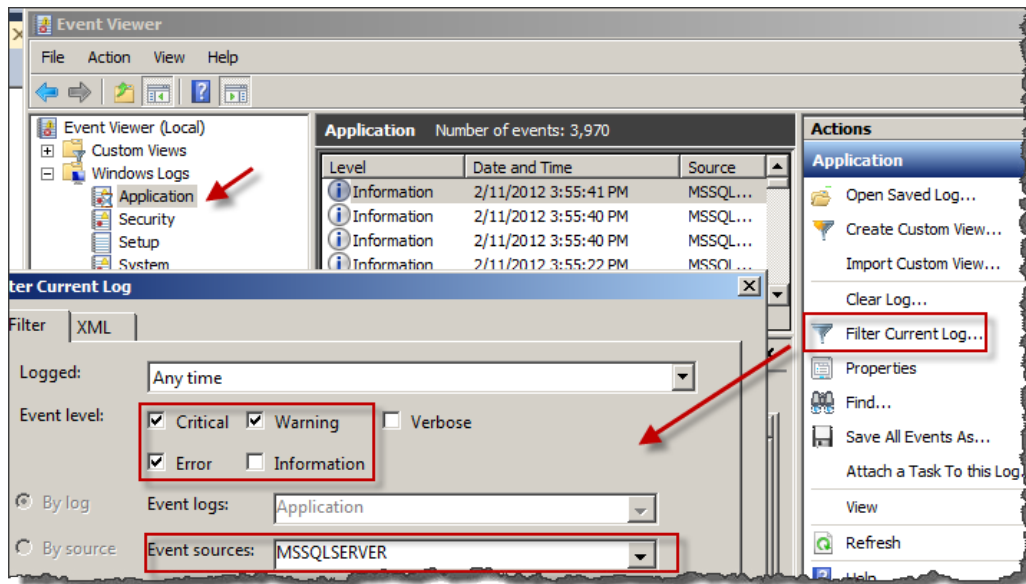
7. Click on **OK** to see the filtered log events.

8. If you want to get generic errors from the **Event log**, add the following script and run:

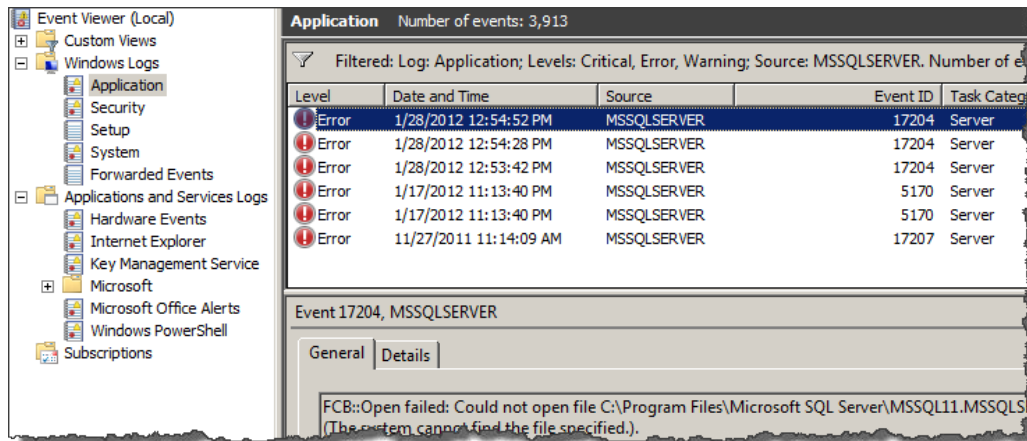
```
#if you want to get all the generic errors from the Event Log
#you can use this
Get-EventLog Application -Source "MSSQLSERVER" -EntryType Error
```

Index	Time	EntryType	Source	InstanceID	Message
10779	Mar 30 23:59	Error	MSSQLSERVER	3221239623	Replication-Replication Distribution Subsystem
10778	Mar 30 23:59	Error	MSSQLSERVER	3221239623	Replication-Replication Distribution Subsystem
10777	Mar 30 23:59	Error	MSSQLSERVER	3221239623	Replication-Replication Transaction-Log Read
10776	Mar 30 23:59	Error	MSSQLSERVER	3221239623	Replication-Replication Transaction-Log Read
10775	Mar 30 23:58	Error	MSSQLSERVER	3221239623	Replication-Replication Distribution Subsystem
10774	Mar 30 23:58	Error	MSSQLSERVER	3221239623	Replication-Replication Distribution Subsystem
10773	Mar 30 23:58	Error	MSSQLSERVER	3221239623	Replication-Replication Transaction-Log Read
10772	Mar 30 23:58	Error	MSSQLSERVER	3221239623	Replication-Replication Transaction-Log Read
10771	Mar 30 23:57	Error	MSSQLSERVER	3221239623	Replication-Replication Distribution Subsystem
10770	Mar 30 23:57	Error	MSSQLSERVER	3221239623	Replication-Replication Transaction-Log Read

To check this visually, you can go to **Administrative Tools | Event Viewer**. Go to **Application**, and **Filter Current Log**. Check **Error**, **Critical**, and **Warning** under **Event level**, and under **Event sources** choose **MSSQLSERVER**.



What you should see after you filter are only the errors pertaining to the default instance **MSSQLSERVER**:

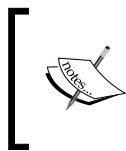


How it works...

SMO provides a way to easily retrieve and display SQL Server-related errors. This is through the `ReadErrorLog` method of the SMO server object. The `ReadErrorLog` method retrieves a list of errors from the SQL Server error log. In our recipe, we filtered only the log entries that contained the word `failed`, and only those ones that happened after November 01, 2011. Note that we are using the simplified PowerShell V3 syntax for the `Where-Object` cmdlet:

```
[datetime]$date = "2011-11-01"

$server.ReadErrorLog() |
Where-Object Text -Like "*failed*" |
Where-Object LogDate -ge $date |
Format-Table -AutoSize
```



Note that to use V2 syntax, you will need to change the `Where-Object` line to:

```
Where-Object {$_.Text -Like "*failed*"
-and $_.LogDate -ge $date}
```

You can read more about the `ReadErrorLog` method from MSDN:

<http://msdn.microsoft.com/en-us/library/ms210384.aspx>

Instead of using the `ReadErrorLog` method, an alternative is to use the `Get-EventLog` cmdlet and filter by source and keyword:

```
Get-EventLog Application -Source "MSSQLSERVER" -Message "*failed*"
```

The `Get-EventLog` cmdlet supports a number of switches that allow you to further filter and sort results. If you want to display strictly `Error` entry types, you can use:

```
Get-EventLog Application -Source "MSSQLSERVER" -EntryType Error
```

Type the following to get more information about the `Get-EventLog` syntax and usage:

```
Get-Help Get-EventLog
```

See also

- ▶ The *Listing failed login attempts* recipe

Listing failed login attempts

This recipe lists failed login attempts in your SQL Server instance.

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 as follows:

```
#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:

```
#According to MSDN:
#ReadErrorLog returns A StringCollection system object
#value that contains an enumerated list of errors
#from the SQL Server error log.

$server.ReadErrorLog() |
Where-Object ProcessInfo -Like "*Logon*" |
```

```
Where-Object Text -Like "*Login failed*" |
Format-List
```

```
LogDate      : 2/1/2012 10:25:16 AM
ProcessInfo  : Logon
Text         : Login failed for user 'QUERYWORKS\sqlservice'. Reason: Failed to open the explicitl

LogDate      : 2/1/2012 10:25:23 AM
ProcessInfo  : Logon
Text         : Login failed for user 'QUERYWORKS\sqlservice'. Reason: Failed to open the explicitl

LogDate      : 2/1/2012 10:25:31 AM
ProcessInfo  : Logon
Text         : Login failed for user 'QUERYWORKS\sqlservice'. Reason: Failed to open the explicitl

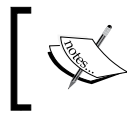
LogDate      : 2/4/2012 2:17:40 PM
ProcessInfo  : Logon
Text         : Login failed for user 'dean'. Reason: Password did not match that for the login pro

LogDate      : 2/4/2012 2:17:48 PM
ProcessInfo  : Logon
Text         : Login failed for user 'eric'. Reason: Password did not match that for the login pro
```

How it works...

One way to get failed login attempts is by using the method `ReadErrorLog` of the SMO Server object and filtering by `ProcessInfo` and `Text` properties. The `ProcessInfo` value we are targeting is `Logon`, and we want to display any login activities that have failed. We are using the simplified PowerShell V3 syntax for `Where-Object` in this code block:

```
$server.ReadErrorLog() |
Where-Object ProcessInfo -Like "*Logon*" |
Where-Object Text -Like "*Login failed*" |
Format-List
```



To use V2 syntax, you will need to change the `Where-Object` line to:

```
Where-Object {$_ .ProcessInfo -Like "*Logon*" -and
$_ .Text -Like "*Login failed*"}
```

See also

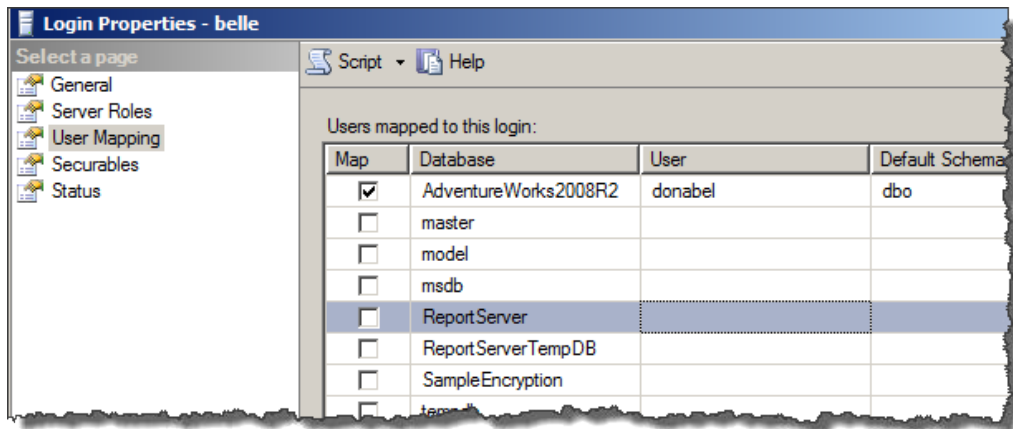
- ▶ The *Listing SQL Server log errors* recipe

Listing logins, users, and database mappings

This recipe lists logins and their corresponding usernames and database mappings.

Getting ready

To check the logins and their database mappings in SQL Server Management Studio, log in to SSMS. Go to the *Security* folder, expand *Logins*, and double-click on a particular login. This will show you the **Login Properties** window. Click on the **User Mapping** option on the left pane, as shown in the following screenshot:



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 as follows:

```
#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:

```
#display login info
#these are two different ways of displaying login info
$server.Logins
$server.EnumWindowsUserInfo()

#List users, and database mappings
$server.Databases |
ForEach-Object {
    #capture database object
    $database = $_

    #capture users in this database
    $users = $_.Users

    $users |
    Where-Object { -not($_.IsSystemObject)} |
    Select @{N="Login";E={$_.Login}},
    @{N="User";E={$_.Name}},
    @{N="DatabaseName";E={$database.Name}},
    @{N="LoginType";E={$_.LoginType}},
    @{N="UserType";E={$_.UserType}}
} |
Format-Table -AutoSize
```

This should give a result similar to the following:

Login	User	DatabaseName	LoginType	UserType
-----	----	-----	-----	-----
belle	donabel	AdventureWorks2008R2	SqlLogin	SqlLogin
eric	eric	AdventureWorks2008R2	SqlLogin	SqlLogin
QUERYWORKS\aterra	QUERYWORKS\aterra	AdventureWorks2008R2	WindowsUser	SqlLogin
QUERYWORKS\jraynor	QUERYWORKS\jraynor	AdventureWorks2008R2	WindowsUser	SqlLogin
##MS_AgentSigningCertificate##	##MS_AgentSigningCertificate##	master	Certificate	Certificate
##MS_PolicyEventProcessingLogin##	##MS_PolicyEventProcessingLogin##	master	SqlLogin	SqlLogin
QUERYWORKS\sqlservice	QUERYWORKS\sqlservice	master	WindowsUser	SqlLogin
##MS_PolicyEventProcessingLogin##	##MS_PolicyEventProcessingLogin##	msdb	SqlLogin	SqlLogin
##MS_PolicyTsqlExecutionLogin##	##MS_PolicyTsqlExecutionLogin##	msdb	SqlLogin	SqlLogin
MS_DataCollectorInternalUser	MS_DataCollectorInternalUser	msdb	SqlLogin	NoLogin
QUERYWORKS\sqlservice	QUERYWORKS\sqlservice	msdb	WindowsUser	SqlLogin
QUERYWORKS\sqlservice	QUERYWORKS\sqlservice	ReportServer	WindowsUser	SqlLogin

How it works...

To just display the logins, you can use the server object and the `Logins` property.

```
$server.Logins
```

An alternative way, if you are only interested in a Windows account, is using the `EnumWindowsUserInfo` method of the SMO server class, which returns the Windows users who have been explicitly given SQL Server access:

```
$server.EnumWindowsUserInfo()
```

To display only database users, you can get a handle to a specific database and use the `Users` property of that database's handle.

The most straightforward way of getting all the mappings is by looping through all the databases, and getting a handle to all the users in that database. Once there is a handle to the database object's `Users`, you can display properties such as `Login`, `User`, `LoginType`, and `UserType`. Note that we create a custom table so we can display the results with a more meaningful format and headers. To do this, we provide formatting instructions to our `Select` cmdlet; `N` refers to the Name of the property, and `E` refers to the Expression that will derive the value:

```
$server.Databases |  
ForEach-Object {  
    #capture database object  
    $database = $_  
  
    #capture users in this database  
    $users = $_.Users  
  
    $users |  
    Where-Object { -not($_.IsSystemObject)} |  
    Select @{N="Login";E={$_.Login}},  
    @{N="User";E={$_.Name}},  
    @{N="DatabaseName";E={$database.Name}},  
    @{N="LoginType";E={$_.LoginType}},  
    @{N="UserType";E={$_.UserType}}  
} |  
Format-Table -AutoSize
```

There's more...

Logins and users are two terms that are often interchanged, but shouldn't be. A login is a server principal that is used for authenticating who can connect and who will have access, on the instance level.

SQL Server supports two types of logins—Windows Login and SQL Login. A **Windows Login** is a Windows-level principal, which means that this is seen and shared with the Windows OS or domain. A **SQL Login** is a SQL Server principal or a login known only to SQL Server.

A user, on the other hand, is a database principal. This means that it is a database-level object and not a server-level object. A user is often mapped to a valid login using the login's Security ID (SID). There are cases when the user isn't mapped; this is when the user is orphaned. This can happen when the database has been moved or restored to a different instance that does not contain the original login. This can also happen when a login has been removed from the instance, and the related database users have not been cleaned up or reassigned.

See also

- ▶ *The Listing login/user roles and permissions recipe*

Listing login/user roles and permissions

This recipe shows how you can list a login- and user-related roles and permissions.

How to do it...

Let's check the code needed to list the login/user roles and permissions.

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 as follows:

```
#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:

```
$server.Databases |
ForEach-Object {
    #capture database object
    $database = $_

    #capture users in this database
    $users = $_.Users
```




```

    $users |
    Where-Object { -not($_.IsSystemObject)} |
    Select @{N="Login";E={$_.Login}},
           @{N="User";E={$_.Name}},
           @{N="DatabaseName";E={$_.DatabaseName}},
           @{N="DBRoles";E={$_.EnumRoles()}},
           @{N="ObjectPermissions";
              E={$_.Database.EnumObjectPermissions($_.Name)}}
    } |
    Format-Table -AutoSize

```

You should see a display similar to the following screenshot:



```

Login          : QUERYWORKS\sqlservice
User           : QUERYWORKS\sqlservice
DatabaseName   : ReportServer
DBRoles        : RSExecRole db_owner
ObjectPermissions :
|
Login          : QUERYWORKS\sqlservice
User           : QUERYWORKS\sqlservice
DatabaseName   : ReportServerTempDB
DBRoles        : RSExecRole db_owner
ObjectPermissions :
|
Login          : EncryptionLogin
User           : EncryptionUser
DatabaseName   : SampleEncryption
DBRoles        :
ObjectPermissions : [CustomerDetails] ObjectOrColumn: EncryptionUser, Grant, INSERT [Cus

```

How it works...

A database mapping determines which logins are related to which database users. Remember that a database user is a database-level principal that is mapped to a login via a Security ID (SID).

To display the database mappings, we will need to loop through all the databases and display the mappings using each individual `User` class' objects. In this recipe, we ignored all system objects (such as `sys`, `guest`, `information_schema`). For each user, we also displayed their respective database roles using the `EnumRoles` method of the `User` class, and their respective database-level permissions using `EnumObjectPermissions` of the database class:

```

$server.Databases |
ForEach-Object {
    #capture database object
    $database = $_

    #capture users in this database
    $users = $_.Users

```

```

$users |
Where-Object { -not($_.IsSystemObject)} |
Select @{N="Login";E={$_.Login}},
       @{N="User";E={$_.Name}},
       @{N="DatabaseName";E={$databaseName}},
       @{N="DBRoles";E={$_.EnumRoles()}},
       @{N="ObjectPermissions";
         E={$database.EnumObjectPermissions($_.Name)}}
} |
Format-Table -AutoSize

```

See also

- ▶ *The Listing logins, users, and database mappings recipe*

Creating a login

This recipe shows how you can create a login using PowerShell and SMO.

Getting ready

For this recipe, we will create a SQL login called `eric`. The T-SQL equivalent of what we are trying to accomplish is:

```

CREATE LOGIN [eric]
WITH PASSWORD=N'YourSuperStrongPassword',
CHECK_EXPIRATION=OFF
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 as follows:

```

#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:

```
$loginName = "eric"

# drop login if it exists
if ($server.Logins.Contains($loginName))
{
    $server.Logins[$loginName].Drop()
}

$login = New-Object `
-TypeName Microsoft.SqlServer.Management.Smo.Login `
-ArgumentList $server, $loginName
$login.LoginType = [Microsoft.SqlServer.Management.Smo.
LoginType]::SqlLogin
$login.PasswordExpirationEnabled = $false

# prompt for password
$pw = Read-Host "PW" -AsSecureString
$login.Create($pw)
```

How it works...

The first thing we need to do, after getting an SMO `server` object handle, is create an SMO Login object:

```
$login = New-Object `
-TypeName Microsoft.SqlServer.Management.Smo.Login `
-ArgumentList $server, $loginName
```

The next step is to identify what type of login this is. The possible `LoginTypes` are `AsymmetricKey`, `Certificate`, `SQLLogin`, `WindowsGroup`, and `WindowsUser`. In our recipe, we are using a `SQLLogin`:

```
$login.LoginType = [Microsoft.SqlServer.Management.Smo.
LoginType]::SqlLogin
```

The login object also has a few settable properties, such as `PasswordPolicyEnforced` and `PasswordExpirationEnabled`.

```
$login.PasswordExpirationEnabled = $false
```

When ready, you can invoke the `Create` method of the `Login` class. Note that the `Create` method has a few overloads, some of which allow you to pass `LoginCreateOptions`. In our recipe, we are only passing in a password, which we collect using a `Read-Host` cmdlet. We prompt the user for the password instead of hardcoding it with our script ourselves:

```
$pw = Read-Host "PW" -AsSecureString
$login.Create($pw)
```

See also

- ▶ The *Assigning permissions and roles to a login* recipe
- ▶ The *Creating a database user* recipe

Assigning permissions and roles to a login

This recipe shows you how to assign permissions and roles to a login by using PowerShell and SMO.

Getting ready

If you haven't already done so in the *Creating a login* recipe, create a SQL login name `eric`. We will be assigning the `dbcreator` and `setupadmin` server role to this login, as well as granting `ALTER` permissions to any setting or database. The T-SQL equivalent of what we are trying to accomplish is:

```
ALTER SERVER ROLE [dbcreator]
ADD MEMBER [eric]
GO
ALTER SERVER ROLE [setupadmin]
ADD MEMBER [eric]
GO
GRANT
    ALTER ANY DATABASE,
    ALTER SETTINGS
TO [eric]
```

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 as follows:

```
#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:

```
#assumption is this login already exists
$loginName = "eric"

#assign server level roles
$login = $server.Logins[$loginName]
$login.AddToRole("dbcreator")
$login.AddToRole("setupadmin")
$login.Alter()

#grant server level permissions
$permissionset = New-Object Microsoft.SqlServer.Management.Smo.
ServerPermissionSet ( [Microsoft.SqlServer.Management.Smo.ServerPermi
ssion]::AlterAnyDatabase)
$permissionset.Add([Microsoft.SqlServer.Management.Smo.ServerPermi
ssion]::AlterSettings)
$server.Grant($permissionset, $loginName)

#confirm server roles
$login.ListMembers()

#confirm permissions
$server.EnumServerPermissions($loginName) |
Select Grantee, PermissionType, PermissionState |
Format-Table -AutoSize
```

You should get a result similar to the following:

Grantee	PermissionType	PermissionState
eric	ALTER ANY DATABASE	Grant
eric	ALTER SETTINGS	Grant
eric	CONNECT SQL	Grant

How it works...

After we create an SMO server object, we create a handle to the SMO login object that we want to query:

```
$loginName = "eric"
```

```
#assign server level roles
$login = $server.Logins[$loginName]
```

The login object has an `AddToRole` method, which we can use to add the login as a member of fixed server roles:

```
$login.AddToRole("dbcreator")
$login.AddToRole("setupadmin")
```

When we're ready to send this command to SQL Server, we issue the `Alter` method of the login object.

```
$login.Alter()
```

Now we also have the option to assign specific permissions outside of the role, for the login. This requires creating a `ServerPermissionSet` object. The following code creates the permission set and adds the permission `AlterAnyDatabase` to the list of permissions that we will be assigning:

```
$permissionset = New-Object Microsoft.SqlServer.Management.Smo.
ServerPermissionSet ([Microsoft.SqlServer.Management.Smo.
ServerPermission]::AlterAnyDatabase)
```

This permission set can accommodate multiple server-level permissions. In our recipe, we add another permission—`AlterSettings`—by issuing this command:

```
$permissionset.Add ([Microsoft.SqlServer.Management.Smo.
ServerPermission]::AlterSettings)
```

To finalize the process, we issue the grant statement on the object with the parameters being the permission set that we have created, and the login.

```
$server.Grant($permissionset, $loginName)
```

See also

- ▶ The *Creating a login* recipe
- ▶ The *Creating a database user* recipe
- ▶ Read more about the `ServerPermissionSet` class from MSDN:
[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.serverpermissionset\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.serverpermissionset(v=sql.110).aspx)
- ▶ Check out all the `ServerPermission` properties from:
<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.serverpermission.aspx>

Creating a database user

This recipe demonstrates how to create a database user by using PowerShell and SMO.

Getting ready

If you haven't already done so in the *Creating a login* recipe, create a SQL login called `eric`.

In our recipe, we will use a login called `eric`, which we will map to a user called `eric` in the `AdventureWorks2008R2` database. The T-SQL equivalent of what we are trying to accomplish is:

```
USE [AdventureWorks2008R2]
GO
```

```
CREATE USER [eric]
FOR LOGIN [eric]
```

How to do it...

Here are the steps for creating a database user:

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 as follows:

```
#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:

```
$loginName = "eric"

#get login
$login = $server.Logins[$loginName]

#add a database mapping
$databasename = "AdventureWorks2008R2"
$database = $server.Databases[$databasename]
```

```

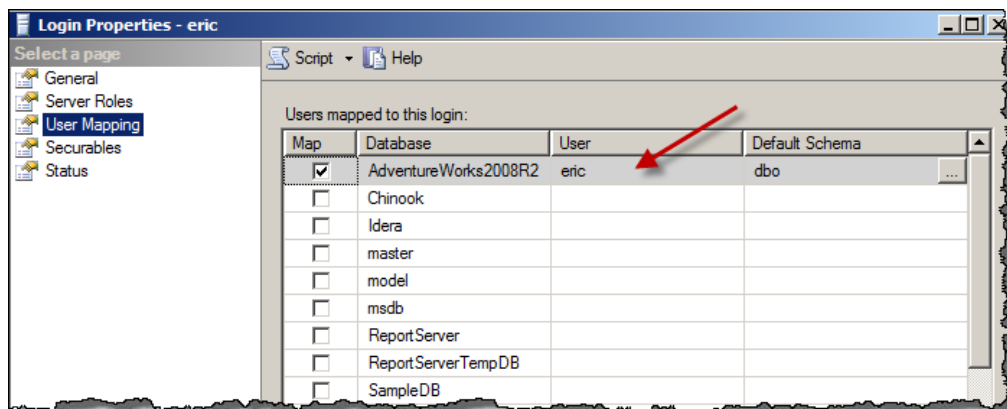
if ($database.Users [$dbUserName])
{
    $database.Users [$dbUserName].Drop()
}

$dbUserName = "eric"
$dbuser = New-Object `
-TypeName Microsoft.SqlServer.Management.Smo.User `
-ArgumentList $database, $dbUserName

$dbuser.Login = $loginName
$dbuser.Create()

```

4. To confirm that the user has been created:
 1. Open SQL Server Management Studio.
 2. Expand **Security** and expand **Logins**.
 3. Double-click the login called **eric**.
 4. Highlight **User Mapping** from the left-hand pane. You should see that **eric** has been mapped to the **AdventureWorks2008R2** database:



How it works...

After creating the SMO server object, create a handle to the login you wish to use:

```

$loginName = "eric"

#get login
$login = $server.Logins[$loginName]

```


Next, you need to get a handle to the database that you want this login to have a corresponding user to. In our case, we will be using AdventureWorks2008R2:

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

To create a database user, we need to instantiate a `Microsoft.SqlServer.Management.Smo.User` object, and pass the database and database username as arguments:

```
$dbUserName = "eric"
$dbuser = New-Object `
-TypeName Microsoft.SqlServer.Management.Smo.User `
-ArgumentList $database, $dbUserName

$dbuser.Login = $loginName
```

The final step is to issue the `Create` method on the `$dbuser` object.

```
$dbuser.Create()
```

See also

- ▶ The *Creating a login* recipe
- ▶ The *Assigning permissions and roles to a database user* recipe

Assigning permissions to a database user

This recipe shows how to assign permissions to a database user via SMO and PowerShell.

Getting ready

In this recipe, we will use the AdventureWorks2008R2 database user `eric` that we created in the previous recipes. We will grant this user `ALTER` and `CREATE TABLE` permissions. The T-SQL equivalent of what we are trying to accomplish is:

```
USE [AdventureWorks2008R2]
GO
GRANT
    ALTER,
    CREATE TABLE
TO [eric]
```

You can substitute this with any database user you have in your database.

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 as follows:

```
#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:

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

#get a handle to the database user we want
#to assign permissions to
$dbusername = "eric"
$dbuser = $database.Users[$dbusername]

#assign database permissions
$permissionset = New-Object Microsoft.SqlServer.Management.
Smo.DatabasePermissionSet ([Microsoft.SqlServer.Management.Smo.
DatabasePermission]::Alter)
$permissionset.Add ([Microsoft.SqlServer.Management.Smo.DatabasePer
mission]::CreateTable)

#grant the permissions
$database.Grant ($permissionset, $dbuser.Name)

#confirm permissions
$database.Users |
ForEach-Object {
    $database.EnumDatabasePermissions($_.Name) |
    Select PermissionState, PermissionType, Grantee
} |
Format-Table -AutoSize
```

When the script has been successfully executed, you should see a screen similar to the following:

PermissionState	PermissionType	Grantee
Grant	CONNECT	belle
Grant	CONNECT	dbo
Grant	CONNECT	donabel
Grant	ALTER	eric
Grant	CONNECT	eric
Grant	CREATE TABLE	eric

How it works...

To add specific permissions to a database user, you must first get a handle to the database user.

```
$dbusername = "eric"
$dbuser = $database.Users[$dbusername]
```

The next step is to define a `DatabasePermissionSet` object. This object will contain all the permissions you want to assign to your database user:

```
#assign database permissions
$permissionset = New-Object Microsoft.SqlServer.Management.
Smo.DatabasePermissionSet ([Microsoft.SqlServer.Management.Smo.
DatabasePermission]::Alter)
$permissionset.Add ([Microsoft.SqlServer.Management.Smo.
DatabasePermission]::CreateTable)
```

Once you've added all the permissions, invoke the `Grant` method of the database object:

```
#grant the permissions
$database.Grant($permissionset, $dbuser.Name)
```

To list all the permissions, we can go through each of the database users, and pass each user to the database's `EnumDatabasePermissions` method. This should list whether `GRANT`, `DENY`, or `REVOKE` has been assigned to a particular permission and principal:

```
$database.Users |
ForEach-Object {
    $database.EnumDatabasePermissions($_.Name) |
    Select PermissionState, PermissionType, Grantee
} |
Format-Table -AutoSize
```

See also

- ▶ The *Creating a database user* recipe
- ▶ Read more about the `DatabasePermissionSet` class from MSDN:
[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.databasepermissionset\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.databasepermissionset(v=sql.110).aspx)

Creating a database role

In this recipe, we will walk through creating a custom database role.

Getting ready

In this recipe, we will create a database role called `Custom Role`, and we will grant it `SELECT` permissions to the `HumanResources` schema, and `ALTER` and `CREATE TABLE` permissions to the database.

The T-SQL equivalent of what we are trying to accomplish is:

```
USE AdventureWorks2008R2
GO

CREATE ROLE [Custom Role]
GO

GRANT SELECT
ON SCHEMA::[HumanResources]
TO [Custom Role]

GRANT ALTER, CREATE TABLE
TO [Custom Role]
```

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:

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

#role
$rolename = "Custom Role"
if($database.Roles[$rolename])
{
    $database.Roles[$rolename].Drop()
}

#let's assume this custom role, we want to grant
#everyone in this role select, insert access
#to the HumanResources Schema, in addition to the
#CreateTable permission

$dbrole = New-Object Microsoft.SqlServer.Management.Smo.
DatabaseRole -ArgumentList $database, $rolename
$dbrole.Create()

#verify; list database roles
$database.Roles

#create a permission set to contain SELECT permissions
#for the HumanResources schema
$permissionset1 = New-Object Microsoft.SqlServer.Management.
Smo.ObjectPermissionSet([Microsoft.SqlServer.Management.Smo.
ObjectPermission]::Select)

$permissionset1.Add([Microsoft.SqlServer.Management.Smo.
ObjectPermission]::Select)
$hrschema = $database.Schemas["HumanResources"]

$hrschema.Grant($permissionset1, $dbrole.Name)

#create another permission set that contains
#CREATE TABLE and ALTER on this database
$permissionset2 = New-Object Microsoft.SqlServer.Management.Smo.
DatabasePermissionSet([Microsoft.SqlServer.Management.Smo.
DatabasePermission]::CreateTable)

$permissionset2.Add([Microsoft.SqlServer.Management.Smo.
DatabasePermission]::Alter)
```

```

$database.Grant($permissionset2, $dbrole.Name)

#to add member
#assume eric is already a user in the database
$username = "eric"
$dbrole.AddMember($username)

#confirm permissions
$database.Roles[$rolename] |
ForEach-Object {
    $currentrole = $_
    $database.EnumDatabasePermissions($_.Name) |
    Select PermissionState, PermissionType, Grantee,
    @{N="Members";E={$currentrole.EnumMembers()}}
} |
Format-Table -AutoSize

```

When the script has been successfully executed, you should see a screen similar to the following:

PermissionState	PermissionType	Grantee	Members
Grant	ALTER	Custom Role	eric
Grant	CREATE TABLE	Custom Role	eric

How it works...

A database role enables easier management of users and permissions on the database level.

To create a database role, you need to create an instance of an SMO `DatabaseRole` first:

```

$dbrole = New-Object Microsoft.SqlServer.Management.Smo.DatabaseRole
-ArgumentList $database, "Custom Role"
$dbrole.Create()

```

The next step is to identify what permissions this group needs to have. You will need to create a different permission set for each type of securable that you want to assign permissions to.

In our recipe, we created two permission sets. The first one is at the schema level, allowing the database user to use the `SELECT` statement against all objects belonging to the `HumanResources` schema:

```

#create a permission set to contain SELECT permissions
#for the HumanResources schema

```

```
$permissionset1 = New-Object Microsoft.SqlServer.Management.Smo.ObjectPermissionSet ([Microsoft.SqlServer.Management.Smo.ObjectPermission]::Select)

$permissionset1.Add ([Microsoft.SqlServer.Management.Smo.ObjectPermission]::Select)

$hrschema = $database.Schemas["HumanResources"]
$hrschema.Grant($permissionset1, $dbrole.Name)
```

Our second permission set pertains to the database securable, allowing CREATE and ALTER for the AdventureWorks2008R2 database:

```
#create another permission set that contains
#CREATE TABLE and ALTER on this database
$permissionset2 = New-Object Microsoft.SqlServer.Management.Smo.DatabasePermissionSet ([Microsoft.SqlServer.Management.Smo.DatabasePermission]::CreateTable)

$permissionset2.Add ([Microsoft.SqlServer.Management.Smo.DatabasePermission]::Alter)

$database.Grant($permissionset2, $dbrole.Name)
```

The last step in our recipe is to add users to this role. This step does not need to follow granting permissions. It can be completed as soon as the role is set up:

```
#to add member
#assume eric is already a user in the database
$username = "eric"
$dbrole.AddMember($username)
```

To confirm the settings, we use PowerShell to target this specific role. We use the `EnumDatabasePermissions` method of the SMO database class to display the `PermissionState`, `PermissionType`, and `Grantee` properties. In addition, we display the members of this database role by using the `EnumMembers` method of the SMO Role class:

```
#confirm permissions
$database.Roles[$rolename] |
ForEach-Object {
    $currentrole = $_
    $database.EnumDatabasePermissions($_.Name) |
    Select PermissionState, PermissionType, Grantee,
    @{N="Members";E={$currentrole.EnumMembers()}}
} |
Format-Table -AutoSize
```

See also

- ▶ *The Creating a database user recipe*

Fixing orphaned users

This recipe shows how you can remap orphaned database users to valid logins.

Getting ready

Let us create an orphaned user to use in this recipe. Open up SQL Server Management Studio, and execute the following T-SQL statements:

```
USE [master]
GO
CREATE LOGIN [marymargaret]
WITH PASSWORD=N'P@ssword',
DEFAULT_DATABASE=[master],
CHECK_EXPIRATION=OFF,
CHECK_POLICY=OFF
GO
USE [AdventureWorks2008R2]
GO
CREATE USER [marymargaret]
FOR LOGIN [marymargaret]
GO
USE [master]
GO
DROP LOGIN [marymargaret]
GO
-- create another login, this will generate a
-- different SID
CREATE LOGIN [marymargaret]
WITH PASSWORD=N'P@ssword',
DEFAULT_DATABASE=[master],
CHECK_EXPIRATION=OFF,
CHECK_POLICY=OFF
```

This code has created an orphaned user called `marymargaret` in the `AdventureWorks2008R2` database. Although we have recreated a login with the same name, this would generate a different Security ID (SID), thus leaving the database user orphaned.

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:

```
$databasename = "AdventureWorks2008R2"
$database = $server.Databases[$databasename]
$loginname = "marymargaret"
$username = "marymargaret"
$user = $database.Users[$username]

#display current status
$user | Select Parent, Name, Login, LoginType, UserType
```

When the script successfully finishes executing, you should see a screen similar to this. We can confirm that **marymargaret** is an orphaned user because the **Login** value in the result is blank, and the **UserType** is **NoLogin**:

```
Parent      : [AdventureWorks2008R2]
Name        : marymargaret
Login       :
LoginType   : SqlLogin
UserType    : NoLogin
```

4. Let's fix the orphaned user by remapping it to the new valid login. Note that we are going to use a combination of T-SQL and the `Invoke-Sqlcmd` cmdlet to accomplish the remapping, because of some issues with SMO, which are discussed in the *There's more...* section.

```
$query = "ALTER USER $($username) WITH LOGIN=$(($loginname))"
Invoke-Sqlcmd -ServerInstance $instanceName -Query $query
-Database $databasename
Start-Sleep -Seconds 1
```

```
#display current status
$user.Refresh()
$user | Select Parent, Name, Login, LoginType, UserType
```

```
Parent      : [AdventureWorks2008R2]
Name        : marymargaret
Login       : marymargaret
LoginType   : SqlLogin
UserType    : SqlLogin
```

How it works...

An orphaned user is a database user that is not mapped to a valid login anymore. This may stem from a number of scenarios, but more often, it happens when you move a database from server to server, for example, from production to development.

To fix an orphaned user, you need to remap this orphaned user to a valid, recognized login in your instance. The core of the solution lies in these statements:

```
$query = "ALTER USER $($username) WITH LOGIN=$($loginname)"
Invoke-Sqlcmd -ServerInstance $instanceName -Query $query -Database
$databasename
```

We acquired handles to the `User` objects merely to display the status of the user. While it's still orphaned, the `UserType` will indicate `NoLogin`.

There's more...

Following the patterns of the previous recipes, you may have thought that we should be able to use SMO to fix our orphaned user. This snippet of code *should* allow us to remap the user:

```
#unfortunately this doesn't work
$user.Login = "marymargaret"
$user.Alter()
$user.Refresh()
```

The code makes sense syntax-wise, however when you execute this, it will give an exception:

```
System.Management.Automation.MethodInvocationException: Exception calling
"Alter" with "0" argument(s): "Alter failed for User 'marymargaret'. "
--->Microsoft.SqlServer.Management.Smo.FailedOperationException: Alter
failed for User 'marymargaret'. --->Microsoft.SqlServer.Management.
Smo.SmoException: Modifying the Login property of the User object is not
allowed. You must drop and recreate the object with the desired property.
```

This error complains that the `Login` property cannot be modified unless the `User` object is dropped. Therefore to make it work using SMO, we will need to drop and recreate the database user. Dropping and recreating can work to an extent, but you will have to remember to reassign all the permissions and roles to this user. For some situations, this may not be the ideal solution.

See also

- ▶ *The Listing logins, users, and database mappings recipe*

Creating a credential

This recipe goes through the code needed for creating a SQL Server credential.

Getting ready

In this recipe, we create a credential for a domain account that has access to certain files and folders in our system, `QUERYWORKS\filemanager`. The equivalent T-SQL for what we are trying to accomplish is:

```
CREATE CREDENTIAL [filemanagercred]
WITH IDENTITY = N'QUERYWORKS\filemanager',
SECRET = N'YourSuperStrongPassword'
```

You can substitute this with another known Windows account that you have in your environment.

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:

```
$identity = "QUERYWORKS\filemanager"
$credentialname = "filemanagercred"
if($server.Credentials.Contains($credentialname))
{
```

```

$server.Credentials[$credentialname].Drop()
}
$credential=New-Object "Microsoft.SqlServer.Management.Smo.
Credential" $server, $credentialname
$credential.Create($identity, "YourSuperStrongPassword")

#list credentials
$server.Credentials

```

When the script has been successfully executed, you should see a screen similar to the following:

```

Name
----
filemanagercred

```

This should confirm that the credential has been created.

How it works...

A credential in SQL Server allows a server principal to connect to resources outside of SQL Server, using a different identity or username/password combination. This is often used to map SQL Server logins to a Windows account needed to access files/folders/programs outside of SQL Server.


Creating a credential in PowerShell is short and straightforward. To create a credential, you will need to know the username and password of the external account that you want to use as a credential:

```

$credential=New-Object "Microsoft.SqlServer.Management.Smo.Credential"
$server, $credentialname
$credential.Create($identity, "YourSuperStrongPassword")

```

You may not want to hardcode the password in your script. In that case, you can use the `Get-Credential` cmdlet to capture the password.

 The `Get-Credential` cmdlet is used and discussed further in the *Change SQL Server service account* recipe.

See also

- ▶ The *Creating a proxy* recipe
- ▶ The *Changing SQL Server service account* recipe

Creating a proxy

In this recipe, we will create a SQL Server proxy.

Getting ready

In this recipe, we will map out our SQL Server Agent service account (QUERYWORKS\sqlagent) to the credential we created in the previous recipe, filemanagercred. We are also going to grant this proxy with rights to run the PowerShell agent steps and operating system (CmdExec) steps. The equivalent T-SQL statements of what we are trying to achieve are as follows:

```
EXEC msdb.dbo.sp_add_proxy
@proxy_name = N'filemanagerproxy',
@credential_name = N'filemanagercred',
@enabled = 1,
@description = N'Proxy Account for PowerShell Agent Job steps'

EXEC msdb.dbo.sp_grant_login_to_proxy
@proxy_name = N'filemanagerproxy',
@login_name = N'QUERYWORKS\sqlagent'

-- PowerShell subsystem
EXEC msdb.dbo.sp_grant_proxy_to_subsystem
@proxy_name = N'filemanagerproxy',
@subsystem_id = 12

-- CmdExec subsystem
EXEC msdb.dbo.sp_grant_proxy_to_subsystem
@proxy_name = N'filemanagerproxy',
@subsystem_id = 12
```

You can substitute this with known SQL Server principals and credentials in your environment.

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 as follows:

```
#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:

```
$proxyname = "filemanagerproxy"
$credentialname = "filemanagercred"
$jobserver = $server.JobServer

if($jobserver.ProxyAccounts[$proxyname])
{
    $jobserver.ProxyAccounts[$proxyname].Drop()
}

$proxy=New-Object "Microsoft.SqlServer.Management.Smo.Agent.
ProxyAccount" $jobserver, $proxyname, $credentialname, $true,
"Proxy Account for PowerShell Agent Job steps"
$proxy.Create()

#add sql server agent account - QUERYWORKS\sqlagent
$agentlogin = "QUERYWORKS\sqlagent"
$proxy.AddLogin($agentlogin)
$proxy.AddSubSystem([Microsoft.SqlServer.Management.Smo.Agent.
AgentSubsystem]::PowerShell)
$proxy.AddSubSystem([Microsoft.SqlServer.Management.Smo.Agent.
AgentSubsystem]::CmdExec)

#confirm, list proxy accounts
$jobserver.ProxyAccounts |
ForEach-Object {
    $currproxy = $_
    $subsystems = ($currproxy.EnumSubSystems() |
        Select -ExpandProperty Name) -Join ","
    $currproxy |
    Select Name, CredentialName, CredentialIdentity,
    @{N="Subsystems";E={$subsystems}}
} |
Format-Table -AutoSize
```

When the script has been successfully executed, you should see a screen similar to this. This should confirm that the proxy has been created and subsystems have been assigned.

Name	CredentialName	CredentialIdentity	Subsystems
filemanagerproxy	filemanagercred	QUERYWORKS\filemanager	CmdExec,PowerShell

How it works...

The first step is to create an SMO proxy instance:

```
$proxy=New-Object "Microsoft.SqlServer.Management.Smo.Agent.  
ProxyAccount" $jobserver, $proxyname, $credentialname, $true, "Proxy  
Account for PowerShell Agent Job steps"  
$proxy.Create()
```

To create a proxy, you will need two pieces of information—the server principal (login) you want to use, and the SQL Server credential to map it to. In our recipe, we mapped our SQL Server Agent service account `QUERYWORKS\sqlagent` to a domain account called `QUERYWORKS\filemanager` via the `filemanagercred` credential.

```
$agentlogin = "QUERYWORKS\sqlagent"  
$proxy.AddLogin($agentlogin)
```

In SQL Server, we also need to narrow down on which specific subsystems the proxy can be used:

```
$proxy.AddSubSystem([Microsoft.SqlServer.Management.Smo.Agent.  
AgentSubsystem]::PowerShell)  
$proxy.AddSubSystem([Microsoft.SqlServer.Management.Smo.Agent.  
AgentSubsystem]::CmdExec)
```

In our recipe, we specified the `PowerShell` and `CmdExec` subsystems. Other common options include `TransactSQL`, `ActiveScripting`, `AnalysisCommand`, `AnalysisQuery`, and `SSIS`.

To confirm, we iterate through all `ProxyAccounts`, and we also use the method `EnumSubsystems` of the `Microsoft.SqlServer.Management.Smo.Agent.ProxyAccount` class to display which subsystems are tied to a proxy.

```
#confirm, list proxy accounts  
$jobserver.ProxyAccounts |  
ForEach-Object {  
    $currproxy = $_
```

```
$subsystems = ($currproxy.EnumSubSystems() |  
                Select -ExpandProperty Name) -Join ", "  
$currproxy |  
Select Name, CredentialName, CredentialIdentity,  
@{N="Subsystems";E={$subsystems}}  
} |  
Format-Table -AutoSize
```

You can find the complete enumeration values from MSDN:

<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.agent.agentsubsystem.aspx>

There's more...

You will often encounter the need to use proxies when you have some principals that need to access external resources, but you don't want to grant them those extra permissions outside of SQL Server. One common scenario is with your SSIS packages. SQL Server Agent would usually not have the extra rights to access files and folders. To avoid granting these extra rights, you will need to map the agent account to another account that already has these rights.

See also

- ▶ The *Creating a credential* recipe

5

Advanced Administration

In this chapter, we will cover:

- ▶ Listing facets and facet properties
- ▶ Listing policies
- ▶ Exporting a policy
- ▶ Importing a policy
- ▶ Creating a condition
- ▶ Creating a policy
- ▶ Evaluating a policy
- ▶ Enabling/disabling change tracking
- ▶ Running and saving a profiler trace event
- ▶ Extracting the contents of a trace file
- ▶ Creating a database master key
- ▶ Creating a certificate
- ▶ Creating symmetric and asymmetric keys
- ▶ Setting up Transparent Data Encryption (TDE)

Introduction

The most recent versions of SQL Server have seen new features that can help IT professionals get a better handle on the instances and databases they are managing. Policies can now be created on SQL Server, and applied to single or multiple instances, to ensure compliance of settings and configurations with company rules. SQL Server also supports different levels of encryption, including cell-level or column-level encryption, and database-level encryption. PowerShell can help with setting up security policies, or enabling **Transparent Database Encryption (TDE)** for encrypting your whole database. In this chapter, we will also look at how we can work with SQL Server Profiler trace files and trace events.

Listing facets and facet properties

In this recipe, we will list all available facets and their properties.

How to do it...

4. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

5. Import the SQLPS module as follows:

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

6. Add the following script and run:

```
[Microsoft.SqlServer.Management.Dmf.PolicyStore]::Facets |
ForEach-Object {
    $facet = $_
    $facet.FacetProperties |
    Select @{N="FacetName";E={$facet.Name}},
    @{N="PropertyName";E={$_.Name}},
    @{N="PropertyType";E={$_.PropertyType}}
} |
Format-Table
```

When the script successfully finishes executing, the resulting screen should display all the facets and their properties.

FacetName	PropertyName	PropertyType
ApplicationRole	CreateDate	System.DateTime
ApplicationRole	DateLastModified	System.DateTime
ApplicationRole	ID	System.Int32
ApplicationRole	DefaultSchema	System.String
ApplicationRole	Name	System.String
ApplicationRole	ID	System.Int32
AsymmetricKey	KeyEncryptionAlgorithm	Microsoft.Sqls
AsymmetricKey	KeyLength	System.Int32
AsymmetricKey	Owner	System.String
AsymmetricKey	PrivateKeyEncryptionType	Microsoft.Sqls
AsymmetricKey	PublicKey	System.Byte[]
AsymmetricKey	Sid	System.Byte[]
AsymmetricKey	Thumbprint	System.Byte[]
AsymmetricKey	ProviderName	System.String
AsymmetricKey	Name	System.String
Audit	CreateDate	System.DateTime
Audit	DateLastModified	System.DateTime
Audit	DestinationType	Microsoft.Sqls
Audit	Enabled	System.Boolean
Audit	FileName	System.String
Audit	FilePath	System.String
Audit	Guid	System.Guid
Audit	ID	System.Int32
Audit	MaximumFileSize	System.Int32
Audit	MaximumFileCount	System.Int32

How it works...

Facets are introduced with SQL Server 2008's **Policy Based Management (PBM)**. Facets are defined in MSDN as follows:

a set of logical properties that model the behavior or characteristics for certain types of managed targets.

Simply, these are the SQL Server components manageable through PBM.

For exploring facets, you need to connect to the `PolicyStore` parameter, using the `Microsoft.SqlServer.Management.Dmf.PolicyStore` namespace.

```
[Microsoft.SqlServer.Management.Dmf.PolicyStore]::Facets
```

Note that DMF is the *old* PBM name, which stands for Declarative Management Framework.

In this recipe we iterate through all the facets, and display the facet name, facet property name, and type:

```
[Microsoft.SqlServer.Management.Dmf.PolicyStore]::Facets |
ForEach-Object {
```

For each facet we extract the respective facet properties:

```
[Microsoft.SqlServer.Management.Dmf.PolicyStore]::Facets |  
ForEach-Object {  
    $facet = $_  
    $facet.FacetProperties |  
    Select @{N="FacetName";E={$facet.Name}},  
    @{N="PropertyName";E={$_.Name}},  
    @{N="PropertyType";E={$_.PropertyType}}  
} |  
Format-Table
```

To explore facets more, use the `$facet` object and pipe it to `Get-Member`.

```
$facet | Get-Member
```

See also

- ▶ The *Listing policies* recipe
- ▶ The *Creating a policy* recipe
- ▶ Dan Jones, Project Manager on the SQL Server Manageability team at Microsoft, explains facets in his blog post:

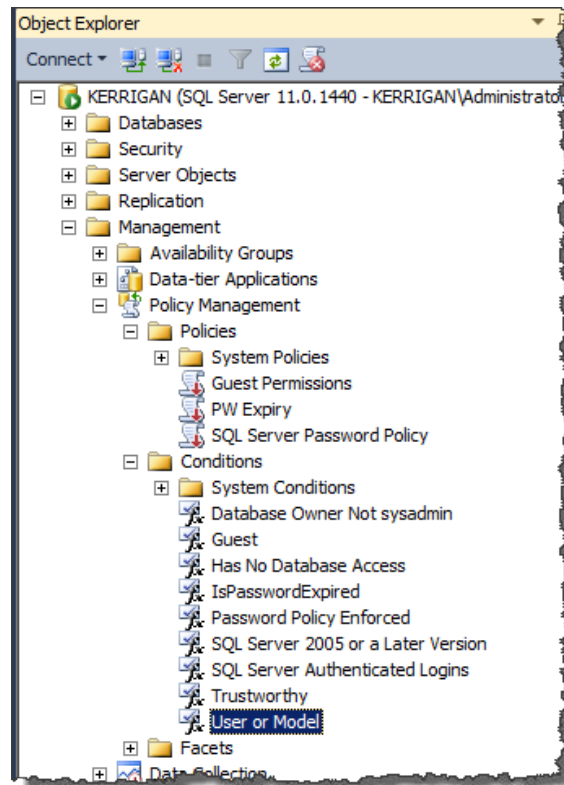
<http://blogs.msdn.com/b/sqlpbm/archive/2008/05/24/facets.aspx>

Listing policies

In this recipe, we will list policies deployed in our SQL Server instance.

Getting ready

Check which policies are being used in your environment using SQL Server Management Studio. Connect to SSMS, and expand **Management** | **Policy Management** | **Policies**:



These are the same policies you should get when you run the PowerShell script in this 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 as follows:

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

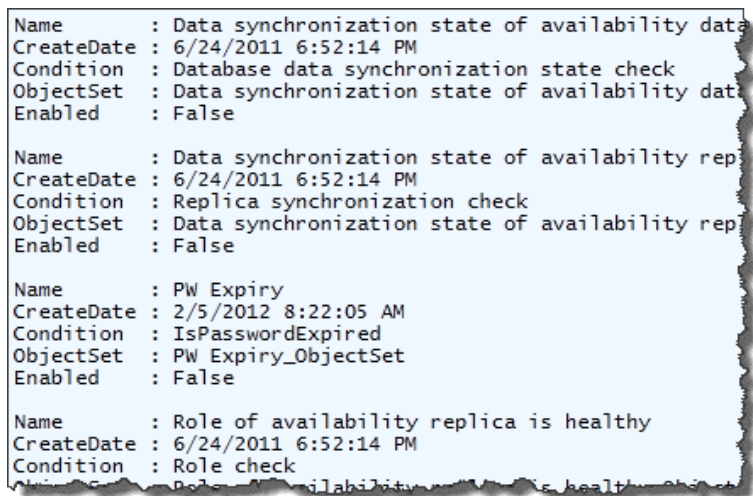
3. Add the following script and run:

```
$connectionstring = "server='KERRIGAN';Trusted_Connection=true"
```

```
$conn = New-Object Microsoft.SqlServer.Management.Sdk.Sfc.SqlStore
Connection($connectionstring)
```

```
#NOTE notice how the namespace is still called DMF
#DMF - declarative management framework
#DMF was the old reference to Policy Based Management
$PolicyStore = New-Object Microsoft.SqlServer.Management.DMF.
PolicyStore($conn)
$PolicyStore.Policies |
Select Name, CreateDate, Condition, ObjectSet, Enabled |
Format-List
```

When the script successfully finishes executing, the resulting screen should display all the policies registered in your instance:



```
Name      : Data synchronization state of availability data
CreateDate : 6/24/2011 6:52:14 PM
Condition  : Database data synchronization state check
ObjectSet  : Data synchronization state of availability data
Enabled    : False

Name      : Data synchronization state of availability replica
CreateDate : 6/24/2011 6:52:14 PM
Condition  : Replica synchronization check
ObjectSet  : Data synchronization state of availability replica
Enabled    : False

Name      : PW Expiry
CreateDate : 2/5/2012 8:22:05 AM
Condition  : IsPasswordExpired
ObjectSet  : PW Expiry_ObjectSet
Enabled    : False

Name      : Role of availability replica is healthy
CreateDate : 6/24/2011 6:52:14 PM
Condition  : Role check
ObjectSet  : Role of availability replica is healthy_ObjectSet
```

How it works...

To list the policies in your instance, you need to connect to the `PolicyStore` parameter. Note that the `PolicyStore` parameter requires a different type of Connection compared to the SMO server connections we have been making in the previous recipes. To connect to the `PolicyStore` parameter, you first need to create an `Sfc.SqlStoreConnection` object:

```
$connectionstring = "server='KERRIGAN';Trusted_Connection=true"

$conn = New-Object Microsoft.SqlServer.Management.Sdk.Sfc.
SqlStoreConnection($connectionstring)
```

Once `Sfc.SqlStoreConnection` has been established, you can connect to the `PolicyStore` parameter:

```
#NOTE notice how the namespace is still called DMF
#DMF - declarative management framework
#DMF was the old reference to Policy Based Management
$PolicyStore = New-Object Microsoft.SqlServer.Management.DMF.
PolicyStore($conn)
```

Once you have a handle to the `PolicyStore` parameter, you can use the `Policies` object and list `Name`, `CreateDate`, and `Condition`—among other properties:

```
$PolicyStore.Policies |
Select Name, CreateDate, Condition, ObjectSet, Enabled |
Format-List
```

See also

- ▶ The *Listing facets and facet properties* recipe
- ▶ The *Creating a facet* recipe
- ▶ To learn more about `SqlStoreConnection`, check out this MSDN article:
<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.sdk.sfc.sqlstoreconnection.aspx>

Exporting a policy

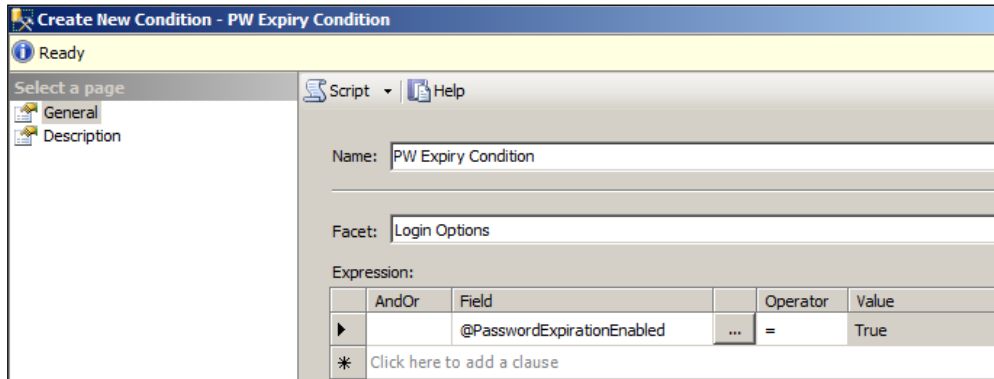
In this recipe, we will export a policy to an XML file using PowerShell.

Getting ready

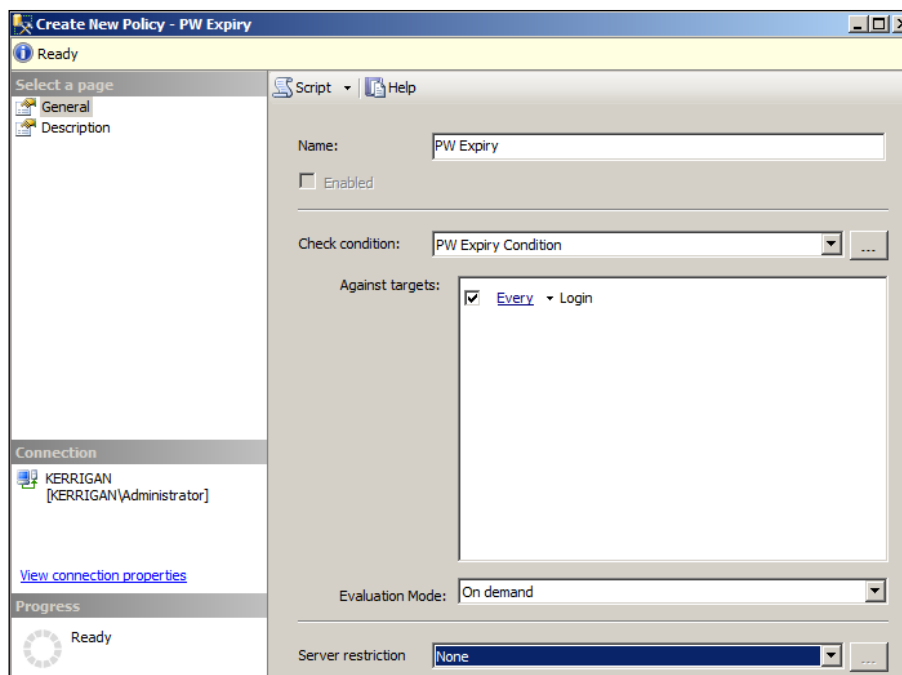
We will export a policy called `PW Expiry` to an XML file. To do this we must first create this policy by performing the following steps:

1. Log in to SQL Server Management Studio, and expand **Management | Policy Management**.
2. Right-click on **Conditions** and select **New Condition**.

3. Create a new condition:
 1. Set **Name** to PW Expiry Condition.
 2. Select **Login Options** for **Facet**.
 3. Use **@PasswordExpirationEnabled = True** for **Expression**.
 4. Click on **OK** when done.



4. Right-click on **Policies** and select **New Policy**.
5. Create a new policy:
 1. Type PW Expiry for **Name**.
 2. Use **PW Expiry Condition** for **Check condition**.
 3. Leave the checkbox for **Against targets** checked, since we want to target every login.
 4. Leave **Evaluation Mode** to **On demand**.
 5. Leave **Server restriction** to **None**.
 6. Click on **OK** when done.



Alternatively, you can substitute this with another policy that exists in your system.

How to do it...

To export a policy to an XML file, perform the following steps:

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the SQLPS module as follows:

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

3. Add the following script and run:

```
$connectionstring = "server='KERRIGAN';Trusted_Connection=true"
```

```
$conn = New-Object Microsoft.SqlServer.Management.Sdk.Sfc.SqlStore
Connection($connectionstring)
```

```
#NOTE this is still called DMF, which stands for
#PBM's old name, Declarative Management Framework
$polycystore = New-Object Microsoft.SqlServer.Management.DMF.
PolicyStore($conn)

#change this to your policy name
$policyname = "PW Expiry"
$policy = $polycystore.Policies[$policyname]

#create an XML writer, to enable us to
#write an XML file
$folder = "C:\Temp\"
$policyfilename = "$($policy.Name).xml"
$fullpath = Join-Path $folder $policyfilename

$xmlwriter = [System.Xml.XmlWriter]::Create($fullpath)
$policy.Serialize($xmlwriter)
$xmlwriter.Close()
```

How it works...

Policies are stored as XML documents, so these policies can be easily exported as XML files.

To export a policy, you first need to get a handle to the `PolicyStore` parameter:

```
$polycystore = New-Object Microsoft.SqlServer.Management.DMF.
PolicyStore($conn)
```

Once the connection to the `PolicyStore` parameter is established, you can get a handle to the policy you want to export:

```
$policyname = "PW Expiry"
$policy = $polycystore.Policies[$policyname]
```

Exporting the policy requires writing the contents of the policy to an XML file in your file system. We will need to use `XMLWriter` in this case:

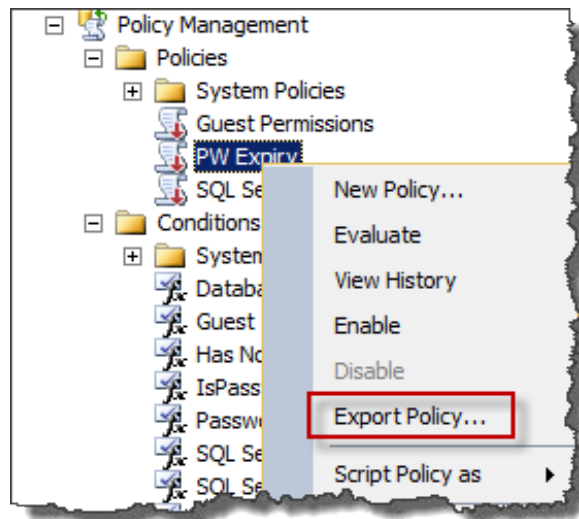
```
#create an XML writer, to enable us to
#write an XML file
$folder = "C:\Temp\"
$policyfilename = "$($policy.Name).xml"
$fullpath = Join-Path $folder $policyfilename

$xmlwriter = [System.Xml.XmlWriter]::Create($fullpath)
$policy.Serialize($xmlwriter)
$xmlwriter.Close()
```

Once that's done, double-check the file that was created. When you open it, you should see the XML structure used to store your policies.

There's more...

To export a policy from SQL Server Management Studio, you can right-click on a policy and select **Export Policy** as shown in the following screenshot:



See also

- ▶ The *Listing policies* recipe
- ▶ The *Importing a policy* recipe

Importing a policy

This recipe will show how you can import a policy stored as an XML file into SQL Server.

Getting ready

In this recipe, we will use an XML policy that comes with the default SQL Server installation. This policy is called `Guest Permissions.xml`, and is stored in `C:\Program Files (x86)\Microsoft SQL Server\110\Tools\Policies\DatabaseEngine\1033`

Feel free to substitute this with a policy you have available in 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
```

3. Add the following script and run:

```
$connectionstring = "server='KERRIGAN';Trusted_Connection=true"
$conn = New-Object Microsoft.SqlServer.Management.Sdk.Sfc.
SqlStoreConnection($connectionstring)

#connect to polycystore
$policyStore = New-Object Microsoft.SqlServer.Management.DMF.
PolicyStore($conn)

#you can replace this with your own file
$policyXmlPath = "C:\Program Files (x86)\Microsoft SQL Server\110\
Tools\Policies\DatabaseEngine\1033\Guest Permissions.xml"

$xmlReader = [System.Xml.XmlReader]::Create($policyXmlPath)

#ready to import
$policyStore.ImportPolicy($xmlReader, [Microsoft.SqlServer.
Management.Dmf.ImportPolicyEnabledState]::Unchanged, $true, $true)

#list policies to confirm
$policyStore.Policies
```

All the loaded policies should be listed when the script has finished executing. Check that the **Guest Permissions** policy is included in the list.

Name	Category	Created	Enabled
Guest Permissions	Microsoft Best...	2/11/2012 11:24 AM	False
All availability replicas a...	Warnings in av...	6/24/2011 6:52 PM	False
All availability replicas a...	Warnings in av...	6/24/2011 6:52 PM	False
All availability replicas h...	Warnings in av...	6/24/2011 6:52 PM	False
All synchronous replicas ar...	Warnings in av...	6/24/2011 6:52 PM	False
Availability database is jo...	Warnings in av...	6/24/2011 6:52 PM	False
Availability database is no...	Warnings in av...	6/24/2011 6:52 PM	False
Availability group is online	Errors in avai...	6/24/2011 6:52 PM	False
Availability group is ready...	Errors in avai...	6/24/2011 6:52 PM	False
Availability replica is con...	Errors in avai...	6/24/2011 6:52 PM	False
Data synchronization state ...	Warnings in av...	6/24/2011 6:52 PM	False

How it works...

To import a policy defined in an XML file, you will first need to connect to the `PolicyStore` parameter.

```
$connectionstring = "server='KERRIGAN';Trusted_Connection=true"

$conn = New-Object Microsoft.SqlServer.Management.Sdk.Sfc.
SqlStoreConnection($connectionstring)

#connect to polycystore
$policyStore = New-Object Microsoft.SqlServer.Management.DMF.
PolicyStore($conn)
```

You will also need to specify which file you want to import:

```
#you can replace this with your own file
$policyXmlPath = "C:\Program Files (x86)\Microsoft SQL Server\110\
Tools\Policies\DatabaseEngine\1033\Guest Permissions.xml"
```

You will need to load this using an `XMLReader` class, which we will pass to our import method:

```
$xmlReader = [System.Xml.XmlReader]::Create($policyXmlPath)
```

When you are ready to import, you can use the `ImportPolicy` method of the `PolicyStore` object:

```
$policyStore.ImportPolicy($xmlReader, [Microsoft.SqlServer.Management.
Dmf.ImportPolicyEnabledState]::Unchanged, $true, $true)
```

If you want to import all policies, you can get all the XML files from the default path for the policies using the `Get-ChildItem` cmdlet. Iterate through each file, and load each of them using the `ImportPolicy` method.

```
$xmlPath = "C:\Program Files (x86)\Microsoft SQL Server\110\Tools\
Policies\DatabaseEngine\1033\"

Get-ChildItem -Path "$($xmlPath)*.xml" |
ForEach-Object {
    $xmlReader = [System.Xml.XmlReader]::Create($_.FullName)
    $policyStore.ImportPolicy($xmlReader, [Microsoft.SqlServer.
Management.Dmf.ImportPolicyEnabledState]::Unchanged, $true, $true) |
    Out-Null
}
```

There's more...

The `ImportPolicy` method accepts four parameters:

- ▶ `XMLReader` that contains the policy
- ▶ `ImportEnabledState`
- ▶ A Boolean value for `overwriteExistingPolicy`
- ▶ A Boolean value for `overwriteExistingCondition`

See also

- ▶ The *Listing policies* recipe
- ▶ The *Exporting a policy* recipe

Creating a condition

In this recipe, we will create a condition to be later used programmatically for a policy.

Getting ready

In this recipe, we will create a condition called `xp_cmdshell is disabled`, which checks the `Server Security` facet, `XPCmdShellEnabled`.

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 connection object as follows:

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

$connectionstring ="server='KERRIGAN';Trusted_Connection=true"

$conn = New-Object Microsoft.SqlServer.Management.Sdk.Sfc.
SqlStoreConnection($connectionstring)

$polycystore = New-Object Microsoft.SqlServer.Management.DMF.
PolicyStore($conn)
```

3. Add the following script and run:

```
$conditionName = "xp_cmdshell is disabled"

if ($policystore.Conditions[$conditionName])
{
    $policystore.Conditions[$conditionName].Drop()
}

#facet name
#we are retrieving facet name in this manner because
#some facet names are different from the display names
#note this is PowerShell V3 syntax Where-Object syntax
$selectedfacetdisplayname = "Server Security"
$selectedfacet = [Microsoft.SqlServer.Management.Dmf.
PolicyStore]::Facets |
Where-Object DisplayName -eq $selectedfacetdisplayname

#if you want to use PowerShell V2 syntax, use the
#following for the Where-Object clause:
#Where-Object {
#$_ .DisplayName -eq $selectedfacetdisplayname
#}

#display, for visual reference
$selectedfacet.Name

#create condition

$condition = New-Object Microsoft.SqlServer.Management.Dmf.
Condition($conn, $conditionName)
$condition.Facet = $selectedfacet.Name

#a condition consists of a facet, an operator,
#and a value to compare to
$op = [Microsoft.SqlServer.Management.Dmf.OperatorType]::EQ
$attr = New-Object Microsoft.SqlServer.Management.Dmf.ExpressionNo
deAttribute("XPcmdShellEnabled")
$value = [Microsoft.SqlServer.Management.Dmf.ExpressionNode]::Cons
tructNode($false)

#create the expression node
#this is equivalent to "@XPcmdShellEnabled = false"
$expressionNode = New-Object Microsoft.SqlServer.Management.Dmf.
ExpressionNodeOperator($op, $attr, $value)
```



```
#display expression node that was constructed
$expressionNode

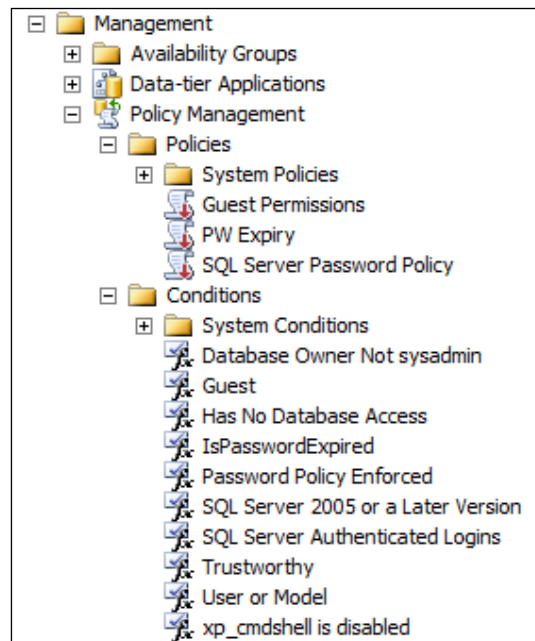
#assign the expression node to the condition, and create
$condition.ExpressionNode = $expressionNode
$condition.Create()

#confirm by displaying conditions in PolicyStore
$policystore.Conditions |
Where Name -eq $conditionName |
Select Name, Facet, ExpressionNode |
Format-Table -AutoSize
```

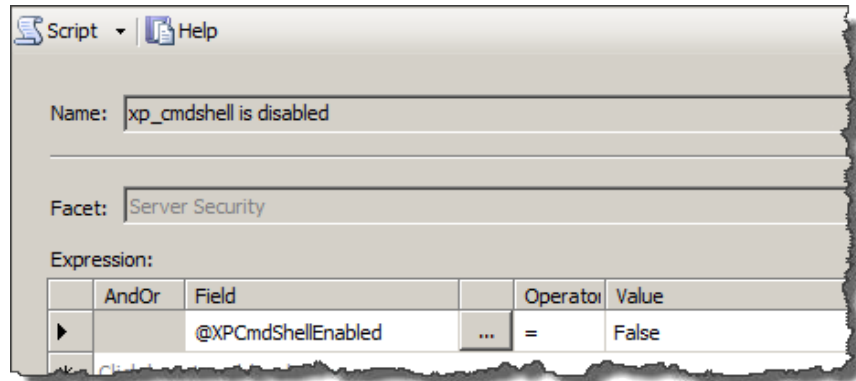
When the script finishes, you should see the new condition displayed in the resulting output:

Name	Facet	ExpressionNode
xp_cmdshell is disabled	IServerSecurityFacet	@XPcmdShellEnabled = False()

4. Confirm visually from SQL Server Management Studio:
 1. Connect to SSMS.
 2. Go to **Management** and expand **Policy Management | Conditions:**



3. Double-click on the **xp_cmdshell is disabled** condition.



How it works...

Creating a condition for policy-based management requires creating what is called an expression node. This is the expression that will be utilized by policies, and will be evaluated to be true or false.

```
#a condition consists of a facet, an operator,
#and a value to compare to
$op = [Microsoft.SqlServer.Management.Dmf.OperatorType]::EQ
$attr = New-Object Microsoft.SqlServer.Management.Dmf.
ExpressionNodeAttribute("XPcmdShellEnabled")
$value = [Microsoft.SqlServer.Management.Dmf.ExpressionNode]::
ConstructNode($false)
```

To put these together, we use the `ExpressionNodeOperator` class of the `Microsoft.SqlServer.Management.Dmf` namespace to construct the final expression node. The constructor, or special method to create a new object, of this class accepts an operator type, a left expression, and a right expression.

```
#create the expression node
#this is equivalent to "@XPcmdShellEnabled = false"
$expressionNode = New-Object Microsoft.SqlServer.Management.Dmf.
ExpressionNodeOperator($op, $attr, $value)
```

Some conditions are straightforward, and will not require an `ExpressionNodeOperator` class to construct them. For example:

- ▶ "@Size <= 100"
- ▶ "@ID >= 4"
- ▶ "Name = 'sqlagent'"

This expression node is what we need to assign to the condition object.

```
#assign the expression node to the condition, and create
$condition.ExpressionNode = $expressionNode
```

Once the expression has been assigned, we can now invoke the `Create` method of the `Microsoft.SqlServer.Management.Dmf.Condition` class to create the condition in SQL Server:

```
$condition.Create()
```

See also

- ▶ The *Creating a policy* recipe
- ▶ Here are a few useful links to `ExpressionNodes` and `ExpressionNodeOperator`.
 - `ExpressionNode`:
[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.dmf.expressionnode\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.dmf.expressionnode(v=sql.110).aspx)
 - `ExpressionNodeOperator`:
[http://msdn.microsoft.com/en-us/library/cc286169\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/cc286169(v=sql.110).aspx)

Creating a policy

In this recipe, we will create a policy programmatically using PowerShell.

Getting ready

In this recipe, we will use a condition called `xp_cmdshell is disabled`, which we created in a previous recipe. Feel free to substitute this with a condition that is available in your instance.

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 connection object as follows:

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

```
$connectionstring = "server='KERRIGAN';Trusted_Connection=true"
$conn = New-Object Microsoft.SqlServer.Management.Sdk.Sfc.
SqlStoreConnection($connectionstring)
$policystore = New-Object Microsoft.SqlServer.Management.Dmf.
PolicyStore($conn)
```

3. Add the following script and run:

```
$policyName = "xp_cmdshell must be disabled"
$conditionName = "xp_cmdshell is disabled"

if ($policystore.Policies[$policyName])
{
    $policystore.Policies[$policyName].Drop()
}

#facet name this policy refers to
#note we are using PowerShell V3 syntax in
#Where-Object
$selectedfacetdisplayname = "Server Security"
$selectedfacet = [Microsoft.SqlServer.Management.Dmf.
PolicyStore]::Facets |
Where DisplayName -eq $selectedfacetdisplayname

#if you want to use PowerShell V2 syntax, use the
#following for the Where-Object clause:
#Where-Object {$_.DisplayName -eq
#$selectedfacetdisplayname}

#create objectset
#objectset represents a policy-based management set of objects
$objectsetName = "$($policyName)_ObjectSet"
$objectset = New-Object Microsoft.SqlServer.Management.Dmf.
ObjectSet($policystore, $objectsetName)
$objectset.Facet = $selectedfacet.Name
$objectset.Create()

#confirm, display objectset name
#again we are using PowerShell V3 simplified
#Where-Object syntax here
$objectset.Name
$policystore.ObjectSets |
Where Name -eq $objectsetName |
Format-List
```

```
#if using PowerShell V2, use
#Where {$_.Name -eq $objectsetName} | Format-List

#create policy
$policy = New-Object Microsoft.SqlServer.Management.Dmf.Policy
($conn, $policyName)

#assumption here is conditions have been pre-created
#if not, see recipe for creating a condition
$policy.Condition=$conditionName
$policy.ObjectSet = $objectsetName
$policy.AutomatedPolicyEvaluationMode= [Microsoft.SqlServer.
Management.Dmf.AutomatedPolicyEvaluationMode] :None
$policy.Create()

#confirm, display policies
#PowerShell V3 syntax
$polycystore.Policies |
Where-Object Name -eq $policyName


#PowerShell V2
#Where-Object {$_.Name -eq $policyName}
```

How it works...

To start, you need to create a Policy instance:

```
#create policy
$policy = New-Object Microsoft.SqlServer.Management.Dmf.Policy
($conn, $policyName)
```

Before you create a policy, you need to make sure you have available condition(s) you can use to attach to your policy. In our recipe, we will use the condition `xp_cmdshell is disabled`.

 The `xp_cmdshell is disabled` condition is created in the *Create a condition* recipe.

To attach a condition to a policy, you can assign this to the policy's `Condition` property.

```
#assumption here is conditions have been pre-created
$policy.Condition=$conditionName
```

PBM also requires an object set. An object set is defined in MSDN as an object that *represents a policy-based management set of objects*. The object set provides the target objects for the policy, in our case, our facet.

```
#create objectset
#objectset represents a policy-based management set of objects
$objectsetName = "$($policyName)_ObjectSet"
$objectset = New-Object Microsoft.SqlServer.Management.Dmf.
ObjectSet($policystore, $objectsetName)
$objectset.Facet = $selectedfacet.Name
$objectset.Create()
```

You will also need to specify what the evaluation mode is. The valid values for evaluation mode are:

Evaluation mode	Description
None	No policy checking
Enforce	Use DDL triggers to evaluate or prevent policy violations
CheckOnChanges	Use event notification to evaluate a policy when changes happen
CheckOnSchedule	Use SQL Server Agent to evaluate a policy based on schedule

Not all facets support all possible evaluation modes. Most support `OnDemand` (ie, `None`) and `OnSchedule`. Aaron Bertrand posted a blog called *Policy-Based Management : Which facets support which evaluation methods?* that provides a way to determine which evaluation methods are supported by each facet (http://sqlblog.com/blogs/aaron_bertrand/archive/2011/10/03/policy-based-management-which-facets-support-which-evaluation-methods.aspx).

For our purposes, we will just choose `None`, or `OnDemand`:

```
$policy.AutomatedPolicyEvaluationMode=[Microsoft.SqlServer.Management.
Dmf.AutomatedPolicyEvaluationMode]::None
```

When ready, invoke the `Create` method of the policy object:

```
$policy.Create()
```

See also

- ▶ The *Creating a condition* recipe
- ▶ The complete `EvaluationMode` enumeration values can be found in:
[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.dmf.automatedpolicyevaluationmode\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.dmf.automatedpolicyevaluationmode(v=sql.110).aspx)

Evaluating a policy

In this recipe, we will evaluate a policy against our SQL Server instance.

Getting ready

In this recipe, we will evaluate the policy `xp_cmdshell must be disabled`, which we created in a previous recipe. We also want to export this to an XML file, so we can see two different ways of evaluating the policy. Use the *Exporting a policy* recipe to export the policy `xp_cmdshell must be disabled` and save it in `C:\Temp`. Alternatively you can:

1. Log in to SQL Server Management Studio.
2. Go to **Management | Policy Management** and expand **Policies**.
3. Right-click on the policy **xp_cmdshell must be disabled**, and select **Export Policy**.
4. Save this policy in `C:\Temp`.

Feel free to substitute this with a policy that is available in your instance.

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 connection object as follows:

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

$instanceName = "KERRIGAN"

$connectionstring = "server='KERRIGAN';Trusted_Connection=true"

$conn = New-Object Microsoft.SqlServer.Management.Sdk.Sfc.
SqlStoreConnection($connectionstring)

$polycystore = New-Object Microsoft.SqlServer.Management.DMF.
PolicyStore($conn)
```

3. Add the following script and run:

```
$policyName = "xp_cmdshell must be disabled"

$policy = $polycystore.Policies[$policyName]
```

```

#evaluate using the Evaluate() method
$policy.Evaluate([Microsoft.SqlServer.Management.DMF.AdHocPolicyEvaluationMode]::Check,$conn)

#check evaluation history
Write-Host "$("= " * 100)`n Evaluation Histories`n $("=" * 100)"
$policy.EvaluationHistories

#an alternative way to invoke a policy is
#to use the Invoke-PolicyEvaluation cmdlet instead
#of using the Evaluate() method
#however you need to have a handle to the actual XML file
#this alternative way allows you to capture the results
#which you can save to another XML file
#assuming we have this policy definition in
$file = "C:\Temp\$($policyName).xml"
$result = Invoke-PolicyEvaluation -Policy $file -TargetServer
$instanceName

#display results
Write-Host "$("= " * 100)`n Invocation Result`n $("=" * 100)"
$result

```

This is what your result should look similar to:

Evaluation Histories					
ID	Policy Name	Result	Start Date	End Date	Messages
15	xp_cmdshell must be disabled	True	2/11/2012 12:58 PM	2/11/2012 12:58 PM	
16	xp_cmdshell must be disabled	True	2/11/2012 1:10 PM	2/11/2012 1:11 PM	
17	xp_cmdshell must be disabled	True	2/11/2012 1:13 PM	2/11/2012 1:13 PM	
18	xp_cmdshell must be disabled	True	2/11/2012 1:18 PM	2/11/2012 1:18 PM	
19	xp_cmdshell must be disabled	True	2/11/2012 1:20 PM	2/11/2012 1:20 PM	
20	xp_cmdshell must be disabled	True	2/11/2012 1:21 PM	2/11/2012 1:21 PM	
21	xp_cmdshell must be disabled	True	2/11/2012 1:21 PM	2/11/2012 1:21 PM	
22	xp_cmdshell must be disabled	True	2/11/2012 1:22 PM	2/11/2012 1:22 PM	
Results					
1	xp_cmdshell must be disabled	True	2/11/2012 1:22 PM	2/11/2012 1:22 PM	

How it works...

In this recipe, we covered a couple of ways to evaluate a policy.

The first way is by using the `Policy` object. We first need to get a handle to the `Policy` object:

```
$policyName = "xp_cmdshell must be disabled"  
$policy = $policystore.Policies[$policyName]
```

The `Policy` object has a method called `Evaluate`, which we can invoke as follows:

```
$policy.Evaluate([Microsoft.SqlServer.Management.DMF.  
AdHocPolicyEvaluationMode]::Check,$conn)
```

The `Evaluate` method returns a `Boolean` value—true if every object you evaluated the policy against are in compliance to the policy, and false otherwise.

An alternative way to invoke a policy is by using the `Invoke-PolicyEvaluation` cmdlet. You will need to provide the full path of the XML file that contains the policy. This cmdlet also returns the result of the evaluation, also in XML format, which you can either display or save to a file:

```
$result = Invoke-PolicyEvaluation -Policy $file -TargetServer  
$instanceName
```

There's more...

To get more information about `Invoke-PolicyEvaluation`, type:

```
Get-Help Invoke-PolicyEvaluation
```

You will quickly find out that this cmdlet allows you to:

- ▶ Evaluate policies against your target objects
- ▶ Retrieve results in an XML format, which you can redirect to an XML file for storage
- ▶ Reconfigure objects in the target set that do not comply with the policy, if run in `Configure` mode

See also

- ▶ The *Creating a policy* recipe

Enabling/disabling change tracking

This recipe shows you how you can enable and disable change tracking to your target database.

Getting ready

In this recipe, we will use a test database called `TestDB`. If you don't already have this database, log in to SQL Server Management Studio and execute the following T-SQL code:

```
IF DB_ID('TestDB') IS NULL
CREATE DATABASE TestDB
GO
```

Check which of your databases have change tracking enabled. Connect to your instance using SQL Server Management Studio, and type in this T-SQL statement:

```
SELECT
    DB_NAME(database_id) AS 'DB',
    *
FROM
    sys.change_tracking_databases
```

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 as follows:

```
#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:

```
$databasename = "TestDB"
$database = $server.Databases[$databasename]

$database.ChangeTrackingEnabled
$database.ChangeTrackingEnabled = $true
```

```
$database.Alter()  
$database.Refresh()  
$database.ChangeTrackingEnabled
```

To disable change tracking, you just need to set the database property `ChangeTrackingEnabled` to `false`, and invoke the `Alter` method again.

```
$database.ChangeTrackingEnabled = $false  
$database.Alter()
```

How it works...

Change tracking is a database-level feature that can be turned on or off using the database object's `ChangeTrackingEnabled` property. Once you get a handle to the database, you can set this property to a true or false Boolean value, followed by an invocation of the `Alter` method:

```
$database.ChangeTrackingEnabled  
$database.ChangeTrackingEnabled = $true  
$database.Alter()
```

There's more...

Change Tracking (CT) is a feature introduced in SQL Server 2008. It is a lightweight solution that enables developers and administrators alike to detect if changes have been done to a user table they are monitoring. This is a pretty lightweight solution, because it only tracks those changes that have occurred, and does not keep track of all intermediate changes.

See also

- ▶ *The Altering database properties recipe in Chapter 2, SQL Server and PowerShell Basic Tasks*

Running and saving a profiler trace event

In this recipe, we will run and save a profiler trace event using PowerShell.

Getting ready

To run and save a profiler trace event, we will need to use the x86 version of PowerShell and/or PowerShell ISE. This is unfortunate, but some of the classes we need to use are only supported in 32-bit mode.

In this recipe, we will need to use the standard trace **Template Definition File (TDF)** as our starting template for the trace we're going to run. This can be found in `C:\Program Files (x86)\Microsoft SQL Server\110\Tools\Profiler\Templates\Microsoft SQL Server\110\Standard.tdf`

For our purposes, we are also going to limit the number of events to 50.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE (x86)**.
2. Import the `SQLPS` module, and create a new SMO Server object as follows:

```
#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. Import additional libraries. These are needed to use our `TraceFile` and `TraceServer` classes. We do this as follows:

```
#load ConnectionInfoExtended, this contains TraceFile class
[Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.
ConnectionInfoExtended") |
Out-Null

#load ConnectionInfo, contains SqlConnectionInfo class
[Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.
ConnectionInfo") |
Out-Null
```

4. Add the following script and run:

```
#create SqlConnectionInfo object,
#specifically required to run the traces
#need to specifically use the ConnectionInfoBase type
[Microsoft.SqlServer.Management.Common.ConnectionInfoBase]$conn =
New-Object Microsoft.SqlServer.Management.Common.SqlConnectionInfo
-ArgumentList "KERRIGAN"

$conn.UseIntegratedSecurity = $true

#create new TraceServer object
#The TraceServer class can start and read traces
$trcserver = New-Object -TypeName Microsoft.SqlServer.Management.
Trace.TraceServer
```

```
#need to get a handle to a Trace Template
#in this case we are using the Standard template
#that comes with Microsoft
$standardTemplate = "C:\Program Files (x86)\Microsoft SQL
Server\110\Tools\Profiler\Templates\Microsoft SQL Server\110\
Standard.tdf"

$trcserver.InitializeAsReader($conn,$standardTemplate) | Out-Null

$received = 0

#where do you want to write the trace?
#here we compose a timestamped file
$folder = "C:\Temp\"
$currdate = Get-Date -Format "yyyy-MM-dd_hmmtt"
$filename = "$($instanceName)_trace_{$currdate}.trc"
$outputtrace = Join-Path $folder $filename

#number of events to capture
$numevents = 10

#create new TraceFile object
#and initialize as writer
#The TraceFile class can read and write a Trace File
$trcwriter = New-Object Microsoft.SqlServer.Management.Trace.
TraceFile

$trcwriter.InitializeAsWriter($trcserver,$outputtrace) | Out-Null

while ($trcserver.Read())
{
    #write incoming trace to file
    $trcwriter.Write() | Out-Null
    $received++

    #we dont know how many columns are included
    #in the template so we will have to loop if we
    #want to capture and display all of them

    #get number of columns
    #we need to subtract 1 because column array
    #is zero-based, ie index starts at 0
    $cols = ($trcserver.FieldCount) -1
```

```
#we'll need to dynamically create a hash to
#contain the trace events
#because we need to dynamically build this hash
#based on number of columns included in a template,
#we'll have to store the code to build the hash
#as string first and then invoke expression
#to actually build the hash in PowerShell
$hashstr = "`$hash = `null; `n `$hash = @{ `n"
for($i = 0;$i -le $cols; $i++)
{
    $colname = $trcserver.GetName($i)

    #add each column to our hash
    #we will not capture the binary data
    if($colname -ne "BinaryData")
    {
        $colvalue = $trcserver.GetValue($trcserver.
GetOrdinal($colname))

        $hashstr += "`"$($colname)`"="`"$($colvalue)`" `n"
    }
}
$hashstr += "}"

#create the real hash
Invoke-Expression $hashstr

#display
$item = New-Object PSObject -Property $hash
$item | Format-List

if($received -ge $numevents)
{
    break
}
}

$trcwriter.Close()
$trcserver.Close()
```

What you should see in your **PowerShell ISE** results pane is a stream of events that are happening in SQL Server, much like what you would see if you were running **SQL Server Profiler**.

```
Duration      : 1111
ApplicationName : SQL Server Profiler - 77c05b3e-7fd3-446c-8
Reads         : 0
EventClass    : RPC:Completed
ClientProcessID : 124
NTUserName    : Administrator
SPID         : 51
StartTime     : 02/11/2012 14:06:32
CPU          : 0
TextData      : exec sp_trace_setstatus 9,1
Writes       : 0
LoginName     : KERRIGAN\Administrator
EndTime      : 02/11/2012 14:06:32
```

How it works...

This is a long recipe. There are quite a few things going on here. What we are doing is simulating what you can do and see with **SQL Server Profiler** using PowerShell. There will be cases where this will be useful and cases where **SQL Server Profiler** is still the right tool for the job. Regardless, it is good to know how to do it using PowerShell.

To start, it is important to use PowerShell ISE (x86), instead of the usual (x64) version we have been using in other recipes. The classes we need to use are only supported in 32-bit mode.

We first need to load a few extra libraries, `ConnectionInfo` and `ConnectionInfoExtended`, because we will need to pass these as arguments to the `TraceServer` class constructor when we are creating our `TraceServer` object.

```
#load ConnectionInfoExtended, this contains TraceFile class
[Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.
ConnectionInfoExtended") |
Out-Null

#load ConnectionInfo, contains SqlConnectionInfo class
[Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.
ConnectionInfo") |
Out-Null
```

Next, we need to create a `SqlConnectionInfo` connection object, which needs to be stored into a `ConnectionInfoBase` class.

```
#create SqlConnectionInfo object,
#specifically required to run the traces
[Microsoft.SqlServer.Management.Common.ConnectionInfoBase]
$conn = New-Object Microsoft.SqlServer.Management.Common.
SqlConnectionInfo -ArgumentList "KERRIGAN"

$conn.UseIntegratedSecurity = $true
```

There are a couple of Trace-specific classes we need to initialize. The first one is the `TraceServer`— which will enable us to start and read the traces.

```
#create new TraceServer object
#The TraceServer class can start and read traces
$trcserver = New-Object -TypeName Microsoft.SqlServer.Management.
Trace.TraceServer
```

We will need to initialize this as `Reader`, and we need to pass our connection object and the path to our Standard Trace Template:

```
#need to get a handle to a Trace Template
#in this case we are using the Standard template
#that comes with Microsoft
$standardTemplate = "C:\Program Files (x86)\Microsoft SQL Server\110\
Tools\Profiler\Templates\Microsoft SQL Server\110\Standard.tdf"

$trcserver.InitializeAsReader($conn,$standardTemplate) |
Out-Null
```

The goal of our recipe is to both start and read the trace, as well as write new trace events to a trace file. To achieve this, we need to create a `TraceFile` object, which allows for writing the Trace file.

```
#create new TraceFile object
#and initialize as writer
#The TraceFile class can read and write a Trace File
$trcwriter = New-Object Microsoft.SqlServer.Management.Trace.TraceFile

$trcwriter.InitializeAsWriter($trcserver,$outputtrace) |
Out-Null
```


Once the `TraceServer` and `TraceFile` objects are set up, we can start reading the trace. This will need to happen in a loop:

```
while ($trcserver.Read())
```

This start of the while loop will go on as long as there are events being captured by our `TraceServer` object.

Inside the loop, we do two things. The first one is we write these events to a trace file, using our `TraceFile` object called `$trcwriter`:

```
$trcwriter.Write()
```

The second thing we do is display the trace. For this particular exercise, we want to capture the events and be able to display them in a tabular fashion if we need to. To do this, we can store this event data in a hash, and display this before the end of the loop. This is a little bit challenging to do if you do not know which columns, and how many columns, are being captured. This will depend on the trace template you are using. To accommodate different templates, we'll determine first how many columns are being captured by the `TraceServer` object. Note that when we retrieve the columns from the `TraceServer`, the column index will start at zero, so we need to subtract one from the total number of columns to avoid any index out of bounds errors.

```
$cols = ($trcserver.FieldCount) -1
```

Based on the columns, we can dynamically build our hash. We can use the `GetName` method of the `TraceServer` object to get the name of the incoming column, and the `GetValue` and `GetOrdinal` methods of the `TraceServer` class to extract the value of the column coming in.

```
$hashstr = "`$hash = `null; `n `$hash = @{ `n"
for($i = 0;$i -le $cols; $i++)
{
    $colname = $trcserver.GetName($i)

    #add each column to our hash
    #we will not capture the binary data
    if($colname -ne "BinaryData")
    {
        $colvalue = $trcserver.GetValue($trcserver.
        GetOrdinal($colname))

        $hashstr += "`$($colname)`="`${$colvalue}`" `n"
    }
}
$hashstr += "}"
```

This is an example of the dynamically constructed hash code:

```
$hash = $null;
$hash = @{
  "EventClass"="Audit Logout"
  "TextData"=""
  "ApplicationName"="Report Server"
  "NTUserName"="sqlservice"
  "LoginName"="QUERYWORKS\sqlservice"
  "CPU"="0"
  "Reads"="36"
  "Writes"="0"
  "Duration"="7950000"
  "ClientProcessID"="2032"
  "SPID"="52"
  "StartTime"="02/11/2012 11:52:55"
  "EndTime"="02/11/2012 11:53:03"
}
```

We then take this dynamically created code to create the actual hash using the `Invoke-Expression` cmdlet:

```
Invoke-Expression $hashstr
```

Once the hash is created, we can display it on the screen:

```
#display
$item = New-Object PSObject -Property $hash
$item | Format-List
```

When done with our loop, we need to close both the `TraceServer` and `TraceFile` handles:

```
$trcwritter.Close()
$trcserver.Close()
```

See also

- ▶ *The Extracting the contents of a trace file recipe*
- ▶ Check out this article in MSDN called *Trace and Replay Objects: A New API for SQL Server Tracking and Replay* ([http://msdn.microsoft.com/en-us/library/ms345134\(v=sql.90\).aspx](http://msdn.microsoft.com/en-us/library/ms345134(v=sql.90).aspx)).

It is a little bit outdated, but is still very relevant if you want to programmatically work with traces using .NET languages.

Extracting the contents of a trace file

In this recipe, we will extract the contents of a trace file (.trc) using PowerShell.

Getting ready

We will need to use the x86 version of PowerShell and/or PowerShell ISE for this recipe. This is unfortunate, but some of the classes we need to use are only supported in 32-bit mode.

In this recipe, we will use a previously saved trace (.trc) file. Feel free to substitute this with a trace file that you have available.

How to do it...

Let's look at how we can extract the contents of a trace file.

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE (x86)**.

2. Import the SQLPS module, and create a new SMO Server object:

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

3. Import additional libraries. These are needed to use our TraceFile and TraceServer classes. We do this as follows:

```
#load ConnectionInfoExtended, this contains TraceFile class
[Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.
ConnectionInfoExtended") |
Out-Null
```

```
#load ConnectionInfo, contains SqlConnectionInfo class
[Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.
ConnectionInfo") |
Out-Null
```

4. Add the following script and run:

```
#replace this with your own filename
$path = "C:\Temp\KERRIGAN_trace_2012-02-11_206PM.trc"
$trcreader = New-Object Microsoft.SqlServer.Management.Trace.
TraceFile
$trcreader.InitializeAsReader($path)
```

```

#extract all
$result = @()
if($trcreader.Read())
{
    while($trcreader.Read())
    {
        #let's extract only the ones that
        #took more than 1000ms
        $duration = $trcreader.GetValue($trcreader.
GetOrdinal("Duration"))

        if($duration -ge 1000)
        {
            $cols = ($trcreader.FieldCount) -1
            #we need to dynamically build the hash string
            #because we don't know how many columns
            #are in the incoming trace file
            $hashstr = "`$hash = @{ `n"
            for($i = 0;$i -le $cols; $i++)
            {
                $colname = $trcreader.GetName($i)
                #don't include binary data
                if($colName -ne "BinaryData")
                {
                    $colvalue = $trcreader.GetValue($trcreader.
GetOrdinal($colname))
                    $hashstr += "`"$($colname)`"="`"$($colvalue)`" `n"
                }
            }
            $hashstr += "`"}"

            #create the real hash
            Invoke-Expression $hashstr

            $item = New-Object PSObject -Property $hash
            $result += $item
        }
    }
}
#display
$result | Format-List

```

Once the script finishes executing, the results on your screen should look like this:

```
Duration      : 1111
ApplicationName : SQL Server Profiler - 77c05b
Reads         : 0
EventClass    : RPC:Completed
ClientProcessID : 124
NTUserName    : Administrator
SPID          : 51
StartTime     : 02/11/2012 14:06:32
CPU           : 0
TextData      : exec sp_trace_setstatus 9,1
Writes        : 0
LoginName     : KERRIGAN\Administrator
EndTime       : 02/11/2012 14:06:32

Duration      : 2390000
ApplicationName : Report Server
Reads         : 506
EventClass    : Audit Logout
ClientProcessID : 2004
NTUserName    : sqlservice
SPID          : 60
StartTime     : 02/11/2012 14:06:37
CPU           : 0
TextData      :
Writes        : 0
LoginName     : QUERYWORKS\sqlservice
EndTime       : 02/11/2012 14:06:39
```

How it works...

To extract the contents of a trace file (.trc), we first need to load a few extra libraries, `ConnectionInfo` and `ConnectionInfoExtended`. These contain the `TraceFile` class we need to use in this recipe.

```
#load ConnectionInfoExtended, this contains TraceFile class
[Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.
ConnectionInfoExtended") |
Out-Null

#load ConnectionInfo, contains SqlConnectionInfo class
[Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.
ConnectionInfo") |
Out-Null
```

We then need to create a `TraceFile` object, initialized as a reader:

```
#replace this with your own filename
$path = "C:\Temp\KERRIGAN_trace_2012-02-11_206PM.trc"

$strcreader = New-Object Microsoft.SqlServer.Management.Trace.TraceFile
$strcreader.InitializeAsReader($path)
```

To read all the contents, we need to put the reader in a while loop, and keep on iterating while there are events in the trace file to be read:

```
while($strcreader.Read())
```

In our recipe, we only cared about any events that had a duration of over 1000 ms.

```
#let's extract only the ones that took more than 1000ms
$duration = $strcreader.GetValue($strcreader.GetOrdinal("Duration"))
```

The `GetOrdinal` method of the `TraceFile` class allows you to get the *n*th column in which `Duration` is. Using this, we can pass it to the `GetValue` method of the `TraceFile` class to extract the value in that column position.

Also note that in our recipe, we extract all the columns except the `BinaryData` in the trace file. We do this by looping through all the columns, and putting them into a hash we dynamically build:

```
#we need to dynamically build the hash string
#because we don't know how many columns are in the
#incoming trace file
$hashstr = "`$hash = @{ `n"
for($i = 0;$i -le $cols; $i++)
{
    $colname = $strcreader.GetName($i)
    #don't include binary data
    if($colName -ne "BinaryData")
    {
        $colvalue = $strcreader.GetValue($strcreader.GetOrdinal($colname))
        $hashstr += "`"$($colname)`"="`"$($colvalue)`" `n"
    }
}
$hashstr += "}"
```

This is an example of the dynamically constructed hash code:

```
$hash = $null;
$hash = @{
  "EventClass"="Audit Logout"
  "TextData"=""
  "ApplicationName"="Report Server"
  "NTUserName"="sqlservice"
  "LoginName"="QUERYWORKS\sqlservice"
  "CPU"="0"
  "Reads"="36"
  "Writes"="0"
  "Duration"="7950000"
  "ClientProcessID"="2032"
  "SPID"="52"
  "StartTime"="02/11/2012 11:52:55"
  "EndTime"="02/11/2012 11:53:03"
}
```

Once the hash string is built, we can use the `Invoke-Expression` cmdlet to create the real hash.

```
#create the real hash
Invoke-Expression $hashstr
```

We then store this to an array, which we display after the loop is finished:

```
$item = New-Object PSObject -Property $hash
$result += $item

    }
}
}
#display
$result | Format-List
```

An alternative to dynamically building the hash is explicitly identifying which columns you want included in the hash. This is doable only if you are familiar with the template used when capturing the trace file. The syntax you would use would be similar to this:

```
$hash = @{
  "EventClass"=$trcreader.GetValue($trcreader.GetOrdinal("EventClass"))
  "TextData"=$trcreader.GetValue($trcreader.GetOrdinal("TextData"))
  "Duration"=$trcreader.GetValue($trcreader.GetOrdinal("Duration"))
}
$item = New-Object PSObject -Property $hash
$result += $item
```

See also

- ▶ *The Running and saving a profiler trace event recipe*

Creating a database master key

In this recipe, we will create a database master key.

Getting ready

We will create a database master key for the master database in this recipe. You can substitute a different database for this exercise if you wish.

The T-SQL equivalent of what we are trying to accomplish is:

```
USE master
GO
CREATE MASTER KEY ENCRYPTION
BY PASSWORD = 'P@ssword'
```

How to do it...

Let's list the steps required to complete the task:

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 as follows:

```
#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:

```
$VerbosePreference = "Continue"
$masterdb = $server.Databases["master"]
```



```
if($masterdb.MasterKey -eq $null)
{
    $masterkey = New-Object Microsoft.SqlServer.Management.Smo.
MasterKey -ArgumentList $masterdb
    $masterkey.Create("P@ssword")
    Write-Verbose "Master Key Created : $($masterkey.CreateDate)"
}
$VerbosePreference = "SilentlyContinue"
```

If successful, in your output, you should see a one-line message containing the success message, and the date on which the master key was created.

How it works...

A database master key is required if you want to do any database-level encryption. It is used to encrypt keys and certificates in a specific database.

Creating a database master key is straightforward. You need to create an SMO `MasterKey` object:

```
$masterkey = New-Object Microsoft.SqlServer.Management.Smo.MasterKey
-ArgumentList $masterdb

$masterkey.Create("P@ssword")
```

There are a couple of overloads to the `Create` method of the `MasterKey` class. In our recipe, we chose to provide a single password. The alternative is to pass both a decryption and encryption password.

If the database master key already exists, you may not necessarily be able to drop it right away. If there are encryption objects already created that are being protected by the database master key, you must drop those encryption objects first before you can drop the database master key. Once there are no more dependent objects, you can use the following PowerShell code to drop the master key:

```
#drop master key
$masterkey.Drop()
```

See also

- ▶ The *Creating a certificate* recipe
- ▶ You can learn more about the `MasterKey` class from MSDN:

[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.masterkey\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.masterkey(v=sql.110).aspx)

Creating a certificate

This recipe demonstrates how you can create a certificate using PowerShell and SMO.

Getting ready

In this recipe, we will create a certificate called `Test Certificate`, protected by the database master key. You will need to make sure that the database master key has been created first for the database.

The T-SQL equivalent of what we are trying to accomplish is:

```
CREATE CERTIFICATE [Test Certificate]
WITH SUBJECT = N'This is a test certificate.',
START_DATE = N'02/10/2012',
EXPIRY_DATE = N'01/01/2015'
```

How to do it...

Let's list the steps required to complete the task.

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:

```
$certificateName = "Test Certificate"
$masterdb = $server.Databases["master"]

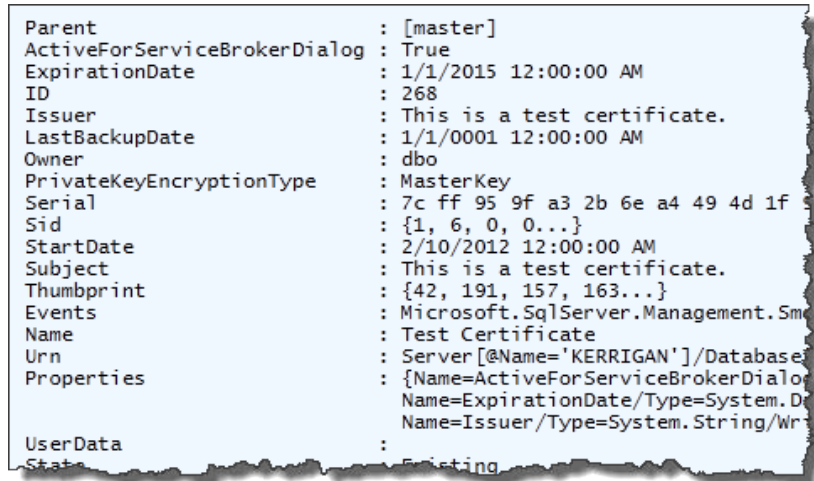
if ($masterdb.Certificates[$certificateName])
{
    $masterdb.Certificates[$certificateName].Drop()
}
$certificate = New-Object -TypeName Microsoft.SqlServer.
Management.Smo.Certificate -ArgumentList $masterdb,
$certificateName
```

```
#set properties
$certificate.StartDate = "February 10, 2012"
$certificate.Subject = "This is a test certificate."
$certificate.ExpirationDate = "January 01, 2015"

#create certificate
#you can optionally provide a password, but this
#certificate we created is protected by the master key
$certificate.Create()

#display all properties
$certificate | Select *
```

When the certificate is created and the script is done executing, the resulting screen will look similar to the following screenshot:



4. To confirm this via T-SQL, we can use the `sys.certificates` DMV to list all certificates. Open SQL Server Management Studio, and execute the following T-SQL statement:

```
SELECT *
FROM sys.certificates
WHERE [name] = 'Test Certificate'
```

How it works...

To create a certificate, you need to first create an SMO Certificate object:

```
$certificate = New-Object -TypeName Microsoft.SqlServer.Management.  
Smo.Certificate -ArgumentList $masterdb, $certificateName
```

There are a few properties we can set for an SMO `Certificate` object. In this recipe, we set the `StartDate`, `Subject`, and `ExpirationDate` values:

```
$certificate.StartDate = "February 10, 2012"  
$certificate.Subject = "This is a test certificate."  
$certificate.ExpirationDate = "January 01, 2015"
```

If you want to create a certificate that is protected by the database master key, you can just invoke the `Create` method of the `Certificate` class. You can optionally provide a password:

```
$certificate.Create()
```

There's more...

A certificate is essentially a digitally signed document that binds a public key with an identity, and is used to prove authenticity of ownership. This helps prevent malicious impersonations, in other words, somebody or something pretending to be someone or something they are not.

Learn more about certificates from MSDN:

[http://msdn.microsoft.com/en-us/library/ms189586\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms189586(v=sql.110).aspx)

See also

- ▶ The *Creating a database master key* recipe

Creating symmetric and asymmetric keys

In this recipe, we will create symmetric and asymmetric keys.

Getting ready

In this recipe, we will use the `TestDB` database. If you don't already have this database, log in the SQL Server Management Studio and execute the following T-SQL code:

```
IF DB_ID('TestDB') IS NULL  
CREATE DATABASE TestDB  
GO
```

We will also need a user called `eric` in our `TestDB` database. This user will map to the SQL login `eric`. Feel free to create this user using the *Creating a database user* recipe. Alternatively, execute the following T-SQL code from SQL Server Management Studio:

```
Use TestDB
GO
CREATE USER [eric]
FOR LOGIN [eric]
```

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 as follows:

```
#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 handle
$databasename = "TestDB"
$database = $server.Databases[$databasename]

#=====
# Create Database Master Key
#=====
#this is equivalent to:
<#
USE TestDB
GO
CREATE MASTER KEY ENCRYPTION
BY PASSWORD = 'P@ssword'
#>
#create (user) database master key
#if this doesn't exist yet
$dbmk = New-Object Microsoft.SqlServer.Management.Smo.MasterKey
-ArgumentList $database
$dbmk.Create("P@ssword")
```

```
#=====
# Create Asymmetric Key
#=====
#this is equivalent to:
<#
USE TestDB
GO
CREATE ASYMMETRIC KEY [EncryptionAsymmetricKey]
AUTHORIZATION [eric]
WITH ALGORITHM = RSA_2048
#>
$asymk = New-Object Microsoft.SqlServer.Management.Smo.
AsymmetricKey -ArgumentList $database, "EncryptionAsymmetricKey"

#replace this with a known database user in the
#database you are using for this recipe
$asymk.Owner = "eric"
$asymk.Create([Microsoft.SqlServer.Management.Smo.AsymmetricKeyEnc
ryptionAlgorithm]::Rsa2048)

#=====
# Create Symmetric Key
#=====
#this is equivalent to :
<#
CREATE CERTIFICATE [Encryption]
WITH SUBJECT = N'This is a test certificate.',
START_DATE = N'02/10/2012',
EXPIRY_DATE = N'01/01/2015'
#>

#create certificate first to be used for Symmetric Key
$cert = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Certificate -ArgumentList $database, "Encryption"
$cert.StartDate = "February 10, 2012"
$cert.Subject = "This is a test certificate."
$cert.ExpirationDate = "January 01, 2015"
$cert.Create()

#create a symmetric key based on certificate
#this is equivalent to :
```

```

<#
CREATE SYMMETRIC KEY [EncryptionSymmetricKey]
WITH ALGORITHM = TRIPLE_DES
ENCRYPTION BY CERTIFICATE [Encryption]
#>
$symk = New-Object Microsoft.SqlServer.Management.Smo.
SymmetricKey -ArgumentList $database, "EncryptionSymmetricKey"
$symkenc = New-Object Microsoft.SqlServer.Management.Smo.
SymmetricKeyEncryption ([Microsoft.SqlServer.Management.Smo.
KeyEncryptionType]::Certificate, "Encryption")
$symk.Create($symkenc, [Microsoft.SqlServer.Management.Smo.
SymmetricKeyEncryptionAlgorithm]::TripleDes)

#list each object we created
$dbmk.Parent
$cert.Name
$asymk
$symk

```

The resulting screen should look similar to the following:

```

CreateDate       : 2/10/2012 2:46:34 PM
DateLastModified : 2/10/2012 2:46:34 PM
IsEncryptedByServer : True
IsOpen           : False
Parent           : [TestDB]
Urn               : Server[[@Name='KERRIGAN']/Databases]
Properties        : {Name=CreateDate/Type=System.DateTime
                    Name=DateLastModified/Type=System.DateTime
                    Name=IsEncryptedByServer/Type=System.Boolean
                    Name=IsOpen/Type=System.Boolean}
UserData         :
State            : Existing

Name : Encryption

Name : EncryptionAsymmetricKey

Name : EncryptionSymmetricKey

```

Alternatively, you can use the following T-SQL statement to confirm the existence of the database master key, certificate, symmetric, and asymmetric keys we created in this recipe:

```

SELECT 'DB Master Key' ,
       is_master_key_encrypted_by_server
FROM   sys.databases
WHERE  [name] = 'TestDB'

SELECT 'Certificate' , *
FROM   sys.certificates

```

```

WHERE [name] = 'Encryption'

SELECT 'Asymmetric Key' , *
FROM sys.asymmetric_keys
WHERE [name] = 'EncryptionAsymmetricKey'

SELECT 'Symmetric Key' , *
FROM sys.symmetric_keys
WHERE [name] = 'EncryptionSymmetricKey'

```

How it works...

Before we can create a symmetric or asymmetric key, we have to first create a database master key. MSDN defines a database master key as follows:

a symmetric key used to protect the private keys of certificates and asymmetric keys that are present in the database.

Consider the following code for creating the master key:

```

$dbmk = New-Object Microsoft.SqlServer.Management.Smo.MasterKey
-ArgumentList $database
$dbmk.Create("P@ssword")

```

Once the database master key is in place, we can create our symmetric and asymmetric keys.

To create the asymmetric key, you need to create an SMO asymmetric key instance, and assign an owner and encryption algorithm. The available `AsymmetricKeyEncryptionAlgorithm` values are `CryptographicProviderDefined`, `Rsa512`, `Rsa1024`, and `Rsa2048`.

```

$asymk = New-Object Microsoft.SqlServer.Management.Smo.AsymmetricKey
-ArgumentList $database, "EncryptionAsymmetricKey"
#replace this with a known user in your instance
$asymk.Owner = "EncryptionUser"
$asymk.Create([Microsoft.SqlServer.Management.Smo.
AsymmetricKeyEncryptionAlgorithm]::Rsa2048)

```

To create a symmetric key, we must first create a certificate:

```

#create certificate first to be used for Symmetric Key
$cert = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Certificate -ArgumentList $database, "Encryption"
$cert.StartDate = "February 10, 2012"
$cert.Subject = "This is a test certificate."
$cert.ExpirationDate = "January 01, 2015"
$cert.Create()

```


To create a symmetric key based on the certificate, we should first instantiate an SMO `SymmetricKey` object:

```
$symk = New-Object Microsoft.SqlServer.Management.Smo.SymmetricKey  
-ArgumentList $database, "EncryptionSymmetricKey"
```

We then need to specify the `SymmetricKey` encryption type. The available values are `SymmetricKey`, `Certificate`, `Password`, `AsymmetricKey`, and `Provider`.

```
$symkenc = New-Object Microsoft.SqlServer.Management.Smo.  
SymmetricKeyEncryption ([Microsoft.SqlServer.Management.Smo.  
KeyEncryptionType]::Certificate, "Encryption")
```

When we create the `SymmetricKey`, we must also specify which algorithm to use. The available `SymmetricKeyAlgorithm` values are `CryptographicProviderDefined`, `RC2`, `RC4`, `Des`, `TripleDes`, `DesX`, `Aes128`, `Aes192`, `Aes256`, and `TripleDes3Key`:

```
$symk.Create($symkenc, [Microsoft.SqlServer.Management.Smo.  
SymmetricKeyEncryptionAlgorithm]::TripleDes)
```

There's more...

Symmetric and asymmetric keys can be used to set up cell-level encryption in SQL Server. The typical steps to setting up cell-level encryption are:

- ▶ Create master key
- ▶ Create certificate or asymmetric key
- ▶ Create symmetric key protected by certificate or asymmetric key
- ▶ Open symmetric key—encrypt or decrypt—close symmetric key

You can learn more about symmetric and asymmetric keys from this MSDN article:

<http://support.microsoft.com/kb/246071>

Another MSDN article that walks you through how you can encrypt a column of data using T-SQL is [http://msdn.microsoft.com/en-us/library/ms179331\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms179331(v=sql.110).aspx).

See also

- ▶ *The Creating a database master key recipe*
- ▶ *The Creating a certificate recipe*

Setting up Transparent Data Encryption (TDE)

This recipe shows how you can set up Transparent Data Encryption using PowerShell and SMO.

Getting ready

In this recipe, we will enable **Transparent Data Encryption (TDE)** on the `TestDB` database. If you don't already have this test database, log in the SQL Server Management Studio and execute the following T-SQL code:

```
IF DB_ID('TestDB') IS NULL
CREATE DATABASE TestDB
GO
```

You must already have a database master key for this `TestDB` database. If not, create it using the *Creating a database master key* recipe.

How to do it...

These are the steps to set up Transparent Data Encryption (TDE) programmatically:

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 as follows:

```
#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:

```
$databasename = "TestDB"
$database = $server.Databases[$databasename]

#if not yet created, create or obtain a certificate
#protected by the master key
#this is equivalent to
```

```
<#
USE master
GO
CREATE CERTIFICATE [Encryption]
WITH SUBJECT = N'This is a test certificate.',
START_DATE = N'02/10/2012',
EXPIRY_DATE = N'01/01/2015'
#>
$certificateName = "Test Certificate"
$masterdb = $server.Databases["master"]

if ($masterdb.Certificates[$certificateName])
{
    $masterdb.Certificates[$certificateName].Drop()
}
$certificate = New-Object -TypeName Microsoft.SqlServer.
Management.Smo.Certificate -ArgumentList $masterdb,
$certificateName

#create certificate protected by the master key
$certificate.StartDate = "February 10, 2012"
$certificate.Subject = "This is a test certificate."
$certificate.ExpirationDate = "January 01, 2015"

#you can optionally provide a password, but this
#certificate we created is protected by the master key
$certificate.Create()

#create a database encryption key
#this is equivalent to
<#
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_256
ENCRYPTION BY SERVER CERTIFICATE [Test Certificate]
#>
$dbencryption = New-Object Microsoft.SqlServer.Management.Smo.
DatabaseEncryptionKey
$dbencryption.Parent = $database
$dbencryption.EncryptionAlgorithm = [Microsoft.SqlServer.
Management.Smo.DatabaseEncryptionAlgorithm]::Aes256
```

```

$dbencryption.EncryptionType = [Microsoft.SqlServer.Management.
Smo.DatabaseEncryptionType]::ServerCertificate

#associate certificate name
$dbencryption.EncryptorName = $certificateName
$dbencryption.Create()

#enable TDE
#this is equivalent to :
<#
ALTER DATABASE [TestDB]
SET ENCRYPTION ON
#>
$database.EncryptionEnabled = $true
$database.Alter()
$database.Refresh()

#display TDE setting
$database.EncryptionEnabled

```

The resulting screen should look similar to the following:

```

PS SQLSERVER:\>
#display TDE setting
$database.EncryptionEnabled
True

```

The final line should say **True**, if Transparent Data Encryption was successfully turned on for TestDB.

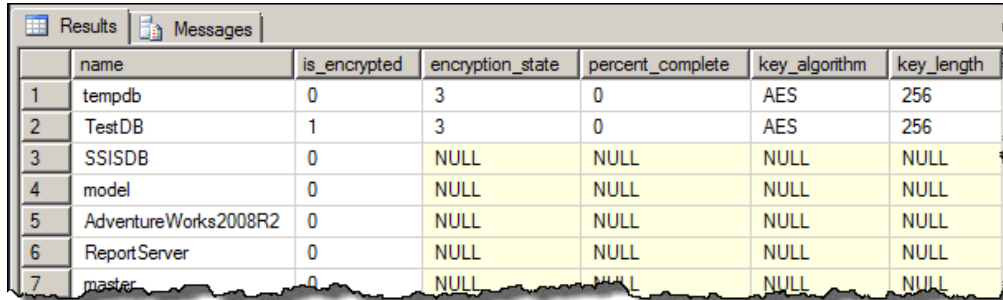
- Alternatively you can use the following T-SQL statement to confirm:

```

SELECT  db.name ,
        db.is_encrypted ,
        dm.encryption_state ,
        dm.percent_complete ,
        dm.key_algorithm ,
        dm.key_length
FROM    sys.databases db
        LEFT OUTER JOIN sys.dm_database_encryption_keys dm
        ON db.database_id = dm.database_id

```

This should give you a result similar to the following:



	name	is_encrypted	encryption_state	percent_complete	key_algorithm	key_length
1	tempdb	0	3	0	AES	256
2	TestDB	1	3	0	AES	256
3	SSISDB	0	NULL	NULL	NULL	NULL
4	model	0	NULL	NULL	NULL	NULL
5	AdventureWorks2008R2	0	NULL	NULL	NULL	NULL
6	ReportServer	0	NULL	NULL	NULL	NULL
7	master	0	NULL	NULL	NULL	NULL

The **encryption_state = 3** means encryption of that database has already completed. Notice also that `tempdb` is also encrypted. By default, if any user databases are encrypted, `tempdb` also automatically gets encrypted.

How it works...

There are a few preparatory steps required to enable Transparent Data Encryption (TDE).

You first need to create a master key. You will then need to create a certificate stored in the master database, and protected by the database master key for the master database.

```
$certificate = New-Object -TypeName Microsoft.SqlServer.Management.Smo.Certificate -ArgumentList $masterdb, $certificateName
```

```
#create certificate protected by the master key
$certificate.StartDate = "February 10, 2012"
$certificate.Subject = "This is a test certificate."
$certificate.ExpirationDate = "January 01, 2015"
```

```
#you can optionally provide a password, but this
#certificate we created is protected by the master key
$certificate.Create()
```

The next step is to create a database encryption key protected by the certificate. This key is needed for transparently encrypting a user database.

```
#create a database encryption key
$dbencryption = New-Object Microsoft.SqlServer.Management.Smo.DatabaseEncryptionKey
```

We need to associate this with the database for which we want to turn on TDE.

```
$dbencryption.Parent = $database
```

When creating a database encryption key, we also need to specify the encryption algorithm. The available encryptions are `Aes128`, `Aes192`, `Aes256`, and `TripleDes`.

```
$dbencryption.EncryptionAlgorithm = [Microsoft.SqlServer.Management.Smo.DatabaseEncryptionAlgorithm]::Aes256
```

We also need to associate this key with the certificate we previously created. The possible `DatabaseEncryptionType` values are `ServerCertificate` and `ServerAsymmetricKey`.

```
$dbencryption.EncryptionType = [Microsoft.SqlServer.Management.Smo.DatabaseEncryptionType]::ServerCertificate
```

```
#associate certificate name
$dbencryption.EncryptorName = $certificateName
```

You are now ready to create the database encryption key:

```
$dbencryption.Create()
```

At this point, the preparatory steps are complete. We can now turn on TDE, and alter our target database.

```
#enable TDE
$database.EncryptionEnabled = $true
$database.Alter()
$database.Refresh()
```

There's more...

Transparent Data Encryption (TDE) is introduced in SQL Server 2008 as a solution for database-level encryption. If TDE is turned on, data in the data and log files are encrypted. This will also automatically encrypt `tempdb`.

See also

- ▶ The *Creating a certificate* recipe
- ▶ The *Altering database properties* recipe in *Chapter 2, SQL Server and PowerShell Basic Tasks*
- ▶ Read more about Transparent Data Encryption from MSDN:
[http://msdn.microsoft.com/en-us/library/bb934049\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/bb934049(v=sql.110).aspx)
- ▶ Check out `encryption_state` values of `sys.dm_database_encryption_keys` from MSDN:
<http://msdn.microsoft.com/en-us/library/bb677274.aspx>

6

Backup and Restore

In this chapter, we will cover:

- ▶ Changing database recovery model
- ▶ Listing backup history
- ▶ Creating a backup device
- ▶ Listing backup header and file list information
- ▶ Creating a full backup
- ▶ Creating a backup on mirrored media sets
- ▶ Creating a differential backup
- ▶ Creating a transaction log backup
- ▶ Creating a filegroup backup
- ▶ Restoring a database to a point in time
- ▶ Performing an online piecemeal restore

Introduction

Knowing how to back up and restore a database is one of the most fundamental skills you need to have when managing your database environment.

There are different ways to do backup and restore. It can be done through SQL Server Management Studio, by using stored procedures, or through SSIS. And now, these tasks can be done with PowerShell. The key is to determine which tool is best suited for the particular task.

Doing the backups and restores using PowerShell has its own advantages, including being able to automate backups across multiple servers, being able to retrieve, consolidate, and filter all backup histories if needed. It is even easier to do these tasks in SQL Server 2012 because of additional cmdlets for backup and restore. It also gives you access to the full power of SMO should you need to add additional parameters.

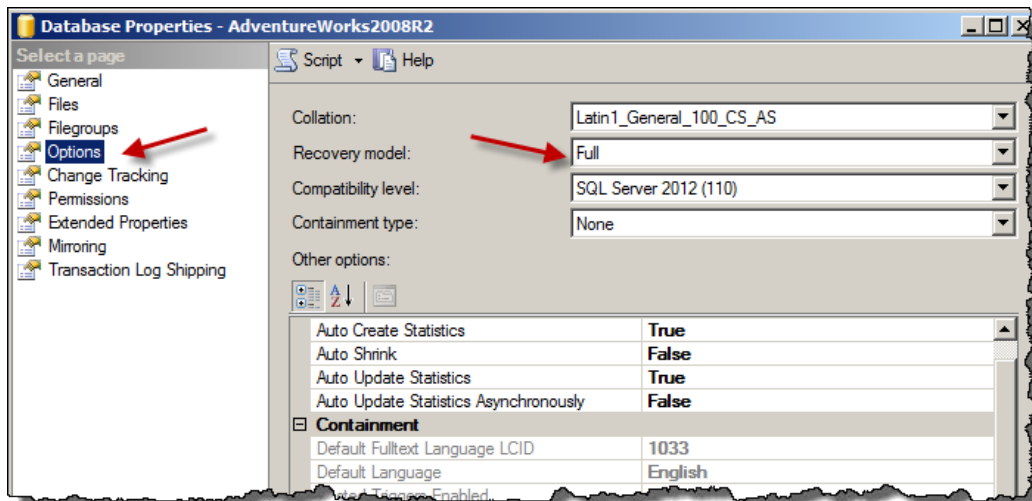
Changing database recovery model

In this recipe, we will explore how to change SQL Server recovery model using PowerShell.

Getting ready

We will use **AdventureWorks2008R2** in this exercise, and change the recovery model from **Full** to **Simple**. Feel free to substitute this with a database of your choice.

Check what SQL Server recovery model your instance is set to, using SSMS. Open your **Object Explorer** and right-click on the database you chose and click on **Properties | Options**:



If your database is set to either **Simple** or **Bulk-logged**, change this to **Full** and click on **OK**. Since we will be using **AdventureWorks2008R2** in later exercises, we need to change this recovery model back to **Full** after this exercise.

How to do it...

- Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

- Import the SQLPS module as follows:

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

- Add the following script and run:

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

$databasename = "AdventureWorks2008R2"
$database = $server.Databases[$databasename]

#possible values for RecoveryModel are
#Full, Simple and BulkLogged
$database.DatabaseOptions.RecoveryModel = [Microsoft.SqlServer.
Management.Smo.RecoveryModel]::Simple
$database.Alter()
$database.Refresh()

#list Recovery Model again
$database.DatabaseOptions.RecoveryModel

#remember to change the recovery model back
#to full for the next recipes
```

How it works...

To change a database's `RecoveryModel` property, get a handle to that database first:

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

Once you have the handle, use the `DatabaseOptions` property of the database object to set the `RecoveryModel` property to `Simple`:

```
#possible values for RecoveryModel are
#Full, Simple and BulkLogged
$database.DatabaseOptions.RecoveryModel = [Microsoft.SqlServer.
Management.Smo.RecoveryModel]::Simple
$database.Alter()
$database.Refresh()
```

There's more...

`RecoveryModel` is a database property that specifies what backup and restore operations are permitted. There are three possible values for `RecoveryModel`: `Full`, `BulkLogged`, and `Simple`.

`Full` and `BulkLogged` recovery models allow the use of logfiles for backup and restore purposes. The `Full` recovery model heavily uses the transaction logfiles, and allows for point-in-time recovery.

The `BulkLogged` recovery model minimally logs the bulk events. If there are no bulk events in the system, then point-in-time recovery is possible. If there are bulk events, however, point-in-time recoverability will be affected, and it is possible not to be able to recover from your logfiles at all. See Paul Randal's blog post on *A SQL Server DBA myth a day: (28/30) BULK_LOGGED recovery model*:

[http://www.sqlskills.com/BLOGS/PAUL/post/A-SQL-Server-DBA-myth-a-day-\(2830\)-BULK_LOGGED-recovery-model.aspx](http://www.sqlskills.com/BLOGS/PAUL/post/A-SQL-Server-DBA-myth-a-day-(2830)-BULK_LOGGED-recovery-model.aspx)

The `Simple` recovery model does not support transaction log backups and restores at all. This means that there is no point-in-time recovery possible, and the window for data loss could be large. `Simple` recovery model, therefore, is not a recommended setting for production servers; it can be a setting used for development and sandbox servers, or any instance where data loss would not be critical.

The `RecoveryModel` you choose in your environment will typically be determined by the company's **Recovery Point Objective (RPO)** and **Recovery Time Objective (RTO)**, although in most cases the recommended setting would be `Full` recovery model.

Read more about `RecoveryModel` from MSDN:

[http://msdn.microsoft.com/en-us/library/ms189275\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms189275(v=sql.110).aspx)

See also

- ▶ The *Altering database properties* recipe in *Chapter 2, SQL Server and PowerShell Basic Tasks*

Listing backup history

In this recipe, we will list the backup history for a SQL Server instance.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the SQLPS module as follows:

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

3. Add the following script and run:

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

#display date of last backup
$server.Databases |
Select Name, RecoveryModel, LastBackupDate,
LastDifferentialBackupDate, LastLogBackupDate |
Format-Table -AutoSize
```

Your result should look similar to the following screenshot:

Name	RecoveryModel	LastBackupDate	LastDifferentialBackupDate	LastLogBackupDate
AdventureWorks2008R2	Full	2/26/2012 7:14:16 PM	2/26/2012 7:14:29 PM	2/26/2012 7:15:00 PM
master	Simple	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM
model	Full	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM
msdb	Simple	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM
ReportServer	Full	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM
ReportServerTempDB	Simple	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM
SampleEncryption	Full	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM
StudentDB	Full	2/26/2012 5:48:30 PM	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM
tempdb	Simple	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM	1/1/0001 12:00:00 AM
TestDB	Full	2/26/2012 12:32:44 PM	2/26/2012 8:49:34 AM	2/26/2012 8:55:30 AM

Note that when you see a date of **1/1/0001 12:00:00 AM**, then it means no backup has ever been taken for that database.

How it works...

Listing the backup history is a simple task, using a little bit of PowerShell and SMO. After you get a database handle, you can display the last backup dates onto the screen.

```
#display days ago since last backup
$server.Databases |
```

```
Select Name, RecoveryModel, LastBackupDate,  
LastDifferentialBackupDate, LastLogBackupDate |  
Format-Table -AutoSize
```

Alternatively, you can capture this in a file, or a table, whichever your requirements specify.

See also

- ▶ The *Listing SQL Server jobs* recipe in *Chapter 3, Basic Administration*

Creating a backup device

This recipe shows how you can create a backup device using PowerShell.

Getting ready

We are going to create a backup device in this recipe. The equivalent T-SQL of what we are trying to accomplish is:

```
EXEC master.dbo.sp_addumpdevice @devtype = N'disk',  
    @logicalname = N'Full Backups',  
    @physicalname = N'C:\Backup\backupfile.bak'
```

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the `SQLPS` module as follows:

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

3. Add the following script and run:

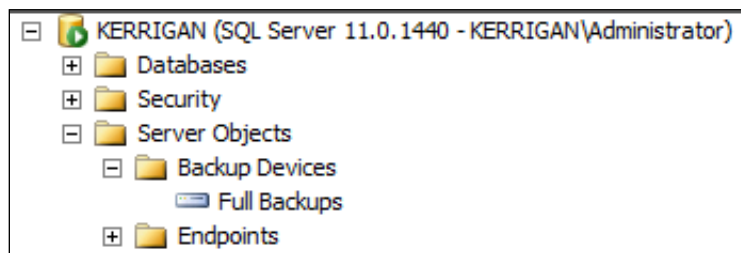
```
$instanceName = "KERRIGAN"  
$server = New-Object -TypeName Microsoft.SqlServer.Management.  
Smo.Server -ArgumentList $instanceName  
#this file will be created by PowerShell/SMO  
$backupfilename = "Full Backups"  
$backupfile = "C:\Backup\backupfile.bak"
```

```
$backupdevice = New-Object Microsoft.SqlServer.Management.Smo.  
BackupDevice($server,$backupfilename)
```

```
#BackupDeviceType values are:
#CDRom, Disk, FloppyA, FloppyB, Tape, Pipe, Unknown
$backupdevice.BackupDeviceType = [Microsoft.SqlServer.Management.Smo.BackupDeviceType]::Disk
$backupdevice.PhysicalLocation = $backupfile
$backupdevice.Create()
```

```
#list backup devices
$server.BackupDevices
```

4. Confirm by using SQL Server Management Studio. Log in to your instance and expand **Backup Devices**. You should see the new backup device you created in PowerShell.



How it works...

A backup device is a layer of abstraction that allows you to reference a backup medium—be it a file, a network share, or a tape—using a logical name instead of specifying the full physical path.

To create a backup device using PowerShell and SMO, you will need to first create a handle to an SMO BackupDevice object:

```
$backupdevice = New-Object Microsoft.SqlServer.Management.Smo.BackupDevice($server,$backupfilename)
```

You will also need to specify BackupDeviceType, and the physical location of the media. BackupDeviceType can be one of CDRom, Disk, FloppyA, FloppyB, Tape, Pipe, and Unknown. This is illustrated in the following code:

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

$backupdevice.BackupDeviceType = [Microsoft.SqlServer.Management.Smo.BackupDeviceType]::Disk
$backupdevice.PhysicalLocation = $backupfile
$backupdevice.Create()
```

See also

- ▶ The *Listing backup header information and file list information* recipe
- ▶ Read up on backup devices:
[http://msdn.microsoft.com/en-us/library/ms179313\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms179313(v=sql.110).aspx)

Listing backup header and file list information

In this recipe, we will look at listing backup header information from a backup file.

Getting ready

For this task, we will look at listing an existing backup's header information.



If you do not have any backups in your system yet, you can do any of this chapter's backup recipes prior to performing this recipe.

How to do it...

To list the header information, follow these steps:

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the SQLPS module as follows:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking
```
3. Add the following script and run:

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

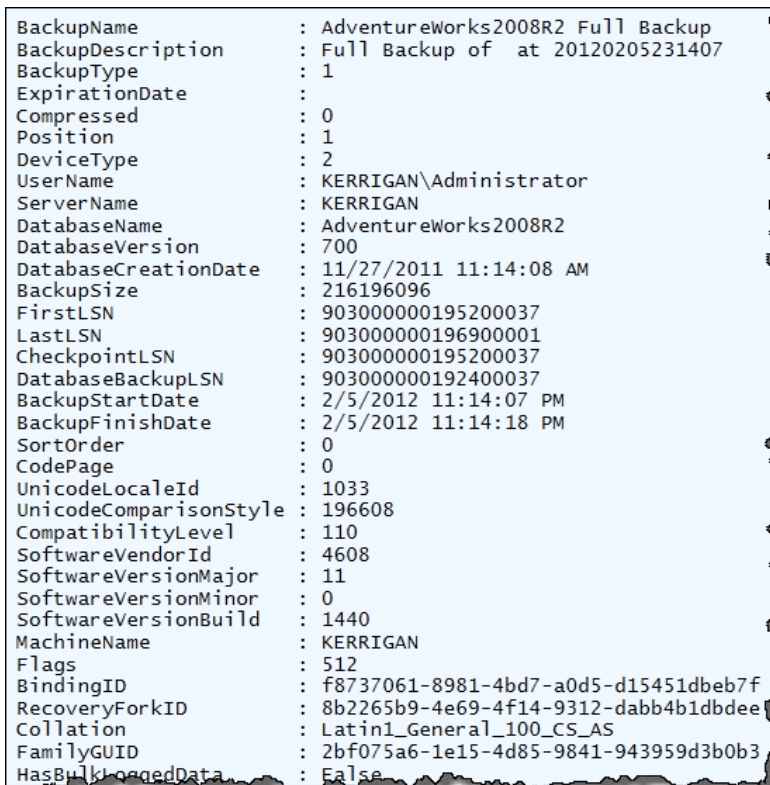
$smoRestore = New-Object Microsoft.SqlServer.Management.Smo.Restore

#replace this with your backup file
$backupfile = "AdventureWorks2008R2_Full_20120205231407.bak"
```

```
#change this to where your backup directory is
#in our case we're using default backup directory
$backupfilepath = Join-Path $server.Settings.BackupDirectory
$backupfile

$smoRestore.Devices.AddDevice($backupfilepath, [Microsoft.
SqlServer.Management.Smo.DeviceType]::File)
$smoRestore.ReadBackupHeader($server)
$smoRestore.ReadFileList($server)
```

The result you are going to get will be similar to the following screenshot:



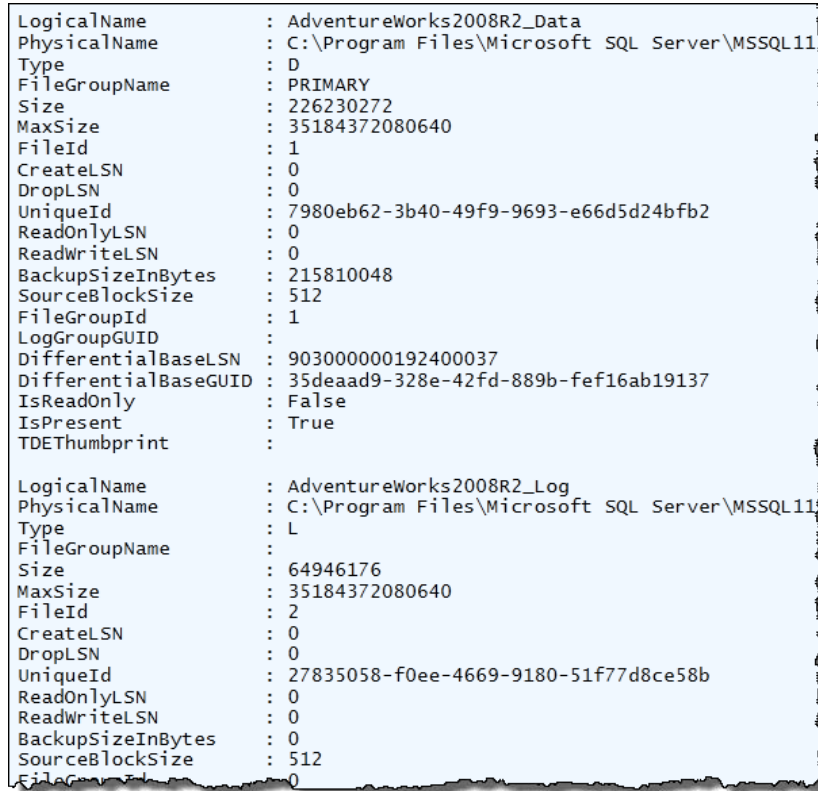
```
BackupName           : AdventureWorks2008R2 Full Backup
BackupDescription    : Full Backup of  at 20120205231407
BackupType           : 1
ExpirationDate       :
Compressed           : 0
Position             : 1
DeviceType           : 2
UserName             : KERRIGAN\Administrator
ServerName           : KERRIGAN
DatabaseName         : AdventureWorks2008R2
DatabaseVersion      : 700
DatabaseCreationDate : 11/27/2011 11:14:08 AM
BackupSize           : 216196096
FirstLSN             : 903000000195200037
LastLSN              : 903000000196900001
CheckpointLSN       : 903000000195200037
DatabaseBackupLSN   : 903000000192400037
BackupStartDate      : 2/5/2012 11:14:07 PM
BackupFinishDate     : 2/5/2012 11:14:18 PM
SortOrder            : 0
CodePage             : 0
UnicodeLocaleId     : 1033
UnicodeComparisonStyle : 196608
CompatibilityLevel   : 110
SoftwareVendorId     : 4608
SoftwareVersionMajor : 11
SoftwareVersionMinor : 0
SoftwareVersionBuild : 1440
MachineName          : KERRIGAN
Flags                : 512
BindingID            : f8737061-8981-4bd7-a0d5-d15451dbeb7f
RecoveryForkID       : 8b2265b9-4e69-4f14-9312-dabb4b1dbdee
Collation             : Latin1_General_100_CS_AS
FamilyGUID           : 2bf075a6-1e15-4d85-9841-943959d3b0b3
HasBulkLoggedData    : False
```

Notice that you can see the **BackupName**, **BackupType**, **ServerName**, **BackupSize**, **BackupStartDate**, **BackupFinishDate**, and different LSN values.

4. To display the file list information, add the following script and run:

```
$smoRestore.ReadFileList ($server)
```

The result you are going to get will be similar to the following screenshot:



```
LogicalName      : AdventureWorks2008R2_Data
PhysicalName     : C:\Program Files\Microsoft SQL Server\MSSQL11
Type            : D
FileGroupName    : PRIMARY
Size            : 226230272
MaxSize         : 35184372080640
FileId          : 1
CreateLSN       : 0
DropLSN        : 0
UniqueId        : 7980eb62-3b40-49f9-9693-e66d5d24bfb2
ReadOnlyLSN     : 0
ReadWriteLSN    : 0
BackupSizeInBytes : 215810048
SourceBlockSize : 512
FileGroupId     : 1
LogGroupGUID    :
DifferentialBaseLSN : 903000000192400037
DifferentialBaseGUID : 35deaad9-328e-42fd-889b-fef16ab19137
IsReadOnly      : False
IsPresent       : True
TDEThumbprint   :

LogicalName      : AdventureWorks2008R2_Log
PhysicalName     : C:\Program Files\Microsoft SQL Server\MSSQL11
Type            : L
FileGroupName    :
Size            : 64946176
MaxSize         : 35184372080640
FileId          : 2
CreateLSN       : 0
DropLSN        : 0
UniqueId        : 27835058-f0ee-4669-9180-51f77d8ce58b
ReadOnlyLSN     : 0
ReadWriteLSN    : 0
BackupSizeInBytes : 0
SourceBlockSize : 512
FileGroupId     : 0
```

Notice that you can see properties such as **LogicalName**, **PhysicalName**, **FileGroupName**, and **Size** of both the data and logfiles associated with this backup file.

How it works...

You will often want to find out more information about the contents of your backup files. The backup header and the file list of the backup files allow you to retrieve additional information about the contents of a backup file or backup device. Starting with SQL Server 2008, one must have the `CREATE DATABASE` permission before the header information can be listed.

To start, we must first create a reference to an SMO `Restore` object:

```
$smoRestore = New-Object Microsoft.SqlServer.Management.Smo.Restore;
```

The `ReadBackupHeader` method of the `Microsoft.SqlServer.Management.Smo.Restore` class lists all the backup headers for all backup sets contained in a backup device or file. The information it returns includes:

- ▶ BackupName and Description
- ▶ BackupType
- ▶ Compressed
- ▶ ServerName
- ▶ DatabaseName
- ▶ DatabaseVersion and DatabaseCreationDate
- ▶ BackupSize
- ▶ CheckpointLSN
- ▶ DatabaseBackupLSN
- ▶ Backup start and finish date

We will also need to create a reference to the backup file or backup device from which we wish to read the information. We do this by adding the backup file using the `AddDevice` method of the `Restore` object.

```
$backupfile = "AdventureWorks2008R2_Full_20120205231407.bak"

#change this to where your backup directory is
#in our case we're using default backup directory
$backupfilepath = Join-Path $server.Settings.BackupDirectory
$backupfile

$smoRestore.Devices.AddDevice($backupfilepath, [Microsoft.SqlServer.
Management.Smo.DeviceType]::File)
```

To retrieve the backup header, just invoke the `ReadBackupHeader` method of the `Restore` object and pass in the `server` object as an argument.

```
$smoRestore.ReadBackupHeader($server)
```

The file list contains the actual database and logfiles associated in a particular backup set. Listing the file list requires a very similar syntax to reading the backup header. We need to invoke the method `ReadFileList`, passing the `server` object as an argument again.

```
$smoRestore.ReadFileList($server)
```

See also...

- ▶ Read more about the Restore class methods:

[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.restore_methods\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.restore_methods(v=sql.110).aspx)

Creating a full backup

In this recipe, we will look at how we can do a full database backup using PowerShell.

Getting ready

We will use the AdventureWorks2008R2 database for this recipe. We will create a full compressed backup of the database to a timestamped .bak file in the C:\Backup folder. Feel free to use a database of your choice for this task.

The T-SQL syntax that will be generated by this PowerShell recipe will look similar to:

```
BACKUP DATABASE [AdventureWorks2008R2]
TO DISK = N'C:\Backup\AdventureWorks2008R2_Full_20120227092409.bak'
WITH NOFORMAT, INIT,
NAME = N'AdventureWorks2008R2 Full Backup',
NOSKIP, REWIND, NOUNLOAD, COMPRESSION,
STATS = 10, CHECKSUM
```

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the SQLPS module as follows:

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

3. Add the following script and run:

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

$databasename = "AdventureWorks2008R2"
$timestamp = Get-Date -Format yyyyMMddHHmmss
```

```
$backupfolder = "C:\Backup\"
$backupfile = "$($databasename)_Full_$( $timestamp ).bak"
$fullBackupFile = Join-Path $backupfolder $backupfile
```

```
Backup-SqlDatabase `
-ServerInstance $instanceName `
-Database $databasename `
-BackupFile $fullBackupFile `
-Checksum `
-Initialize `
-BackupSetName "$databasename Full Backup" `
-CompressionOption On
```

Check your C:\Backup directory and confirm that the timestamped backup file has been created.

4. Confirm by reading the backup header. Add the following script and run:

```
#confirm by reading the header
#backup type for full is 1
#this is a block of code you would want to put
#in a function so you can use anytime
$smoRestore = New-Object Microsoft.SqlServer.Management.Smo.
Restore
$smoRestore.Devices.AddDevice($fullBackupFile, [Microsoft.
SqlServer.Management.Smo.DeviceType]::File)
$smoRestore.ReadBackupHeader($server)
$smoRestore.ReadFileList($server)
```

How it works...

In this recipe, we first create a timestamped filename:

```
$databasename = "AdventureWorks2008R2"
$timestamp = Get-Date -Format yyyyMMddHHmmss
$backupfolder = "C:\Backup\"
$backupfile = "$($databasename)_Full_$( $timestamp ).bak"
$fullBackupFile = Join-Path $backupfolder $backupfile
```

This will give you a filename similar to this:

```
C:\Backup\AdventureWorks2008R2_Full_20120227092409.bak
```

Next, you need to invoke the Backup-SqlDatabase cmdlet. The Backup-SqlDatabase cmdlet has been introduced for SQL Server 2012, and this cmdlet encapsulates a lot of the options that used to be available only via SMO.

It is imperative, for this recipe, that we use the `Get-Help` cmdlet for the `Backup-SqlDatabase` cmdlet first, to know which parameters are available.

Here is one part of the help content:

```
SYNTAX
    Backup-SqlDatabase [-Database] <string> [[-BackupFile]
<string[]>] [-BackupAction <BackupActionType>] [-BackupDevice
<BackupDeviceItem[]>]
    [-BackupSetDescription <string>] [-BackupSetName
<string>] [-BlockSize <int>] [-BufferCount <int>] [-Checksum]
[-CompressionOption
    <BackupCompressionOptions>] [-ContinueAfterError] [-CopyOnly]
[-DatabaseFile <string[]>] [-DatabaseFileGroup <string[]>]
[-ExpirationDate
    <DateTime>] [-Format] [-Incremental] [-Initialize]
[-LogTruncationType <BackupTruncateLogType>] [-MaxTransferSize <int>]
[-MediaDescription
    <string>] [-MediaName <string>] [-MirrorDevices
<BackupDeviceList[]>] [-NoRecovery] [-NoRewind] [-Passthru] [-Path
<string[]>] [-Restart]
    [-RetainDays <int>] [-SkipTapeHeader] [-UndoFileName <string>]
[-UnloadTapeAfter] [-Confirm] [-WhatIf] [<CommonParameters>]
```

At the time of writing this book, there are still some corrections that need to be made to the help contents for `Backup-SqlDatabase`. This is documented in this MS Connect item <http://connect.microsoft.com/SQLServer/feedback/details/683594/backup-sqldatabase-cmdlet-help>. The content of the help, nevertheless, is still useful in getting you up and running with the cmdlet.

In our recipe, this is the command we executed:

```
Backup-SqlDatabase `
-ServerInstance $instanceName `
-Database $databasename `
-BackupFile $fullBackupFile `
-Checksum `
-Initialize `
-BackupSetName "$databasename Full Backup" `
-CompressionOption On
```

Note that we used the line continuation character back tick (```) for readability purposes, so we can align each parameter at the same position on each line.

Let's explain in more detail these options that we have chosen:

Parameter	Explanation
-ServerInstance \$instanceName	Instance to backup
-Database \$databasename	Database to backup
-BackupFile \$fullBackupFile	Backup file name
-Checksum	Enable backup checksum, which can be used in restore operation to determine if backup file is corrupt
-Initialize	Specifies backup set contained in the file or backup device will be overwritten
-BackupSetName "\$databasename Full Backup"	Backup set name
-CompressionOption On	Specifies whether compression should be applied to the backup file You can also provide the complete enum reference for the CompressionOption value: -CompressionOption ([Microsoft.SqlServer.Management.Smo.BackupCompressionOptions]::On)

Once you get more familiar with the `Backup-SqlDatabase` cmdlet, you will soon realize that all other backup types will be just a matter of adding or changing some of these parameters.

There's more...

Although there is already a cmdlet available for backing up databases, it will also be useful to look at how you can do the backups via SMO. Using SMO may be the more code-heavy way of tackling a database backup in PowerShell, but it is nonetheless still very powerful and flexible.

The cmdlet can be viewed as simply a wrapper to the SMO backup methods. Taking a peek at how this is done can be a beneficial exercise.

The first few steps for this approach are similar to the steps we have for this recipe: import `SQLPS`, and create the SMO server object. After that, we will need to create an SMO Backup object.

```
$databasename = "AdventureWorks2008R2"
$timestamp = Get-Date -Format yyyyMMddHHmmss
$backupfolder = "C:\Backup\"
```

Backup and Restore

```
$backupfile = "$($databasename)_Full_$(timestamp).bak"
$fullBackupFile = Join-Path $backupfolder $backupfile

#This belongs in Microsoft.SqlServer.SmoExtended assembly
$smoBackup = New-Object Microsoft.SqlServer.Management.Smo.Backup
```

With a handle to the SMO backup object, you will have more granular control over what values are set to which properties. Action can be any of Database, File, or Log.

```
$smoBackup.Action = [Microsoft.SqlServer.Management.Smo.
BackupActionType]::Database
$smoBackup.BackupSetName = "$databasename Full Backup"
$smoBackup.Database = $databasename
$smoBackup.MediaDescription = "Disk"
$smoBackup.Devices.AddDevice($fullBackupFile, "File")
$smoBackup.Checksum = $true
$smoBackup.Initialize = $true
$smoBackup.CompressionOption = [Microsoft.SqlServer.Management.Smo.
BackupCompressionOptions]::On
```

You can also optionally set up your own event notification on the backup progress using the `Microsoft.SqlServer.Management.Smo.PercentCompleteEventHandler` and `Microsoft.SqlServer.Management.Common.ServerMessageEventHandler` classes.

```
#the notification part below is optional
#it just creates an
#event handler that indicates progress every 20%
$smoBackup.PercentCompleteNotification = 20
$percentEventHandler = [Microsoft.SqlServer.Management.Smo.
PercentCompleteEventHandler] {
    Write-Host "Backing up $($databasename)...$( $_.Percent)%"
}
$completedEventHandler = [Microsoft.SqlServer.Management.Common.
ServerMessageEventHandler] {
    Write-Host $_.Error.Message
}
$smoBackup.add_PercentComplete($percentEventHandler)
$smoBackup.add_Complete($completedEventHandler)
```

When done setting the properties, you can just invoke the `SqlBackup` method of the SMO Backup class and pass the server object:

```
#backup
$smoBackup.SqlBackup($server)
```

Conversely, when you do a restore with SMO, the steps are going to be pretty similar. You will need to create the SMO `Restore` object, set the properties, and call the `SqlRestore` method of the `Restore` class in the end.

More about Backup and PercentCompleteEventHandler

Learn more about these SMO classes:

- ▶ BackupRestoreBase: <http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.backuprestorebase.percentcomplete.aspx>
- ▶ PercentCompleteEventHandler: <http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.percentcompleteeventhandler.aspx>

See also

- ▶ *The Creating a backup on mirrored media sets recipe*
- ▶ *The Creating a differential backup recipe*
- ▶ *The Creating a transaction log backup recipe*
- ▶ *The Creating a filegroup backup recipe*

Creating a backup on mirrored media sets

In this recipe, we will create a full database backup on mirrored backup files.

Getting ready

We will use the AdventureWorks2008R2 database for this recipe. We will create a mirrored backup of the database, and both timestamped backup files will be stored in C:\Backup. Feel free to substitute this with the database you want to use with mirrored backups.

The T-SQL syntax that will be generated by this PowerShell recipe will look similar to:

```
BACKUP DATABASE [AdventureWorks2008R2]
TO DISK = N'AdventureWorks2008R2.bak'
MIRROR
TO DISK = N'C:\Backup\AdventureWorks2008R2_Full_20120227092409_Copy1.
bak'
MIRROR TO DISK = N'C:\Backup\AdventureWorks2008R2_
Full_20120227092409_Copy2.bak'
WITH FORMAT, INIT,
NAME = N'AdventureWorks2008R2 Full Backup', SKIP, REWIND,
NOUNLOAD, COMPRESSION, STATS = 10, CHECKSUM
```


How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the SQLPS module as follows:

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

3. Add the following script and run:

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

$databasename = "AdventureWorks2008R2"

#create filenames, which we will use as Device
$databasename = "AdventureWorks2008R2"
$timestamp = Get-Date -Format yyyyMMddHHmmss
$backupfolder = "C:\Backup\"

$backupfile1 = Join-Path $backupfolder "$($databasename)_
Full_$( $timestamp)_Copy1.bak"
$backupfile2 = Join-Path $backupfolder "$($databasename)_
Full_$( $timestamp)_Copy2.bak"

#create a backup device list
#in this example, we will only use two (2)
#mirrored media sets
#note a maximum of four (4) is allowed
$backupDevices = New-Object Microsoft.SqlServer.Management.Smo.
BackupDeviceList(2)
$backupDevices.AddDevice($backupfile1, [Microsoft.SqlServer.
Management.Smo.DeviceType]::File)
$backupDevices.AddDevice($backupfile2, [Microsoft.SqlServer.
Management.Smo.DeviceType]::File)

#backup database
Backup-SqlDatabase `
-ServerInstance $instanceName `
-Database $databasename `
-BackupSetName "$databasename Full Backup" `
-Checksum `
-Initialize `
-FormatMedia `
```

```
-SkipTapeHeader `
-MirrorDevices $backupDevices `
-CompressionOption On
```

4. Open your C:\Backup folder and confirm that the two timestamped backup files have been created.

How it works...

With SQL Server, it is possible to create a backup with up to four mirrors per media set. Mirrored media sets allow you to have multiple copies of that backup, which are stored in different backup devices.

For our recipe, we must first create a set of files that we will use to save our backup to.

```
#create backup devices
#in this example, we will only use two (2) mirrored media sets
#note a maximum of four (4) is allowed
$databasename = "AdventureWorks2008R2"
$timestamp = Get-Date -Format yyyyMMddHHmmss
$backupfolder = "C:\Backup\"

$backupfile1 = Join-Path $backupfolder "$($databasename)_
Full_{$timestamp}_Copy1.bak"
$backupfile2 = Join-Path $backupfolder "$($databasename)_
Full_{$timestamp}_Copy2.bak"
```

We then need to add these files' backup devices to our BackupDeviceList object. The value that we pass to our BackupDeviceList constructor, represents the number of backup devices we are adding. A maximum of four is allowed for mirrored media.

```
$backupDevices = New-Object Microsoft.SqlServer.Management.Smo.
BackupDeviceList(2)
$backupDevices.AddDevice($backupfile1, [Microsoft.SqlServer.
Management.Smo.DeviceType]::File)
$backupDevices.AddDevice($backupfile2, [Microsoft.SqlServer.
Management.Smo.DeviceType]::File)
```

In the Backup-SqlDatabase cmdlet, the highlighted code in the following snippet shows the options that enable mirrored backups. Note that we used the line continuation character backtick (`) for readability purposes, so we can align each parameter at the same position on each line.

```
#backup database
Backup-SqlDatabase `
-ServerInstance $instanceName `
-Database $databasename `
```

Backup and Restore

```
-BackupSetName "$databasename Full Backup" `
-Checksum `
-Initialize `
-FormatMedia `
-SkipTapeHeader `
-MirrorDevices $backupDevices `
-CompressionOption On
```

Let's explain a bit more about some of these highlighted options:

Parameter	Explanation
-Initialize	Specifies backup set contained in the file or backup device will be overwritten
-FormatMedia	Overwrites existing media header information, and creates a new media set
-SkipTapeHeader	Skip checking backup tape expiration
-MirrorDevices	Allows backup on mirrored media sets; accepts a BackupDeviceList array

See also

- ▶ The *Creating a full backup* recipe
- ▶ The *Creating a differential backup* recipe
- ▶ The *Creating a transaction log backup* recipe
- ▶ The *Creating a filegroup backup* recipe
- ▶ Learn more about mirrored backup media sets:

[http://msdn.microsoft.com/en-us/library/ms175053\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms175053(v=sql.110).aspx)

Creating a differential backup

This recipe shows how you can create a differential backup on your database.

Getting ready

We will use the AdventureWorks2008R2 database for this recipe. We will create a differential compressed backup of the database to a timestamped .bak file in the C:\Backup folder. Feel free to use a database of your choice for this task.

The T-SQL syntax that will be generated by this PowerShell recipe will look similar to:

```
BACKUP DATABASE [AdventureWorks2008R2]
TO DISK = N'C:\Backup\AdventureWorks2008R2_Diff_20120227092409.bak'
WITH DIFFERENTIAL , NOFORMAT, INIT,
NAME = N'AdventureWorks2008R2 Diff Backup',
NOSKIP, REWIND, NOUNLOAD, COMPRESSION,
STATS = 10, CHECKSUM
```

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the SQLPS module as follows:

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

3. Add the following script and run:

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

```
$databasename = "AdventureWorks2008R2"
$timestamp = Get-Date -Format yyyyMMddHHmmss
$backupfolder = "C:\Backup\"
$backupfile = "$($databasename)_Diff_$( $timestamp ).bak"
$difffBackupFile = Join-Path $backupfolder $backupfile
```

```
Backup-SqlDatabase `
-ServerInstance $instanceName `
-Database $databasename `
-BackupFile $difffBackupFile `
-Checksum `
-Initialize `
-Incremental `
-BackupSetName "$databasename Diff Backup" `
-CompressionOption On
```

4. Confirm by reading the backup header. Add the following script and run:

```
#confirm by reading the header
#backup type for differential is 5
#this is a block of code you would want to put
#in a function so you can use anytime
```

```
$smoRestore = New-Object Microsoft.SqlServer.Management.Smo.  
Restore  
$smoRestore.Devices.AddDevice($diffBackupFile, [Microsoft.  
SqlServer.Management.Smo.DeviceType]::File)  
$smoRestore.ReadBackupHeader($server)  
$smoRestore.ReadFileList($server)
```

How it works...

A differential backup captures all changes to a database since the last full backup. Creating a differential backup in PowerShell is very similar to creating a full backup when using the `Backup-SqlDatabase` cmdlet, with a slight change in the set of options that need to be specified.

```
Backup-SqlDatabase `
  -ServerInstance $instanceName `
  -Database $databasename `
  -BackupFile $diffBackupFile `
  -Checksum `
  -Initialize `
  -Incremental `
  -BackupSetName "$databasename Diff Backup" `
  -CompressionOption On
```

The one option that differentiates a full and differential backup is the option `-Incremental`.



More information about these options used with the `Backup-SqlDatabase` cmdlet is explained in more details in the *Creating a full backup* recipe.

There's more...

To do a differential backup using SMO, the code will be similar to the SMO code you would use with a full backup. The one line that you will need to add is:

```
$smoBackup.Incremental = $true
```



Check out a more detailed example and explanation of how to use SMO for backups, instead of the `Backup-SqlDatabase` cmdlet, in the *Creating a full backup* recipe.

See also

- ▶ The *Creating a full backup* recipe
- ▶ The *Creating a backup on mirrored media sets* recipe
- ▶ The *Creating a transaction log backup* recipe
- ▶ The *Creating a filegroup backup* recipe

Creating a transaction log backup

In this recipe, we will create a transaction log backup.

Getting ready

We will use the `AdventureWorks2008R2` database for this recipe. We will create a timestamped transaction log backup file in the `C:\Backup` folder. Feel free to use a database of your choice for this task.

Ensure the recovery model of the database you are backing up is either `Full` or `BulkLogged`. You can use the *Changing database recovery model* recipe as a reference. The main code you can execute to query the current recovery model setting of your database is:

```
$database.DatabaseOptions.RecoveryModel
```

You can also check this using SQL Server Management Studio.

- ▶ Log in to SSMS
- ▶ Expand **Databases**, and right-click on **AdventureWorks2008R2**
- ▶ Go to **Properties | Options** and check the **Recovery Model** value

The T-SQL syntax that will be generated by this PowerShell recipe will look similar to:

```
BACKUP LOG [AdventureWorks2008R2]
TO DISK = N'C:\Backup\AdventureWorks2008R2_Txn_20120815235319.bak'
WITH NOFORMAT, NOINIT, NOSKIP, REWIND, NOUNLOAD, STATS = 10
```

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the `SQLPS` module as follows:

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

3. Add the following script and run:

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

#create a transaction log backup
$databasename = "AdventureWorks2008R2"
$timestamp = Get-Date -Format yyyyMMddHHmmss
$backupfolder = "C:\Backup\"
$backupfile = "$($databasename)_Txn_{$($timestamp)}.bak"
$txnBackupFile = Join-Path $backupfolder $backupfile

Backup-SqlDatabase `
-BackupAction Log `
-ServerInstance $instanceName `
-Database $databasename `
-BackupFile $txnBackupFile
```

How it works...

Transaction log backups are only permitted if the database you are backing up is in either the Full or BulkLogged Recovery Model. To create a transaction log backup using the Backup-SqlDatabase cmdlet, there is one option that must be specified:

```
Backup-SqlDatabase `
-BackupAction Log `
-ServerInstance $instanceName `
-Database $databasename `
-BackupFile $txnBackupFile
```

When backing up databases, one of the most important parameters is BackupAction, which accepts three valid values: Database, Files, and Log.

You can also optionally use the fully qualified name of the BackupActionType enumeration:

```
-BackupAction ([Microsoft.SqlServer.Management.Smo.
BackupActionType]::Log
```

Additional options you can specify when doing transaction log backups are:

Parameter	Explanation
-NoRecovery	Required when you are taking tail log backups; this puts the database in the Restoring state, and the log is not truncated
-LogTruncationType	Accepts an SMO BackupTruncateLogType enumeration value, which is one of: NoTruncate, Truncate, and TruncateOnly

There's more...

Tail log backups will contain anything that hasn't been backed up yet. These backups are usually taken in the event of a disaster, or just before a restore operation. Taking a tail log backup leaves the database in a Restoring state, that is, in an inaccessible state to prevent further changes.

See also

- ▶ The *Creating a backup on mirrored media sets* recipe
- ▶ The *Creating a full backup* recipe
- ▶ The *Creating a differential backup* recipe
- ▶ The *Creating a filegroup backup* recipe
- ▶ Learn more about tail log backups here:

[http://msdn.microsoft.com/en-us/library/ms179314\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms179314(v=sql.110).aspx)

Creating a filegroup backup

In this recipe, we will create a filegroup backup using the Backup-SqlDatabase PowerShell cmdlet.

Getting ready

For testing purposes, let's create a small sample database called `StudentDB` that contains a couple of filegroups called `FG1` and `FG2`. Each filegroup will have two datafiles.

Open up SQL Server Management Studio and run the following script:

```
CREATE DATABASE [StudentDB]
ON PRIMARY
```



```
( NAME = N'StudentDB', FILENAME = N'C:\Temp\StudentDB.mdf'),
FILEGROUP [FG1]
( NAME = N'StudentData1', FILENAME = N'C:\Temp\StudentData1.ndf'),
( NAME = N'StudentData2', FILENAME = N'C:\Temp\StudentData2.ndf'),
FILEGROUP [FG2]
( NAME = N'StudentData3', FILENAME = N'C:\Temp\StudentData3.ndf')
LOG ON
( NAME = N'StudentDB_log', FILENAME = N'C:\Temp\StudentDB.ldf')
GO
```

We will use this database to do our filegroup backup.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the SQLPS module as follows:

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

3. Add the following script and run:

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

$databasename = "StudentDB"
$timestamp = Get-Date -Format yyyyMMddHHmmss

#create a file to backup FG1 filegroup
$backupfolder = "C:\Backup\"
$backupfile = "$($databasename)_FG1_$( $timestamp).bak"
$fgBackupFile = Join-Path $backupfolder $backupfile

Backup-SqlDatabase `
-BackupAction Files `
-DatabaseFileGroup "FG1" `
-ServerInstance $instanceName `
-Database $databasename `
-BackupFile $fgBackupFile `
-Checksum `
-Initialize `
-BackupSetName "$databasename FG1 Backup" `
-CompressionOption On
```

```
#confirm by reading the header
#backup type for files is 4
#this is a block of code you would want to put
#in a function so you can use anytime
$smoRestore = New-Object Microsoft.SqlServer.Management.Smo.
Restore
$smoRestore.Devices.AddDevice($fgBackupFile, [Microsoft.SqlServer.
Management.Smo.DeviceType]::File)

$smoRestore.ReadBackupHeader($server)
```

How it works...

Backing up filegroups can be considered a practical alternative for VLDBs, or very large databases, where a full backup can take up impractical amounts of space and time. With filegroup backups, you can strategize which filegroups to back up more frequently and which ones less frequently. Filegroup backups also enable you to take advantage of online piecemeal restores for Enterprise Edition of SQL Server, starting with SQL Server 2005.



See the *Performing an online piecemeal restore* recipe for more details.

In our recipe, we chose to backup FG1. Our main backup command looks like this:

```
Backup-SqlDatabase `
-BackupAction Files `
-DatabaseFileGroup "FG1" `
-ServerInstance $instanceName `
-Database $databasename `
-BackupFile $fgBackupFile `
-Checksum `
-Initialize `
-BackupSetName "$databasename FG1 Backup" `
-CompressionOption On;
```

Notice the highlighted lines of code. These lines enable the filegroup backups. For the `BackupAction` parameter, we have to specify `Files`. The other options for `BackupAction` are `Database` and `Log`.

Once we have specified that we want the `Files` value for the `BackupAction` parameter, we should also pass the name of the filegroup we want to back up using the `DatabaseFileGroup` parameter.

See also

- ▶ The *Creating a backup on mirrored media sets* recipe
- ▶ The *Creating a full backup* recipe
- ▶ The *Creating a differential backup* recipe
- ▶ The *Creating a transaction log backup* recipe
- ▶ The *Performing an online piecemeal restore* recipe
- ▶ Learn more about backing up files and filegroups:
[http://msdn.microsoft.com/en-us/library/ms179401\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms179401(v=sql.110).aspx)

Restoring a database to a point in time

In this recipe, we will use the different backup files we have to restore to a point in time.

Getting ready

In this recipe, we will use the `AdventureWorks2008R2` database. You can also substitute this with your preferred database on your development environment.

The `AdventureWorks2008R2` database has a single filegroup that contains a single datafile. We will restore this database to another SQL Server instance at a different point in time using three different backup files from three different backup types:

- ▶ Full backup
- ▶ Differential backup
- ▶ Transaction log backup

We can create these three types of backups on the `AdventureWorks2008R2` database using PowerShell as illustrated in previous recipes. If you are fairly comfortable with T-SQL, this can also be done with T-SQL backup commands.

To help us verify if our point-in-time restore worked as expected, create a timestamped table before taking any type of backup. Alternatively, create a table and insert a timestamped record in the table before taking a backup.

Place these backups in the folder called `C:\Backup\`.

Name ^	Date modified	Type
AdventureWorks2008R2_Diff_201204070819.bak	4/7/2012 8:19 AM	BAK File
AdventureWorks2008R2_Full_201204070818.bak	4/7/2012 8:18 AM	BAK File
AdventureWorks2008R2_Txn_201204070820.bak	4/7/2012 8:20 AM	BAK File
AdventureWorks2008R2_Txn_201204070821.bak	4/7/2012 8:21 AM	BAK File
AdventureWorks2008R2_Txn_201204070822.bak	4/7/2012 8:22 AM	BAK File
AdventureWorks2008R2_Txn_201204070823.bak	4/7/2012 8:23 AM	BAK File

You can use the following script to create your files *6464 - Ch05 - 10 - Restore a database to a point in time - Prep.ps1*, which is included in the downloadable files for this book. When the script has finished executing, you should have timestamped `Student` tables in the `AdventureWorks2008R2` database, created within one minute intervals, similar to the following screenshot:

Folder/Item
KERRIGAN (SQL Server 11.0.2100 - KERRIGAN\Administrator)
Databases
System Databases
Database Snapshots
AdventureWorks2008R2
Database Diagrams
Tables (filtered)
System Tables
FileTables
dbo.Student
dbo.StudentDiff_201204070819
dbo.StudentFull_201204070818
dbo.StudentTxn_201204070820
dbo.StudentTxn_201204070821
dbo.StudentTxn_201204070822
dbo.StudentTxn_201204070823

For our recipe, we will restore the `AdventureWorks2008R2` database to a second instance, `KERRIGAN\SQL01`, up to `2012-04-07 08:21:59`. This means that after the point-in-time restore, we should have only four timestamped `Student` tables in `KERRIGAN\SQL01` restored database:

- ▶ `StudentFull_201204070818`
- ▶ `StudentDiff_201204070819`
- ▶ `StudentTxn_201204070820`
- ▶ `StudentTxn_201204070821`

How to do it...

To restore to a point in time using a full, differential, and several transaction logfiles, follow these steps:

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the SQLPS module as follows:

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

3. Add the following script and run:

```
$instanceName = "KERRIGAN\SQL01"
$server = New-Object -TypeName Microsoft.SqlServer.Management.
Smo.Server -ArgumentList $instanceName

#backupfilefolder
$backupfilefolder = "C:\Backup\"

#look for the last full backupfile
#you can be more specific and specify filename
$fullBackupFile =
Get-ChildItem $backupfilefolder -Filter "*Full*" |
Sort -Property LastWriteTime -Descending |
Select -Last 1

#read the filelist info within the backup file
#so that we know which other files we need to restore
$smoRestore = New-Object Microsoft.SqlServer.Management.Smo.
Restore

$smoRestore.Devices.AddDevice($fullBackupFile.FullName,
[Microsoft.SqlServer.Management.Smo.DeviceType]::File)

$filelist = $smoRestore.ReadFileList($server)

#read headers of the full backup file,
#because we are restoring to a default instance, we will
#need to specify we want to move the files
#to the default data directory of our KERRIGAN\SQL01 instance
$relocateFileList = @()
$relocatePath = "C:\Program Files\Microsoft SQL Server\MSSQL11.
SQL01\MSSQL\DATA"
```

```

#we are putting this in an array in case we have
#multiple data and logfiles associated with the database
foreach($file in $fileList)
{
    #restore to different instance
    #replace default directory path for both
    $relocateFile = Join-Path $relocatePath (Split-Path $file.
PhysicalName -Leaf)

    $relocateFileList += New-Object Microsoft.SqlServer.
Management.Smo.RelocateFile($file.LogicalName, $relocateFile)
}

#let's timestamp our restored databasename
#this is strictly for testing our recipe
$timestamp = Get-Date -Format yyyyMMddHHmmss
$restoredDBName = "AWRestored_{$timestamp}"

#=====
#restore the full backup to the new instance name
#=====
#note we have a NoRecovery option, because we have
#additional files to restore
Restore-SqlDatabase `
-ReplaceDatabase `
-ServerInstance $instanceName `
-Database $restoredDBName `
-BackupFile $fullBackupFile.FullName `
-RelocateFile $relocateFileList `
-NoRecovery

#=====
#restore last differential
#note the database is still in Restoring State
#=====
#using PowerShell V2 Where syntax
$diffBackupFile =
Get-ChildItem $backupfilefolder -Filter "*Diff*" |
Where {$_.LastWriteTime -ge $fullBackupFile.LastWriteTime} |
Sort -Property LastWriteTime -Descending |
Select -Last 1

Restore-SqlDatabase `
-ReplaceDatabase `

```

Backup and Restore

```
-ServerInstance $instanceName `
-Database $restoreddbname `
-BackupFile $diffBackupFile.FullName `
-NoRecovery

#=====
#restore all transaction log backups from last
#differential up to 2012-04-07 08:21:59
#=====
#identify the last txn log backup file we need to restore
#we need this so we can specify point in time
$lastTxnFileName = "AdventureWorks2008R2_Txn_201204070821"

$lastTxnBackupFile =
Get-ChildItem $backupfilefolder -Filter "$lastTxnFileName"

#restore all transaction log backups after the
#last differential, except the last transaction
#backup that requires the point-in-time restore
foreach ($txnBackup in Get-ChildItem $backupfilefolder -Filter
"*Txn*" |
Where {$_.LastWriteTime -ge $diffBackupFile.LastWriteTime -and
$_.LastWriteTime -lt $lastTxnBackupFile.LastWriteTime} |
Sort -Property LastWriteTime)
{
    Restore-SqlDatabase `
    -ReplaceDatabase `
    -ServerInstance $instanceName `
    -Database $restoreddbname `
    -BackupFile $txnBackup.FullName `
    -NoRecovery
}

#restore last txn backup file to point in time
#restore only up to 2012-04-07 08:21:59
#this time we are going to restore using with recovery
Restore-SqlDatabase `
-ReplaceDatabase `
-ServerInstance $instanceName `
-Database $restoreddbname `
-BackupFile $lastTxnBackupFile.FullName `
-ToPointInTime "2012-04-07 08:21:59"
```

How it works...

In this recipe, we are using the `Restore-SqlDatabase` cmdlet, the counterpart of the `Backup-SqlDatabase` cmdlet that was introduced in SQL Server 2012.

Let's get a high-level overview of how to perform a point-in-time restore, and then we can break it down and explain the pieces involved in this recipe:

1. Gather your backup files.
 - Identify the last transaction log backup file that contains the point you want to restore to.
2. Restore the last good full backup with `NORECOVERY`.
3. Restore the last good differential backup taken after the full backup you just restored with `NORECOVERY`.
4. Restore the transaction logs taken after your differential backup:
 - You can restore *up to and including* the log backup that contains the data to the point in time you want to restore with `NORECOVERY`. You need to restore the last log backup to a point in time, that is, you need to specify up to when to restore. Lastly, restore the database using `WITH RECOVERY` to make the database accessible and ready to use.
 - Or, you can restore all transaction log backup files *before* the log backup that contains the data to the point in time you want to restore with `NORECOVERY`. Next, restore the last log backup using `WITH RECOVERY` to a point in time, that is, you need to specify up to when to restore.

Step 1 – Gather your backup files

You will need to collect your backup files. They don't necessarily have to reside in the same folder or drive, but it will be ideal, as it can simplify your restore script because you will have a uniform folder/drive to refer to. You will also need read permissions for these files.

In our recipe, we have simplified this step. We have collected our full, differential, and transaction log backup files and stored them in the `C:\Backup\` folder for ease of access. If your backup files reside in different locations, you will just need to adjust the directory references in your script appropriately.

Once you have the backup files, assuming you follow a file naming convention, you can filter out all the full backups in your directory. In our sample, we are using the convention `dbname_type_timestamp.bak`. For this scenario, we can extract that one full backup file by specifying the keyword or pattern in our filename. We use the `Get-ChildItem` cmdlet to filter for the latest full backup file:

```
#look for the last full backupfile
#you can be more specific and specify filename
```


Backup and Restore

```
$fullBackupFile =  
Get-ChildItem $backupfilefolder -Filter "*Full*" |  
Sort -Property LastWriteTime -Descending |  
Select -Last 1
```

Once you have the full backup handle, you can read the filelist that is stored in that backup file. You can use the `ReadFileList` method that is available with an SMO `Restore` object. Reading the filelist can help you automate by extracting the filenames of the data and logfiles you will need to restore.

```
#read the filelist info within the backup file  
#so that we know which other files we need to restore  
$smoRestore = New-Object Microsoft.SqlServer.Management.Smo.Restore  
  
$smoRestore.Devices.AddDevice($fullBackupFile.FullName, [Microsoft.  
SqlServer.Management.Smo.DeviceType]::File)  
  
$filelist = $smoRestore.ReadFileList($server)
```

When reading the filelist, one property you can extract is the type of file that is stored:



```
LogicalName      : AdventureWorks2008R2_Data  
PhysicalName     : C:\Program Files\Microsoft SQL Server\MSSQL...  
Type             : D  
FileGroupName   : PRIMARY  
Size            : 226230272  
MaxSize         : 35184372080640  
FileId          : 1  
CreateLSN       : 0  
DropLSN         : 0  
UniqueId        : 7980eb62-3b40-49f9-9693-e66d5d24bfb2  
ReadOnlyLSN     : 0  
ReadWriteLSN    : 0  
BackupSizeInBytes : 216858624  
SourceBlockSize : 512  
FileGroupId      : 1  
LogGroupGUID    :  
DifferentialBaseLSN : 1047000000391700143  
DifferentialBaseGUID : da9e2691-7b3a-4988-94a7-f46fd03f2725  
IsReadOnly      : False  
IsPresent       : True  
TDEThumbprint   :
```

The different types are:

- ▶ **L** = Logfile
- ▶ **D** = Database file
- ▶ **F** = FullText catalog

Step 2 – Restore the last good full backup, with NORECOVERY

The first step in restore operations is to restore the last known good full backup. This provides you a baseline to which you can restore additional files. The `NORECOVERY` option is very important, as it preserves (or does not roll back) uncommitted transactions and allows additional files to be restored. We will be using the `NORECOVERY` option throughout our restore process.

Because the full backup is always the first file that needs to be restored, all the prep work required when moving files also happens at this stage.

For our recipe, we want to restore the database, originally from the default instance `KERRIGAN`, to another instance, `KERRIGAN\SQL01`. For this reason, we will need to move our files from the path stored with our backup file, to the new path we want to use. In this example we only want to move from the default data directory of our default instance to the data directory of our named instance `KERRIGAN\SQL01`. We do this by retrieving the full paths of the original data and logfiles from the `filelist`, and replacing the full path with the new location we want to restore to. The highlighted code in the following snippet shows how we change this location:

```
$relocateFileList = @()
$relocatePath = "C:\Program Files\Microsoft SQL Server\MSSQL11.SQL01\
MSSQL\DATA"

#we are putting this in an array in case we have
#multiple data and logfiles associated with the database
foreach($file in $fileList)
{
    #restore to different instance
    #replace default directory path for both
    $relocateFile = Join-Path $relocatePath (Split-Path $file.
PhysicalName -Leaf)

    $relocateFileList += New-Object Microsoft.SqlServer.Management.
Smo.RelocateFile($file.LogicalName, $relocateFile)
}
```

Note that our array contains the `Microsoft.SqlServer.Management.Smo.RelocateFile` object, which will contain the logical and (relocated) physical names of our database files.

```
$relocateFileList += New-Object Microsoft.SqlServer.Management.Smo.
RelocateFile($file.LogicalName, $relocateFile)
```

To restore our database, we are simply going to use the `Backup-SqlDatabase` cmdlet. There are a couple of really important options here such as `RelocateFile` and `NoRecovery`.

```
#restore the full backup to the new instance name
#note we have a NoRecovery option, because we have
#additional files to restore
```

```
Restore-SqlDatabase `
-ReplaceDatabase `
-ServerInstance $instanceName `
-Database $restoredDBName `
-BackupFile $fullBackupFile.FullName `
-RelocateFile $relocateFileList `
-NoRecovery
```

Step 3 – Restore the last good differential backup taken after the full backup you just restored, with NORECOVERY

Once the full backup is restored, we can add the last good differential backup following our full backup. This is going to be a less involved process, because at this point we've already restored our base database and relocated our files. We need to restore the differential backup with NORECOVERY to prevent uncommitted transactions from being rolled back:

```
#using PowerShell V2 Where syntax
$diffBackupFile =
Get-ChildItem $backupfilefolder -Filter "*Diff*" |
Where {$_.LastWriteTime -ge $fullBackupFile.LastWriteTime} |
Sort -Property LastWriteTime -Descending |
Select -Last 1
```

```
Restore-SqlDatabase `
-ReplaceDatabase `
-ServerInstance $instanceName `
-Database $restoreddbname `
-BackupFile $diffBackupFile.FullName `
-NoRecovery
```

Note that you may, or may not, have a differential backup file in your environment. If you don't, don't worry, it does not affect your recoverability as long as you have all the transaction log backup files intact and available for restore.

Step 4 – Restore the transaction logs taken after your differential backup

After we restore our differential backup file, we can start restoring our transaction log backup files. These transaction log backup files should be the ones following your differential backup. You may, or may not, need the complete set of logfiles following your differential backup. If you need to restore up to the point of a database crash, you will need to restore all transaction log backups including the tail log backup. If not, you will only need the backup files up to the time to which you want to restore.

For our recipe, we identify first the last transaction log backup file we want to restore. This is important because we need to know how to use a `PointInTime` parameter when we use this particular transaction log backup file.

```
#identify the last txn log backup file we need to restore
#we need this so we can specify point in time
$lastTxnFileName = "AdventureWorks2008R2_Txn_201204070821"
```

```
$lastTxnBackupFile =
Get-ChildItem $backupfilefolder -Filter "$lastTxnFileName"
```

For all other transaction log backup files, we loop through our backup folder and restore all .txn files that were taken after the last differential backup, and before the last transaction log backup file we want to restore. We also need to sort the files by the `WriteTime` parameter so that we can restore them sequentially to our database. Note that we need to restore all these files with `NORECOVERY`.

```
foreach ($txnBackup in Get-ChildItem $backupfilefolder -Filter "**Txn*"
|
Where {$_.LastWriteTime -ge $diffBackupFile.LastWriteTime -and
$_LastWriteTime -lt $lastTxnBackupFile.LastWriteTime} |
Sort -Property LastWriteTime)
{
    Restore-SqlDatabase `
    -ReplaceDatabase `
    -ServerInstance $instanceName `
    -Database $restoreddbname `
    -BackupFile $txnBackup.FullName `
    -NoRecovery
}
```

Once all these files are restored, then we are ready to restore the last transaction logfile. Once this file is restored, the database needs to be accessible, and all uncommitted transactions need to be rolled back.

There are two methods to do this. The first method, which we used in the recipe, is to restore the last file with the `ToPointInTime` parameter, and without the `NoRecovery` parameter.

```
Restore-SqlDatabase `
-ReplaceDatabase `
-ServerInstance $instanceName `
-Database $restoreddbname `
-BackupFile $lastTxnBackupFile.FullName `
-ToPointInTime "2012-04-07 08:21:59"
```

An alternative is to restore this last transaction log backup file with `NoRecovery` as well, but add another command in the end to restore the database using `WITH RECOVERY`. In reality, it is safer to restore all the required transaction log backup files with `NORECOVERY` all the way through. This is safer because if we accidentally restore a file using `WITH RECOVERY`, the only way to correct it is to re-do the entire restore process. This may not be such a big deal for smaller databases, but for bigger databases this could be very time-consuming.

Once we have confirmed that all the required files have been restored, we can restore the database using `WITH RECOVERY`. One way to achieve this in our recipe, is by using a T-SQL statement, and passing this statement to our `Invoke-Sqlcmd` cmdlet:

```
#get the database out of Restoring state
#make the database accessible
$sql = "RESTORE DATABASE $restoreddbname WITH RECOVERY"
Invoke-Sqlcmd -ServerInstance $instanceName -Query $sql
```

The `RESTORE DATABASE` command takes our database from a restoring state, to an accessible and ready-to-use state. The `RESTORE` command rolls back all unfinished transactions and readies the database for use.

See also

- ▶ The *Creating a backup on mirrored media sets* recipe
- ▶ The *Creating a full backup* recipe
- ▶ The *Creating a differential backup* recipe
- ▶ The *Creating a transaction log backup* recipe
- ▶ The *Performing an online piecemeal restore* recipe
- ▶ You can check out how to do point-in-time restore using T-SQL:

[http://msdn.microsoft.com/en-us/library/ms179451\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms179451(v=sql.110).aspx)

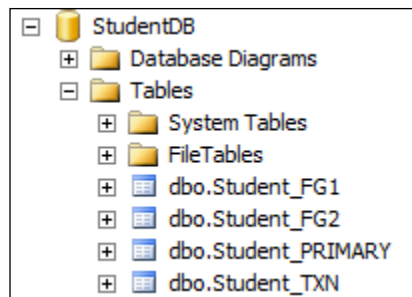
Performing an online piecemeal restore

In this recipe, we will perform an online piecemeal restore.

Getting ready

We will use a test database called `StudentDB` database, which has three filegroups—one primary, two custom filegroups `FG1` and `FG2`—in this recipe. Each of `FG1` and `FG2` will have one secondary datafile stored in the `C:\Temp` folder.

You can use the script `6464 - Ch05 - 11 - Perform an Online PieceMeal Restore - Prep.ps1` to create your files, which is included in the downloadable files for this book. When the script has finished executing, you should see the following database:



This is how the tables will be structured:

Table	Filegroup	Datafile name	Datafile location
Student_PRIMARY	PRIMARY	StudentDB.mdf	Default data directory
Student_FG1	FG1	Student_FG1_data	C:\Temp
Student_FG2	FG2	Student_FG2_data	C:\Temp
Student_TXN	PRIMARY	StudentDB.mdf	Default data directory

For our recipe, we will restore only the PRIMARY filegroup, and filegroup FG2 to our second SQL Server instance KERRIGAN\SQL01. At the end of our task, only Student_PRIMARY and Student_FG2 tables will be accessible.

Feel free to substitute this with a database available in your development environment that already has separate filegroups and filegroup backups.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the SQLPS module as follows:

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

3. Add the following script and run:

```
$instanceName = "KERRIGAN\SQL01"
$server = New-Object -TypeName Microsoft.SqlServer.Management.
Smo.Server -ArgumentList $instanceName
$backupfolder = "C:\Backup\"
```

```
#let's timestamp our databasename
#this is strictly for testing and checking purposes
$timestamp = Get-Date -Format yyyyMMddHHmmss
$restoreddbname = "StudentDBRestored_$( $timestamp )"
$relocatePath = "C:\Program Files\Microsoft SQL Server\MSSQL11.
SQL01\MSSQL\DATA"

#for this piecemeal restore, we need to specify
#files to restore

#primary filegroup
$primaryfgbackup = "C:\Backup\StudentDB_PRIMARY.bak"

#additional filegroup(s) to restore, and filegroup name
$fg2backup = "C:\Backup\StudentDB_FG2.bak"
$fg2name = "Student_FG2_data"

#transaction log backup
$txnbackup = "C:\Backup\StudentDB_TXN.bak"

#=====
#primary fg
#=====

#because we want to restore to a different instance,
#we need to create an array of files which will
#contain the new file locations of data and log
#files in the primary filegroup
$relocateFileList = @()

$smoRestore = New-Object Microsoft.SqlServer.Management.Smo.
Restore
$smoRestore.Devices.AddDevice($primaryfgbackup , [Microsoft.
SqlServer.Management.Smo.DeviceType]::File)
$smoRestore.ReadFileList($server) |
ForEach-Object {
    $relocateFile = Join-Path $relocatePath (Split-Path
    $_.PhysicalName -Leaf)
    $relocateFileList += New-Object Microsoft.SqlServer.
Management.Smo.RelocateFile($_.LogicalName, $relocateFile)
}

#=====
#restore primary fg
#partial must be used if restoring primary fg
```

```

#needs to be only mdf and ldf
#=====
Restore-SqlDatabase `
-Partial `
-ReplaceDatabase `
-ServerInstance $instanceName `
-Database $restoreddbname `
-BackupFile $primaryfgbackup `
-RelocateFile $relocateFileList `
-NoRecovery

#=====
#fg2
#=====
$relocateFileList = @()

#for the custom filegroup we want to restore, we want to
#relocate only that filegroup's datafiles
$smoRestore = New-Object Microsoft.SqlServer.Management.Smo.
Restore
$smoRestore.Devices.AddDevice($fg2backup , [Microsoft.SqlServer.
Management.Smo.DeviceType]::File)
$smoRestore.ReadFileList($server) |
ForEach-Object {
    if($_.LogicalName -eq $fg2name)
    {
        $relocateFile = Join-Path $relocatePath (Split-Path
$_PhysicalName -Leaf)
        $relocateFileList += New-Object Microsoft.SqlServer.
Management.Smo.RelocateFile($_.LogicalName, $relocateFile)
    }
}

#=====
#restore fg2
#dont need partial anymore
#=====
Restore-SqlDatabase `
-ReplaceDatabase `
-ServerInstance $instanceName `
-Database $restoreddbname `
-BackupFile $fg2backup `
-RelocateFile $relocateFileList `
-NoRecovery

```



```
#=====
#restore transaction log backup
#this will restore using with recovery
#=====
Restore-SqlDatabase `
-ReplaceDatabase `
-ServerInstance $instanceName `
-Database $restoredbname `
-BackupFile $txnbackup
```

How it works...

Online piecemeal restore is an Enterprise feature available starting from SQL Server 2005. This type of restore, also referred to as partial restore, allows you to stage your restores. With each restore sequence, one or more filegroups are available online, leaving the rest offline. The power of this feature is that as soon as your first filegroup is restored, the objects you have in this filegroup already become accessible to your end users or applications.

The first thing you will need to do is line up your files. You will need to specify where the PRIMARY filegroup backup, any user filegroups you want to restore, and the transaction log backup files are. In our recipe, we are also restoring the database to a different instance, so we will need to relocate our database files. For this reason, we must also specify what the filegroup names are for the filegroups we are restoring.

```
#primary filegroup
$primaryfgbackup = "C:\Backup\StudentDB_PRIMARY.bak"
```

```
#additional filegroup(s) to restore, and filegroup name
$fg2backup = "C:\Backup\StudentDB_FG2.bak"
$fg2name = "Student_FG2_data"
```

```
#transaction log backup
$txnbackup = "C:\Backup\StudentDB_TXN.bak"
```

Once we have the files lined up, we need to create an array that contains the files we are relocating:

```
$relocateFileList = @()
```

```
$smoRestore = New-Object Microsoft.SqlServer.Management.Smo.Restore
$smoRestore.Devices.AddDevice($primaryfgbackup , [Microsoft.SqlServer.
Management.Smo.DeviceType]::File)
$smoRestore.ReadFileList($server) |
ForEach-Object {
```

```

        $relocateFile = Join-Path $relocatePath (Split-Path
$_PhysicalName -Leaf)
        $relocateFileList += New-Object Microsoft.SqlServer.
Management.Smo.RelocateFile($_LogicalName, $relocateFile)
    }

```

We can then use our `Restore-SqlDatabase` cmdlet to restore the primary filegroup first with `NORECOVERY`. Note that when restoring the `PRIMARY` filegroup, you will need to specify the option `Partial`:

```

#=====
#restore primary fg
#partial must be used if restoring primary fg
#needs to be only mdf and ldf
#=====
Restore-SqlDatabase `
  -Partial `
  -ReplaceDatabase `
  -ServerInstance $instanceName `
  -Database $restoreddbname `
  -BackupFile $primaryfgbackup `
  -RelocateFile $relocateFileList `
  -NoRecovery

```

Next, for our user filegroups, we must still create an array that contains the specific filenames of the filegroup(s) we are restoring.

```

$relocateFileList = @()

#for the custom filegroup we want to restore, we want to
#relocate only that filegroup's datafiles
$smoRestore = New-Object Microsoft.SqlServer.Management.Smo.Restore
$smoRestore.Devices.AddDevice($fg2backup , [Microsoft.SqlServer.
Management.Smo.DeviceType]::File)
$smoRestore.ReadFileList($server) |
ForEach-Object {
    if($_LogicalName -eq $fg2name)
    {
        $relocateFile = Join-Path $relocatePath (Split-Path
$_PhysicalName -Leaf)
        $relocateFileList += New-Object Microsoft.SqlServer.
Management.Smo.RelocateFile($_LogicalName, $relocateFile)
    }
}

```

If we add items in the array that pertain to filegroups that we are not restoring, we are going to get an error like this:

```
Microsoft.SqlServer.Management.Smo.SmoException: System.Data.SqlClient.
SqlError: The operating system returned the error '5(Access is denied.)'
while attempting 'RestoreContainer::ValidateTargetForCreation' on ...
'c:\\Temp\\Student_FG1_data.ndf'
```

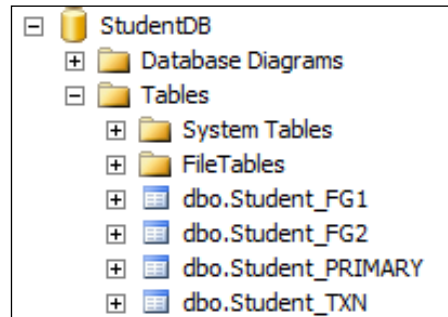
Once we have the array of relocated files, we can restore our user filegroup. Note that for this statement, we no longer need to specify the option `Partial`:

```
#=====
#restore fg2
#dont need partial anymore
#=====
Restore-SqlDatabase `
-ReplaceDatabase `
-ServerInstance $instanceName `
-Database $restoreddbname `
-BackupFile $fg2backup `
-RelocateFile $relocateFileList `
-NoRecovery
```

Lastly, we need to restore the transaction logfile(s). If there are multiple transaction logfiles, each transaction logfile before the final transaction logfile needs to be restored with `NORECOVERY`. The last transaction logfile can be restored using `WITH RECOVERY`.

```
#=====
#restore transaction log backup
#=====
Restore-SqlDatabase `
-ReplaceDatabase `
-ServerInstance $instanceName `
-Database $restoreddbname `
-BackupFile $txnbackup
```

What you should see after you restore this sequence is shown in the following screenshot:



It is a little bit deceiving because it looks like the whole database is already available and accessible. However, since we only restored FG2, only objects in FG2 are truly accessible. If you try to access any of the objects that reside in the unrestored filegroup, you will get an error similar to this:

```
Msg 8653, Level 16, State 1, Line 2
```

```
The query processor is unable to produce a plan for the table or  
view 'Student_FG1' because the table resides in a filegroup which  
is not online.
```

To restore the rest of your filegroups, you can use the same steps as described previously until the final filegroup is restored. Remember to always restore the filegroup, and then the transaction log backup. Lather, rinse, and repeat.

See also

- ▶ The *Creating a filegroup backup* recipe
- ▶ Learn more about performing piecemeal restores:

[http://msdn.microsoft.com/en-us/library/ms177425\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms177425(v=sql.110).aspx)

7

SQL Server Development

In this chapter, we will cover:

- ▶ Inserting XML into SQL Server
- ▶ Extracting XML from SQL Server
- ▶ Creating an RSS feed from SQL Server content
- ▶ Applying XSL to an RSS feed
- ▶ Storing binary data into SQL Server
- ▶ Extracting binary data from SQL Server
- ▶ Creating a new assembly
- ▶ Listing user-defined assemblies
- ▶ Extracting user-defined assemblies

Introduction

The last few versions of SQL Server have seen immense enhancements and support to different components that were traditionally not supported natively in databases, such as XML and **Common Language Runtime (CLR)** assemblies. This chapter explores how you can use PowerShell to simplify and automate some of the tasks you need to do with these items.

To do the exercises in this chapter:

4. Create a sample database named `SampleDB`, and use it for the tasks in this chapter:

```
CREATE DATABASE SampleDB
```

5. Download the files for this chapter from the Packt website, and save them to your local drive. You will find three folders in your downloaded package:
 - a. BLOB Files
 - b. CLR Files
 - c. XML Files

Inserting XML into SQL Server

In this recipe, we will insert the content of some XML files into a SQL Server table that has XML columns.

Getting ready

We will create a sample table that we can use for this recipe. Run the following in SQL Server Management Studio to create a table named `SampleXML` that has an XML field:

```
USE SampleDB
GO
IF OBJECT_ID('SampleXML') IS NOT NULL
    DROP TABLE SampleXML
GO

CREATE TABLE SampleXML
(
    ID INT IDENTITY(1, 1) NOT NULL PRIMARY KEY,
    FileName VARCHAR(200),
    InsertedDate DATETIME DEFAULT GETDATE(),
    InsertedBy VARCHAR(100) DEFAULT SUSER_SNAME(),
    XMLStuff XML,
    FileExtension VARCHAR(50)
)
```

Create a directory called `C:\XML Files\` and copy the sample XML files that come with the book scripts. Alternatively, you can use your own directory and XML files.

How to do it...

These are the steps to insert the contents of XML files into SQL Server:

1. Open the PowerShell console application by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the SQLPS module as follows:

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

3. Add the following script and run:

```
$VerbosePreference = "Continue"

#define variables for directory, instance, database
$xmlDirectory = "C:\XML Files\"
$instanceName = "KERRIGAN"
$databaseName = "SampleDB"

#get all XML files from your XML directory
Get-ChildItem $xmlDirectory -Filter "*.xml" |
ForEach-Object {

    #need to replace some illegal XML characters
    Write-Verbose "Importing $($_.FullName)    "

    #we need to escape single quotes
    #because we are passing the
    #XML content to a T-SQL statement
    [string]$xml = (Get-Content $_.FullName) -replace "'", "''"

    $query = @"
INSERT INTO SampleXML
(File Name, XML Stuff, File Extension)
VALUES ('$($_.Name)', '$xml', '.xml')
"@

    Invoke-Sqlcmd -ServerInstance $instanceName -Database
$databaseName -Query $query

}

$VerbosePreference = "SilentlyContinue"
```

When you are done, your result should look similar to this:

```
VERBOSE: Importing C:\XML Files\books.xml ...
VERBOSE: Importing C:\XML Files\sqlmusings_rss.xml ...
```


How it works...

Inserting the contents of an XML file into a SQL Server XML column is easily done with a combination of T-SQL and PowerShell.

PowerShell can perform file-related functions, while T-SQL can do `INSERT` statements more effectively.

The first step in this recipe is to loop through a set of XML files:

```
Get-ChildItem $xmlDirectory -Filter "*.xml"
```

We then pipe this to a `Foreach-Object` cmdlet that enables each file to be inserted into the table. Inside the `Foreach-Object` cmdlet, we display which file we are importing first:

```
#need to replace some illegal XML characters  
Write-Verbose "Importing $($_.FullName) ..."
```

We then extract the content of each XML file. Because we will be passing the content as text back to the server, we need to make sure we escape all single quotes. Otherwise the string we are inserting will be erroneously terminated.

```
[string]$xml = (Get-Content $_.FullName) -replace "'", "''"
```

Once the XML content is saved into a variable, we can compose an `INSERT` statement to insert into our table that has the XML column. Note that our `INSERT` statement is using a `here-string` variable.

```
$query = @"  
INSERT INTO SampleXML  
(FileName,XMLStuff,FileExtension)  
VALUES ('$($_.Name)', '$xml', '.xml')  
"@
```



Remember a `here-string` variable allows you to more easily create variables containing multi-line text. The text needs to start with `@` at the end of a line, and end with `"@` in a line by itself.

To perform the insert, we can use the `Invoke-SqlCmd` cmdlet and pass our `INSERT` query:

```
Invoke-Sqlcmd -ServerInstance $instanceName -Database  
$databaseName -Query $query
```

See also

- ▶ *The Extracting XML from SQL Server* recipe
- ▶ Learn more about SQL Server XML support from MSDN:
<http://msdn.microsoft.com/en-us/library/ms187339.aspx>

Extracting XML from SQL Server

In this recipe, we will extract the XML content from SQL Server and save each record back to individual files in the filesystem.

Getting ready

For this recipe, we will use the table we created in the previous recipe, *Inserting XML into SQL Server*, to extract files. Feel free to use your own tables that have XML columns; just ensure you change the table name in the script.

How to do it...

These are the steps to extract XML from SQL Server:

1. Open the PowerShell console application by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the SQLPS module as follows:

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

3. Add the following script and run:

```
$VerbosePreference = "Continue"
$instanceName = "KERRIGAN"
$databaseName = "SampleDB"
$foldername = "C:\XML Files\"

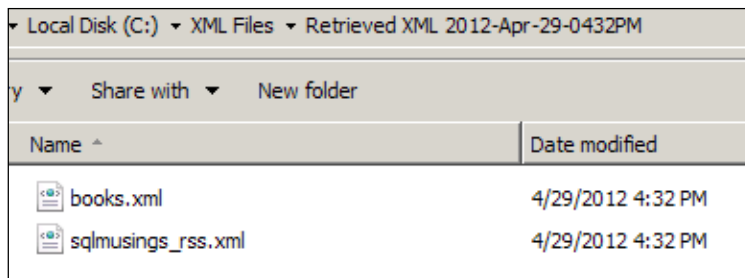
#we will save all retrieved files in a new folder
$newchildfolder = "Retrieved XML $(Get-Date -format 'yyyy-MMM-dd-
hhmm') "
$newfolder = Join-Path -Path "$($foldername)" -ChildPath
$newchildfolder

#If the path exists, will error silently and continue
New-Item -ItemType directory -Path $newfolder -ErrorAction
SilentlyContinue

#query to get XML content from database
$query = @"
SELECT FileName, XMLStuff
FROM SampleXML
WHERE XMLStuff IS NOT NULL
"@
```

```
Invoke-Sqlcmd -ServerInstance $instanceName -Database
$databaseName -Query $query -MaxCharLength 99999999 |
ForEach-Object {
    Write-Verbose "Retrieving $($_.FileName) ..."
    [xml]$xml = $_.XmlStuff
    $xml.Save((Join-Path -Path $newfolder -ChildPath
"$($_.FileName)"))
}
explorer $newfolder
$VerbosePreference = "SilentlyContinue"
```

When you are done, go to your folder and you will see something similar to this:



How it works...

SQL Server has great support for querying and manipulating XML stored in SQL Server tables, but needs external support if these files need to be extracted and saved back to the filesystem. PowerShell can definitely help in this area.

We first create a new timestamped folder where we can store our retrieved XML files. This will help us keep track of which files were downloaded at any specific time. We use the `New-Item` cmdlet to create this new folder. If the folder already exists, no error will be displayed since we specified the parameter `-ErrorAction SilentlyContinue`.

```
#we will save all retrieved files in a new folder
$newchildfolder = "Retrieved XML $(Get-Date -format 'yyyy-MMM-dd-
hhmmst') "
$newfolder = Join-Path -Path "$($foldername)" -ChildPath
$newchildfolder

#If the path exists, will error silently and continue
New-Item -ItemType directory -Path $newfolder -ErrorAction
SilentlyContinue
```

We then construct our T-SQL statement to retrieve the XML data from our table.

```
$query = @"
SELECT FileName, XMLStuff
FROM SampleXML
WHERE XMLStuff IS NOT NULL
"@
```

We can pass this to the `Invoke-Sqlcmd` cmdlet to retrieve all our XML records. We also have to specify a big number for the variable `MaxCharLength`, which defines the maximum number of characters returned for columns, because the content of the XML files we are retrieving will be big. By default, the `MaxCharLength` value is 4000.

```
Invoke-Sqlcmd -ServerInstance $instanceName -Database $databaseName
-Query $query -MaxCharLength 99999999 |
ForEach-Object {
    Write-Verbose "Retrieving $($_.FileName) ..."
    [xml]$xml = $_.XmlStuff
    $xml.Save((Join-Path -Path $newfolder -ChildPath "$($_.FileName)"))
}
```

For each record returned in our query result, we save the content back to a strongly typed XML variable, by putting `[xml]` right beside our `$xml` variable.

```
ForEach-Object {
    Write-Verbose "Retrieving $($_.FileName) ..."
    [xml]$xml = $_.XmlStuff
    $xml.Save((Join-Path -Path $newfolder -ChildPath "$($_.FileName)"))
}
```

The XML variable, because it is an XML object, will have inherited a `Save` method that allows us to save the content back to the filesystem.

```
ForEach-Object {
    Write-Verbose "Retrieving $($_.FileName) ..."
    [xml]$xml = $_.XmlStuff
    $xml.Save((Join-Path -Path $newfolder -ChildPath "$($_.FileName)"))
}
```

See also

- ▶ *The Inserting XML into SQL Server recipe*

Creating an RSS feed from SQL Server content

In this recipe, we will create an RSS feed from SQL Server content.

Getting ready

For this task, we will use a trivial query to populate our RSS feed. We will just list our database list from `sys.databases`, and use that as fictional content for our RSS file.

How to do it...

These are the steps to create an RSS feed using T-SQL and PowerShell.

1. Open the PowerShell console application by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the SQLPS module as follows:

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

3. Add the following script and run:

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

$instanceName = "KERRIGAN"
$databaseName = "SampleDB"
$timestamp = Get-Date -Format "yyy-MM-dd-hhmm"
$rssFileName = "C:\XML Files\rss_{$timestamp}.xml"

#values to be used for RSS
$rssTitle = "QueryWorks Latest News"
$rssLink = "http://www.queryworks.ca/rss.xml"
$rssDescription = "What's new in the world of QueryWorks"

#use r as date formatter to get
#date in RFC1123Pattern
$rssDate = (Get-Date -Format r)
$rssManagingEditor = "info@queryworks.ca"
$rssGenerator = "SQL Server 2012 XML and PowerShell"
$rssDocs = "http://www.queryworks.ca/rss.xml"
```

```

$query = @"
DECLARE @rssbody XML
SET @rssbody = ( SELECT
                    name AS 'title' ,
                    collation_name AS 'description' ,
                    'false' AS 'guid/@isPermaLink' ,
                    'http://www.queryworks.ca/?p=' +
                    CAST(database_id AS VARCHAR(5)) AS 'guid'
                FROM
                    sys.databases
                FOR XML PATH('item') , TYPE)

SELECT @rssbody
"@

$rssFromSQL = Invoke-Sqlcmd -ServerInstance $instanceName
-Database $databaseName -Query $query

#extract the RSS from the SQL Server result
[string] $rssBody = $rssFromSQL.Column1.ToString()

#create the final RSS
$rsstext = @"
<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
<channel>
    <title><![CDATA[$rssTitle]]></title>
    <atom:link href="http://www.queryworks.ca/rss.xml" rel="self"
type="application/rss+xml" />
    <link>$rssLink</link>
    <description><![CDATA[$rssDescription]]></description>
    <pubDate>$rssDate</pubDate>
    <lastBuildDate>$rssDate</lastBuildDate>
    <managingEditor>$rssManagingEditor</managingEditor>
    <generator>$rssGenerator</generator>
    <docs>$rssDocs</docs>
    $rssBody
</channel>
</rss>
"@

[xml] $rss = $rsstext
$rss.Save($rssFileName)

```

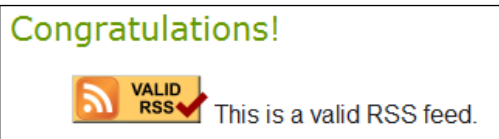
When the script has finished executing, open the RSS file. The content of the file should look similar to this:

```

rss_2012-Apr-29-1002AM.xml X
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
  <channel>
    <title><![CDATA[QueryWorks Latest News]]></title>
    <atom:link href="http://www.queryworks.ca/rss.xml" rel="self" type="applicat
    <link>http://www.queryworks.ca/rss.xml</link>
    <description><![CDATA[What's new in the world of QueryWorks]]></description>
    <pubDate>Sun, 29 Apr 2012 10:02:40 GMT</pubDate>
    <lastBuildDate>Sun, 29 Apr 2012 10:02:40 GMT</lastBuildDate>
    <managingEditor>info@queryworks.ca</managingEditor>
    <generator>SQL Server 2012 XML and PowerShell</generator>
    <docs>http://www.queryworks.ca/rss.xml</docs>
    <item>
      <title>master</title>
      <description>SQL_Latin1_General_CP1_CI_AS</description>
      <guid isPermaLink="false">http://www.queryworks.ca/?p=1</guid>
    </item>
    <item>
      <title>tempdb</title>
      <description>SQL_Latin1_General_CP1_CI_AS</description>
      <guid isPermaLink="false">http://www.queryworks.ca/?p=2</guid>
    </item>
    <item>
      <title>model</title>
      <description>SQL_Latin1_General_CP1_CI_AS</description>
      <guid isPermaLink="false">http://www.queryworks.ca/?p=3</guid>
    </item>
  </channel>
</rss>

```

To validate, [www.w3.org](http://validator.w3.org/feed/check.cgi) has an RSS feed validator at <http://validator.w3.org/feed/check.cgi>. Use the tab **Validate by Direct Input**, and copy the contents of the file into the textarea. Click on the **Validate** button. If validated, you should see a message like this:



How it works...

SQL Server has embraced support for XML since version 2005. While creating the content for RSS feeds is doable using T-SQL in SQL Server, there are still some challenges with composing the RSS file. For example, the RSS file should have the following header:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Although adding this line at the beginning of the content is doable in SQL Server, it is not very straightforward. It will take a few `CAST` functions to get your RSS feed content properly formatted. When you are done with the formatting, you will still need to use another means or tool to save this back to an XML file.

Combining T-SQL with PowerShell allows you to accomplish creating the RSS feed file with ease.

The first thing we do is define a timestamped filename:

```
$timestamp = Get-Date -Format "yyyy-MMM-dd-hhmmtt"
$rssFileName = "C:\XML Files\rss_{$timestamp}.xml"
```

We then have to define the parameters we want to use to populate our RSS header. These include the `title`, `link`, `description`, `date`, `managingEditor`, `generator`, and `docs` variables. We will insert these variables later in the actual RSS feed string:

```
#values to be used for for RSS
$rssTitle = "QueryWorks Latest News"
$rssLink = "http://www.queryworks.ca/rss.xml"
$rssDescription = "What's new in the world of QueryWorks"

#use r as date formatter to get
#date in RFC1123Pattern
$rssDate = (Get-Date -Format r)
$rssManagingEditor = "info@queryworks.ca"
$rssGenerator = "SQL Server 2012 XML and PowerShell"
$rssDocs = "http://www.queryworks.ca/rss.xml"
```

To retrieve data from our SQL Server table, we define a here-string query. Note here, to get the content in the XML format that we want, we use the `FOR XML PATH` with our query:

```
$query = @"
DECLARE @rssbody XML
SET @rssbody = ( SELECT
                    name AS 'title' ,
                    collation_name AS 'description' ,
                    'false' AS 'guid/@isPermaLink' ,
                    'http://www.queryworks.ca/?p=' +
                    CAST(database_id AS VARCHAR(5)) AS 'guid'
                FROM
                    sys.databases
                FOR XML PATH('item') , TYPE)

SELECT @rssbody
"@
```


This query will give you a result similar to this:

```
<item>
  <title>master</title>
  <description>SQL_Latin1_General_CP1_CI_AS</description>
  <guid isPermaLink="false">http://www.queryworks.ca/?p=1</guid>
</item>
<item>
  <title>tempdb</title>
  <description>SQL_Latin1_General_CP1_CI_AS</description>
  <guid isPermaLink="false">http://www.queryworks.ca/?p=2</guid>
</item>
```

When we execute the query, we can use the `Invoke-Sqlcmd` cmdlet, and capture the result using another PowerShell variable.

```
$rssFromSQL = Invoke-Sqlcmd -ServerInstance $instanceName -Database
$databaseName -Query $query
```

Remember, though, that our result from our `Invoke-Sqlcmd` cmdlet is still a table, so we still need to extract just the XML content from the result. We do this by extracting what's been returned in `Column1` (that is, the first column of the result), and saving this as a string:

```
#extract the RSS from the SQL Server result
[string] $rssBody = $rssFromSQL.Column1.ToString()
```

Once we have all the information, we can formulate the RSS file. Note that we are using a here-string variable as the main template, and each tag is populated by the values we set for our RSS-related variables. These are the variables (shown in bold) embedded in the here-string query below:

```
#create the final RSS
$rsstext = @"
<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
<channel>
  <title><![CDATA[$rssTitle]]></title>
  <atom:link href="http://www.queryworks.ca/rss.xml" rel="self"
type="application/rss+xml" />
  <link>$rssLink</link>
  <description><![CDATA[$rssDescription]]></description>
  <pubDate>$rssDate</pubDate>
  <lastBuildDate>$rssDate</lastBuildDate>
  <managingEditor>$rssManagingEditor</managingEditor>
  <generator>$rssGenerator</generator>
  <docs>$rssDocs</docs>
  $rssBody
</channel>
```

```
</rss>
"@
```

To validate and create the file, we need to create a strongly typed XML variable. We are hitting two birds with one stone this way. This can check for well-formed XML. If the XML is not well formed, we will get an error when we try to assign our content to the XML variable.

```
#this can validate the RSS file
[xml]$rss = $rsstext
```

The XML object also comes with a `Save` method that allows us to save the content to a file on a disk.

```
$rss.Save($rssFileName)
```

There's more...

RSS stands for **Really Simple Syndication**. It allows items such as blog entries and news items to be syndicated or published automatically, and consumed by RSS readers from different devices. An RSS feed is nothing more than a specific-formatted XML file that contains specific information such as author, title, description, and the like.

Learn more about RSS feeds and their variations from <http://cyber.law.harvard.edu/rss/rss.html> and <http://www.rss-specifications.com/rss-specifications.htm>.

On the SQL Server side, to learn more about creating XML documents from your records, read up on the FOR XML clause from <http://msdn.microsoft.com/en-us/library/ms190922.aspx>.

See also

- ▶ *The Applying XSL to an RSS feed recipe*

Applying XSL to an RSS feed

In this recipe, we will create a styled HTML file based on an existing RSS feed and XSL (stylesheet).

Getting ready

The files needed for this recipe are included in the downloadable book scripts from Packt. Once downloaded, copy the XML Files\RSS folder to your local C:\ directory. This folder will have two files: one sample RSS feed (`sample_rss.xml`) and one sample XSL file (`rss_style.xsl`).

How to do it...

These are the steps for styling an RSS feed:

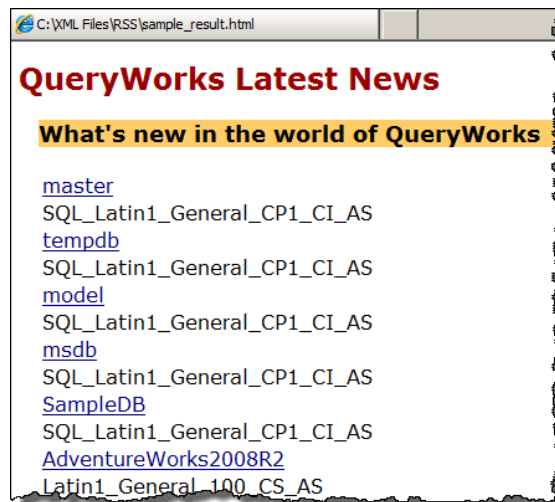
1. Open the PowerShell console application by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run:

```
$xsl = "C:\XML Files\RSS\rss_style.xsl"
$rss = "C:\XML Files\RSS\sample_rss.xml"
$styled_rss = "C:\XML Files\RSS\sample_result.html"

$xslt = New-Object System.Xml.Xsl.XslCompiledTransform
$xslt.Load($xsl)
$xslt.Transform($rss, $styled_rss)

#load the resulting styled html
#in Internet Explorer
Set-Alias ie "$env:programfiles\Internet Explorer\iexplore.exe"
ie $styled_rss
```

When done, an Internet Explorer browser will open and show a page similar to this:



How it works...

XSL stands for **Extensible Stylesheet Language**. It is a stylesheet, similar to its *cousin* **Cascading Style Sheets (CSS)**, which defines how an XML document can be styled and potentially transformed.

Although this recipe may not be directly related to SQL Server, knowing how to apply this may have some benefits to the SQL Server professional.

To style our RSS feed, we will first create some variables that contain our .xsl file and our .xml file (or the RSS feed file). For our recipe, we will style the RSS to produce an HTML file, so we will create a variable to reference this new file as well:

```
$xsl = "C:\XML Files\RSS\rss_style.xsl"
$rss = "C:\XML Files\RSS\sample_rss.xml"
$styled_rss = "C:\XML Files\RSS\sample_result.html"
```

The content of our XSL file looks like this:

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3 <xsl:template match="/rss">
4 <html>
5 <head>
6 <style type="text/css">
7   body {
8     font-family: Verdana,"sans-serif";
9   }
10  h2 a, h2 a:link
11  {
12    text-decoration:none;
13    color: #990000;
14  }
15  .channeldescrip
16  {
17    background-color: #FFCC66;
18  }
19 </style>
20 </head>
21 <body>
22   <div id="rssheader">
23     <h2>
24       <xsl:element name="a">
25         <xsl:attribute name="href">
26           <xsl:value-of select="channel/link" />
27         </xsl:attribute>
28         <xsl:value-of select="channel/title" />
29       </xsl:element>
30     </h2>
31   </div>
32   <div class="rsscontents" style="border-width:0; background-color:#FFF; ma
33     <div class="channeldescrip">
34       <h3>
35         <xsl:value-of select="channel/description" />
36       </h3>
```

It is important to show a sample section of the XSL to help map visually where the RSS items are incorporated.

The styling of the XML with XSL is done using the .NET class `XslCompiledTransform`.

```
$xslt = New-Object System.Xml.Xsl.XslCompiledTransform
```

To transform our RSS feed, which is a simple XML file, into a styled HTML file, the XSL (stylesheet) needs to be loaded using the `Load` method of the `XslCompiledTransform` variable.

```
$xslt.Load($xsl)
```

The actual transformation and styling happens when the `Transform` method of the `XslCompiledTransform` object is invoked, and passed with the XML content and a handle (or variable) to the resulting HTML file.

```
$xslt.Transform($rss, $styled_rss)
```

The last piece we added is just to display the resulting HTML file in Internet Explorer. We create an alias for Internet Explorer using the `Set-Alias` cmdlet, and use it to open our resulting HTML file.

```
#load the resulting styled html
#in Internet Explorer
Set-Alias ie "$env:programfiles\Internet Explorer\iexplore.exe"
ie $styled_rss
```

See also

- ▶ The *Creating an RSS Feed from SQL Server content* recipe
- ▶ To learn more about XSL, visit the www.w3.org official XSL documentation:
<http://www.w3.org/Style/XSL/WhatIsXSL.html>
- ▶ In addition, check out the MSDN documentation on the `XslCompiledTransform` .NET class:
<http://msdn.microsoft.com/en-us/library/system.xml.xsl.xslcompiledtransform.aspx>

Storing binary data into SQL Server

In this recipe, we will store some binary data, including some images, a PDF, and a Word document, into SQL Server.

Getting ready

Let's create a sample table we can use for this recipe. Run the following in SQL Server Management Studio to create a table called `SampleBLOB` that has a BLOB, or `VARBINARY (MAX)`, field:

```
USE SampleDB
GO
IF OBJECT_ID('SampleBLOB') IS NOT NULL
    DROP TABLE SampleBLOB
GO

CREATE TABLE SampleBLOB
(
    ID INT IDENTITY(1, 1) NOT NULL PRIMARY KEY,
    FileName VARCHAR(200) ,
    InsertedDate DATETIME DEFAULT GETDATE() ,
    InsertedBy VARCHAR(100) DEFAULT SUSER_SNAME() ,
    BLOBstuff VARBINARY(MAX) ,
    FileExtension VARCHAR(50)
)
```

Create a directory called `C:\BLOB Files\` and copy the sample BLOB files that come with the book scripts, or use your own directory and BLOB files.

How to do it...

These are the steps to save binary data into SQL Server:

1. Open the PowerShell console application by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the `SQLPS` module as follows:

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

3. Add the following script and run:

```
$VerbosePreference = "Continue"
$instanceName = "KERRIGAN"
$databaseName = "SampleDB"
$folderName = "C:\BLOB Files\"
```

```
#using PowerShell V2 style Where-Object syntax
Get-ChildItem $folderName |
Where-Object {$_.PSIsContainer -eq $false} |
ForEach-Object {
    $blobFile = $_
    $fileExtension = $blobFile.Extension
    Write-Verbose "Importing file $($blobFile.FullName)..."

    $query = @"
INSERT INTO SampleBLOB
(FileName, FileExtension, BLOBStuff)
SELECT '$blobFile','$fileExtension',*
FROM OPENROWSET(BULK N'$folderName$blobFile', SINGLE_BLOB) as
tmpImage
"@

    Invoke-Sqlcmd -ServerInstance $instanceName -Database
    $databaseName -Query $query
    Start-Sleep -Seconds 2

}
$VerbosePreference = "SilentlyContinue"
```

When you're done, you should see something similar to this:

```
VERBOSE: Importing file C:\BLOB Files\Hello SQLSaturday 114.docx...
VERBOSE: Importing file C:\BLOB Files\speakerevals.jpg...
VERBOSE: Importing file C:\BLOB Files\sqlsat108.png...
VERBOSE: Importing file C:\BLOB Files\SSRS CheatSheet.pdf...
```

How it works...

Inserting the contents of a binary file into a SQL Server table can be made easier with the combination of T-SQL and PowerShell.

In this recipe we have a few files—a PDF, a Word document, and a few images—that we want to store to SQL Server.

To start, we first need to define which folder we are importing, and to which instance and which database we are importing from:

```
$instanceName = "KERRIGAN"
$databaseName = "SampleDB"
$folderName = "C:\BLOB Files\"
```

We then pipe a series of cmdlets to accomplish our task. First we use the `Get-ChildItem` cmdlet to get all our files. In our recipe, we import all the files in `C:\BLOB Files`.

```
#using PowerShell V2 style Where-Object syntax
Get-ChildItem $folderName |
Where-Object {$_.PSIsContainer -eq $false} |
```

We exclude folders by specifying `Where-Object {$_.PSIsContainer -eq $false}`. Of course, you have an option of filtering by file extensions if you want. You can just add the `-Include` parameter for `Get-ChildItem` and specify which extensions you want to import, as such:

```
Get-ChildItem -Path "C:\BLOB Files\*.*" -Include *.jpg,*.png
```

The `Foreach-Object` cmdlet then takes each file we retrieve, and composes a T-SQL statement that inserts the file into our `SampleBLOB` table. We use `OPENROWSET` to import the contents of the binary file as a `SINGLE_BLOB` file.

```
$blobFile = $_
$fileExtension = $blobFile.Extension
Write-Verbose "Importing file $($blobFile.FullName)..."

$query = @"
INSERT INTO SampleBLOB
(fileName, FileExtension, BLOBstuff)
SELECT '$blobFile','$fileExtension', *
FROM OPENROWSET(BULK N'$folderName$blobFile', SINGLE_BLOB) as tmpImage
"@
```

This T-SQL statement is then passed to the `Invoke-Sqlcmd` cmdlet, which executes the statement on our instance. We also sleep for 2 seconds to give the command some time to complete.

```
Invoke-Sqlcmd -ServerInstance $instanceName -Database $databaseName
-Query $query
Start-Sleep -Seconds 2
```

There's more...

Read more about the `OPENROWSET` method at <http://msdn.microsoft.com/en-us/library/ms190312.aspx>.

See also

- ▶ *The Extracting binary data from SQL Server recipe*

Extracting binary data from SQL Server

In this recipe, we will extract binary content from SQL Server and save it back to individual files in the filesystem.

Getting ready

For this recipe, we will use the table we created in the previous recipe, *Inserting binary data into SQL Server*, to extract files. Feel free to use your own tables that have `VARBINARY (MAX)` columns; just ensure you change the table name in the script.

In addition to our `SampleBLOB` table, we will create an empty table with a single `VARBINARY (MAX)` table. We will use this for facilitating the creation of a format file we need for exporting binary data out of SQL Server using `bcp`.

```
USE SampleDB
GO
IF OBJECT_ID('EmptyBLOB') IS NOT NULL
    DROP TABLE EmptyBLOB
GO
CREATE TABLE EmptyBLOB
(
    BLOBstuff VARBINARY (MAX)
)
```

How to do it...

These are the steps to extract binary data from SQL Server.

1. Open the **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the SQLPS module as follows:

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

3. First we will create a `bcp` format file. Add the following script and run:

```
$timestamp = Get-Date -Format "yyyy-MMM-dd-hhmmtt"
$emptyBLOB_tableName = "SampleDB.dbo.EmptyBLOB"
$formatFileName = "C:\BLOB Files/blob${$timestamp}.fmt"
$fmtcmd = "bcp `"$emptyBLOB_tableName`" format nul -T -N -f
`"$formatfilename`" -S $instanceName"

#create the format file
Invoke-Expression -Command $fmtcmd
```

```
#now there is a problem, by default the format file
#will use 8 as prefix length for varbinary
#we need this to be zero, so we will replace
(Get-Content $formatFileName) |
ForEach-Object { $_ -replace "8", "0" } |
Set-Content $formatFileName
```

4. After our format file is created, we will export our BLOB content from SQL Server to files in our filesystem. Run the following script:

```
$databaseName = "SampleDB"
$folderName = "C:\BLOB Files\"
$newFolderName = "Retrieved BLOB $timestamp"

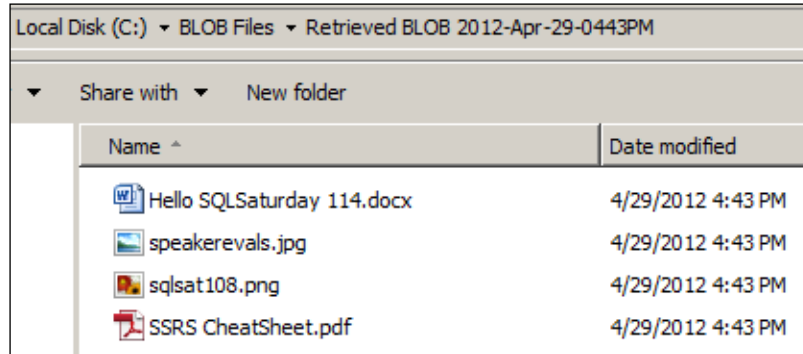
$newFolder = Join-Path -Path "$($foldername)" -ChildPath
$newfoldername

#If the path exists, will error silently and continue
New-Item -ItemType directory -Path $newfolder -ErrorAction
SilentlyContinue

$query = @"
SELECT ID, FileName
FROM SampleBLOB
"@

Invoke-Sqlcmd -ServerInstance $instanceName -Database
$databaseName -Query $query |
ForEach-Object {
    $item = $_
    Write-Verbose "Retrieving $($item.FileName) ..."
    $newFileName = Join-Path $newFolder $item.FileName
    $blobQuery = @"
SELECT BLOBstuff
FROM SampleBLOB
WHERE ID = $($item.ID)
"@
    $cmd = "bcp `"$blobQuery`" queryout `"$newFileName`" -S $server
-T -d $databaseName -f `"$formatFileName`""
    Invoke-Expression $cmd
}
explorer $newFolder
$VerbosePreference = "SilentlyContinue"
```

When you are done, your retrieved files should look like this:



How it works...

To retrieve a BLOB, or binary large object, from SQL Server and saving it back to the filesystem, we utilize a combination of T-SQL and PowerShell cmdlets.

The most important part of retrieving binary data and save them back to a file format is preserving the raw format and encoding. We our data using `bcp` with a format file. To help us create this format file, we created a simple table in our prep section that has a single `VARBINARY (MAX)` column.

To create the format file, we use the dynamically built `bcp` command that will create the format file.

```
$fmtcmd = "bcp `"$emptyBLOB_tableName`" format nul -T -N -f  
`"$formatfilename`" -S $instanceName"
```

A fully composed command will look similar to:

```
bcp "SampleDB.dbo.EmptyBLOB" format nul -T -N -f "C:\BLOB Files\  
blob2012-Apr-29-0443PM.fmt" -S KERRIGAN
```

The options we specified in our `bcp` are (based on Books Online):

Option	Description
<code>format nul -f</code>	Specifies the non-XML format file
<code>-T</code>	Indicates a trusted connection
<code>-N</code>	Specifies to perform <code>bcp</code> using native data types for noncharacter data, and Unicode character data

To create the file, we can use the `Invoke-Expression` command to execute the `bcp` command against the server.

```
Invoke-Expression -Command $fmtcmd
```

This will create a format file that contains:

```
11.0
1
1      SQLBINARY      8      0      ""      1      BLOBstuff      ""
```

Unfortunately, the `bcp` command that creates the format file automatically assigned a prefix length of 8 for our `SQLBINARY` data. This will create problems for our binary file because it adds additional characters to our file, which can "corrupt" the file. We want to replace this prefix length with zero (0), and we do it using this code:

```
(Get-Content $formatFileName) |
ForEach-Object { $_ -replace "8", "0" } |
Set-Content $formatFileName
```

Once our format file is ready, we create our timestamped folder.

```
$newFolderName = "Retrieved BLOB $timestamp"

$newFolder = Join-Path -Path "$($foldername)" -ChildPath
$newfoldername

#If the path exists, will error silently and continue
New-Item -ItemType directory -Path $newfolder -ErrorAction
SilentlyContinue
```

We then get all the records from our `SampleBLOB` table. We will first only get the `ID` and `FileName` variables.

```
$query = @"
SELECT ID, FileName
FROM SampleBLOB
"@

Invoke-Sqlcmd -ServerInstance $instanceName -Database $databaseName
-Query $query |
```

For each record we retrieve that contains the `ID` and `FileName` variables, we query SQL Server again, but this time for the binary content. We use this query in another `bcp` command we are constructing. This `bcp` command uses the format file we created in the previous section. We pass this `bcp` command again to the `Invoke-Expression` cmdlet to create the binary file in the filesystem.

```
ForEach-Object {
    $item = $_
    Write-Verbose "Retrieving $($item.FileName) ..."
    $newFileName = Join-Path $newFolder $item.FileName
    $blobQuery = @"
SELECT BLOBstuff
FROM SampleBLOB
WHERE ID = $($item.ID)
"@
    $cmd = "bcp `"$blobQuery`" queryout `"$newFileName`" -S $server -T -d
    $databaseName -f `"$formatFileName`""
    Invoke-Expression $cmd
}
```

There's more...

Read more about bcp:

<http://msdn.microsoft.com/en-us/library/ms162802.aspx>

See also

- ▶ [The Storing binary data into SQL Server recipe](#)

Creating a new assembly

In this recipe, we will create a new user-defined assembly.

Getting ready

Create a folder named `C:\CLR Files` and copy the `QueryWorksCLR.dll` file that comes with the book's sample files into this folder.

We will load this to the `SampleDB` database. Feel free to use a database accessible to you; just ensure you replace the database name in the script.

How to do it...

These are the steps to create a new assembly in SQL Server:

1. Open the **PowerShell** console application by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the SQLPS module as follows:

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

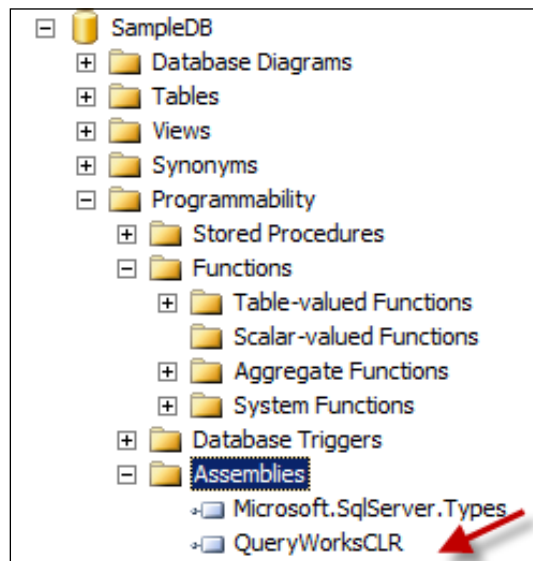
3. Add the following script and run:

```
$instanceName = "KERRIGAN"
$databaseName = "SampleDB"
$assemblyName = "QueryWorksCLR"
$assemblyFile = "C:\CLR Files\QueryWorksCLR.dll"
```

```
#this is for SAFE assemblies only
$query = @"
CREATE ASSEMBLY $assemblyName
FROM '$assemblyFile'
WITH PERMISSION_SET = SAFE
"@
```

```
Invoke-Sqlcmd -ServerInstance $instanceName -Database
$databaseName -Query $query
```

4. When you are done, open **SQL Server Management Studio**. Go to the database and open **Programmability | Assemblies**. Check that the assembly has been created as shown in the following screenshot:



How it works...

Starting with version 2005, SQL Server has supported integration with the Common Language Runtime (CLR). This means that you can create .NET code in your language of preference, compile it into **DLL (Dynamic Linked Library)** files, and create these as SQL Server database objects called **assemblies**.

Creating an assembly in SQL Server can be straightforward. In this recipe, we looked at the simplest case, where we create an assembly with *SAFE* access.

To create the assembly, we need to specify where the DLL is located, and pass it to a `CREATE ASSEMBLY` T-SQL statement:

```
$assemblyFile = "C:\CLR Files\QueryWorksCLR.dll"

#this is for SAFE assemblies only
$query = @"
CREATE ASSEMBLY $assemblyName
FROM '$assemblyFile'
WITH PERMISSION_SET = SAFE
"@
```

Once the parameters are defined, we simply use the `Invoke-Sqlcmd` cmdlet to create the assembly.

```
Invoke-Sqlcmd -ServerInstance $instanceName -Database $databaseName
-Query $query
```

Note that in SQL Server, an assembly might be successfully created and database objects can be created from it (for example, SQLCLR functions and stored procedures), but these will not be usable until SQLCLR integration has been enabled in your instance. This can be done using the T-SQL stored procedure `sp_configure`, or using PowerShell.

To enable SQLCLR using T-SQL, we can use:

```
EXEC sp_configure 'show advanced options', 1
GO
RECONFIGURE
GO
EXEC sp_configure 'clr enabled', 1
GO
RECONFIGURE
GO
```

To do the same thing using PowerShell, we can use the following snippet after we create the `$server` SMO object:

```
$server.Configuration.IsSqlClrEnabled.ConfigValue = 1
$server.Alter()
```

There's more...

CLRs can be very powerful components within a SQL Server environment, thus there needs to be control as to what is allowed and not allowed to do. A lot of this can be controlled through **Code Access Security (CAS)**. There are three security levels, and simply put, these are the differences between them:

Permission Setting	Description
SAFE	Restricted to internal computation, and local SQL Server access Cannot access external resources such as files, folders, and so on
EXTERNAL_ACCESS	Allows external access to files, registry, networks, and so on By default executes as the SQL Server service account
UNSAFE	Least restrictive Can potentially do anything CLRs can do

We have only covered how to deploy `SAFE` assemblies. `EXTERNAL_ACCESS` and `UNSAFE` can be a bit more complicated, and will require creating certificates, logins, and symmetric/asymmetric keys.



Check out the section on Creating `EXTERNAL_ACCESS` and `UNSAFE` Assemblies from the MSDN article *CLR Integration Code Access Security*: <http://msdn.microsoft.com/en-us/library/ms345101.aspx>.

Note that this article strongly encourages *not* to set the `TRUSTWORTHY` property of your database to `ON`.

See also

- ▶ The *Listing user-defined assemblies* recipe
- ▶ The *Extracting user-defined assemblies* recipe

Listing user-defined assemblies

In this recipe, we will list the user-defined assemblies in a SQL Server database.

Getting ready

We can use the `SampleDB` database that we used in the previous recipe, or you can substitute this with any database that is accessible to you that has some user-defined assemblies.

How to do it...

These are the steps to list user-defined assemblies:

1. Open the PowerShell console application by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the SQLPS module as follows:

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

3. Add the following script and run:

```
$instanceName = "KERRIGAN"
$server = New-Object `
-TypeName Microsoft.SqlServer.Management.Smo.Server `
-ArgumentList $instanceName
$databaseName = "SampleDB"
$database = $server.Databases[$databaseName]

#list assemblies except system assemblies
#using PowerShell V3 syntax
$database.Assemblies | Where-Object IsSystemObject -eq $false
```

How it works...

Listing user-defined assemblies is a straightforward task.

After importing the SQLPS module, we create a server handle and database handle:

```
$instanceName = "KERRIGAN"
$server = New-Object `
-TypeName Microsoft.SqlServer.Management.Smo.Server `
-ArgumentList $instanceName
$databaseName = "SampleDB"
$database = $server.Databases[$databaseName]
```

An assembly is a database-level object, which means we can access assemblies through our database variable. We also want to filter out any system assemblies. Note we are using the PowerShell V3 `Where-Object` syntax.

```
$database.Assemblies | Where-Object IsSystemObject -eq $false
```

To do this using the PowerShell V2 `Where-Object` syntax, we need to add the curly braces and use `$_`.

```
$database.Assemblies | Where-Object {$_.IsSystemObject -eq $false}
```

There's more...

Learn more about SQLCLR assemblies from MSDN:

[http://msdn.microsoft.com/en-us/library/ms254498\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms254498(v=vs.110).aspx)

See also

- ▶ *The Extracting user-defined assemblies recipe*

Extracting user-defined assemblies

In this recipe, we will extract user-defined assemblies and resave these back to the filesystem as DLLs.

Getting ready

We can use the `SampleDB` database that we used in the previous recipe, or you can substitute this with any database that is accessible to you that has some user-defined assemblies.

How to do it...

These are the steps to extract user-defined assemblies:

1. Open the **PowerShell** console application by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the SQLPS module as follows:

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

3. First, we will create a bcp format file. Add the following script and run:

```
$VerbosePreference = "Continue"
$instanceName = "KERRIGAN"

$timestamp = Get-Date -Format "yyyy-MMM-dd-hhmm"
$emptyBLOB_tableName = "SampleDB.dbo.EmptyBLOB"
$formatFileName = "C:\CLR Files\clr${timestamp}.fmt"
$fmtcmd = "bcp `"$emptyBLOB_tableName`" format nul -T -N -f
`"$formatFileName`" -S $instanceName"

#create the format file
Invoke-Expression -Command $fmtcmd

#now there is a problem, by default the format file
#will use 8 as prefix length for varbinary
#we need this to be zero, so we will replace
(Get-Content $formatFileName) |
ForEach-Object { $_ -replace "8", "0" } |
Set-Content $formatFileName
```

4. Add the following script and run:

```
$databaseName = "SampleDB"
$folderName = "C:\CLR Files\"
$newFolderName = "Retrieved CLR $timestamp"

$newFolder = Join-Path -Path "$($folderName)" -ChildPath
$newfoldername

#If the path exists, will error silently and continue
New-Item -ItemType directory -Path $newfolder -ErrorAction
SilentlyContinue

#get all user defined assemblies
$query = @"
SELECT
    af.file_id AS ID,
    a.name + '.dll' AS FileName
FROM
    sys.assembly_files af
    INNER JOIN sys.assemblies a
    ON af.assembly_id = a.assembly_id
WHERE
    a.is_user_defined = 1
"@
```

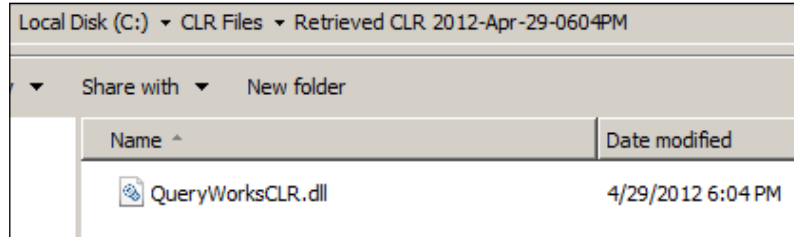
```

Invoke-Sqlcmd -ServerInstance $instanceName -Database
$databaseName -Query $query |
ForEach-Object {
    $item = $_
    Write-Verbose "Retrieving $($item.FileName) ..."

    $newFileName = Join-Path $newFolder $item.FileName
    $blobQuery = @"
SELECT
    af.content
FROM
    sys.assembly_files af
WHERE
    af.file_id = $($item.ID)
"@
    $cmd = "bcp `"$blobQuery`" queryout `"$newFileName`" -S
$instanceName -T -d $databaseName -f `"$formatFileName`""
    Invoke-Expression $cmd
}
explorer $newFolder
$VerbosePreference = "SilentlyContinue"

```

Once done, you can check out the file you generated. Your extracted file(s) will look similar to this:



How it works...

When we deploy SQLCLR assemblies, the definition of each assembly is saved to the target database. There may be times you want to extract these back to their DLL (Dynamic Link Library) binary forms. Retrieving and saving the DLL back into the file system is similar to retrieving and saving BLOB data back into the filesystem.

The first thing we do is to create a format file.



See recipe *Extracting binary data from SQL Server* for details on creating the format file for BLOB retrieval.

Once we have the format file, we create a timestamped folder where we will store our retrieved DLLs. This will help us keep track of what we extracted, and when:

```
$folderName = "C:\CLR Files\"
$newFolderName = "Retrieved CLR $timestamp"

$newFolder = Join-Path -Path "$($foldername)" -ChildPath
$newfoldername

#If the path exists, will error silently and continue
New-Item -ItemType directory -Path $newfolder -ErrorAction
SilentlyContinue
```

We construct a T-SQL statement to retrieve all user-defined assemblies in our target database. We can get the definition of the assemblies from `sys.assembly_files`. To get only user-defined assemblies, we must filter `sys.assembly` for `is_user_defined = 1`. If we do not filter, we may potentially get other files that were deployed with this assembly, such as debug files, especially when the assembly is deployed from SQL Server Data Tools. Alternatively, if you want to export only a selection, you can include a filter in your `SELECT` statement.

```
#get all user defined assemblies
$query = @"
SELECT
    af.file_id AS ID,
    a.name + '.dll' AS FileName
FROM
    sys.assembly_files af
    INNER JOIN sys.assemblies a
    ON af.assembly_id = a.assembly_id
WHERE
    a.is_user_defined = 1
"@
```

We then pass this T-SQL statement to the `Invoke-Sqlcmd` cmdlet:

```
Invoke-Sqlcmd -ServerInstance $instanceName -Database $databaseName
-Query $query |
```

For each record returned to us, we then compose another query that will retrieve the binary contents of the current DLL file from `sys.assembly_files` by passing its `file_id`, and save this back to the filesystem using `bcp` and the format file that we created at the beginning of the recipe.

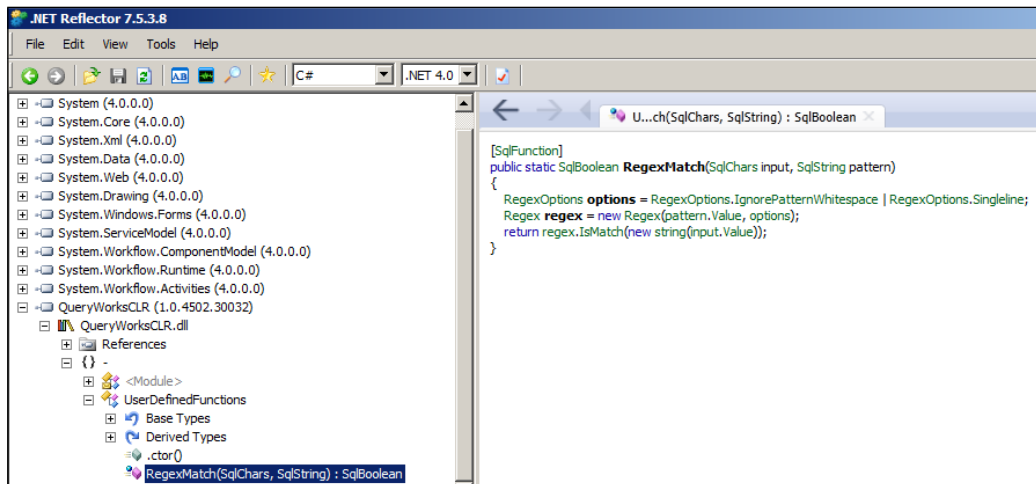
```
ForEach-Object {
    $item = $_
    Write-Verbose "Retrieving $($item.FileName) ..."
```

```

$newFileName = Join-Path $newFolder $item.FileName
$blobQuery = @"
SELECT
    af.content
FROM
    sys.assembly_files af
WHERE
    af.file_id = $($item.ID)
"@
$cmd = "bcp `"$blobQuery`" queryout `"$newFileName`" -S $instanceName
-T -d $databaseName -f `"$formatFileName`""
Invoke-Expression $cmd
}

```

To ensure we have maintained the integrity of the DLL file, we can use Red Gate's .NET Reflector tool to peek into what is in the DLL file. If all is well, you should be able to see all the classes and the definition of the methods when you open up this file in Reflector. Otherwise, Reflector will not be able to load this file.



See also

- ▶ The *Listing user-defined assemblies* recipe
- ▶ The *Creating a new assembly* recipe
- ▶ To learn more about .NET Reflector, visit Red-Gate's site for this product:
<http://www.reflector.net/>

8

Business Intelligence

In this chapter, we will cover:

- ▶ Listing items in your SSRS Report Server
- ▶ Listing SSRS report properties
- ▶ Using `ReportViewer` to view your SSRS report
- ▶ Downloading an SSRS report in Excel and PDF
- ▶ Creating an SSRS folder
- ▶ Creating an SSRS data source
- ▶ Changing an SSRS report's data source reference
- ▶ Uploading an SSRS report to Report Manager
- ▶ Downloading all SSRS report RDL files
- ▶ Adding a user with a role to an SSRS report
- ▶ Creating folders in an SSIS package store and MSDB
- ▶ Deploying an SSIS package to a package store
- ▶ Executing an SSIS package stored in the package store or File System
- ▶ Downloading an SSIS package into a file
- ▶ Creating an `SSISDB` catalog
- ▶ Creating an `SSISDB` folder
- ▶ Deploying an ISPAC file to `SSISDB`
- ▶ Executing an SSIS package stored in `SSISDB`
- ▶ Listing SSAS cmdlets
- ▶ Backing up an SSAS database
- ▶ Restoring an SSAS database
- ▶ Processing an SSAS cube

Introduction

Over the years and the various versions, SQL Server has increased its **Business Intelligence (BI)** support and capabilities. Its BI stack—Reporting Services, Integration Services, and Analysis Services—have become strong players in today's BI market.

PowerShell offers capabilities to automate and manage any BI-related tasks—from rendering **SQL Server Reporting Services (SSRS)** reports, to deploying the new **SQL Server Integration Services (SSIS)** 2012 ISPAC files, to backing up and restoring **SQL Server Analysis Services (SSAS)** cubes.

Listing items in your SSRS Report Server

In this recipe, we will list items in an SSRS Report Server that is configured in native mode.

Getting ready

Identify your SSRS 2012 Report Server URL. We will need to reference the ReportService2010 web service, and you can reference it using `<ReportServer URL>/ReportService2010.asmx`.

For this recipe, we will use the default Windows credential to authenticate to the server.

How to do it...

Let's explore the code required to list items in your SSRS Report Server that is configured in native mode.

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

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri -UseDefaultCredential

#list all children
$proxy.ListChildren("/", $true) |
Select Name, TypeName, Path, CreationDate |
Format-Table -AutoSize
```

```
#if you want to list only reports
#note this is using PowerShell V3 Where-Object syntax
$proxy.ListChildren("/", $true) |
Where TypeName -eq "Report" |
Select Name, TypeName, Path, CreationDate |
Format-Table -AutoSize
```

Here is a sample result:

Name	TypeName	Path	CreationDate
Customers	Folder	/Customers	5/11/2012 10:35:45 PM
Customer Contact Numbers	Report	/Customers/Customer Contact Numbers	5/13/2012 12:13:48 AM
Data Sources	Folder	/Data Sources	5/11/2012 9:59:23 PM
KERRIGAN	DataSource	/Data Sources/KERRIGAN	5/11/2012 9:59:23 PM
Sample	DataSource	/Data Sources/Sample	5/13/2012 12:47:03 AM
SQLSat 114 2012-May-11-0947PM	Folder	/SQLSat 114 2012-May-11-0947PM	5/11/2012 9:47:12 PM

How it works...

The SSRS `ReportService2010` web service provides an API that allows objects in the Report Server to be managed programmatically, whether the Report Server is configured for native mode or SharePoint integrated mode.

This recipe assumes a SQL Server Reporting Services Native Mode install, although listing reports in SSRS SharePoint Integrated mode should employ a similar approach.

The first step is to get a handle to create a web service proxy. A web service proxy in PowerShell allows you to manage the web service as you would for any other PowerShell object. To create a new web service proxy, you need to use the `New-WebServiceProxy` cmdlet and pass to it the web service URL as follows:

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.
asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri
-UseDefaultCredential
```

To display all the items in the Report Server, we just need to call the `ListChildren` method of the `ReportingService2010` web proxy object. This will list all items it can find at the path we specified, in this case the root `"/`.

```
#list all children
$proxy.ListChildren("/", $true) |
Select Name, TypeName, Path, CreationDate |
Format-Table -AutoSize
```

If you want to list just the reports, we can pipe the results of the `ListChildren` method and filter for `TypeName = "Report"`. Note that in the old version of the web service, `ReportService2005`, this property was called `Type` instead of `TypeName`.

```
#if you want to list only reports
#note this is using PowerShell V3 Where-Object syntax
$proxy.ListChildren("/", $true) |
Where TypeName -eq "Report" |
Select Name, TypeName, Path, CreationDate |
Format-Table -AutoSize
```

See also

- ▶ The *Listing SSRS report properties* recipe
- ▶ Learn more about the SSRS Report Server Web Service Endpoints from MSDN:
<http://msdn.microsoft.com/en-us/library/ms155398.aspx>
- ▶ Check out the MSDN articles for New-WebServiceProxy:
<http://msdn.microsoft.com/en-us/library/dd315258.aspx>
- ▶ Check out the MSDN articles for Report Server Namespace Management Methods:
<http://msdn.microsoft.com/en-us/library/ms152872>

Listing SSRS report properties

In this recipe, we will list a single SSRS report's properties.

Getting ready

Identify your SSRS 2012 report server URL. We will need to reference the `ReportService2010` web service, and you can reference it using:

```
<ReportServer URL>/ReportService2010.asmx
```

Specify your Report Manager URI in the variable `$ReportServerUri`.

Pick a report deployed in your SSRS 2012 Report Manager. Note the path to the item, and replace the variable `$reportPath` with your own path.

How to do it...

Here are the steps required to list SSRS report properties.

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

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri
-UseDefaultCredential

$reportPath = "/Customers/Customer Contact Numbers"

#using PowerShell V3 Where-Object syntax
$proxy.ListChildren("/", $true) |
Where-Object Path -eq $reportPath
```

A sample result follows:

```
ID                : 15b3dd87-d0de-43a0-8692-030dcfdab945
Name              : Customer Contact Numbers
Path              : /Customers/Customer Contact Numbers
VirtualPath       :
TypeName          : Report
Size              : 23870
SizeSpecified    : True
Description       :
Hidden            : False
HiddenSpecified  : False
CreationDate      : 5/13/2012 12:13:48 AM
CreationDateSpecified : True
ModifiedDate      : 5/13/2012 12:13:48 AM
ModifiedDateSpecified : True
CreatedBy         : KERRIGAN\Administrator
ModifiedBy        : KERRIGAN\Administrator
ItemMetadata      : {}
```

How it works...

To get SSRS 2012 Report Properties, we must first get a web service proxy.

```
$ReportServerUri = http://localhost/ReportServer/ReportService2010.asmx
$proxy = New-WebServiceProxy -Uri $ReportServerUri
-UseDefaultCredential
```

We must also identify which report we want to display properties for:

```
$reportPath = "/Customers/Customer Contact Numbers"
```

Once we get a proxy and once we know which report we are querying, we need to get the catalog items that are related to this Report Server instance. We do this by using the `ListChildren` method of the proxy object. This method accepts a starting path to traverse, and we will pass `"/` to indicate we want to get all items from the root path of the Report Server. We specify recursive lookup by passing the Boolean value `$true` as a second parameter in `ListChildren`.

```
#using PowerShell V3 Where-Object syntax
$proxy.ListChildren("/", $true) |
Where-Object Path -eq $reportPath
```

To narrow down the displayed properties to just our report's, we can pipe the result of the `ListChildren` method to the `Where-Object` cmdlet and filter only by reports that match `$reportPath`. Note that we are using the PowerShell V3 `Where-Object` syntax here:

```
#using PowerShell V3 Where-Object syntax
$proxy.ListChildren("/", $true) |
Where-Object Path -eq $reportPath
```

To do this in PowerShell V2:

```
#using PowerShell V2 Where-Object syntax
$proxy.ListChildren("/", $true) |
Where-Object {$_.Path -eq $reportPath}
```

Note that a report in ReportServer2010 web service is a `CatalogItem` class, not a `Report` class, which was available in previous SSRS versions. If you pipe the previous code to the `Get-Member` cmdlet, you will see `TypeName` at the beginning of the displayed results:

```
TypeName: Microsoft.PowerShell.Commands.NewWebserviceProxy.
AutogeneratedTypes.WebServiceProxyltServer_Report
Service2010_asmx.CatalogItem
```

See also

- ▶ *The Listing items in your SSRS Report Server recipe*
- ▶ To learn more about the `CatalogItem` class, check out:
<http://msdn.microsoft.com/en-us/library/reportservice2010.catalogitem>

Using ReportViewer to view your SSRS report

This recipe shows how to display a report using the `ReportViewer` redistributable.

Getting ready

First, you need to download `ReportViewer` redistributable and install it on your machine. At the time of writing of this book, the download link is at:

<http://www.microsoft.com/en-us/download/details.aspx?id=6442>

Identify your SSRS 2012 Report Server URL. We will need to reference the `ReportService2010` web service, and you can reference it using:

```
<ReportServer URL>/ReportService2010.asmx
```

Pick a report you want to display using the `ReportViewer` control. Identify the full path, and replace the value of the variable `$reportViewer.ServerReport.ReportPath` in the script.

How to do it...

This list shows how we can display a report using `ReportViewer`.

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Load the assembly for `ReportViewer` as follows:

```
#load the ReportViewer WinForms assembly
Add-Type -AssemblyName "Microsoft.ReportViewer.WinForms,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080
cc91"
```

```
#load the Windows.Forms assembly
Add-Type -AssemblyName "System.Windows.Forms"
```

3. Add the following script and run:

```
$reportViewer = New-Object Microsoft.Reporting.WinForms.
ReportViewer
$reportViewer.ProcessingMode = "Remote"
```

```
$reportViewer.ServerReport.ReportServerUrl = "http://localhost/
ReportServer"
$reportViewer.ServerReport.ReportPath = "/Customers/Customer
Contact Numbers"

#if you need to provide basic credentials, use the following
#$reportViewer.ServerReport.ReportServerCredentials.
NetworkCredentials= New-Object System.Net.
NetworkCredential("sqladmin", "P@ssword");

$reportViewer.Height = 600
$reportViewer.Width = 800
$reportViewer.RefreshReport()

#create a new Windows form
$form = New-Object Windows.Forms.Form

#we're going to make the form just slightly bigger
#than the ReportViewer
$form.Height = 610
$form.Width= 810

#form is not resizable
$form.FormBorderStyle = "FixedSingle"

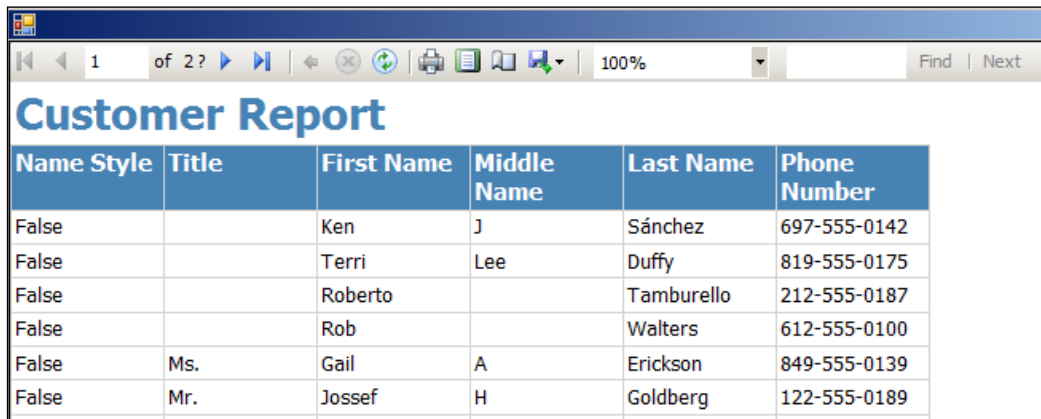
#do not allow user to maximize
$form.MaximizeBox = $false

$form.Controls.Add($reportViewer)

#show the report in the form
$reportViewer.Show()

#show the form
$form.ShowDialog()
```

After you run the script, here is a sample result. Notice how the top bar is similar to the top bar in your Report Manager:



Name Style	Title	First Name	Middle Name	Last Name	Phone Number
False		Ken	J	Sánchez	697-555-0142
False		Terri	Lee	Duffy	819-555-0175
False		Roberto		Tamburello	212-555-0187
False		Rob		Walters	612-555-0100
False	Ms.	Gail	A	Erickson	849-555-0139
False	Mr.	Jossef	H	Goldberg	122-555-0189

How it works...

The `ReportViewer` is a control that allows you to embed and display an SSRS report into a web or Windows form, and supply the user with the familiar interface they might be accustomed to seeing when using the Report Manager. This control always connects back to the Report Server when processing and rendering the report.

The `ReportViewer` is a redistributable package that does not come with Reporting Services installations; you will need to download and install this separately. See the *Getting Ready* section.

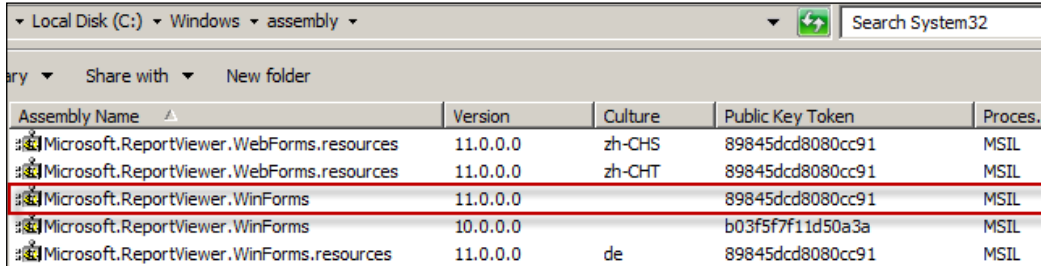
In this recipe, we are displaying a specific report in a Windows form.

To start, we have to load the assemblies related to `ReportViewer` and Windows forms:

```
#load the ReportViewer WinForms assembly
Add-Type -AssemblyName "Microsoft.ReportViewer.WinForms,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91"

#load the Windows.Forms assembly
Add-Type -AssemblyName "System.Windows.Forms"
```


We need to load the strong name of the `ReportViewer.WinForms` assembly using the `Add-Type` cmdlet, that is, to load it with the assembly name, version, culture, and public key token information. To determine the strong name, you can open up `C:\Windows\assembly` and check the properties of the `Microsoft.ReportViewer.WinForms` assembly. Note that you may get multiple versions of the assembly if you have different versions of `ReportViewer` redistributable installed in your system.



Assembly Name	Version	Culture	Public Key Token	Proces...
Microsoft.ReportViewer.WebForms.resources	11.0.0.0	zh-CHS	89845dcd8080cc91	MSIL
Microsoft.ReportViewer.WebForms.resources	11.0.0.0	zh-CHT	89845dcd8080cc91	MSIL
Microsoft.ReportViewer.WinForms	11.0.0.0		89845dcd8080cc91	MSIL
Microsoft.ReportViewer.WinForms	10.0.0.0		b03f5f7f11d50a3a	MSIL
Microsoft.ReportViewer.WinForms.resources	11.0.0.0	de	89845dcd8080cc91	MSIL

If you use the partial name to load the assembly, you can get an error similar to this:

```
Add-Type : Could not load file or assembly 'Microsoft.ReportViewer.  
WinForms, Version=8.0.0.0, Culture=neutral,  
PublicKeyToken=b03f5f7f11d50a3a' or one of its dependencies. The system  
cannot find the file specified.
```

A Microsoft Connect item (<https://connect.microsoft.com/PowerShell/feedback/details/417844/ctp3-v2b-add-type-a-microsoft-sqlserver-smo-wont-load-smo-assemblies>) filed regarding this issue still appears to be true at the time of writing of this book. The answer to the Connect item explains that `Add-Type <partial name>` looks at a hardcoded list of assembly versions, which seems to be `Version=8.0.0.0` for `Microsoft.ReportViewer.WinForms`.

Once the assemblies are loaded, we then have to create a `ReportViewer` object:

```
$reportViewer = New-Object Microsoft.Reporting.WinForms.ReportViewer
```

We also need to set some properties that specify where and how the report is going to be fetched:

```
$reportViewer.ProcessingMode = "Remote"  
$reportViewer.ServerReport.ReportServerUrl = "http://localhost/  
ReportServer"  
$reportViewer.ServerReport.ReportPath = "/Customers/Customer Contact  
Numbers"
```

`ProcessingMode` can either be `Local` or `Remote`. `ReportServerUrl` and `ReportPath` are properties of the `ServerReport` object, and these should point to your Report Server and the full path to your report. Should you need to specify the credentials to connect to the Report Manager, you will need to set the `ReportCredentials` property, like this:

```
$reportViewer.ServerReport.ReportServerCredentials.NetworkCredentials  
= New-Object System.Net.NetworkCredential("sqladmin", "P@ssword");
```

We then also specify the `ReportViewer` dimensions:

```
$reportViewer.Height = 600  
$reportViewer.Width = 800  
$reportViewer.RefreshReport()
```

For this recipe, we embedded the `ReportViewer` object in a Windows form, and lastly, showed it as a dialog form. Since we have pre-set the size of the report to 800 x 600, we are going to disable the maximize button, and the resizability of the window to prevent the users from resizing the form and seeing only empty spaces when the form is resized.

```
#create a new Windows form  
$form = New-Object Windows.Forms.Form  
  
#we're going to make the form just slightly bigger  
#than the ReportViewer  
$form.Height = 610  
$form.Width= 810  
  
#form is not resizable  
$form.FormBorderStyle = "FixedSingle"  
  
#do not allow user to maximize  
$form.MaximizeBox = $false  
  
$form.Controls.Add($reportViewer)  
  
#show the report in the form  
$reportViewer.Show()  
  
#show the form  
$form.ShowDialog()
```

See also

- ▶ *The Downloading an SSRS report in Excel and PDF* recipe
- ▶ Learn more about the `ReportViewer` class:
<http://msdn.microsoft.com/en-us/library/microsoft.reporting.winforms.reportviewer.aspx>
- ▶ `ReportViewer` properties:
http://msdn.microsoft.com/en-us/library/microsoft.reporting.webforms.reportviewer_properties
- ▶ `ReportViewer` Web Server and Windows form controls:
<http://msdn.microsoft.com/en-us/library/ms251771.aspx>

Downloading an SSRS report in Excel and PDF

This recipe shows how to download an SSRS report in Excel and PDF format.

Getting ready

To perform this recipe, you must first download and install the `ReportViewer` control. The `ReportViewer` control allows SSRS reports to be displayed and viewed to a web or Windows form.



See the *Using ReportViewer to view your SSRS report* recipe on how and where to download the `ReportViewer` control.

After installing the `ReportViewer` control, select a report that you wish to download into an Excel or PDF version.

In this recipe, we will download a report `/Customers/Customer Contact Numbers` into Excel and PDF. Alternatively, choose a report you wish to download and replace the `$reportViewer.ServerReport.ReportPath` variable.

How to do it...

Let's explore the code required to view your report in Excel and PDF.

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Load the ReportViewer assembly:

```
Add-Type -AssemblyName "Microsoft.ReportViewer.WinForms,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080
cc91"
```

3. Add the following script and run:

```
$reportViewer = New-Object Microsoft.Reporting.WinForms.
ReportViewer
$reportViewer.ProcessingMode = "Remote"
$reportViewer.ServerReport.ReportServerUrl = "http://localhost/
ReportServer"
$reportViewer.ServerReport.ReportPath = "/Customers/Customer
Contact Numbers"

#required variables for rendering
$mimeType = $null
$encoding = $null
$extension = $null
$streamids = $null
$warnings = $null

#export to Excel
$excelFile = "C:\Temp\Customer Contact Numbers.xls"
$bytes = $reportViewer.ServerReport.Render("Excel", $null,
    [ref] $mimeType,
    [ref] $encoding,
    [ref] $extension,
    [ref] $streamids,
    [ref] $warnings)
$fileStream = New-Object System.IO.FileStream($excelFile, [System.
IO.FileMode]::OpenOrCreate)
$fileStream.Write($bytes, 0, $bytes.Length)
$fileStream.Close()
```

```
#let's open up our Excel document
$excel = New-Object -comObject Excel.Application
$excel.visible = $true
$excel.Workbooks.Open($excelFile) | Out-Null

#export to PDF
$pdfFile = "C:\Temp\Customer Contact Numbers.pdf"
$bytes = $reportViewer.ServerReport.Render("PDF", $null,
    [ref] $mimeType,
    [ref] $encoding,
    [ref] $extension,
    [ref] $streamids,
    [ref] $warnings)

$fileStream = New-Object System.IO.FileStream($pdfFile, [System.
IO.FileMode]::OpenOrCreate)
$fileStream.Write($bytes, 0, $bytes.Length)
$fileStream.Close()

#let's open up our PDF application
[System.Diagnostics.Process]::Start($pdfFile)
```

How it works...

For this recipe, we will need to load a few assemblies. We need to load the ReportViewer assembly, which will render the SSRS report from Report Manager into different formats:

```
Add-Type -AssemblyName "Microsoft.ReportViewer.WinForms,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91"
```

We will also need to set the properties of the report:

```
$reportViewer = New-Object Microsoft.Reporting.WinForms.ReportViewer
$reportViewer.ProcessingMode = "Remote"
$reportViewer.ServerReport.ReportServerUrl = "http://localhost/
ReportServer"
$reportViewer.ServerReport.ReportPath = "/Customers/Customer Contact
Numbers"
```

There are also few variables we need to declare, to render our report. We need to declare these because they need to be passed by reference to the Render method of ReportViewer.

```
#required variables for rendering
$mimeType = $null
$encoding = $null
```

```

$extension = $null
$streamids = $null
$warnings = $null

```

We want to render the report first as an Excel file. The `ReportViewer` handle has a `Render` method that allows the report to be rendered in different formats, including Excel, PDF, and image. To render a report to Excel, we must invoke the `ServerReport.Render` method. The first parameter that we pass is for format, and it should be Excel. We are also going to pass five output parameters for MIME type, encoding, extension, stream IDs, and warnings respectively. We need to assign the result of this method's invocation into a byte variable.

```

#export to Excel
$excelFile = "C:\Temp\Customer Contact Numbers.xls"
$bytes = $reportViewer.ServerReport.Render("Excel", $null,
    [ref] $mimeType,
    [ref] $encoding,
    [ref] $extension,
    [ref] $streamids,
    [ref] $warnings)

```

To create an Excel file based on what was rendered, we should use a `System.IO.FileStream` object:

```

$fileStream = New-Object System.IO.FileStream($excelFile, [System.
IO.FileMode]::OpenOrCreate)
$fileStream.Write($bytes, 0, $bytes.Length)
$fileStream.Close()

```

When done, we create an `Excel.Application` COM object. We pass the filename, and open the workbook using the Excel object's `Workbooks.Open` method.

```

#let's open up our excel document
$excel = New-Object -comObject Excel.Application
$excel.visible = $true
$excel.Workbooks.Open($excelFile) | Out-Null

```

To render the report in PDF format, the same `ServerReport.Render` method can be invoked, but this time passing PDF instead of Excel as the first parameter:

```

$pdfFile = "C:\Temp\Customer Contact Numbers.pdf"
$bytes = $reportViewer.ServerReport.Render("PDF", $null,
    [ref] $mimeType,
    [ref] $encoding,
    [ref] $extension,
    [ref] $streamids,
    [ref] $warnings);

```

Saving the rendered PDF document also requires using the `System.IO.FileStream` object.

```
$fileStream = New-Object System.IO.FileStream($pdfFile, [System.IO.FileMode]::OpenOrCreate)
$fileStream.Write($bytes, 0, $bytes.Length)
$fileStream.Close()
```

The `[System.Diagnostics.Process]::Start` method is then used to open the PDF using the default application installed to run PDFs:

```
#let's open up our PDF application
[System.Diagnostics.Process]::Start($pdfFile)
```

See also

- ▶ *The Using ReportViewer to view your SSRS report recipe*
- ▶ ReportViewer Web Server and Windows form controls:
<http://msdn.microsoft.com/en-us/library/ms251771.aspx>
- ▶ `ServerReport.Render` method:
[http://msdn.microsoft.com/en-us/library/microsoft.reporting.webforms.serverreport.render\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.reporting.webforms.serverreport.render(v=vs.100).aspx)

Creating an SSRS folder

In this recipe, we create a timestamped SSRS folder.

Getting ready

Identify your SSRS 2012 Report Server URL. We will need to reference the `ReportService2010` web service, and you can reference it using:

```
<ReportServer URL>/ReportService2010.asmx
```

How to do it...

Let's explore the code required to create an SSRS folder programmatically.

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Add the following script and run:

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri -UseDefaultCredential

#A workaround we have to do to ensure
#we don't get any namespace clashes is to
#capture the auto-generated namespace, and
#create our objects based on this namespace

#capture automatically generated namespace
#this is a workaround to avoid namespace clashes
#resulting in using -Class with New-WebServiceProxy
$type = $Proxy.GetType().Namespace

#formulate data type we need
$datatype = ($type + '.Property')

#display datatype, just for our reference
$datatype

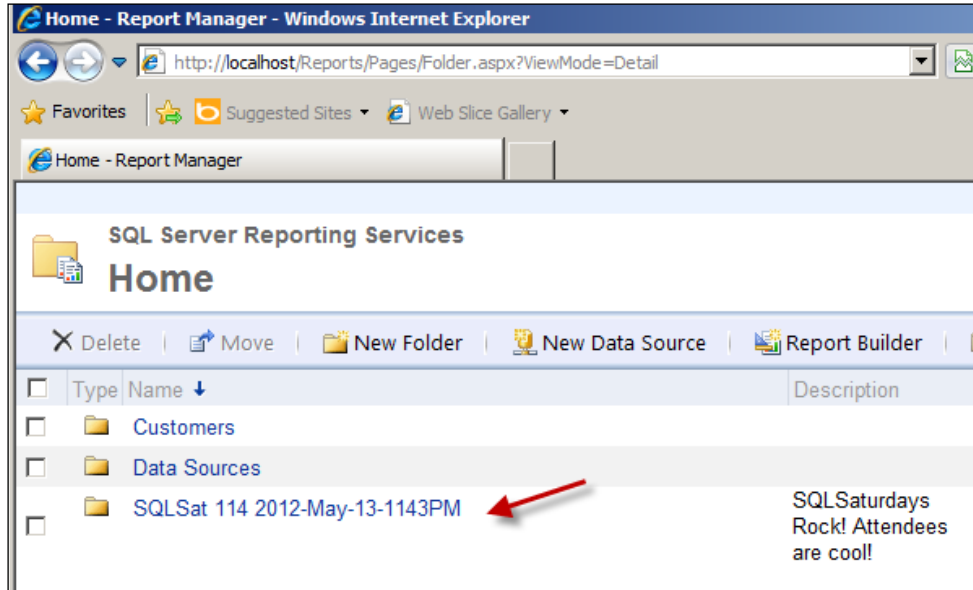
#create new Property
#if we were using -Class SSRS, this would be similar to
#$property = New-Object SSRS.Property
$property = New-Object ($datatype)
$property.Name = "Description"
$property.Value = "SQLSaturdays Rock! Attendees are cool!"
$folderName = "SQLSat 114 " + (Get-Date -format "yyyy-MMM-dd-hmmmtt")

#Report SSRS Properties
#http://msdn.microsoft.com/en-us/library/ms152826.aspx
$numProperties = 1
$properties = New-Object ($datatype + '[]') $numProperties
$properties[0] = $property

$proxy.CreateFolder($folderName, "/", $properties)

#display new folder in IE
Set-Alias ie "$env:programfiles\Internet Explorer\iexplore.exe"
ie "http://localhost/Reports"
```


Once done, go to your **Report Manager** and verify that the folder has been created:



How it works...

To create a folder, or any item, in your Report Server, we have to first create a handle to the Report Server web service by creating a proxy:

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.
asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri
-UseDefaultCredential
```


Typically, when you check sample code, you will find that the `-Class` switch is specified with the `New-WebServiceProxy` class, like this:

```
$proxy = New-WebServiceProxy -Uri $ReportServerUri
-UseDefaultCredential -Class SSRS2008
```

We don't use `-Class` in this recipe because of a couple of issues:

- ▶ When run from the PowerShell console, the script runs once and does not run subsequent times in the same session. You will have to close the shell (Command Line Interface or CLI) to release the previously created proxy object that holds that namespace. As far as the web server proxy is concerned, this namespace has already been created and we will not re-create it again. Remember what we created is just a proxy. The actual object was allocated not in our session but on the server.

- ▶ When run from the PowerShell ISE, you will get different host of issues, including errors that say the namespace cannot be recognized.

 See <http://www.sqlmusings.com/2012/02/04/resolving-ssrs-and-powershell-new-webserviceproxy-namespace-issue/> for more details on using the `-Class` switch for the `New-WebServiceProxy` cmdlet.

On the other hand, if we do not use namespace, a different issue arises. The automatically generated namespace is unpredictable. For example, a sample namespace is:

```
PS C:\Users\Administrator> $Proxy.GetType().Namespace
Microsoft.PowerShell.Commands.NewWebServiceProxy.AutogeneratedTypes.
WebServiceProxyltServer_ReportService2010
_asmx
```

This poses a problem because we need to refer to this namespace when we create any `ReportService2010` object. To work around this issue, we can omit the `-Class` and dynamically capture the namespace, and subsequently use it when creating our SSRS objects.

In the following script, we are creating a `Property` object that we are going to use for our folder:

```
#capture automatically generated namespace
#this is a workaround to avoid namespace clashes
#resulting in using -Class with New-WebServiceProxy
$type = $Proxy.GetType().Namespace

#formulate data type we need
$datatype = ($type + '.Property')

#display datatype, just for our reference
$datatype

#create new Property
#if we were using -Class SSRS, this would be similar to
#$property = New-Object SSRS.Property
$property = New-Object ($datatype)
```

Once we have created the `Property` object, we can assign the values. One property we can set for a folder is `Description`:

```
$property.Name = "Description"
$property.Value = "SQLSaturdays Rock! Attendees are cool!"
$folderName = "SQLSat 114 " + (Get-Date -format "yyyy-MMM-dd-hhmmtt")
```

We then need to add this to a `Property []` array, which is what the `CreateFolder` method of the proxy accepts. Note that when we create this array, we still need to create this dynamically, similar to how we created our `Property` object:

```
#Report SSRS Properties
#http://msdn.microsoft.com/en-us/library/ms152826.aspx
$numProperties = 1
$properties = New-Object ($datatype + '[]')$numProperties
$properties[0] = $property
```

When done, we can create the folder using the `CreateFolder` method, which accepts the folder name, the parent, and a properties array:

```
$proxy.CreateFolder($folderName, "/", $properties)
```

The last step we have in the recipe is creating an alias for IE, and launching our Report Manager to verify the folder has been created:

```
#display new folder in IE
Set-Alias ie "$env:programfiles\Internet Explorer\iexplore.exe"
ie "http://localhost/Reports"
```

See also

- ▶ The *Creating an SSRS folder* recipe
- ▶ To learn more about the namespace collision issue mentioned in this recipe when dealing with `New-WebServiceProxy`, check out:

<http://www.sqlmusings.com/2012/02/04/resolving-ssrs-and-powershell-new-webserviceproxy-namespace-issue/>

- ▶ Check out the `ReportService2010.CreateFolder` method:

<http://msdn.microsoft.com/en-us/library/reportservice2010.reporting-service2010.createfolder.aspx>

Creating an SSRS data source

In this recipe, we will create an SSRS data source.

Getting ready

In our recipe, we will create a data source called `Sample` that is stored in the `/Data Sources` folder. This data source uses Integrated authentication and points to the `AdventureWorks2008R2` database.

Before we start, we will need to identify the typical information needed for a data source, including:

Property	Value
Data source name	Sample
Data source type	SQL
Connection string	Data Source=KERRIGAN;Initial Catalog=AdventureWorks2008R2
Credentials	Integrated
Parent (that is, folder where this data source will be placed; must exist already)	/Data Sources

These are the same pieces of information you can find when you go to a data source's properties in your Report Manager:

Name:

Description:

Hide in tile view

Enable this data source

Data source type:

Connection string:

Connect using:

Credentials supplied by the user running the report

Display the following text to prompt user for a user name and password:

Use as Windows credentials when connecting to the data source

Credentials stored securely in the report server

User name:

Password:

Use as Windows credentials when connecting to the data source

Impersonate the authenticated user after a connection has been made to the data source

Windows integrated security

Credentials are not required

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:

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri -UseDefaultCredential
$type = $Proxy.GetType().Namespace

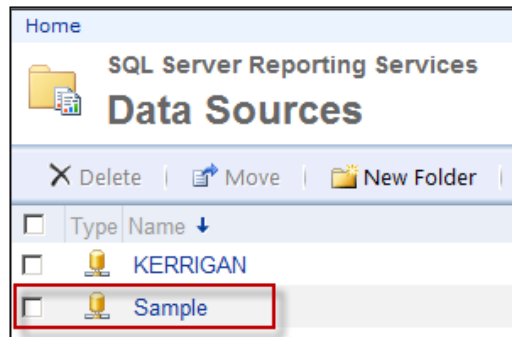
#create a DataSourceDefinition
$dataSourceDefinitionType = ($type + '.DataSourceDefinition')
$dataSourceDefinition = New-Object($dataSourceDefinitionType)
$dataSourceDefinition.CredentialRetrieval = "Integrated"
$dataSourceDefinition.ConnectionString = "Data Source=KERRIGAN;Initial Catalog=AdventureWorks2008R2"
$dataSourceDefinition.extension = "SQL"
$dataSourceDefinition.enabled = $true
$dataSourceDefinition.Prompt = $null
$dataSourceDefinition.WindowsCredentials = $false

#NOTE this is SSRS native mode
#CreateDataSource method accepts the following parameters:
#datasource name
#parent (data folder) - must already exist
#overwrite
#data source definition
#properties

$dataSource = "Sample"
$parent = "/Data Sources"
$overwrite = $true

$newDataSource = $proxy.CreateDataSource($dataSource, $parent, $overwrite,$dataSourceDefinition, $null)
```

When done, open up your Report Manager and confirm that the data source has been created:



How it works...

To create a data source programmatically, we first need to get a web service proxy:

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.
asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri
-UseDefaultCredential
```

We then need to capture the automatically generated namespace. We will use this in succeeding steps:

```
$type = $Proxy.GetType().Namespace
```

We then need to create a `DataSourceDefinition` class. We start by using our automatically generated namespace to help us create a new `DataSourceDefinition` object:

```
#create a DataSourceDefinition
$dataSourceDefinitionType = ($type + '.DataSourceDefinition')
$dataSourceDefinition = New-Object($dataSourceDefinitionType)
```



See the *How it works...* section of the *Creating an SSRS folder* recipe for additional details on automatically generated namespace issues.

We then need to specify the properties of `DataSourceDefinition`:

- ▶ `CredentialRetrieval` specifies how to retrieve credentials when Report Server needs to connect to the data source. It can be one of the following: `None`, `Prompt`, `Integrated`, or `Store`.

- ▶ `ConnectionString` is the connection string to the data source.
- ▶ `Extension` is the data source extension, and can be either: `SQL`, `OleDb`, `Odbc`, or a custom extension.

We are also setting the report so that it does not prompt for credentials when run by setting the `Prompt` property to `null`; and `WindowsCredentials` to be `false`, for the report not to pass credentials as Windows credentials:

```
$dataSourceDefinition.CredentialRetrieval = "Integrated"
dataSourceDefinition.ConnectionString = "Data Source=KERRIGAN;Initial
Catalog=AdventureWorks2008R2"
dataSourceDefinition.extension = "SQL"
dataSourceDefinition.enabled = $true
dataSourceDefinition.Prompt = $null
dataSourceDefinition.WindowsCredentials = $false
```

To create a data source in native mode, we need to use the `CreateDataSource` method, which accepts five parameters:

- ▶ Data source name
- ▶ Parent
- ▶ Overwrite
- ▶ Data source definition
- ▶ Properties

This is illustrated in the following code:

```
$dataSource = "Sample"
$parent = "/Data Sources"
$overwrite = $true

$newDataSource = $proxy.CreateDataSource($dataSource, $parent,
$overwrite,$dataSourceDefinition, $null)
```

See also

- ▶ The *Creating an SSRS folder* recipe
- ▶ The *Changing an SSRS report's data source reference* recipe
- ▶ Learn more about the `ReportService.DataSourceDefinition` class:
<http://msdn.microsoft.com/en-us/library/reportservice2010.datasourcedefinition.aspx>

Changing an SSRS report's data source reference

In this recipe, we will update an SSRS report's data source.

Getting ready

In our recipe we will change the data source of our report `/Customers/Customer Contact Numbers`, which originally uses the data source reference `/Data Sources/Sample` to point to `/Data Sources/KERRIGAN`.

Alternatively, pick an existing report in your environment and the data source you want this report to reference. Note the names and the path to these items.

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:

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri -UseDefaultCredential

#get autogenerated namespace
$type = $proxy.GetType().Namespace

#specify which report's data source to change
$reportPath = "/Customers/Customer Contact Numbers"

#look for the report in the catalog items array
#note we are using PowerShell V3 Where-Object syntax
$report = $proxy.ListChildren("/", $true) |
    Where-Object Path -eq $reportPath

#get current data source name
#this needs to be the same name in the RDL
$dataSourceName = $($proxy.GetItemDataSources($report.Path)).Name

#specify new data source reference
$newDataSourcePath = "/Data Sources/Sample"
```

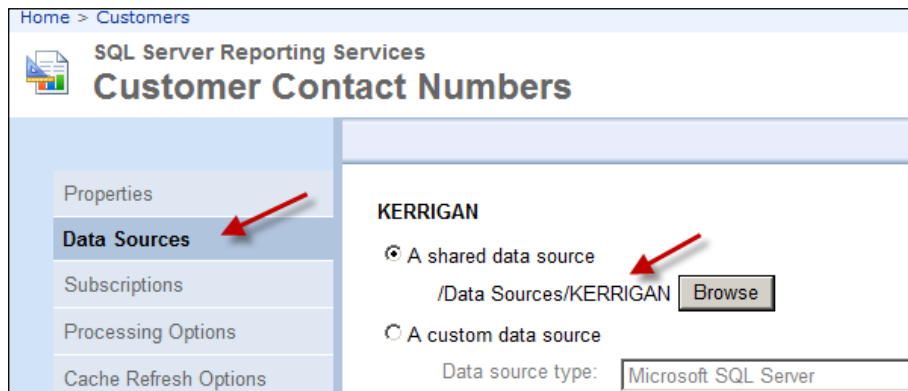


```
#dynamically create data types based on the new
#autogenerated namespace
$dataSourceType = ($type + '.DataSource')
$numItems = 1
$dataSourceArrayType = ($type + '.DataSource[]')
$dataSourceReferenceType = ($type + '.DataSourceReference')

#create a data source array containing
#the new data source path
$dataSourceArray = New-Object ($dataSourceArrayType)$numItems
$dataSourceArray[0] = New-Object ($dataSourceType)
$dataSourceArray[0].Name = $dataSourceName
$dataSourceArray[0].Item = New-Object ($dataSourceReferenceType)
$dataSourceArray[0].Item.Reference = $newDataSourcePath

#set the new data source
$proxy.SetItemDataSources($report.Path, $dataSourceArray)
```

You can confirm the changes by opening the **Report Manager**, and opening that report's **Data Sources**. Ensure that the data source reference now points to the correct path:



How it works...

In order to change a report's data source, we must first get a handle to this report.

The first step is to create a web server proxy:

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.
asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri
-UseDefaultCredential
```

In this recipe, we will also be creating a few `ReportService2010` objects, so we will need to capture the dynamically generated namespace:

```
$type = $proxy.GetType().Namespace
```



See the *How it works...* section of the *Creating an SSRS folder* recipe for additional details on automatically generated namespace issues.

We then need to get a handle to the report. In order to do this, we need to capture all the Report Server objects, and extract the report that matches the path we specified:

```
#look for the report in the catalog items array
#note we are using PowerShell V3 Where-Object syntax
$report = $proxy.ListChildren("/", $true) |
    Where-Object Path -eq $reportPath
```

We also need to capture the report's current data source name by using the `GetItemDataSources` method of the proxy object. We need to keep the same name.



Note that paths, report names, and data source names and references are case sensitive.

Later in the code, we will need to change the data source path it references:

```
#get current data source name
#this needs to be the same name in the RDL
$dataSourceName = $($proxy.GetItemDataSources($report.Path)).Name

#specify new data source reference
$newDataSourcePath = "/Data Sources/Sample"
```

The next step is to create a data source array (`DataSource []` object). Because we have a dynamically generated namespace, we must first compose the data types dynamically based on the namespace—here, stored in the variable `$type`:

```
#dynamically create data types based on the new
#autogenerated namespace
$dataSourceType = ($type + '.DataSource')
$numItems = 1
$dataSourceArrayType = ($type + '.DataSource[]')
$dataSourceReferenceType = ($type + '.DataSourceReference')
```

To create a data source array, we use the new types:

```
#create a data source array containing
#the new data source path
$dataSourceArray = New-Object ($datasourceArrayType)$numItems
$dataSourceArray[0] = New-Object ($dataSourceType)
$dataSourceArray[0].Name = $dataSourceName
$dataSourceArray[0].Item = New-Object ($dataSourceReferenceType)
$dataSourceArray[0].Item.Reference = $newDataSourcePath
```

We are now ready to call the `SetItemDataSources` method of the proxy object to change our report's data source reference. This method accepts a catalog item name path, and a data source array.

```
$proxy.SetItemDataSources($report.Path, $dataSourceArray);
```

See also

- ▶ The *Creating an SSRS folder* recipe
- ▶ The *Creating an SSRS data source* recipe
- ▶ Learn more about the `SetItemDataSources` method:

<http://msdn.microsoft.com/en-us/library/reportservice2010.reportingervice2010.setitemdatasources.aspx>

Uploading an SSRS report to Report Manager

In this recipe, we will upload an SSRS Report (.rdl file) to the Report Manager.

Getting ready

You can use the sample RDL file that comes with this cookbook and save it to the `C:\SSRS` folder. The sample RDL file uses the `AdventureWorks2008R2` sample database. Alternatively, use an RDL file that is readily available to you. Be sure to update the RDL file reference in the script to reflect where your report file is located.

How to do it...

Here is how we can upload an RDL file to the Report Manager:

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Add the following script and run:

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri -UseDefaultCredential
$type = $proxy.GetType().Namespace

#specify where the RDL file is
$rdl = "C:\SSRS\Customer Sales.rdl"

#extract report name from the RDL file
$reportName = [System.IO.Path]::GetFileNameWithoutExtension($rdl)

#get contents of the RDL
$byteArray = Get-Content $rdl -Encoding Byte

#The fully qualified URL for the parent folder that will contain
#the item.
$parent = "/Customers"
$overwrite = $true
$warnings = $null

#create report
$report = $proxy.CreateCatalogItem("Report", $reportName, $parent,
$overwrite, $byteArray, $null, [ref]$warnings )

#data source name must match what's in the RDL
$dataSourceName = "KERRIGAN"

#data source path should match what's in the report server
$dataSourcePath = "/Data Sources/KERRIGAN"

#when we upload the report, if the
#data source from the source is different
#or has a different path from what's in the
#report manager, the data source will be broken
#and we will need to update

#create our data type references
$dataSourceArrayType = ($type + '.DataSource[]')
$dataSourceType = ($type + '.DataSource')
$dataSourceReferenceType = ($type + '.DataSourceReference')
```

```
#create data source array
$numDataSources = 1
$dataSourceArray = New-Object ($dataSourceArrayType) $numDataSources
$dataSourceReference = New-Object ($dataSourceReferenceType)

#update data source
$dataSourceArray[0] = New-Object ($dataSourceType)
$dataSourceArray[0].Name = $dataSourceName
$dataSourceArray[0].Item = New-Object ($dataSourceReferenceType)
$dataSourceArray[0].Item.Reference = $dataSourcePath
$proxy.SetItemDataSources($report.Path, $dataSourceArray)
```

How it works...

First, we create a web service proxy object:

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.
asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri
-UseDefaultCredential
$type = $Proxy.GetType().Namespace
```

We will then need to specify the path to the RDL file. In this recipe, we will keep the file name the same as the RDL filename, without the extension:

```
#specify where the RDL file is
$rddl = "C:\SSRS\Customer Sales.rdl"

#extract report name from the RDL file
$reportName = [System.IO.Path]::GetFileNameWithoutExtension($rddl)
```

We need to extract the contents of the RDL file to create the report programmatically. To do so, we will use the `Get-Content` cmdlet, but using the switch `-Encoding` to ensure we preserve the encoding used in the report:

```
#get contents of the RDL
$byteArray = Get-Content $rddl -Encoding Byte
```

To create the report, we need to use the `CreateCatalogItem` method of the proxy object, which accepts the catalog item type, report name, parent, overwrite `Boolean` flag, the contents of the RDL file, and a warnings variable:

```
#The fully qualified URL for the parent folder that will contain
#the item.
$parent = "/Customers"
```

```

$overwrite = $true
$warnings = $null
#create report
$report = $proxy.CreateCatalogItem("Report", $reportName, $parent,
$overwrite, $byteArray, $null, [ref]$warnings )

```

The supported `CatalogItem` types in native mode are:

- ▶ Report
- ▶ DataSet
- ▶ Resource
- ▶ DataSource
- ▶ Model

At this point, the report is already uploaded to the server. However, if the data source path stored in the report is different from where the data source is located in the server, the report will still not be usable.

To change the data source, we must create a `DataSource` array, and change only the `DataSourceReference` value. We change the report's data source reference by using the `SetItemDataSources` method of the proxy object:

```

#data source name must match what's in the RDL
$dataSourceName = "KERRIGAN"

#data source path should match what's in the report server
$dataSourcePath = "/Data Sources/KERRIGAN"

#when we upload the report, if the
#data source from the source is different
#or has a different path from what's in the
#report manager, the data source will be broken
#and we will need to update

#create our data type references
$dataSourceArrayType = ($type + '.DataSource[]')
$dataSourceType = ($type + '.DataSource')
$dataSourceReferenceType = ($type + '.DataSourceReference')

#create data source array
$numDataSources = 1
$dataSourceArray = New-Object ($dataSourceArrayType)$numDataSources
$dataSourceReference = New-Object ($dataSourceReferenceType)

```

```
#update data source
$dataSourceArray[0] = New-Object ($dataSourceType)
$dataSourceArray[0].Name = $dataSourceName
$dataSourceArray[0].Item = New-Object ($dataSourceReferenceType)
$dataSourceArray[0].Item.Reference = $dataSourcePath
$proxy.SetItemDataSources($report.Path, $dataSourceArray)
```



See the *Changing an SSRS report's data source reference* recipe for more details on the steps.

See also

- ▶ The *Using ReportViewer to view your SSRS report* recipe
- ▶ The *Downloading an SSRS report in Excel and PDF* recipe
- ▶ Check out more information on the `CreateCatalogItem` method:
<http://msdn.microsoft.com/en-us/library/reportservice2010.reporting-service2010.createcatalogitem.aspx>

Downloading all SSRS report RDL files

This recipe shows how you can download all RDL files from your Report Server.

Getting ready

In this recipe, we will download all RDL files from the SSRS Report Server into `C:\SSRS\` in a subfolder structure that mimics the folder structure in the Report Server.

Identify your SSRS 2012 Report Server URL. We will need to reference the `ReportService2010` web service, and you can reference it using:

```
<ReportServer URL>/ReportService2010.asmx
```

How to do it...

Let's explore the code required to download the RDL files from your Report Server.

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Add the following script and run:

```
$VerbosePreference = "Continue"
$ReportServerUri = "http://localhost/ReportServer/
ReportService2010.asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri
-UseDefaultCredential

$destinationFolder = "C:\SSRS\"

#create a new folder where we will save the files
#we'll use a time-stamped folder, format similar
#to 2012-Mar-28-0850PM
$ts = Get-Date -format "yyyy-MMM-dd-hhmmtt"
$folderName = "RDL Files $($ts)"
$fullFolderName = Join-Path -Path "$($destinationFolder)"
-ChildPath $folderName

#If the path exists, will error silently and continue
New-Item -ItemType Directory -Path $fullFolderName -ErrorAction
SilentlyContinue

#get all reports
#second parameter means recursive
#CHANGE ALERT:
#in ReportingService2005 - Type
#in ReportingService2010 - TypeName
$proxy.ListChildren("/", $true) |
Select TypeName, Path, ID, Name |
Where-Object TypeName -eq "Report" |
ForEach-Object {
    $item = $_
    [string]$path = $item.Path
    $pathItems=$path.Split("/")

    #get path name;we will mirror structure
    #when we save the file
    $reportName = $pathItems[$pathItems.Count -1]
    $subfolderName = $path.Trim($reportName)

    $fullSubfolderName = Join-Path -Path "$($fullFolderName)"
-ChildPath $subfolderName
```



```
#If the path exists, will error silently and continue
New-Item -ItemType directory -Path $fullSubfolderName
-ErrorAction SilentlyContinue

#CHANGE ALERT:
#in ReportingService2005 - GetReportDefinition
#in ReportingService2010 - GetItemDefinition
#use $Proxy | gm to learn more
[byte[]] $reportDefinition = $proxy.GetItemDefinition($item.
Path)

#note here we're forcing the actual definition to be
#stored as a byte array
#if you take out the @() from the
#MemoryStream constructor,
#you'll get an error
[System.IO.MemoryStream] $memStream = New-Object System.
IO.MemoryStream(@($reportDefinition))

#save the XML file
$rdlFile = New-Object System.Xml.XmlDocument
$rdlFile.Load($memStream) | Out-Null

$fullReportFileName = "$($fullSubfolderName)$($item.Name).rdl"
Write-Verbose "Saving $($fullReportFileName)"
$rdlFile.Save($fullReportFileName)

}

Write-Verbose "Done downloading your RDL files to
 $($fullFolderName)"
$VerbosePreference = "SilentlyContinue"
```

How it works...

This recipe will re-create the entire folder structure of the Report Manager, and save the appropriate RDL files in their respective folders.

To do this, we first create a proxy to the ReportService2010 web service:

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.
asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri
-UseDefaultCredential
```

We will also need to specify where we want to store the downloaded RDL files:

```
$destinationFolder = "C:\SSRS\"
```

We also want to create a new timestamped folder where we will store the RDL files:

```
$ts = Get-Date -format "yyyy-MMM-dd-hhmm"
$folderName = "RDL Files $($ts)"
$fullFolderName = Join-Path -Path "$($destinationFolder)" -ChildPath
$folderName
```

We then get all report items. Note that we have to filter these to return only items with `TypeName = Report`:

```
$proxy.ListChildren("/", $true) |
Select TypeName, Path, ID, Name |
Where-Object TypeName -eq "Report" |
ForEach-Object {
```

We can pass all `Report` items to the `ForEach-Object` cmdlet so we can download each RDL file from Report Manager. For each report, we want to investigate the path. If the path contains a series of folders, we want to recreate these folders in our destination folder:

```
$item = $_
[string]$path = $item.Path
$pathItems=$path.Split("/")

#get path name; we will mirror structure
#when we save the file
$reportName = $pathItems[$pathItems.Count -1]
$subfolderName = $path.Trim($reportName)

$fullSubfolderName = Join-Path -Path "$($fullFolderName)"
-ChildPath $subfolderName

#If the path exists, will error silently and continue
New-Item -ItemType directory -Path $fullSubfolderName -ErrorAction
SilentlyContinue
```

Once we have created the folder structure, we can get the report definition using the `GetItemDefinition` method of the proxy object. This needs to be stored in a byte array, to ensure we store unaltered, raw bytes of the report:

```
#CHANGE ALERT:
#in ReportingService2005 - GetReportDefinition
#in ReportingService2010 - GetItemDefinition
#use $Proxy | gm to learn more
[byte[]] $reportDefinition = $proxy.GetItemDefinition($item.Path)
```

```
#note here we're forcing the actual definition to be
#stored as a byte array
#if you take out the @() from the
#MemoryStream constructor,
#you'll get an error
[System.IO.MemoryStream] $memStream = New-Object System.
IO.MemoryStream(@($reportDefinition))
```

We can then store the memory stream in an `XmlDocument` object, which can in turn save the file back to the filesystem, given a full file path and name:

```
#save the XML file
$rdlFile = New-Object System.Xml.XmlDocument
$rdlFile.Load($memStream) | Out-Null

$fullReportFileName = "$($fullSubfolderName)$($item.Name).rdl"
Write-Verbose "Saving $($fullReportFileName)"
$rdlFile.Save($fullReportFileName)
```

See also

- ▶ *The Using ReportViewer to view your SSRS report recipe*
- ▶ *The Downloading an SSRS Report in Excel and PDF recipe*
- ▶ Check out these MSDN articles related to:
 - `GetItemDefinition`:
<http://msdn.microsoft.com/en-us/library/reportservice2010.reporting-service2010.getitemdefinition.aspx>
 - `MemoryStream`:
<http://msdn.microsoft.com/en-us/library/system.io.memorystream.aspx>

Adding a user with a role to an SSRS report

In this recipe, we will add a user with a few roles to SSRS.

Getting ready

In this recipe, we will add `QUERYWORKS\aterra` as a browser and Content Manager to the Customer Contact Numbers report.

For your environment, instead of using `QUERYWORKS\aterra`, you can identify a user you want to add to an existing report, and which roles you want to assign to them.

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:

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri -UseDefaultCredential
$type = $Proxy.GetType().Namespace

$itemPath = "/Customers/Customer Contact Numbers"

#this will hold all the groups/users for a report
$newPolicies = @()
$inherit = $null

#list current report users
$proxy.GetPolicies($itemPath, [ref]$inherit)

#NOTE that when we change policies, it will
#automatically break inheritance
#ALSO NOTE that when you programmatically mess
#with policies, you will need to "re-add" users that were
#already there, if you want them to keep on having access
#to your reports

#this gets all users who currently have
#access to this report
#need to pass $inherit by reference
```

```
$proxy.GetPolicies($itemPath, [ref]$inherit) |
ForEach-Object {
    #re-add existing policies
    $newPolicies += $_
}

$policyDataType = ($type + '.Policy')
$newPolicy = New-Object ($policyDataType)
$newPolicy.GroupUserName = "QUERYWORKS\aterra"

#a policy must have roles
$roleDataType = ($type + '.Role')
$newRole = New-Object ($roleDataType)
$newRole.Name = "Browser"

#add the role to the policy
$newPolicy.roles += $newRole

#a policy must have roles
$roleDataType = ($type + '.Role')
$newRole = New-Object ($roleDataType)
$newRole.Name = "Content Manager"

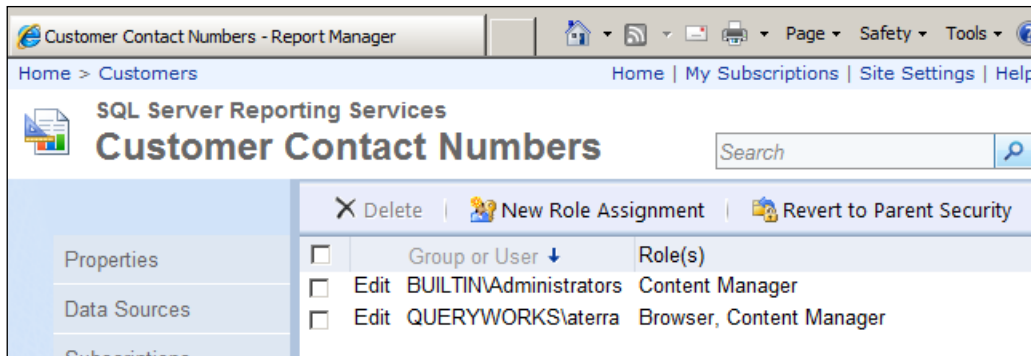
#add the role to the policy
$newPolicy.roles += $newrole

#check if this user already exists in your policy array
#if user does not exist yet with current role, add policy
if ($($newPolicies | ForEach-Object {$_.GroupUserName})
-notcontains $newPolicy.GroupUserName)
{
    $newPolicies += $newPolicy
}

#set the policies
$proxy.SetPolicies($itemPath,$newPolicies)

#list new report users
$proxy.GetPolicies($itemPath, [ref]$inherit)
```

When done, check the report that you just added a user to, from **Report Manager**. Go to its **Properties** and look at its security settings. Note that the user has been added, but inheritance is broken—as illustrated by the checkboxes, and the extra menu item called **Revert to Parent Security** has been added:



How it works...

When adding or changing users in an SSRS report programmatically, we will need to get a handle to the whole policy object, add or change the users or roles, and then re-apply the policy. Because this is manually changing a single item's security, inheritance is automatically broken for this item.

First, we need to create a proxy, and extract the dynamically created namespace.

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.
asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri
-UseDefaultCredential
$type = $proxy.GetType().Namespace
```



See the *How it works...* section of the *Create an SSRS folder* recipe for additional details on automatically generated namespace issues.

We also need to specify the path of the report we want to change:

```
$itemPath = "/Customers/Customer Contact Numbers"
```

To change policies, we will need to re-save existing policies. We will do this by retrieving the current users and roles using the `GetPolicies` method of the proxy object, and saving them to an array. The `$inherit` variable will hold whether that item inherits its security policy from its parent or not:

```
$newPolicies = @()
$inherit = $null

#this gets all users who currently have
#access to this report
```

```
#need to pass $inherit by reference
$proxy.GetPolicies($itemPath, [ref]$inherit) |
ForEach-Object {
    #re-add existing policies
    $newPolicies += $_
}
```

We then need to specify the account we are adding. This needs to be held in a ReportingService2010.Policy object, and can either be a user or group name:

```
$policyDataType = ($type + '.Policy')
$newPolicy = New-Object ($policyDataType)
$newPolicy.GroupUserName = "QUERYWORKS\aterra"
```

Next, we add the roles that will be associated with this group or user:

```
#a policy must have roles
$roleDataType = ($type + '.Role')
$newRole = New-Object ($roleDataType)
$newRole.Name = "Browser"
```

```
#add the role to the policy
$newPolicy.roles += $newRole
```

```
#a policy must have roles
$roleDataType = ($type + '.Role')
$newRole = New-Object ($roleDataType)
$newRole.Name = "Content Manager"
```

```
#add the role to the policy
$newPolicy.roles += $newrole
```

Once the new account and roles are in place, we need to add it to our policy array, which contains all existing policies for the item:

```
#check if this user already exists in your policy array
#if user does not exist yet with current role, add policy
if ($($newPolicies | ForEach-Object {$_.GroupUserName}) -notcontains
$newPolicy.GroupUserName)
{
    $newPolicies += $newPolicy
}
```

When everything is set, we can call the `SetPolicies` method of the proxy object:

```
#set the policies
$proxy.SetPolicies($itemPath,$newPolicies)
```

See also

- ▶ Check out these MSDN articles related to:
 - ❑ `GetPolicies`:
<http://msdn.microsoft.com/en-us/library/reportservice2010.reporting-service2010.getpolicies>
 - ❑ `SetPolicies`:
<http://msdn.microsoft.com/en-us/library/reportservice2010.reporting-service2010.setpolicies>
 - ❑ `InheritParentSecurity`:
<http://msdn.microsoft.com/en-us/library/reportservice2010.reporting-service2010.inheritparentsecurity>

Creating folders in an SSIS package store and MSDB

In this recipe, we will see how to create a folder in the SSIS instance and the package store.

Getting ready

For this recipe, we will create a timestamped folder prefixed with the word `QueryWorks`. Feel free to replace it with your folder name by changing the variable `$newfolder`.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the `ManagedDTS` assembly as follows:

```
#add ManagedDTS assembly
Add-Type -AssemblyName "Microsoft.SqlServer.ManagedDTS,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080
cc91"
```


3. Add the following script and run:

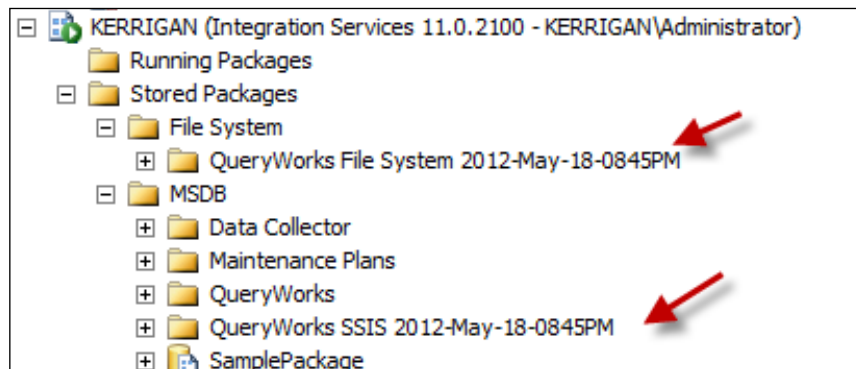
```
$server = "KERRIGAN"

#create new app
$app = New-Object ("Microsoft.SqlServer.Dts.Runtime.Application")
$ts = Get-Date -format "yyyy-MMM-dd-hhmm"
$newfolder = "QueryWorks File System $($ts)"

#folder in package store
#will appear under "Stored Packages > File System"
if (!$app.FolderExistsOnDtsServer("\File System\$($newfolder)",
$server))
{
    $app.CreateFolderOnDtsServer("\File System\", $newfolder,
$server)
}

#folder in SSIS instance
#will appear under "Stored Packages > MSDB"
$newfolder = "QueryWorks SSIS $($ts)"
if (!$app.FolderExistsOnSqlServer($newfolder, $server, $null,
$null))
{
    $app.CreateFolderOnSqlServer("", $newfolder, $server, $null,
$null)
}
```

When the script finishes, connect to the Integration Services instance. Expand both **File System** and **MSDB** nodes, and confirm that the folders have been created.



How it works...

The assembly `Microsoft.SqlServer.ManagedDTS` exposes SSIS 2005 and 2008 objects for programmatic access. Although this can be considered legacy SSIS when SQL Server 2012 came out, this method was still supported, and will still be used by developers.

To create folders in the package store and the SSIS instance, we must first load the `ManagedDTS` assembly. We need to do this explicitly because this assembly does not come with the `SQLPS` module:

```
#add ManagedDTS assembly
Add-Type -AssemblyName "Microsoft.SqlServer.ManagedDTS,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91"
```

We then need to create an application object, which contains the methods to create the folders:

```
$server = "KERRIGAN"

#create new app
$app = New-Object ("Microsoft.SqlServer.Dts.Runtime.Application")
```

To create the folder in the SSIS package store, we first check if the folder is created already. If not, we create the folder using the `CreateFolderOnDtsServer` method of the `DTS Application` object, which accepts the parent path, the new folder name, and the server name:

```
#folder in package store
#will appear under "Stored Packages > File System"
if (!$app.FolderExistsOnDtsServer("\File System\$($newfolder)",
$server))
{
    $app.CreateFolderOnDtsServer("\File System\", $newfolder, $server)
}
```

Creating the folder in the SSIS instance is very similar to creating folders in the package store. However, the methods to check and create the instance folders accept more parameters. Both the `FolderExistsOnSqlServer` and `CreateFolderOnSqlServer` methods of the `DTS application` object accept two extra parameters for username and password used to authenticate to SQL Server:

```
#folder in SSIS instance
#will appear under "Stored Packages > MSDB"
$newfolder = "QueryWorks SSIS $($ts)"
if (!$app.FolderExistsOnSqlServer($newfolder, $server, $null, $null))
{
    $app.CreateFolderOnSqlServer("", $newfolder, $server, $null,
    $null)
}
```

See also

- ▶ The *Creating an SSISDB folder* recipe
- ▶ Learn more about the Application class:
<http://msdn.microsoft.com/en-us/library/ms211665>

Deploying an SSIS package to the package store

In this recipe, we will deploy an SSIS package (.dtmx) to the SSIS package store.

Getting ready

Use the sample SSIS package—`Customer Package.dtsx`—that came with the downloadable code of this book. Save this file to `C:\SSIS`. We will deploy this to our SSIS instance, and save it under the `\File System\QueryWorks` package folder. Alternatively, use a .dtmx package that is readily available in your environment.

How to do it...

Let's explore the code required to deploy an SSIS .dtmx file.

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the ManagedDTS assembly as follows:

```
#add ManagedDTS assembly
Add-Type -AssemblyName "Microsoft.SqlServer.ManagedDTS,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080
cc91"
```

3. Add the following script and run:

```
$server = "KERRIGAN"

#create new app
$app = New-Object "Microsoft.SqlServer.Dts.Runtime.Application"

#specify package to be deployed
$dtsx = "C:\SSIS\Customer Package.dtsx"
$package = $app.LoadPackage($dtsx, $null)

#where are we going to deploy?
```

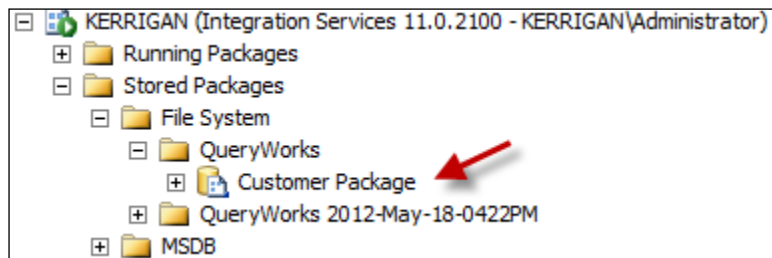
```

$SSISPackageStorePath = "\File System\QueryWorks"
$destinationName = "$($SSISPackageStorePath)\$($package.Name)"

#save to the package store
$app.SaveToDtsServer($package, $events, $destinationName, $server)


```

When done, log in to the SSIS instance in Management Studio, and confirm that the package has been deployed:



How it works...

Deploying a `.dtsx` file to the package store in the filesystem, or the `msdb` database, is considered a legacy way of deploying SSIS packages in SQL Server 2012. This is now referred to as a Package Deployment model.

 See the *Deploying an ISPAC file to SSISDB* recipe for more details on deploying SSIS projects in SQL Server 2012.

Although this may be considered legacy already, this may still be the preferred way to deploy packages in some environments for a while.

To deploy programmatically, we must first create a handle to the `ManagedDTS` assembly:

```

#add ManagedDTS assembly
Add-Type -AssemblyName "Microsoft.SqlServer.ManagedDTS,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91"

```

After loading the `ManagedDTS` object, we need to create an `Application` object:

```

$server = "KERRIGAN"

#create new app
$app = New-Object "Microsoft.SqlServer.Dts.Runtime.Application"

```

We then need to load the SSIS `.dtsx` package into a variable using the `LoadPackage` method of the `DTS Application` object. We will load the package from the `C:\SSIS` folder where we saved the `Customer Package.dtsx` package:

```
#deploy a package
$dtsx = "C:\SSIS\Customer Package.dtsx"
$package = $app.LoadPackage($dtsx, $null)
```

We also need to specify where the package is going to be deployed. If the package is to be deployed to the File System, we prefix the path with `\File System\`; if to the database, we prefix `\MSDB\`.

```
#where are we going to deploy?
$SSISPackageStorePath = "\File System\QueryWorks"
$destinationName = "$($SSISPackageStorePath)\ $($package.Name)"

#save to the package store
$app.SaveToDtsServer($package, $events, $destinationName, $server)
```

If you want to save to the MSDB folder, you will have to use the `SaveToSQLServer` method instead of the `SaveToDtsServer` method.

See also

- ▶ The *Deploying an ISPAC file to SSISDB* recipe
- ▶ Check out the `Application.LoadPackage` method documentation from MSDN:
<http://msdn.microsoft.com/en-us/library/ms188550.aspx>

Executing an SSIS package stored in the package store or File System

In this recipe, we will execute an SSIS package using PowerShell.

Getting ready

In our recipe, we will execute `Customer Package`, which is saved in the package store, and we will also execute the `C:\SSIS\SamplePackage.dtsx` file—also included in the downloadable files for this chapter—directly from the filesystem.

Alternatively, you can locate an available SSIS package in your system that you want to execute instead. Identify whether this package is stored in the filesystem, or in the SSIS package store.

How to do it...

Let's explore the code required to execute an SSIS package programmatically using PowerShell.

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the ManagedDTS assembly as follows:

```
#add ManagedDTS assembly
Add-Type -AssemblyName "Microsoft.SqlServer.ManagedDTS,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080
cc91"
```

3. Add the following script and run:

```
$server = "KERRIGAN"

#create new app we'll use for SSIS
$app = New-Object "Microsoft.SqlServer.Dts.Runtime.Application"

#execute package in SSIS Package Store
$packagePath = "\File System\QueryWorks\Customer Package"
$package = $app.LoadFromDtsServer($packagePath, $server,$null)
$package.Execute()

#execute package saved in filesystem
$packagePath = "C:\SSIS\SamplePackage.dtsx"
$package = $app.LoadPackage($packagePath, $null)
$package.Execute()
```

How it works...

In SQL Server 2012, a new method of storing is introduced to SSIS. Using the Project Deployment model, SSIS packages are deployed with their corresponding parameters and environments to the SSISDB catalog. SQL Server 2012 still supports the legacy way of storing packages, however—which is through the filesystem, or package store.

The default package store is in:

```
<SQL Server Install Directory>\110\DTS\Packages
```

The first step is to load the ManagedDTS assembly, and create an Application object:

```
#add ManagedDTS assembly
Add-Type -AssemblyName "Microsoft.SqlServer.ManagedDTS,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91"
```

```
#create new app we'll use for SSIS
$app = New-Object "Microsoft.SqlServer.Dts.Runtime.Application"
```

To load a package stored in the package store, we need to use the `LoadFromDtsServer` method of the DTS Application object and supply it with three parameters—the path to the package relative to the File System, the server name, and a third parameter for events, which we will leave null.

```
$packagePath = "\File System\QueryWorks\Customer Package"
$package = $app.LoadFromDtsServer($packagePath, $server,$null)
```

If a package is stored in the filesystem, we have to use the method `LoadPackage` of the DTS Application object, and pass to it the path of the package:

```
$packagePath = "C:\SSIS\SamplePackage.dtsx"
$package = $app.LoadPackage($packagePath, $null)
```

If you still have packages deployed in msdb, you can also execute these packages by using the `LoadFromSqlServer` method of the DTS Application object:

```
$packagePath = "\MSDB\SamplePackage"
$package = $app.LoadFromSqlServer($packagePath, $server, $null, $null,
    $null)
$package.Execute()
```

There's more...

Before a package can be executed, it must be loaded first. Check out different methods to load an SSIS package:

- ▶ `LoadFromSqlServer`:
<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.dts.runtime.application.loadfromsqlserver.aspx>
- ▶ `LoadFromDtsServer`:
<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.dts.runtime.application.loadfromdtsserver.aspx>
- ▶ `LoadPackage`:
<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.dts.runtime.application.loadpackage.aspx>

See also

- ▶ The *Executing an SSIS package stored in SSISDB* recipe

Downloading an SSIS package to a file

This recipe will download an SSIS package back to a `.dtsx` file.

Getting ready

Locate a package stored in the package store that you want to download to the filesystem. Note the path to this package.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Add the ManagedDTS assembly as follows:

```
#add ManagedDTS assembly
Add-Type -AssemblyName "Microsoft.SqlServer.ManagedDTS,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080
cc91"
```

3. Add the following script and run:

```
$server = "KERRIGAN"

#create new app
$app = New-Object "Microsoft.SqlServer.Dts.Runtime.Application"

$timestamp = Get-Date -format "yyyy-MMM-dd-hhmm"
$destinationFolder = "C:\SSIS"

$packageToDownload = "Customer Package"
$packageParentPath = "\File System\QueryWorks"

#download the specified package
#here we're dealing with a package in
#the SSIS Package store
$app.GetDtsServerPackageInfos($packageParentPath,$server) |
Where-Object Flags -eq "Package" |
ForEach-Object {
    $package = $_
    $packagePath = "$($package.Folder)\$($package.Name)"
```



```
#check if this package does exist in the Package Store
if($app.ExistsOnDtsServer($packagePath, $server))
{
    $fileName = Join-Path $destinationFolder "$($package.
Name)_${$timestamp}.dtsx"
    $newPackage = $app.LoadFromDtsServer($packagePath,
$server,$null)
    $app.SaveToXml($fileName, $newPackage, $null)
}
}
```

How it works...

The first step is to load the ManagedDTS assembly and create an application object:

```
#add ManagedDTS assembly
Add-Type -AssemblyName "Microsoft.SqlServer.ManagedDTS,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91"
$server = "KERRIGAN"

#create new app
$app = New-Object "Microsoft.SqlServer.Dts.Runtime.Application"
```

We will also define our variables for timestamp, destination folder, and which package we want to download:

```
$timestamp = Get-Date -format "yyyy-MMM-dd-hhmm"
$destinationFolder = "C:\SSIS"

$packageToDownload = "Customer Package"
$packageParentPath = "\File System\QueryWorks"
```

We then retrieve all packages using the `GetDtsServerPackageInfos` method of the DTS application object:

```
$app.GetDtsServerPackageInfos($packageParentPath,$server) |
Where-Object Flags -eq "Package" |
ForEach-Object {
```

For each package, we check if this matches the package we wanted to download. If it does, we can use the `LoadFromDtsServer` method to load the package, and use the `SaveToXml` method to save the package back to the filesystem. Remember that a `.dtsx` file is simply an XML file.

```
ForEach-Object {
    $package = $_
```

```

$packagePath = "$($package.Folder)\$($package.Name) "

#check if this package does exist in the Package Store
if($app.ExistsOnDtsServer($packagePath, $server))
{
    $fileName = Join-Path $destinationFolder "$($package.
Name)_${timestamp}.dtsx"
    $newPackage = $app.LoadFromDtsServer($packagePath,
$server,$null)
    $app.SaveToXml($fileName, $newPackage, $null)
}
}

```

Note that we constructed a timestamped filename for our recipe; you can definitely change this filename to whatever suits your requirements.

See also

- ▶ The *Deploying an SSIS package to a package store* recipe
- ▶ Learn more about:
 - `Application.SaveToXml`:
<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.dts.runtime.application.savetoxml.aspx>
 - `LoadFromDtsServer`:
<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.dts.runtime.application.loadfromdtsserver.aspx>

Creating an SSISDB catalog

In this recipe, we will create an SSISDB catalog.

Getting ready

To create an SSISDB catalog, we must first enable `SQLCLR` on the instance. Log in to SQL Server Management Studio, and use the system stored procedure `sp_configure` to enable CLR. Execute the following T-SQL script:

```

sp_configure 'clr enabled', 1
GO
RECONFIGURE
GO

```

How to do it...

Let's step through creating SSISDB programmatically.

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the SQLPS module as follows:

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

3. Load the IntegrationServices assembly as follows:

```
Add-Type -AssemblyName "Microsoft.SqlServer.Management.
IntegrationServices, Version=11.0.0.0, Culture=neutral,
PublicKeyToken=89845dcd8080cc91"
```

4. Add the following script and run:

```
$instanceName = "KERRIGAN"
$connectionString = "Data Source=$instanceName;Initial
Catalog=master;Integrated Security=SSPI"

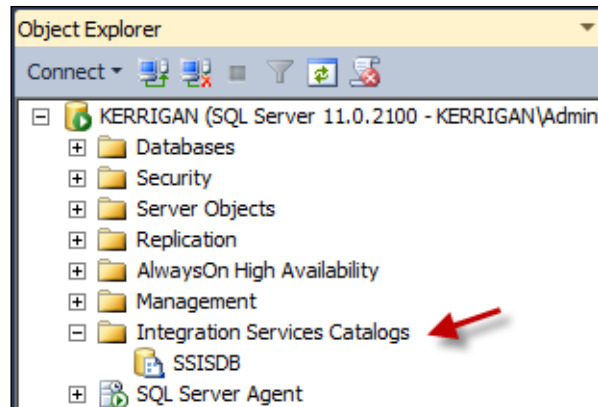
$conn = New-Object System.Data.SqlClient.SqlConnection
$conn.ConnectionString

$SSIServer = New-Object Microsoft.SqlServer.Management.
IntegrationServices.IntegrationServices $conn

if(!$SSIServer.Catalogs["SSISDB"])
{
    #constructor accepts three (3) parameters:
    #parent, name, password
    $SSISDB = New-Object Microsoft.SqlServer.Management.
IntegrationServices.Catalog ($SSIServer, "SSISDB", "P@ssword")
    $SSISDB.Create()
}
```

How it works...

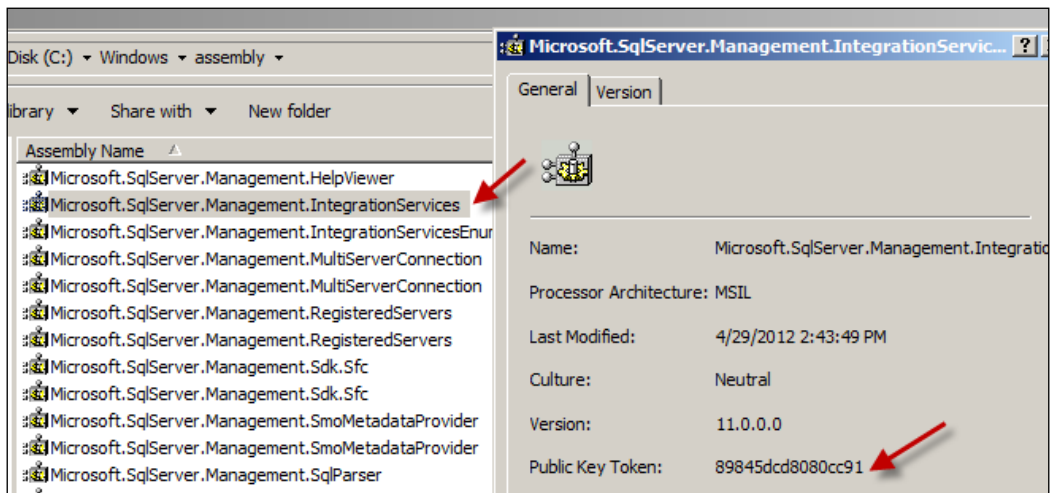
SQL Server 2012 introduces SSIS catalog for Integration Services. The catalog is implemented as a database called **SSISDB** that stores Integration Services objects (projects, packages, and parameters) and logs when projects are deployed using the new Project Deployment model. This database is accessible from SQL Server Management Studio and can be queried like any regular database:



To create **SSISDB** programmatically, we must first load the `IntegrationServices` assembly. This assembly exposes the SSIS Catalog Managed Object Model to allow programmatic access to the new SSIS objects:

```
Add-Type -AssemblyName "Microsoft.SqlServer.Management.
IntegrationServices, Version=11.0.0.0, Culture=neutral,
PublicKeyToken=89845dcd8080cc91"
```

To figure out the version and public key token, you can check out `C:\Windows\assembly`, and check the properties of this assembly:



First, we need to create a `SqlConnection` object, which we will need to pass to the `IntegrationServices` constructor:

```
$instanceName = "KERRIGAN"
$connectionString = "Data Source=$instanceName;Initial
Catalog=master;Integrated Security=SSPI;"
$conn = New-Object System.Data.SqlClient.SqlConnection
$connectionString
```

We then need to create an `IntegrationServices` object:

```
$SSISServer = New-Object Microsoft.SqlServer.Management.
IntegrationServices.IntegrationServices $conn
```

To create an SSISDB catalog, we create a new `Catalog` object, which accepts three parameters: `IntegrationServices` server object, name of the catalog (SSISDB), and a password:

```
if(!$SSISServer.Catalogs["SSISDB"])
{
    #constructor accepts three (3) parameters:
    #parent, name, password
    $SSISDB = New-Object Microsoft.SqlServer.Management.
IntegrationServices.Catalog ($SSISServer, "SSISDB", "P@ssword")
    $SSISDB.Create()
}
```

See also

- ▶ The *Deploying an ISPAC file to SSISDB* recipe
- ▶ Check out additional information about SQL Server 2012 SSIS:
<http://msdn.microsoft.com/en-us/library/gg471508.aspx>
- ▶ Learn more about SSISDB catalog from MSDN:
<http://msdn.microsoft.com/en-us/library/hh479588.aspx>
- ▶ See the properties and methods for the new `IntegrationServices` assembly:
[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.integrationservices\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.integrationservices(v=sql.110).aspx)

Creating an SSISDB folder

In this recipe, we will create a folder in the SSISDB catalog.

Getting ready

In this recipe, we assume that the SSISDB catalog has been created. We will create a folder called `QueryWorks` inside the SSISDB catalog.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the `SQLPS` module as follows:

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

3. Load the `IntegrationServices` assembly:

```
Add-Type -AssemblyName "Microsoft.SqlServer.Management.
IntegrationServices, Version=11.0.0.0, Culture=neutral,
PublicKeyToken=89845dcd8080cc91"
```

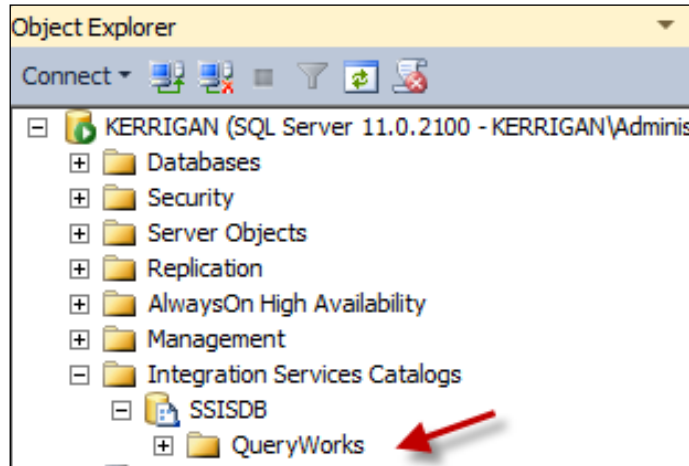
4. Add the following script and run:

```
$instanceName = "KERRIGAN"
$connectionString = "Data Source=$instanceName;Initial
Catalog=master;Integrated Security=SSPI"
$conn = New-Object System.Data.SqlClient.SqlConnection
$connectionString

$SSISServer = New-Object Microsoft.SqlServer.Management.
IntegrationServices.IntegrationServices $conn
$SSISDB = $SSISServer.Catalogs["SSISDB"]

#create QueryWorks catalog folder here
$folderName = "QueryWorks"
$folderDescription = "New SSISDB folder"
$SSISDBFolder = New-Object Microsoft.SqlServer.Management.
IntegrationServices.CatalogFolder ($SSISDB, $folderName,
$folderDescription)
$SSISDBFolder.Create()
```

When done, log in to Management Studio and connect to your database engine. Expand **Integration Services Catalogs**, and check that the folder has been created under the **SSISDB** node:



How it works...

A folder in an SSISDB catalog can hold multiple projects and environments.

To create a folder inside SSISDB, also called a catalog folder, we must first get a handle to SSISDB. The core code required to do this is as follows:

```
$$SSISServer = New-Object Microsoft.SqlServer.Management.  
IntegrationServices.IntegrationServices $conn  
$$SSISDB = $$SSISServer.Catalogs["SSISDB"]
```

Once we have the SSISDB handle, creating the folder is straightforward. It requires creating a new `CatalogFolder` object. The constructor takes in the SSISDB object, the name of the catalog folder, and the description:

```
#create QueryWorks catalog folder here  
$folderName = "QueryWorks"  
$folderDescription = "New SSISDB folder"  
$$SSISDBFolder = New-Object Microsoft.SqlServer.Management.  
IntegrationServices.CatalogFolder ($$SSISDB, $folderName,  
$folderDescription)
```

The `Create()` method will persist the catalog folder in SSISDB:

```
$$SSISDBFolder.Create()
```

See also

- ▶ The *Creating an SSISDB catalog* recipe
- ▶ The *Deploying an ISPAC file to SSISDB* recipe
- ▶ Check out the properties and methods supported by the `CatalogFolder` class:
<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.integrationservices.catalogfolder.aspx>

Deploying an ISPAC file to SSISDB

You will see how to deploy an ISPAC file to SSISDB.

Getting ready

Save the `Customer Package Project.ispac` file provided with the sample code of this book to the `C:\SSIS` folder. Alternatively, if you have an available ISPAC file that you want to use, change the `$ispacFilePath` variable's value to reflect your file.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the `SQLPS` module as follows:

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

3. Load the `IntegrationServices` assembly:

```
Add-Type -AssemblyName "Microsoft.SqlServer.Management.
IntegrationServices, Version=11.0.0.0, Culture=neutral,
PublicKeyToken=89845dcd8080cc91"
```

4. Add the following script and run:

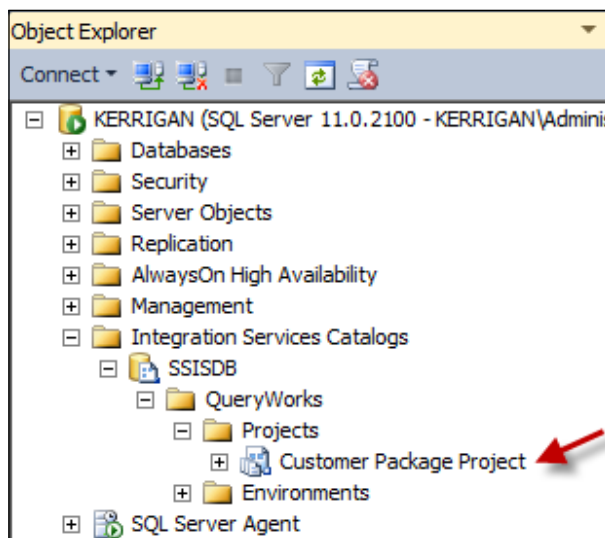
```
$instanceName = "KERRIGAN"
$connectionString = "Data Source=$instanceName;Initial
Catalog=master;Integrated Security=SSPI"
$conn = New-Object System.Data.SqlClient.SqlConnection
$conn.ConnectionString

$SSISServer = New-Object Microsoft.SqlServer.Management.
IntegrationServices.IntegrationServices $conn
$SSISDB = $SSISServer.Catalogs["SSISDB"]
```



```
$SSISDBFolderName = "QueryWorks"  
$SSISDBFolder = $SSISDB.Folders[$SSISDBFolderName]  
  
$ispacFilePath = "C:\SSIS\Customer Package Project.ispac"  
[byte[]] $ispac = [System.IO.File]::ReadAllBytes($ispacFilePath)  
  
$SSISDBFolder.DeployProject("Customer Package Project", $ispac)
```

When done, log in to Management Studio and expand **Integration Services Catalogs**. Under **SSISDB**, open the **QueryWorks** folder, and confirm that **Customer Package Project** has been deployed:



How it works...

SQL Server 2012 Integration Services supports two deployment models: Package Deployment model and Project Deployment model. The Package Deployment model is the older, legacy way of deploying, where packages are deployed as standalone entities. The newer Project Deployment model is the default mode supported when you create a new SSIS project in **SQL Server Data Tools (SSDT)**, previously known as **Business Intelligence Development Studio (BIDS)**.

In the Package Deployment model, everything needed to deploy a project is packaged up into a single file with an `.ispac` extension. This file is created when you deploy the SSIS 2012 project. Although it appears to be a single file, you will discover that this is a series of files that have been compressed. Simply change the `.ispac` extension to `.zip`, and extract the file. You should see something similar to the files shown in the following screenshot:

Name ^	Date modified	Type
@Project.manifest	5/16/2012 1:05 PM	MANIFEST File
[Content_Types].xml	5/16/2012 1:05 PM	XML Document
Package.dtsx	5/16/2012 1:05 PM	Integration Services Package
Project.params	5/16/2012 1:05 PM	PARAMS File

A package manifest has been created when the SSIS was built in SQL Server Data Tools (SSDT), in addition to the package files and parameter file.

To deploy the `.ispac` file programmatically using PowerShell and the new SSIS object model, we first need to load the `IntegrationServices` assembly, and create a handle to the `IntegrationServices` object:

```
Add-Type -AssemblyName "Microsoft.SqlServer.Management.
IntegrationServices, Version=11.0.0.0, Culture=neutral, PublicKeyToken
=89845dcd8080cc91"

$instanceName = "KERRIGAN"
$connectionString = "Data Source=$instanceName;Initial
Catalog=master;Integrated Security=SSPI"
$conn = New-Object System.Data.SqlClient.SqlConnection
$connectionString
$SSISServer = New-Object Microsoft.SqlServer.Management.
IntegrationServices.IntegrationServices $conn
```

The next step is to get a handle to the folder where the ISPAC file will be deployed. This means we need to get a handle to each object in the hierarchy that leads to the folder, that is, create a handle to `SSISDB`, and then to the folder:

```
$SSISDB = $SSISServer.Catalogs["SSISDB"]

$SSISDBFolderName = "QueryWorks"
$SSISDBFolder = $SSISDB.Folders[$SSISDBFolderName]
```

Once we have a handle to the folder, we need to read the byte content of the ISPAC file, and use the `DeployProject` method of the `SSISDBFolder` object available with the catalog folder object:

```
$ispacFilePath = "C:\SSIS\Customer Package Project.ispac"
[byte[]] $ispac = [System.IO.File]::ReadAllBytes($ispacFilePath)
$SSISDBFolder.DeployProject("Customer Package Project", $ispac)
```

See also

- ▶ The *Creating an SSISDB catalog* recipe
- ▶ The *Creating an SSISDB folder* recipe
- ▶ To learn more about the specification, check out "[MS-ISPAC]: Integration Services Project Deployment File Format Structure Specification":
[http://msdn.microsoft.com/en-us/library/ff952821\(v=sql.110\)](http://msdn.microsoft.com/en-us/library/ff952821(v=sql.110))
- ▶ See the properties and methods for the new `IntegrationServices` assembly:
[http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.integrationsservices\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.integrationsservices(v=sql.110).aspx)

Executing an SSIS package stored in SSISDB

In this recipe, we execute a package stored in SSISDB.

Getting ready

In this recipe, we execute the package that comes with the `Customer Package Project` that was deployed in the *Deploying an ISPAC File to SSISDB* recipe. Alternatively, replace the variables for folder, project, and package names.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the `SQLPS` module as follows:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking
```
3. Load the `IntegrationServices` assembly:

```
Add-Type -AssemblyName "Microsoft.SqlServer.Management.
IntegrationServices, Version=11.0.0.0, Culture=neutral,
PublicKeyToken=89845dcd8080cc91"
```

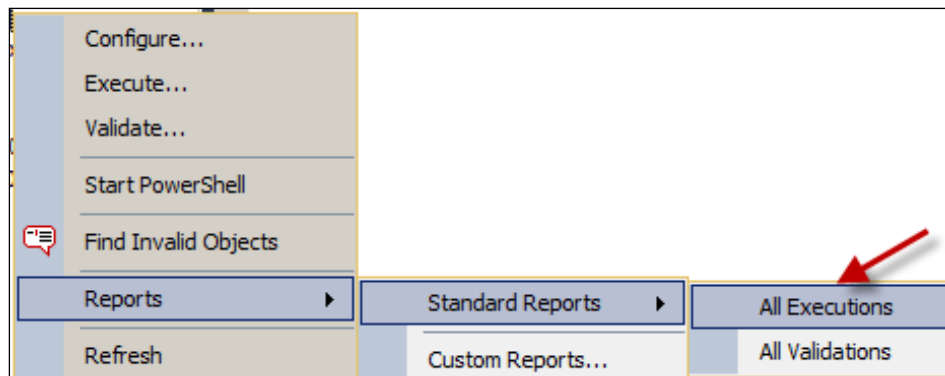
4. Add the following script and run:

```
$instanceName = "KERRIGAN"
$connectionString = "Data Source=$instanceName;Initial
Catalog=master;Integrated Security=SSPI;"
$conn = New-Object System.Data.SqlClient.SqlConnection $constr
$SSISServer = New-Object Microsoft.SqlServer.Management.
IntegrationServices.IntegrationServices $conn
$SSISDB = $SSISServer.Catalogs["SSISDB"]

$SSISDBFolderName = "QueryWorks"
$SSISDBFolder = $SSISDB.Folders[$SSISDBFolderName]
$projectName= "Customer Package Project"
$packageName= "Package.dtsx"
$SSISDBFolder.Projects[$projectName].Packages[$packageName].
Execute($false, $null)
```

Once the script finishes, it will return the process ID of the execution.

5. Confirm this process ID against the executions of the report:
 1. Connect to Management Studio.
 2. Go to Integration Services package.
 3. Right-click on the package you executed through the script.
 4. Go to **Reports | Standard Reports | All Executions:**



You should see the **All Executions** report rendered. Confirm that the ID returned by the script is in the report. You can also check the execution start time (not shown in the screenshot, but is in the third rightmost column of the report):

All Executions
on KERRIGAN at 5/17/2012 10:22:35 PM
This report provides information about the Integration Services package executions that have been performed on the connected SQL Server instance.

Filter: Start time range: 5/11/2012 - 5/17/2012; Status: All; (4 more)

Execution Information

0 Failed 0 Running 4 Succeeded 0 Others

ID	Status	Report	Folder Name	Project Name	Package Name
9	Succeeded	Overview All Messages Execution Performance	QueryWorks	Customer Package Project	Package.dtsx
8	Succeeded	Overview All Messages Execution Performance	QueryWorks	Customer Package Project	Package.dtsx
7	Succeeded	Overview All Messages Execution Performance	QueryWorks	Customer Package Project	Package.dtsx
6	Succeeded	Overview All Messages Execution Performance	QueryWorks	Customer Package Project	Package.dtsx

How it works...

To execute a package stored in the SSISDB catalog, we need to get a handle to the package first. To get a handle to the package, we must first get to the SSISDB catalog:

```
$connectionString = "Data Source=$instanceName;Initial
Catalog=master;Integrated Security=SSPI;"
$conn = New-Object System.Data.SqlClient.SqlConnection $constr
$SSISServer = New-Object Microsoft.SqlServer.Management.
IntegrationServices.IntegrationServices $conn
$SSISDB = $SSISServer.Catalogs["SSISDB"]
```

We also need to have access to the folder where the package is saved:

```
$SSISDBFolderName = "QueryWorks"
$SSISDBFolder = $SSISDB.Folders[$SSISDBFolderName]
```

To execute, we must trace where the package is and invoke the `Execute` method of the `Package` object. The method accepts two parameters: a Boolean value for `use32RuntimeOn64`, and `EnvironmentReference`:

```

$projectName= "Customer Package Project"
$packageName= "Package.dtsx"
$SSISDBFolder.Projects[$projectName].Packages[$packageName].
Execute($false, $null)

```

This method returns the process ID of the execution.

See also

- ▶ The *Executing an SSIS package stored in the package store or File System* recipe
- ▶ Check out these MSDN articles related to PackageInfo:
<http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.integrationservices.packageinfo.aspx>

Listing SSAS cmdlets

This recipe lists the new SSAS cmdlets in SQL Server 2012.

How to do it...

Let's explore the code required to list the SSAS cmdlets.

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

```
Get-Command -Module SQLASCmdlets
```

This should give a result similar to this:

```

PS SQLSERVER:\> Get-Command -Module SQLASCmdlets

```

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

How it works...

SQL Server Analysis Services (SSAS) gets some PowerShell love in SQL Server 2012. You can import the `SQLASCMDLETS` module to start using the new cmdlets.

To list the new `AS` cmdlets, simply use the `Get-Command` as follows:

```
Get-Command -Module SQLASCMDLETS
```

You will notice that some of the common SSAS tasks have been wrapped in cmdlets, such as `Backup-ASDatabase`, `Restore-ASDatabase`, `Invoke-ASCmd`, `Invoke-ProcessCube`, and the like.

See also

- ▶ The *Listing SSAS instance properties* recipe
- ▶ Check out these MSDN articles related to:
 - Analysis Services PowerShell:
<http://msdn.microsoft.com/en-us/library/hh213141.aspx>
 - Analysis Services PowerShell Reference:
<http://msdn.microsoft.com/en-us/library/hh758425.aspx>

Listing SSAS instance properties

We will list SSAS instance properties in this recipe.

How to do it...

Let's explore the code required to list SSAS instance properties.

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the `SQLPS` module as follows:

```
Import-Module SQLASCMDLETS -DisableNameChecking
```

3. Add the following script and run:

```
#Connect to your Analysis Services server
$SSASServer = New-Object Microsoft.AnalysisServices.Server

$instanceName = "KERRIGAN"
$SSASServer.connect($instanceName)
```

```
#get all properties
$SSASServer | Select *
```

You should see a result similar to this:

```
ConnectionString : KERRIGAN
ConnectionInfo  : Microsoft.AnalysisServices.ConnectionInfo
SessionID       : CE7B36E7-4F48-421B-B22E-E53B31129472
CaptureXml      : False
CaptureLog      : {}
Connected       : True
SessionTrace    : Microsoft.AnalysisServices.SessionTrace
Version         : 11.0.2100.60
Edition         : Developer64
EditionID       : 2176971986
ProductLevel    : CTP
Databases       : {SampleDW}
Assemblies      : {System, EXCELXLINTERNAL, VBAMDXINTERNAL, V
Traces          : {FlightRecorder}
Roles           : {Administrators}
ServerProperties : {DataDir, TempDir, LogDir, BackupDir...}
ProductName     : Microsoft SQL Server Analysis Services
ServerMode      : Multidimensional
IsLoaded        : True
```

How it works...

To get SSAS instance properties, we first need to load the SQLASCMDLETS module:

```
Import-Module SQLASCMDLETS -DisableNameChecking
```

We can then create an Analysis Server object and connect to our instance:

```
#Connect to your Analysis Services server
$SSASServer = New-Object Microsoft.AnalysisServices.Server

$instanceName = "KERRIGAN"
$SSASServer.connect($instanceName)
```

Once we get a handle to our SSAS instance, we can display its properties:

```
#get all properties
$SSASServer | Select *
```

Note that in SQL Server 2012, there are two flavors of Analysis Services: multidimensional and tabular. You can identify this by checking the `ServerMode` properties.

See also

- ▶ The *Listing SSAS cmdlets* recipe
- ▶ Check out these MSDN articles related to SQL Server Analysis Services class:
<http://msdn.microsoft.com/en-us/library/microsoft.analysissservices.server.aspx>

Backing up an SSAS database

In this recipe, we will create an SSAS database backup.

Getting ready

Choose an SSAS database you want to back up, and replace the `-Name` parameter in the recipe. Ensure that you are running PowerShell with administrator privileges to the SSAS instance.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the `SQLPS` module as follows:

```
#import SQLASCMDLETS module
Import-Module SQLASCMDLETS -DisableNameChecking
```

3. Add the following script and run:

```
$instanceName = "KERRIGAN"
$backupfile = "C:\Temp\AWDW.abf"
Backup-ASDatabase -BackupFile $backupfile -Name "SampleDW" -Server
$instanceName -AllowOverwrite -ApplyCompression
```

How it works...

The `Backup-ASDatabase` cmdlet allows multidimensional or tabular SSAS databases to be backed up to a file. In our recipe, we chose to do a compressed backup for the `SampleDW` SSAS database to an Analysis Services Backup file (`.abf`).

```
$instanceName = "KERRIGAN"
$backupfile = "C:\Temp\AWDW.abf"
Backup-ASDatabase -BackupFile $backupfile -Name "SampleDW" -Server
$instanceName -AllowOverwrite -ApplyCompression
```

Other switches that can be set using the `Backup-ASDatabase` cmdlet are:

- ▶ `-BackupRemotePartitions <SwitchParameter>`
- ▶ `-FilePassword <SecureString>`
- ▶ `-Locations <Microsoft.AnalysisServices.BackupLocation[] >`
- ▶ `-Server <string>`
- ▶ `-Credential <PSCredential>`

See also

- ▶ The *Restoring an SSAS database* recipe
- ▶ Learn more about the `Backup-ASDatabase` cmdlet:
<http://msdn.microsoft.com/en-us/library/hh479574.aspx>

Restoring an SSAS database

You will see how to restore an SSAS database in this recipe.

Getting ready

Locate your SSAS backup file, and replace the backup file parameter with the location of your file.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Import the `SQLASCMDLETS` module as follows:

```
#import SQLASCMDLETS module
Import-Module SQLASCMDLETS -DisableNameChecking
```
3. Add the following script and run:

```
$instanceName = "KERRIGAN"
$backupfile = "C:\Temp\AWDW.abf"
Restore-ASDatabase -RestoreFile $backupfile -Server $instanceName
-Name "SampleDW" -AllowOverwrite
```

How it works...

The `Restore-ASDatabase` cmdlet allows multidimensional or tabular SSAS databases to be restored when provided with a backup file:

```
$instanceName = "KERRIGAN"
$backupfile = "C:\Temp\AWDW.abf"
Restore-ASDatabase -RestoreFile $backupfile -Server $instanceName
-Name "SampleDW" -AllowOverwrite
```

See also

- ▶ The *Backing up an SSAS database* recipe
- ▶ Learn more about the `Restore-ASDatabase` cmdlet:
<http://msdn.microsoft.com/en-us/library/hh510169.aspx>

Processing an SSAS cube

In this recipe, we will process an SSAS cube.

Getting ready

Choose a cube that is readily available in your SSAS instance.

How to do it...

1. Open the PowerShell console by going to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.

2. Import the `SQLASCMDLETS` module as follows:

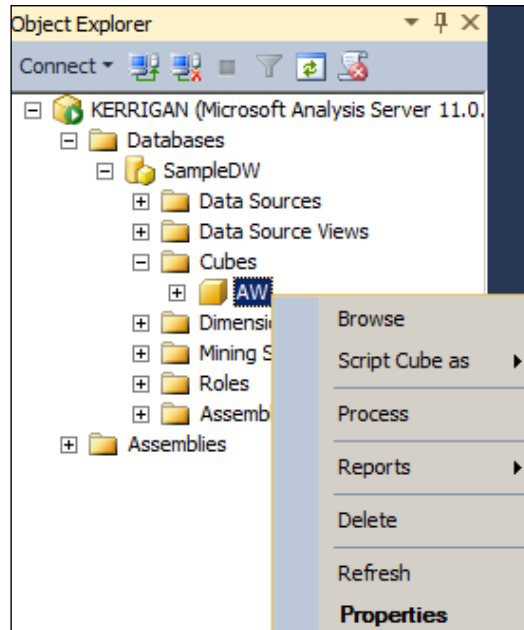
```
#import SQL Server module
Import-Module SQLASCMDLETS -DisableNameChecking
```

3. Add the following script and run:

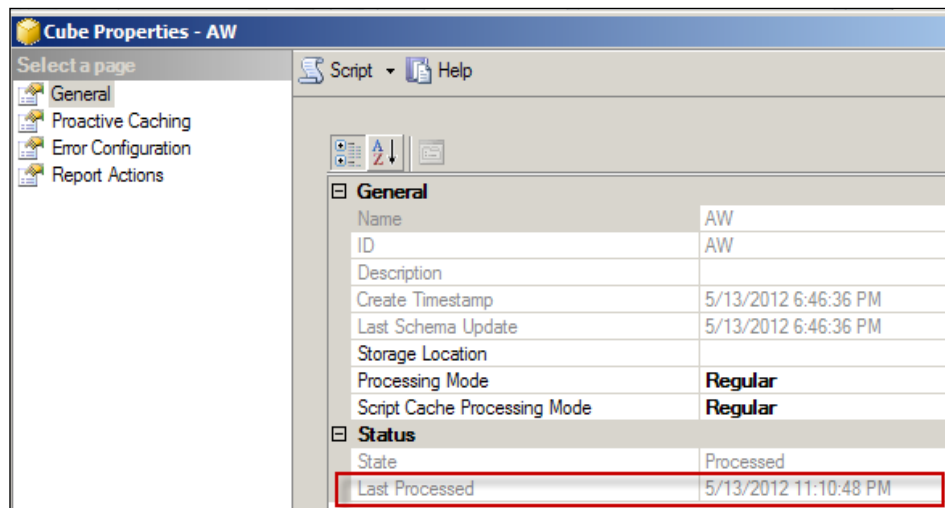
```
$instanceName = "KERRIGAN"
Invoke-ProcessCube -Name "AW" -Server $instanceName -Database
"SampleDW" -ProcessType ([Microsoft.AnalysisServices.
ProcessType]::ProcessFull)
```

To check that the cube has been processed:

1. Go to Management Studio and connect to SQL Server Analysis Services.
2. Right-click on the cube you've just processed and go to **Properties**:



3. In the **General** section, check the **Last Processed** value. It should be updated to when the script finished executing, as shown in the following screenshot:



How it works...

Processing, or reprocessing, a cube is a common task that needs to be done in an SSAS environment on a regular basis. Processing a cube ensures that your cube has the latest data that has been loaded to the source data warehouse, or any changes to the cube's structure are in place.

The `Invoke-Process` cmdlet simplifies this process if you are doing this task through PowerShell:

```
Invoke-ProcessCube -Name "AW" -Database "SampleDW" -ProcessType  
( [Microsoft.AnalysisServices.ProcessType] :: ProcessFull)
```

All we need to specify is the cube name and the SSAS database where this cube belongs. Processing cubes requires administrative privileges on the SSAS instance.

There are different processing types that can be specified with the `-ProcessType` switch, including `ProcessFull`, `ProcessAdd`, and `ProcessUpdate`. Check out the different processing options and settings from MSDN:

<http://msdn.microsoft.com/en-us/library/ms174774.aspx>

See also

- ▶ The *Backing up an SSAS database* recipe
- ▶ The *Restoring an SSAS database* recipe
- ▶ Check out these MSDN articles related to `Invoke-ProcessCube` cmdlet:
<http://msdn.microsoft.com/en-us/library/hh510171.aspx>
- ▶ To learn more about the different processing options and settings for Analysis Services:
<http://msdn.microsoft.com/en-us/library/ms174774.aspx>

9

Helpful PowerShell Snippets

In this chapter, we will cover:

- ▶ Documenting PowerShell script for Get-Help
- ▶ Getting a timestamp
- ▶ Getting additional error messages
- ▶ Listing processes
- ▶ Getting aliases
- ▶ Exporting to CSV and XML
- ▶ Using Invoke-Expression
- ▶ Testing regular expressions
- ▶ Managing folders
- ▶ Manipulating files
- ▶ Searching for files
- ▶ Reading an event log
- ▶ Sending e-mail
- ▶ Embedding C# code
- ▶ Creating an HTML report
- ▶ Parsing XML
- ▶ Extracting Data from a web service
- ▶ Using PowerShell Remoting

Introduction

In this chapter, we tackle a variety of recipes that are not SQL Server specific; but you may find them useful as you work with PowerShell. Often you will need to create files that use a timestamp, analyze event logs for recent system errors, export a list of processes to CSV or XML, or even access web services. Here you will find snippets of code that you can use in existing or new scripts, or whenever you need them.

Documenting PowerShell script for Get-Help

In this recipe, we will use header comments that can be utilized by the Get-Help cmdlet.

How to do it...

In this recipe, we will explore comment-based Help.

4. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
5. Add the following:

```
<#
.SYNOPSIS
    Creates a full database backup
.DESCRPTION
    Creates a full database backup using specified instance name
    and database name
    This will place the backup file to the default backup directory
    of the instance
.PARAMETER instanceName
    instance where database to be backed up resides
.PARAMETER databaseName
    database to be backed up
.EXAMPLE
    PS C:\PowerShell> .\Backup-Database.ps1 -instanceName
    "QUERYWORKS\SQL01" -databaseName "pubs"
```

.EXAMPLE

```
PS C:\PowerShell> .\Backup-Database.ps1 -instance "QUERYWORKS\
SQL01" -database "pubs"
```

.NOTES

To get help:

```
Get-Help .\Backup-Database.ps1
```

```
#>
```

```
param
```

```
(
```

```
    [Parameter(Position=0)]
```

```
    [alias("instance")]
```

```
    [string]$instanceName,
```

```
    [Parameter(Position=1)]
```

```
    [alias("database")]
```

```
    [string]$databaseName
```

```
)
```

```
function main
```

```
{
```

```
    #this is just a stub file
```

```
}
```

```
cls
```

```
#get general help
```

```
Get-Help "C:\PowerShell\Backup-Database.ps1"
```

```
#get examples
```

```
Get-Help "C:\PowerShell\Backup-Database.ps1" -Examples
```

6. Save the script as C:\PowerShell\Backup-Database.ps1.
7. Execute the script from the PowerShell ISE.



Check appendices A and B for executing the script from the PowerShell console.

Here is a sample result:

```
NAME
    C:\PowerShell\Backup-Database.ps1

SYNOPSIS
    Creates a full database backup

SYNTAX
    C:\PowerShell\Backup-Database.ps1 [[-instanceName] <String>]

DESCRIPTION
    Creates a full database backup using specified instance name.
    This will place the backup file to the default backup directory.

RELATED LINKS

REMARKS
    To see the examples, type: "get-help C:\PowerShell\Backup-Database.ps1 -examples"
    For more information, type: "get-help C:\PowerShell\Backup-Database.ps1 -full"
```

How it works...

Starting PowerShell V2, if a script or a function has some header comments formatted in a specific way, these can be displayed when `Get-Help` is invoked for that function or script. This is also called **comment-based help**.

This comment block must be the first section in a script, or must be the first lines in a function. Once composed, the script or the function name can be passed as a parameter to `Get-Help`.

Some of the core keywords of the comment-based help are as follows:

```
<#
.SYNOPSIS
    summary
.DESCRIPTION
    Description
.PARAMETER parameter1Name
    Parameter description
.PARAMETER parameter1Name
    Parameter description
.EXAMPLE
    Usage example; Appears when you use -examples
```

```
.EXAMPLE
    Usage example; Appears when you use -examples
.NOTES
    Additional notes; Appears when you use -full
#>
```

Additional sections that can be used are as follows:

- ▶ .INPUTS
- ▶ .OUTPUTS
- ▶ .LINK
- ▶ .ROLE
- ▶ .FUNCTIONALITY

There's more...

- ▶ To find out more on *about_Comment_Based_Help*, you can use the MSDN and browse to the following link:
<http://msdn.microsoft.com/en-us/library/windows/desktop/dd819489.aspx>
- ▶ Refer to *WTFM: Writing the Fabulous Manual* available at:
<http://technet.microsoft.com/en-us/magazine/ff458353.aspx>

Getting a timestamp

In this recipe, we simply get the system's current timestamp.

How to do it...

This is how we will get the timestamp.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell**.
2. Add the following script and run it:

```
$timestamp = Get-Date -Format "yyyy-MM-dd-hhmmtt"

#display timestamp
$timestamp
```

Following is a sample result:

```
2012-May-27-0717AM
```

How it works...

Often, we find ourselves needing the timestamp to append to different files we create or modify. To get the timestamp in PowerShell, we simply have to use the `Get-Date` cmdlet, which gives the following default format:

```
Sunday, May 27, 2012 7:21:03 AM
```

To change the format, we can use the `-Format` switch, which accepts a format string. In our recipe, we used the following format: `"yyyy-MMM-dd-hhmmstt"`.

There are a number of standard format strings that return preformatted datetime type, or you can also compose your own format string. Common format strings, as documented in MSDN are as follows:

Format Pattern	Description
tt	AM/PM designator
ss	Seconds with leading zero
mm	Minutes with leading zero
dd	Day of month with leading zero
dddd	Full name of the day of the week
hh	12-hour clock with leading zero
HH	24-hour clock with leading zero
dd	Day of month with leading zero
MM	Numeric month with leading zero
MMM	Abbreviated month name
MMMM	Full month name
YY	Two-digit year
YYYY	Four-digit year

There's more...

- ▶ Information on `DateTimeFormatInfo` class is available at:
<http://msdn.microsoft.com/en-us/library/system.globalization.datetimeformatinfo.aspx>

- ▶ Information on standard Date and Time format strings is available at:
<http://msdn.microsoft.com/en-us/library/az4se3k1.aspx>
- ▶ The Windows PowerShell Tip of the Week – Formatting Dates and Times is available at:
<http://technet.microsoft.com/en-us/library/ee692801.aspx>
- ▶ Information on the MSDN Get-Date is available at:
<http://technet.microsoft.com/en-us/library/hh849887>

Getting additional error messages

In this recipe, we will learn to display additional error messages.

How to do it...

Let's take a look at how to display more error messages.

1. Open the **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
Clear-Host  
$error[0] | Format-List -Force
```

How it works...

PowerShell supports some special variables and constants. Some of these display arguments, user directories, and other settings. The `$error` is an array variable that holds all the error objects that are encountered in your PowerShell session. To display the last error message, you can use the following:

```
$error[0] | Format-List -Force
```

To check the number of errors contained in your variable, you can use the following:

```
$error.Count
```

`$error` works like a circular buffer. By default, `$error` stores the last 256 errors in your session. If you want to increase the number of error objects the array can store, you can set the `$MaximumErrorCount` variable to a new value.

```
$MaximumErrorCount = 300
```

Should you want to clear all the errors, you can use the clear method.

```
$error.Clear()
```

To get more information about variables that are set in your session, you can use the following command:

```
Get-Variable |  
Select Name, Value, Options |  
Format-Table -AutoSize
```

A partial list of special variables is presented in the following table:

Special Variable	Description
\$_	Current pipeline object
\$args	Arguments passed to a function
\$error	Stores the last error
\$home	User's home directory
\$host	Host information
\$match	Regex matches
\$PSHome	Install directory of PowerShell
\$pid	Process ID (PID) of PowerShell process
\$pwd	Present working directory
\$true	Boolean true
\$false	Boolean false
\$null	Null value

Listing processes

In this recipe, we will list processes in the system.

How to do it...

Let's list processes using PowerShell.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**

2. Add the following script and run it:

```
#list all processes to screen
Get-Process
#list 10 most recently started processes
Get-Process |
Sort -Property StartTime -Descending |
Select Name, StartTime, Path, Responding -First 10

#save processes to a text file
$txtFile = "C:\Temp\processes.txt"
Get-Process |
Out-File -FilePath $txtFile -Force

notepad $txtFile

#save processes to a csv file,
#and display first five lines in file
$csvFile = "C:\Temp\processes.csv"
Get-Process |
Export-Csv -Path $csvFile -Force -NoTypeInfoation

Get-Content $csvFile -totalCount 5

#save the top 5 CPU-heavy processes that
#start with S to an xml file,
#and display in Internet Explorer
$xmlFile = "C:\Temp\processes.xml"

#note we are using PowerShell V3 Where-Object syntax
Get-Process |
Where-Object ProcessName -like "S*" |
Sort -Property CPU -Descending |
Select Name, CPU -First 5 |
Export-Clixml -path $xmlFile -Force

Set-Alias ie "$env:programfiles\Internet Explorer\iexplore.exe"

ie $xmlFile
```

How it works...

In this recipe, we have used the `Get-Process` cmdlet to display processes in the system. We explored a few variations in this recipe.

The first example lists all processes.

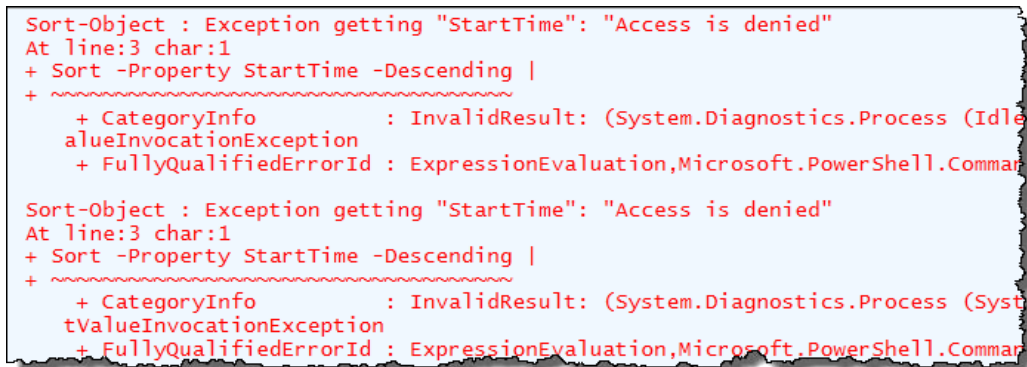
```
Get-Process |
```

The second example is slightly different. We pipe the results of `Get-Process`, and get only the 10 most recently started processes. We achieve this by sorting `StartTime` in descending order, and selecting only the top 10.

```
Sort -Property StartTime -Descending |  
Select Name, StartTime, Path, Responding -First 10
```

Note, however, that this will throw some errors because there are system processes that are not accessible to non-elevated users. Refer to <http://blogs.technet.com/b/heyscriptingguy/archive/2010/08/07/weekend-scripter-boot-tracing-with-windows-powershell.aspx>.

The result is shown in the following screenshot:



```
Sort-Object : Exception getting "StartTime": "Access is denied"  
At line:3 char:1  
+ Sort -Property StartTime -Descending |  
+ ~~~~~  
+ CategoryInfo          : InvalidResult: (System.Diagnostics.Process (Idle  
+ ValueInvocationException  
+ FullyQualifiedErrorId : ExpressionEvaluation,Microsoft.PowerShell.Command  
  
Sort-Object : Exception getting "StartTime": "Access is denied"  
At line:3 char:1  
+ Sort -Property StartTime -Descending |  
+ ~~~~~  
+ CategoryInfo          : InvalidResult: (System.Diagnostics.Process (Syst  
+ ValueInvocationException  
+ FullyQualifiedErrorId : ExpressionEvaluation,Microsoft.PowerShell.Command
```

The results of `Get-Process` can be piped to other cmdlets and exported to different file formats, such as text file, CSV file, or XML.

To pipe results to a text file, we can use the `Out-File` cmdlet.

```
$txtFile = "C:\Temp\processes.txt"  
  
Get-Process |  
Out-File -FilePath $txtFile -Force  
  
notepad $txtFile
```

To create a CSV file, we can use the `Export-Csv` cmdlet. In this sample, we also read back the first five lines of the CSV file that we just created.

```
$csvFile = "C:\Temp\processes.csv"

Get-Process |
Export-Csv -Path $csvFile -Force -NoTypeInfoation

Get-Content $csvFile -totalCount 5
```

If you require an XML format, you can achieve that by using the `Export-Clixml` cmdlet. In this sample, we also filter for only processes that start with `s`, and we only get the top five CPU-heavy processes.

```
#save the top 5 CPU-heavy processes that
#start with S to an xml file,
#and display in Internet Explorer
$xmlFile = "C:\Temp\processes.xml"

#note we are using PowerShell V3 Where-Object syntax
Get-Process |
Where-Object ProcessName -like "S*" |
Sort -Property CPU -Descending |
Select Name, CPU -First 5 |
Export-Clixml -path $xmlFile -Force

Set-Alias ie "$env:programfiles\Internet Explorer\iexplore.exe"

ie $xmlFile
```

The last two lines simply create an alias for Internet Explorer, and then display the XML file.

There's more...

- ▶ Refer to the following link to know more on using the `Get-Process` Cmdlet:
<http://msdn.microsoft.com/en-us/library/ee176855.aspx>
- ▶ Also, refer to MSDN `Get-Process`, available at:
<http://msdn.microsoft.com/en-us/library/hh849832>

See also

- ▶ The *Exporting to CSV and XML* recipe

Getting aliases

In this recipe, we look at aliases in PowerShell.

How to do it...

Let's check out aliases in PowerShell.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
#list all aliases
Get-Alias

#get members of Get-Alias
Get-Alias | Get-Member

#list cmdlet that is aliased as dir
$alias:dir

#list cmdlet that is aliased as ls
$alias:ls

#get all aliases of Get-ChildItem
Get-Alias -Definition "Get-ChildItem"
```

How it works...

The `Get-Alias` cmdlet returns all PowerShell aliases. PowerShell's building blocks are cmdlets, and are named using the `<Verb-Noun>` convention. For example, to list contents of a directory, we use `Get-ChildItem`. There are, however, better-known ways to get this information such as `dir` if running the Windows Command Prompt, and `ls` if running in a Unix environment. Aliases allow most well-known commands to be run from within PowerShell. To list all aliases, use the following:

```
#list all aliases
Get-Alias
```

To get the members of `Get-Alias`, we can pipe the result of `Get-Alias` to `Get-Member`.

```
Get-Alias | Get-Member
```

If there is a well-known command, such as `dir` or `ls` that is supported in PowerShell and you are curious which cmdlet it refers to, you can use the following:

```
#list cmdlet that is aliased as dir
$alias:dir
```

```
#list cmdlet that is aliased as ls
$alias:ls
```

On the other hand, if you want to know all aliases for a cmdlet, you can use the following:

```
Get-Alias -Definition "Get-ChildItem"
```

There's more...

For more information on MSDN Get-Alias, refer to:

<http://technet.microsoft.com/en-us/library/hh849948>

Exporting to CSV and XML

In this recipe, we pipe the results of the `Get-Process` cmdlet to a CSV and XML file.

How to do it...

Following are the steps to export to CSV and XML:

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
$csvFile = "C:\Temp\sample.csv"
Get-Process |
Export-Csv -path $csvFile -Force -NoTypeInfoation
notepad $csvFile
```

```
$xmlFile = "C:\Temp\process.xml"
Get-Process |
Export-Clixml -path $xmlFile -Force
notepad $xmlFile
```

How it works...

PowerShell provides a few cmdlets that support exporting data to files of different formats. `Export-Csv` saves information to a comma-separated value file, and `Export-Clixml` exports the piped data to XML.

```
$csvFile = "C:\Temp\sample.csv"
Get-Process |
Export-Csv -Path $csvFile -Force -NoTypeInfo
notepad $csvFile
```

```
$xmlFile = "C:\Temp\process.xml"
Get-Process |
Export-Clixml -Path $xmlFile -Force
notepad $xmlFile
```

The `Export-Csv` cmdlet converts each object passed to it from the pipeline into a row in the resulting CSV file. Although the default delimiter is a comma, this can be changed to other characters by using the `-Delimiter` switch. You can also start appending data using the `-Append` switch, which was added in PowerShell V3.

The `Export-Clixml` cmdlet converts data passed to it into XML and saves it to a file. The resulting XML is similar to what the `ConvertTo-Xml` cmdlet would return.

There's more...

- ▶ Refer to MSDN `Export-Csv`, available at:
<http://msdn.microsoft.com/en-us/library/hh849932>
- ▶ Refer to MSDN `Export-Clixml`, available at:
<http://msdn.microsoft.com/en-us/library/hh849916>

Using Invoke-Expression

In this recipe, we will use the `Invoke-Expression` cmdlet.

Getting ready

For this recipe, we will use the 7-zip application to compress some files. Download 7-zip from <http://www.7-zip.org/>.

How to do it...

Let's check out the `Invoke-Expression` cmdlet.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
$VerbosePreference = "Continue"

$program = "`"C:\Program Files\7-Zip\7z.exe`"
$7zargs = " a -tzip "
$zipFile = " `C:\Temp\new archive.zip`" "
$directoryToZip = " `C:\Temp\old`" "

$cmd = "& $program $7zargs $zipFile $directoryToZip "

#display final command
Write-Verbose $cmd

Invoke-Expression $cmd

$VerbosePreference = "SilentlyContinue"
```

How it works...

The `Invoke-Expression` cmdlet allows PowerShell expressions to be run from PowerShell. These expressions can consist of other PowerShell statements and functions, or they can contain executables and arguments.

In this recipe, we are composing the command to run `7z.exe` and pass it the name of a folder, which needs to be compressed into a ZIP file.

The challenge faced most often with using `Invoke-Expression` is making sure that the full path of the program, or the full arguments, are all properly escaped. In our recipe, we individually compose the strings for the executable and the arguments. All the strings are escaped with a backtick.

```
$program = "`"C:\Program Files\7-Zip\7z.exe`"
$7zargs = " a -tzip "
$zipFile = " `C:\Temp\new archive.zip`" "
$directoryToZip = " `C:\Temp\old`" "
```

Helpful PowerShell Snippets

```
$cmd = "& $program $7zargs $zipFile $directoryToZip "  
#display final command  
Write-Verbose $cmd
```

When we display the command, we will see that the double quotes are preserved:

```
VERBOSE: & "C:\Program Files\7-Zip\7z.exe" a -tzip "C:\Temp\new archive.zip" "C:\Temp\old"
```

The preceding ampersand is considered as a call operator, and this whole expression is meant to run the `7z.exe` application and compress the `C:\Temp\old` folder into a file called `new archive.zip`.

Finally, running the expression requires using the `Invoke-Expression` cmdlet, and passing the string command argument:

```
Invoke-Expression $cmd
```

There's more...

- ▶ Refer to MSDN `Invoke-Expression`, available at:
<http://msdn.microsoft.com/en-us/library/hh849893>

Testing regular expressions

In this recipe we are going to explore some ways to use and test regular expressions.

How to do it...

Let's check out regular expressions in PowerShell.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
$VerbosePreference = "Continue"  
  
#check if valid email address  
$str = "belle@sqlmusings.com"  
$pattern = "^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.(?:[A-Z]{2}|com|org|net|gov|ca|mil|biz|info|mobi|name|aero|jobs|museum)$"  
if ($str -match $pattern)  
{
```

```

        Write-Verbose "Valid Email Address"
    }
    else
    {
        Write-Verbose "Invalid Email Address"
    }

#another way to test
[Regex]::Match($str, $pattern)

#can also use regex in switch
$str = "V1A 2V1"
$str = "90250"
switch -regex ($str)
{
    "(^\d{5}$) | (^\d{5}-\d{4}$)"
    {
        Write-Verbose "Valid US Postal Code"
    }
    "[A-Za-z]\d[A-Za-z]\s*\d[A-Za-z]\d"
    {
        Write-Verbose "Valid Canadian Postal Code"
    }
    default
    {
        Write-Verbose "Don't Know"
    }
}

#use regex and extract matches
#to create named groups - use format ?<groupname>
$str = "Her number is (604)100-1004. Sometimes she can be reached
at (604)100-1005."
$pattern = @"
(?<phone>\(\d{3}\)\d{3}-\d{4})
"@

$m = [regex]::Matches($str, $pattern)

#list individual phones
$m | Foreach {
    Write-Verbose "$($_.Groups["phone"].Value)"
}

$VerbosePreference = "SilentlyContinue"

```

How it works...

We have looked at a few ways to use and test regular expressions in this recipe.

A **regular expression** is a string pattern—for example, a pattern for a valid ZIP code, or an e-mail address, that can be used to compare strings.

Here are some of the common patterns:

Pattern	Description
\	Escape character
^	Beginning of line
\$	End of line
*	Matches zero or many times
?	Matches zero or one time
+	Matches one or more times
.	Matches a single character except newline
pattern1 pattern2	Matches either of the patterns
pattern{m}	Matches a pattern exactly m times
pattern{m, n}	Matches minimum m to a maximum n times
pattern{m, }	Matches minimum m times
[abcd]	Matches any character in a set
[a-d]	Matches any character in a range
[^abcd]	Matches characters NOT in a set
\n	Newline
\r	Carriage return
\b	Word boundary
\B	Non-word boundary
\d	Digits: 0-9
\D	Non-digit
\w	Word character; equivalent to [A-Za-z0-9_]
\W	Non-word character
\s	Space character
\S	Non whitespace character

PowerShell has the `-match` and `-replace` operators that allow strings to be matched or replaced against a pattern. PowerShell also supports the static methods of the `Regex` class, such as `[regex]::Match`, and `[regex]::Matches`.

In the first example, we will check for a valid e-mail address and we will use the `-match` operator.

```
#check if valid e-mail address
$str = "belle@sqlmusings.com"
$pattern = "^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.(?:[A-Z]{2}|com|org|net|gov|ca|mil|biz|info|mobi|name|aero|jobs|museum)$"
if ($str -match $pattern)
{
    Write-Verbose "Valid Email Address"
}
else
{
    Write-Verbose "Invalid Email Address"
}
```

Regular expressions can also be used in a `switch` statement. In our example, we were checking if our string is either a valid US or Canadian postal code:

```
#can also use regex in switch
$str = "V1A 2V1";
switch -regex ($str)
{
    " (^d{5}$) | (^d{5}-d{4}$) "
    {
        Write-Verbose "Valid US Postal Code"
    }
    "[A-Za-z]d[A-Za-z]s*d[A-Za-z]d"
    {
        Write-Verbose "Valid Canadian Postal Code"
    }
    default
    {
        Write-Verbose "Don't Know"
    }
}
```

If there is a possibility of multiple matches, we can use the `[regex]::Matches` operator, and pipe the result to a `Foreach` cmdlet to display the group matches.

```
#use regex and extract matches
#to create named groups - use format ?<groupname>
$str = "Her number is (604)100-1004. Sometimes she can be reached at (604)100-1005."
$pattern = @"
(?<phone>\(d{3}\)d{3}-d{4})
"@
```



```
"@

$m = [regex]::Matches($str, $pattern)

#list individual phones
$m | Foreach {
    Write-Verbose "$($_.Groups["phone"].Value) "
}
```

The pattern we are using is a named group, specified by the (?<phone>) label. Anything that is matched by the pattern in the parenthesis can later be referred to by the label **phone**.

There's more...

- ▶ For more information on regex methods, visit:
<http://msdn.microsoft.com/en-us/library/axa83z9t>
- ▶ Refer to Regular Expression Language – Quick Reference, available at:
<http://msdn.microsoft.com/en-us/library/az24scfc.aspx>
- ▶ Refer to the PowerShell Admin Regex article, available at:
http://www.powershelladmin.com/wiki/Powershell_regular_expressions

Managing folders

In this recipe, we will explore different cmdlets that support folder management.

How to do it...

Let's take a look at different cmdlets that can be used for folders.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
#list folders ordered by name descending
$path = "C:\Temp"

#get directories only
Get-Childitem $path | Where PSIsContainer

#create folder
```

```

$newFolder = "C:\Temp\NewFolder"
New-Item -Path $newFolder -ItemType Directory -Force

#check if folder exists
Test-Path $newFolder

#copy folder
$anotherFolder = "C:\Temp\NewFolder2"
Copy-Item $newFolder $anotherFolder -Force

#move folder
Move-Item $anotherFolder $newFolder

#delete folder
Remove-Item $newFolder -Force -Recurse

```

How it works...

Here are some cmdlets that support folder manipulation:

Cmdlet	Description
Get-ChildItem	Lists all directories in a path #get directories only Get-Childitem \$path Where PSIsContainer
Test-Path	Checks if a folder exists Test-Path \$newFolder
New-Item	Creates a new folder PS> NewItem -Path \$newFolder -ItemType Directory -Force
Copy-Item	Copies a folder Copy-Item \$newFolder \$anotherFolder -Force
Move-Item	Moves a folder to a different location Move-Item \$anotherFolder \$newFolder
Remove-Item	Deletes a folder and all its contents Remove-Item \$newFolder -Force -Recurse

There's more...

Refer to the following links provided to gain a better understanding on folder management using cmdlets:

- ▶ Files and Folders, Part I (TechNet, by the Microsoft Scripting Guys):
<http://technet.microsoft.com/en-us/library/ee176983>
- ▶ Files and Folders, Part II (TechNet, by the Microsoft Scripting Guys):
<http://technet.microsoft.com/en-us/library/ee176985>
- ▶ Files and Folders, Part III (TechNet, by the Microsoft Scripting Guys):
<http://technet.microsoft.com/en-us/library/ee176988>

See also

- ▶ *The Manipulating files recipe*

Manipulating files

In this recipe, we will look at different cmdlets that help to manipulate files.

How to do it...

Let's explore different ways to manage files.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
#create file
$timestamp = Get-Date -format "yyy-MM-dd-hhmmtt"
$path = "C:\Temp\"
$filename = "$timestamp.txt"
$fullpath = Join-Path $path $filename

New-Item -Path $path -Name $filename -ItemType "File"

#check if file exists
Test-Path $fullpath

#copy file
$path = "C:\Temp\"
```

```

$newfilename = $timestamp + "_2.txt"
$fullpath2 = Join-Path $path $newfilename

Copy-Item $fullpath $fullpath2

#move file
$newfolder = "C:\Data"
Move-Item $fullpath2 $newfolder

#append to file
Add-Content $fullpath "Additional Item"
notepad $fullpath

#merge file contents
$newcontent = Get-Content "C:\Temp\processes.txt"
Add-Content $fullpath $newcontent
notepad $fullpath

#delete file
Remove-Item $fullpath

```

How it works...

Here are some of the cmdlets that support file manipulation:

Cmdlet	Description
Test-Path	Checks if a file exists Test-Path \$fullpath
Join-Path	Combines a path and a child path Join-Path \$path \$filename
New-Item	Creates a new file New-Item -Path \$path -Name \$filename -ItemType "File"
Get-Content	Retrieves the content of a file Get-Content "C:\Temp\processes.txt"
Add-Content	Appends content to a file Add-Content \$fullpath \$newcontent
Copy-Item	Copies a file Copy-Item \$fullpath \$fullpath2

Cmdlet	Description
Move-Item	Moves a file to a different location Move-Item \$fullpath2 \$newfolder
Remove-Item	Deletes a file Remove-Item \$fullpath

There's more...

Refer to the following for more information on file manipulation:

- ▶ Files and Folders, Part I (Technet, by the Microsoft Scripting Guys):
<http://technet.microsoft.com/en-us/library/ee176983>
- ▶ Files and Folders, Part II (Technet, by the Microsoft Scripting Guys):
<http://technet.microsoft.com/en-us/library/ee176985>
- ▶ Files and Folders, Part III (Technet, by the Microsoft Scripting Guys):
<http://technet.microsoft.com/en-us/library/ee176988>

See also

- ▶ *Managing folders*

Searching for files

In this recipe, we will search for files based on filenames, attributes, and content.

How to do it...

Let's explore different ways to use `Get-ChildItem` to search for files.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
#search for file with specific extension
$path = "C:\Temp"
Get-ChildItem -Path $path -Include *.sql -Recurse

#search for file based on date creation
#use LastWriteTime for date modification
```

```

[datetime]$startDate = "2012-05-01"
[datetime]$endDate = "2012-05-20"
#note date is at 12 midnight
#sample date Sunday, May 20, 2012 12:00:00 AM

#PowerShell V3 Where-Object syntax
Get-ChildItem -Path $path -Recurse |
Where CreationTime -ge $startDate |
Where CreationTime -le $endDate |
Sort -Property LastWriteTime

#list files greater than 10MB
#PowerShell V3 syntax
Get-ChildItem $path -Recurse |
Where Length -ge 10Mb |
Select Name,
@{Name="MB";Expression="{0:N2}" -f ($_.Length/1MB)}} |
Sort -Property Length -Descending |
Format-Table -AutoSize

#search for content of file
#search TXT, CSV and SQL files that contain
#the word "QueryWorks"
$pattern = "QueryWorks"
Get-ChildItem -Path $path -Include *.txt, *.csv, *.sql -Recurse |
Select-String -Pattern $pattern

```

How it works...

The `Get-ChildItem` cmdlet displays contents of a given path:

Get-ChildItem

You can also use the aliases `gci`, `ls`, or `dir` instead of `Get-ChildItem` when typing this command.

We can pipe the results of `Get-ChildItem` to a `Where` cmdlet to filter the results. For example, if we wanted to look for only `.sql` files, we would use:

```

#search for file with specific extension
$path = "C:\Temp"
Get-ChildItem -Path $path -Include *.sql -Recurse

```

To get files created within a date range, we pipe the results, and in the `Where-Object` cmdlet, we filter based on the `CreationTime` property. Note that dates are automatically assigned a timestamp of midnight, and the following example actually gets all files created between May 1 and May 19:

```
#search for file based on creation date
#use LastWriteTime for modification date
[datetime]$startDate = "2012-05-01"
[datetime]$endDate = "2012-05-20"
#note date is at 12 midnight
#sample date Sunday, May 20, 2012 12:00:00 AM

#PowerShell V3
Get-ChildItem -Path $path -Recurse |
Where CreationTime -ge $startDate |
Where CreationTime -le $endDate |
Sort -Property LastWriteTime
```

To retrieve the same files in PowerShell V2, we can use the `Where-Object` syntax:

```
#PowerShell V2
Get-ChildItem -Path $path -Recurse |
Where {$_.CreationTime -ge $startDate -and $_.CreationTime -le
$endDate} |
Sort -Property LastWriteTime
```

To filter files based on file size, we can filter the files using the `Length` property. Note that PowerShell supports the constants KB (kilobyte), MB (megabyte), GB (gigabyte), TB (terabyte), and PB (petabyte):

```
#list files greater than 10MB
#PowerShell V3 syntax
Get-ChildItem $path -Recurse |
Where Length -ge 10Mb |
Select Name,
@{Name="MB";Expression="{0:N2}" -f ($_.Length/1MB)}} |
Sort -Property Length -Descending |
Format-Table -AutoSize
```

The last example showcases the use of the `-Include` switch with the `Get-ChildItem` cmdlet, which allows the cmdlet to selectively include only specific files based on the pattern that was passed. This example also highlights how we can search not only filenames and paths, but the actual contents of the file using the `Select-String` cmdlet. The `Select-String` cmdlet can only search for text files, however; it cannot search other proprietary formats such as `.doc`, `.docx`, and `.pdf`.

```
#search for content of file
#search TXT, CSV and SQL files that contain
#the word "QueryWorks"
$pattern = "QueryWorks"
Get-ChildItem -Path $path -Include *.txt, *.csv, *.sql -Recurse |
Select-String -Pattern $pattern
```

There's more...

- ▶ Refer to MSDN Get-ChildItem, available at:
<http://msdn.microsoft.com/en-us/library/hh849800>
- ▶ Refer to MSDN Select-String, available at:
<http://msdn.microsoft.com/en-us/library/hh849903>

See also

- ▶ *Getting aliases*

Reading an event log

In this recipe, we will read the event log.

How to do it...

Let's see how we can read the Windows event log from PowerShell.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
Get-EventLog -LogName Application -Newest 20 -EntryType Error
```

How it works...

Reading the event log is straightforward in PowerShell. We can do this using the `Get-EventLog` cmdlet. This cmdlet accepts a few switches, which includes `LogName` and `EntryType`.

```
Get-EventLog -LogName Application -Newest 20 -EntryType Error
```


Some of the possible `LogName` values are as follows:

- ▶ `Application`
- ▶ `HardwareEvents`
- ▶ `Internet Explorer`
- ▶ `Security`
- ▶ `System`
- ▶ `Windows PowerShell`

You can alternatively pass it the name of a custom log available in your system.

The `EntryType` switch can be of the following types:

- ▶ `Error`
- ▶ `FailureAudit`
- ▶ `Information`
- ▶ `SuccessAudit`
- ▶ `Warning`

In our recipe, we also use the `-Newest` switch, to filter only for the newest 20 error events.

There's more...

Refer to MSDN `Get-EventLog`, available at:

<http://msdn.microsoft.com/en-us/library/hh849834>

Sending e-mail

In this recipe, we send an e-mail with an attachment.

Getting ready

Before proceeding, identify the following in your environment:

- ▶ SMTP server
- ▶ Recipient's e-mail address
- ▶ Sender's e-mail address
- ▶ Attachment

How to do it...

The following are the steps to send an e-mail:

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
$file = "C:\Temp\processes.csv"
$timestamp = Get-Date -format "yyyy-MMM-dd-hhmm"

#note we are using backticks to put each parameter
#in its own line to make code more readable
Send-MailMessage `
  -SmtpServer "queryworks.local" `
  -To "administrator@queryworks.local" `
  -From "powershell@gaia.local" `
  -Subject "Process Email - $file - $timestamp" `
  -Body "Your requested file is attached." `
  -Attachments $file
```

How it works...

One way to send an e-mail using PowerShell is by using the `Send-MailMessage` cmdlet. Some of the switches it accepts are as follows:

- ▶ `-SmtpServer`
- ▶ `-To`
- ▶ `-Cc`
- ▶ `-Bcc`
- ▶ `-Credential`
- ▶ `-From`
- ▶ `-Subject`
- ▶ `-Body`
- ▶ `-Attachments`
- ▶ `-UseSsl`

There's more...

Refer to MSDN Send-MailMessage, available at:

<http://msdn.microsoft.com/en-us/library/hh849925.aspx>

Embedding C# code

In this recipe, we will embed and execute C# code in our PowerShell script.

How to do it...

Let's explore how to embed C# code in PowerShell.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
#define code
#note this can also come from a file

$code = @"
using System;
public class HelloWorld
{
    public static string SayHello(string name)
    {
        return (String.Format("Hello there {0}", name));
    }
    public string GetLuckyNumber(string name)
    {
        Random random = new Random();
        int randomNumber = random.Next(0, 100);
        string message = String.Format("{0}, your lucky" +
            " number for today is {1}",
            name, randomNumber);
        return message;
    }
}
"@

#add this code to current session
Add-Type -TypeDefinition $code
```

```
#call static method
[HelloWorld]::SayHello("belle")

#create instance
$instance = New-Object HelloWorld

#call instance method
$instance.GetLuckyNumber("belle")
```

How it works...

We can use C# code from within PowerShell. This will require constructing a class in a here-string and adding that class as a type to the session using the `Add-Type` cmdlet. The `Add-Type` cmdlet allows the construction of the class in the session, or to all sessions if created within the PowerShell profile.

In the recipe, we use a very simple class defined in a here-string:

```
$code = @"
using System;
public class HelloWorld
{
    public static string SayHello(string name)
    {
        return (String.Format("Hello there {0}", name));
    }
    public string GetLuckyNumber(string name)
    {
        Random random = new Random();
        int randomNumber = random.Next(0, 100);
        string message = String.Format("{0}, your lucky" +
            " number for today is {1}",
            name, randomNumber);
        return message;
    }
}
"@
```

This code does not have to be built and hardcoded within the script. It can be read from another file using the `Get-Content` cmdlet and stored into the `$code` variable.

To put this class in effect in the current session, we use the `Add-Type` cmdlet:

```
#add this code to current session
Add-Type -TypeDefinition $code
```

Note that this class has both a static and non-static method. To call the static method, we must use the class name:

```
#call static method
[HelloWorld]::SayHello("belle")
```

To call the non-static method, we must instantiate an object first, and then call the method using the object:

```
#call instance method
$instance.GetLuckyNumber("belle")
```

There's more...

Refer to MSDN Add-Type, available at:

<http://msdn.microsoft.com/en-us/library/hh849914>

Creating an HTML report

In this recipe, we will create an HTML report based on the system's services.

How to do it...

This is a sample of how we can create an HTML report using PowerShell.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
#simple CSS Style
$style = @"
<style type='text/css'>
  td {border:1px solid gray;}
  .stopped{background-color: #E01B1B;}
</style>
"@

#let's get content from Get-Service
#and output this to styled HTML
Get-Service |
ConvertTo-Html -Property Name, Status -Head $style |
Foreach {
  #if service is running, use green background
```

```

    if ($_ -like "*<td>Stopped</td>*")
    {
        $_ -replace "<tr>", "<tr class='stopped'>"
    }
    else
    {
        #display normally
        $_
    }
} |
Out-File "C:\Temp\sample.html" -force

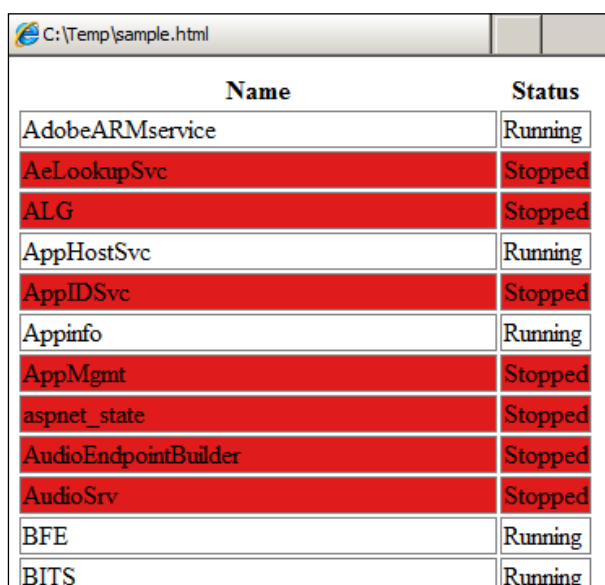
```

```

Set-Alias ie "$env:programfiles\Internet Explorer\iexplore.exe"
ie "C:\Temp\sample.html"

```

The following screenshot shows a sample result:



Name	Status
AdobeARMservice	Running
AeLookupSvc	Stopped
ALG	Stopped
AppHostSvc	Running
AppIDSvc	Stopped
Appinfo	Running
AppMgmt	Stopped
aspnet_state	Stopped
AudioEndpointBuilder	Stopped
AudioSrv	Stopped
BFE	Running
BITS	Running

How it works...

In this recipe, we piped the result of the `Get-Service` cmdlet, which returns all services, into the `ConvertTo-HTML` cmdlet. The `ConvertTo-HTML` cmdlet formats the results as HTML. This cmdlet also allows you to configure what goes into an HTML `<head>` tag. This is where you typically add your CSS styles and JavaScript.

Once the file has been created, we set an alias to Internet Explorer and just display the resulting HTML file in the browser.

There's more...

Refer to MSDN ConvertTo-HTML, available at:

<http://msdn.microsoft.com/en-us/library/hh849944>

Parsing XML

In this recipe, we will parse a sample XML document using PowerShell.

Getting ready

In this recipe, we will use Vancouver's 2012 daily weather data, which can be downloaded from the following URL:

http://www.climate.weatheroffice.gc.ca/climateData/dailydata_e.html?Prov=BC&StationID=889&Year=2012&Month=4&Day=30&timeframe=2

How to do it...

Let's look at how we can parse XML files.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
$vancouverXML = "C:\XML Files\eng-daily-01012012-12312012.xml"
$xml = Get-Content $vancouverXML

#get number of entries
$xml.climatedata.stationdata.Count

#store max temps in array
$maxtemp = $xml.climatedata.stationdata |
Foreach { [int]$_ .maxtemp."#text" }

#list all daily max temperatures
$maxtemp | Sort -Descending

#get max temperature recorded in 2012
$maxtemp | Sort -Descending | Select -First 1
```

How it works...

One of the key things to do, when working with XML data, is to make sure the data is stored as an XML object. In our recipe, we get the contents of the file using `Get-Content`, and store it in the strongly typed variable, `$xml`. We know it is strongly typed because we have placed the `[xml]` data type right at the variable declaration:

```
$vancouverXML = "C:\XML Files\eng-daily-01012012-12312012.xml"
$xml = Get-Content $vancouverXML
```

The following screenshot is an example of how the file is formatted:



To know how many records are in the file, we can traverse the `stationdata` nodes and count the records:

```
#get number of entries
$xml.climatedata.stationdata.Count
```

To manipulate the `maxtemp` data, we can loop through all the nodes and extract the values into an array:

```
#store max temps in array
$maxtemp = $xml.climatedata.stationdata |
Foreach { [int]$_ .maxtemp."#text" }
```


When the data is in the array, we can further manipulate it. For example, we can now more easily sort as needed, or get the overall maximum value if required:

```
#list all daily max temperatures
$maxtemp | Sort -Descending

#get max temperature recorded in 2012
$maxtemp | Sort -Descending | Select -First 1
```

Extracting data from a web service

In this recipe, we will extract data from a free, public web service.

How to do it...

Let's explore how to access and retrieve data from a web service.

1. Open **PowerShell ISE**. Go to **Start | Accessories | Windows PowerShell | Windows PowerShell ISE**.
2. Add the following script and run it:

```
#delayed stock quote URI
$stockUri = "http://ws.cdyne.com/delayedstockquote/
delayedstockquote.asmx"

$stockproxy = New-WebServiceProxy -Uri $stockUri
-UseDefaultCredential

#get quote
$stockresult = $stockProxy.GetQuote("MSFT", "")

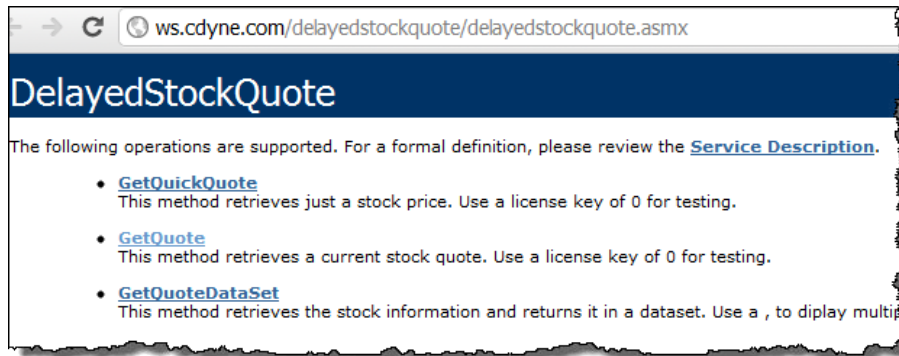
#display results
$stockresult.StockSymbol
$stockresult.DayHigh
$stockresult.DayLow
$stockresult.LastTradeDateTime
```

How it works...

To work with a web service, we first need to create a proxy object that will allow us to access the methods available from a web service. We can achieve this by using the `New-WebProxy` cmdlet, which accepts the web service URL.

```
$stockUri = "http://ws.cdyne.com/delayedstockquote/delayedstockquote.
asmx"
```

This URI points to a free web service that provides delayed stock quote values. If we go to this URI from the browser, the following screenshot is what we are going to see:



We can see that this web service has a method called `GetQuote`, which retrieves the current stock quote. This accepts a stock symbol and a license key. In our script, we call this method through our proxy object:

```
#get quote
$stockresult = $stockProxy.GetQuote("MSFT", "")
```

If we were to plug these values into the browser, the following screenshot is a sample result that we might get:



To display these in our script, we simply need to know how to traverse the nodes from the root to the values we want to display. In our case, we wanted to display `StockSymbol`, `DayHigh`, `DayLow`, and `LastTradeDateTime`.

```
#display results
$stockresult.StockSymbol
$stockresult.DayHigh
$stockresult.DayLow
$stockresult.LastTradeDateTime
```

There's more...

Refer to MSDN `New-WebServiceProxy`, available at:

<http://msdn.microsoft.com/en-us/library/hh849841>

Using PowerShell Remoting

In this recipe, we will use PowerShell Remoting to execute commands on a remote machine.

Getting ready

We first need to identify which remote machine we want to use. In our recipe, we will connect to a remote machine called `ZERATULDC` from our client machine `KERRIGAN`. These two machines are in the same domain.

Log in to `ZERATULDC`, or to a machine you want to use for remoting. We need to enable PowerShell Remoting. Check out the system and permission requirements for running PowerShell Remoting from MSDN *about_Remote_Requirements*, available at <http://msdn.microsoft.com/en-us/library/hh847859.aspx>.

To turn on remoting, open up the PowerShell console using elevated privileges. Right-click on the PowerShell console and go to **Run as Administrator**. Execute the following command:

```
PS> Enable-PSRemoting
```

You will be prompted to confirm a couple of times. Answer **A** (or **Yes to All**) to these questions. Your screen should look similar to the following screenshot:

```

Administrator: Windows PowerShell
PS C:\Users\Administrator> Enable-PSRemoting

WinRM Quick Configuration
Running command "Set-WSManQuickConfig" to enable this machine for remote management through WinRM service.
This includes:
  1. Starting or restarting (if already started) the WinRM service
  2. Setting the WinRM service type to auto start
  3. Creating a listener to accept requests on any IP address
  4. Enabling firewall exception for WS-Management traffic (for http only).

Do you want to continue?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): A
WinRM already is set up to receive requests on this machine.
WinRM already is set up for remote management on this machine.

Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name: microsoft.powershell SDDL:
O:NSG:BAD:P(A;;GA;;;BA)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will allow selected users to remotely run Windows
PowerShell commands on this computer".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): A
PS C:\Users\Administrator>

```

We also need to add our remote computer ZERATULDC as a trusted host. Open a PowerShell console as administrator from KERRIGAN and run the following:

```
Set-Item wsman:localhost\client\trustedhosts -value ZERATULDC
```

How to do it...

Let's explore how to use PowerShell Remoting to execute commands on a remote machine.

1. Open a PowerShell console as administrator from KERRIGAN. Right-click on the PowerShell console icon, and select **Run as administrator**.
2. Let's execute a remote command first.

```
Invoke-Command -ComputerName ZERATULDC -Credential "QUERYWORKS\
Administrator" -ScriptBlock {
    Get-Wmiobject win32_computersystem
}
```

3. Next, let's start an interactive remoting session to ZERATULDC. We will provide our credentials to the machine by specifying the `-Credential` parameter.

```
Enter-PSSession -ComputerName ZERATULDC -Credential "QUERYWORKS\
Administrator"
```

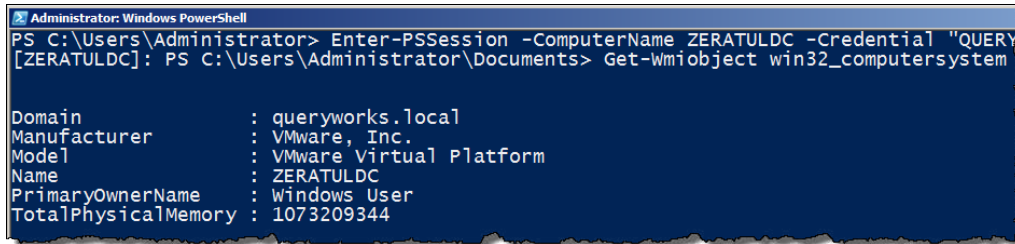
Note that as soon as we are authenticated, the prompt changes to indicate we are now in ZERATULDC. This is shown in the following screenshot:

```
[ZERATULDC]: PS C:\Users\Administrator\Documents>
```

- Let's execute a simple command in our remoting session. Execute the following:

```
Get-Wmiobject win32_computersystem
```

You should see a result similar to the following screenshot. Note the prompt still displays ZERATULDC.



```
Administrator: Windows PowerShell
PS C:\Users\Administrator> Enter-PSSession -ComputerName ZERATULDC -Credential "QUERYWORKS\
[ZERATULDC]: PS C:\Users\Administrator\Documents> Get-Wmiobject win32_computersystem

Domain           : queryworks.local
Manufacturer     : VMware, Inc.
Model            : VMware Virtual Platform
Name             : ZERATULDC
PrimaryOwnerName : windows User
TotalPhysicalMemory : 1073209344
```

- Exit out of the session by typing `exit`.

How it works...

PowerShell Remoting allows you to connect and execute PowerShell commands on remote machines. PowerShell Remoting uses **Web Services for Management (WSMan)** to communicate to a remote machine, and **Windows Remote Management (WinRM)** service on the remote machine to listen for incoming WSMan requests.

There are different ways to execute remote commands. We can use the `Invoke-Command` cmdlet to establish a remote connection, execute our command(s) and get our results, and disconnect. The command(s) we want to execute can either be placed in the `-ScriptBlock` parameter, or in a file specified with the `-FilePath` parameter. In our recipe we used `-ScriptBlock`.

```
Invoke-Command -ComputerName ZERATULDC -Credential "QUERYWORKS\
Administrator" -Authentication Negotiate -ScriptBlock {
    Get-Wmiobject win32_computersystem
}
```

We have also chosen to provide our credentials to ZERATULDC by specifying the `-Credential` parameter. You can choose to prompt for both username and password by using the `Get-Credential` cmdlet, and passing this to the `Invoke-Command` cmdlet.

```
$cred = Get-Credential
```

Another way to execute a remote command is by establishing an interactive session to a remote machine. We do this by using the `Enter-PSSession` cmdlet:

```
Enter-PSSession -ComputerName ZERATULDC -Credential "QUERYWORKS\  
Administrator" -Authentication Negotiate
```

Once the remoting interactive session is started, you will notice that the PowerShell prompt changes to show the remote computer's name. We can then start executing commands in this session.

What we have shown in this recipe is just a very brief example of how you can use PowerShell Remoting. To learn more about PowerShell Remoting, including system and permission requirements, how to set up HTTPS, and so on, be sure to check the recommended additional resources in the *There's more...* section.

There's more...

Check the following resources for additional information on remoting:

- ▶ MSDN Remoting Requirements:
<http://msdn.microsoft.com/en-us/library/hh847859.aspx>
- ▶ Layman's Guide to PowerShell 2.0 Remoting by Ravikanth Chaganti:
<http://www.ravichaganti.com/blog/?p=1305>
- ▶ An Introduction to PowerShell Remoting (5-part series):
<http://blogs.technet.com/b/heyscriptingguy/archive/2012/07/23/an-introduction-to-powershell-remoting-part-one.aspx>
<http://blogs.technet.com/b/heyscriptingguy/archive/2012/07/24/an-introduction-to-powershell-remoting-part-two-configuring-powershell-remoting.aspx>
<http://blogs.technet.com/b/heyscriptingguy/archive/2012/07/25/an-introduction-to-powershell-remoting-part-three-interactive-and-fan-out-remoting.aspx>
<http://blogs.technet.com/b/heyscriptingguy/archive/2012/07/26/an-introduction-to-powershell-remoting-part-four-sessions-and-implicit-remoting.aspx>
<http://blogs.technet.com/b/heyscriptingguy/archive/2012/07/27/an-introduction-to-powershell-remoting-part-five-constrained-powershell-endpoints.aspx>
- ▶ Secrets of PowerShell Remoting:
<http://powershellbooks.com/SecretsofPowerShellRemoting.pdf>

A

SQL Server and PowerShell CheatSheet

Learning PowerShell

- ▶ `Get-Help` lists syntax, usage, examples:
`Get-Help Restore-SqlDatabase`
`Get-Help Backup-SqlDatabase -Examples`
`Get-Help Invoke-Sqlcmd -Full`
`Get-Help Get-Process -Online`
- ▶ `Get-Command` lists cmdlets and functions:
`Get-Command -Module SQLPS`
`Get-Command -Module SQLASCMDLETS`
`Get-Command -Name "*Event*"`
- ▶ `Get-Member` lists properties and methods:
`$server | Get-Member -Name "*Version*" |`
`Select Name, MemberType`

PowerShell V2 versus V3 Where-Object syntax

PowerShell V2 uses {} and \$_:

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

PowerShell V3 is simplified:

```
$server | Get-Member |  
Where-Object MemberType -eq "Property"
```

Changing execution policy

The execution policy determines which PowerShell scripts are allowed to run:

- ▶ To get:
`Get-ExecutionPolicy`
- ▶ To set:
`Set-ExecutionPolicy RemoteSigned`

Execution policies

Execution Policy	Description
Restricted	Default execution policy PowerShell will not run any scripts
AllSigned	PowerShell will run only signed scripts
RemoteSigned	PowerShell will run signed scripts, or locally created scripts
Unrestricted	PowerShell will run any scripts, signed or not
Bypass	PowerShell will not block any scripts, and will prevent any prompts or warnings
Undefined	PowerShell will remove the set execution policy in the current user scope

Running a script

Save your PowerShell code in a file with a `.ps1` extension.

- ▶ From the PowerShell console, if the script is in the current directory and the filename does not have any spaces:

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

- ▶ From the PowerShell console, if the script is in the current directory and the filename has spaces:

```
PS C:\> & '.\My Script.ps1'
```

- ▶ From the PowerShell console, if the script is in a different directory:

```
PS C:\> & 'C:\Temp\My Script.ps1'
```

- ▶ From the PowerShell console, using dot sourcing. Dot sourcing persists variables in session:

```
PS C:\> . '.\My Script.ps1'
```

```
PS C:\> . 'C:\Temp\My Script.ps1'
```

- ▶ From the Command Prompt:

```
C:\>powershell.exe -ExecutionPolicy RemoteSigned -File "C:\PowerShell\My Script.ps1"
```

Common aliases

Command	Alias
Foreach-Object	%, Foreach
Where-Object	?, Where
Sort-Object	Sort
Compare-Object	compare, diff
Write-Output	echo, write
help	man
Get-Content	cat, gc, type
Get-ChildItem	dir, gci, ls
Copy-Item	copy, cp, cpi
Move-Item	mi, move, mv
Remove-Item	del, erase, rd, ri, rm, rmdir

Command	Alias
Get-Process	gps, ps
Stop-Process	kill, spps
Get-Location	gl, pwd
Set-Location	cd, chdir, sl
Clear-Host	clear, cls
Get-History	h, ghy, history

Displaying output

```
PS C:\> Get-Command -Name "*Write*" -CommandType Cmdlet
```

Cmdlet	Description
Write-Debug	Displays a debug message to the console Typically used with: \$DebugPreference = "Continue"
Write-Error	Displays a non-terminating error message to the console
Write-EventLog	Writes a message to Windows Event Log
Write-Host	Displays a string message to the host
Write-Output	Writes an object to the pipeline
Write-Progress	Display a progress bar
Write-Verbose	Displays a verbose message to the console Typically used with: \$VerbosePreference = "Continue"
Write-Warning	Displays a warning message to the console

Special characters

Special character	Special character name	Explanation
\$	Dollar	Variable
\$_	Dollar underscore	Current object in pipeline

Special character	Special character name	Explanation
	Pipe	Command chaining; output from one command to input to another
~	Backtick	Escape or continuation character
@	At sign	Array
#	Hash sign	Comment
[]	Square brackets	For indexes and strongly typing variables
()	Parentheses	For array members; For calling functions
&	Ampersand	Call operator
*	Star or asterisk	Wildcard
%	Percent	Alias for Foreach-Object
?	Question mark	Alias for Where-Object
+	Plus	Addition; String concatenation operator

Special variables

Special variable	Explanation
\$_	Current pipeline object
\$args	Arguments passed to a function
\$error	Array that stores all errors
\$home	User's home directory
\$host	Host information
\$match	Regex matches
\$profile	Path to profile, if available
\$PSHome	Install directory of PowerShell
\$PSISE	PowerShell Scripting Environment object
\$pid	Process ID (PID) of PowerShell process
\$pwd	Present Working Directory
\$true	Boolean true
\$false	Boolean false
\$null	Null value

Common operators

Note that many operators perform case-insensitive string comparisons by default. If you want to do case-sensitive matching, prepend with `c`. For example, `-ceq`, `-clike`, `-cnotlike`.

PowerShell	Traditional	Explanation
<code>-eq</code>	<code>==</code>	Equal to
<code>-ne</code>	<code><></code> or <code>!=</code>	Not equal to
<code>-match</code>		Match using regex; searches anywhere in the string
<code>-notmatch</code>		
<code>-contains</code>		Collection match. Does the item exist in the array or collection?
<code>-notcontains</code>		
<code>-like</code>		Wildcard match
<code>-notlike</code>		* (asterisk) for zero or more characters ? (question mark) for any single character
<code>-clike</code>		Case-sensitive wildcard match
<code>-cnotlike</code>		
<code>-not</code>	<code>!</code>	Negation
<code>-lt</code>	<code><</code>	Less than
<code>-le</code>	<code><=</code>	Less than or equal to
<code>-gt</code>	<code>></code>	Greater than
<code>-ge</code>	<code>>=</code>	Greater than or equal to
<code>-and</code>	<code>&&</code>	Logical and
<code>-or</code>	<code> </code>	Logical or
<code>-bor</code>	<code> </code>	Bitwise or
<code>-band</code>	<code>&</code>	Bitwise and
<code>-xor</code>	<code>^</code>	Exclusive or

Common date-time format strings

```
PS C:\> Get-Date -Format "yyyy-MMM-dd-hhmmtt"
```

Format pattern	Explanation
<code>tt</code>	A.M./P.M. designator
<code>ss</code>	Seconds with leading zero
<code>mm</code>	Minutes with leading zero

Format pattern	Explanation
dd	Day of the month with leading zero
dddd	Full name of the day of the week
hh	12 hour clock with leading zero
HH	24 hour clock with leading zero
MM	Numeric month with leading zero
MMM	Abbreviated month name
MMMM	Full month name
YY	Two digit year
YYYY	Four digit year

Comment based help

To enable comment-based help, put a special comment header at the top of your script, or in the first block of your function.

```
<#
.SYNOPSIS
    Creates a full database backup
.DESCRPTION
    Creates a full database backup using specified instance name and
    database name
    This will place the backup file to the default backup directory
    of the instance
.PARAMETER instanceName
    instance where database to be backed up resides
.PARAMETER databaseName
    database to be backed up
.EXAMPLE
    PS C:\PowerShell> .\Backup-Database.ps1 -instanceName "QUERYWORKS\
SQL01" -databaseName "pubs"
.EXAMPLE
    PS C:\PowerShell> .\Backup-Database.ps1 -instance "QUERYWORKS\
SQL01" -database "pubs"
.NOTES

    To get help:
    Get-Help .\Backup-Database.ps1
.LINK
    http://msdn.microsoft.com/en-us/library/hh245198.aspx
#>
```

Comments

- ▶ Single line comments start with a hash sign:
`#this is a single line comment`
- ▶ Block comments start with `<#` and end with `#>`:
`<#`
`this is a block comment`
`#>`

Here-string

A here-string is a string that often contains large blocks of text. It starts with `@"` and must end with a line that contains only `"@` (no other characters or spaces before it):

```
$query = @"
INSERT INTO SampleXML
(fileName, XMLStuff, FileExtension)
VALUES('$xmlfile', '$xml', '$fileextension')
"@
```

Common regex characters and patterns

<code>\</code>	Escape character
<code>^</code>	Beginning of line
<code>\$</code>	End of line
<code>*</code>	Matches zero or many times
<code>?</code>	Matches zero or one time
<code>+</code>	Matches one or more times
<code>.</code>	Matches a single character except newline
<code>pattern1 pattern2</code>	Matches either pattern
<code>pattern{m}</code>	Matches pattern exactly m times
<code>pattern{m,n}</code>	Matches minimum m to a maximum n times
<code>pattern{m, }</code>	Matches minimum m times
<code>[abcd]</code>	Matches any character in set
<code>[a-d]</code>	Matches any character in range

[^abcd]	Matches characters NOT in set
\n	Newline
\r	Carriage Return
\b	Word boundary
\B	Non word boundary
\d	Digit; 0-9
\D	Non digit
\w	Word character; equivalent to [A-Za-z0-9_]
\W	Non word character
\s	Space character
\S	Non white space character

Arrays and hash tables

An array is a collection of items:

```
#simple array
$simplearray = @(1,2,3,4)
$simplearray.Count

#array of processes consuming >30% CPU
$processes = (Get-Process | Where CPU -gt 30)
```

A hash table is a collection of key-value pairs. It is also referred to as an associative array:

```
#simple hash
$simplehash = @{
  "BCIT" = "BC Institute of Technology"
  "CST" = "Computer Systems Technology"
  "CIT" = "Computer Information Technology"
}
$simplehash.Count

#hash containing process IDs and names

$hash = @{}
Get-Process | Foreach {$hash.Add($_.Id, $_.Name)}
$hash.GetType()
```


Arrays and loops

- ▶ The while loop repeats a block while the condition evaluates to true:

```
$command = ""
while($command.ToLower() -NotMatch "quit" -and $command.ToLower()
-NotMatch "q")
{
    $command = Read-Host "Enter your command >"
}
```

- ▶ The for loop repeats for a predefined number of iterations:

```
for($counter = 0; $counter -lt 10; $counter++)
{
    Write-Verbose "Processing item $counter"
}
```

- ▶ The Foreach loop repeats for all items in the collection:

```
$processes = (Get-Process | Where CPU -gt 30)
foreach($process in $processes)
{
    Write-Verbose "$($process.ProcessName) : $($process.CPU) "
}
```

Note that this Foreach loop can be rewritten as:

```
Get-Process | Where CPU -gt 30 |
Foreach {
    Write-Verbose "$($_.ProcessName) : $_.CPU "
}
```

Logic

- ▶ If/Elseif/Else:

```
$course = "COMP4677"
if ($course -eq "COMP4677") {
    "SQL Server Administration"
}
elseif ($course -eq "COMP4678") {
    "SQL Server Development"
}
```

```

else {
    "Don't know"
}

```

► **Switch:**

```

switch ($course)
{
    "COMP4677" { "SQL Server Administration" }
    "COMP4678" { "SQL Server Development" }
    Default   { "Don't Know" }
}

```

Functions

Functions are a block or blocks of code that are encapsulated into a construct that has a name, can be reused, and can be called with parameters.

- **Sample function:**

```

function Get-SQLErrorLogs
{
    param
    (
        [Parameter(Position=0,Mandatory=$true)]
        [alias("instance")]
        [string]$instanceName
    )
    Import-Module SQLPS -DisableNameChecking | Out-Null

    #replace this with your instance name
    $server = New-Object -TypeName Microsoft.SqlServer.Management.
    Smo.Server -ArgumentList $instanceName
    Write-Output $server.ReadErrorLog()
}

```

- **Sample function call:**

```

Get-SQLErrorLogs -instanceName "KERRIGAN" |
Where LogDate -gt "2012-09-01"

```

Common Cmdlets

Utility	ConvertFrom-Csv
	ConvertFrom-Json
	ConvertTo-Csv
	ConvertTo-Html
	ConvertTo-Json
	ConvertTo-Xml
	Export-Clixml
	Export-Csv
	Format-List
	Format-Table
	Get-Alias
	Get-Date
	Get-Member
	Import-Clixml
	Import-Csv
	Read-Host
	Management
Get-Content	
Get-EventLog	
Get-HotFix	
Get-Process	
Get-Service	
Get-WmiObject	
New-WebServiceProxy	
Start-Process	
Start-Service	
Security	ConvertFrom-SecureString
	ConvertTo-SecureString
	Get-Credential
	Get-ExecutionPolicy
	Set-ExecutionPolicy

Import SQLPS module

Introduced in PowerShell V2:

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

Add SQL Server Snapins

Introduced in PowerShell V1, it can be used with SQL Server 2008/R2:

- ▶ SQL Server Snapins to load:

```
SQLServerCmdletSnapin100  
SqlServerProviderSnapin100
```

- ▶ To load:

```
if (!(Get-PSSnapin -Name SQLServerCmdletSnapin100 -ErrorAction  
SilentlyContinue))  
{  
    Add-PSSnapin SQLServerCmdletSnapin100  
}
```

Add SQL Server Assemblies

Need to be loaded for PowerShell V1 to work with SQL Server from the PowerShell prompt:

- ▶ Common SQL-related Assemblies:

```
Microsoft.SqlServer.Smo  
Microsoft.SqlServer.SmoExtended  
Microsoft.SqlServer.SqlEnum  
Microsoft.SqlServer.SmoEnum  
Microsoft.SqlServer.ConnectionInfo
```

- ▶ To load (if needed):

```
Add-Type -Assembly "Microsoft.SqlServer.Smo, Version=11.0.0.0,  
Culture=neutral, PublicKeyToken=89845dcd8080cc91"
```

Getting credentials

- ▶ Interactive, prompts user for the username and password:

```
$credential = Get-Credential
```

- ▶ Non interactive, password saved as SecureString into the file and read back and passed to Get-Credential:

```
#create file
Read-Host -AsSecureString | ConvertFrom-SecureString |
Out-File "C:\password.txt" -Force

#read back credentials
$pw = (Get-Content "C:\password.txt") |
ConvertTo-SecureString
$username = "QUERYWORKS\Administrator"
$credential = New-Object System.Management.Automation.PSCredential
$username, $pw
```

Running and blocking SQL Server processes

```
#assume $server has already been defined
$server.EnumProcesses() |
Where BlockingSpid -ne 0 |
Select Name, Spid, Command, Status, Login, Database, BlockingSpid |
Format-Table -AutoSize
```

Read file into an array

```
$instances = Get-Content "C:\Temp\sqlinstances.txt"
```

SQL Server-Specific Cmdlets

```
Get-Command -CommandType Cmdlet -Module SQLPS,SQLASCMDLETS |
Select Name, Module |
Sort Module, Name |
Format-Table -AutoSize
```

Results:

Add-RoleMember	SQLASCMDLETS
Backup-ASDatabase	SQLASCMDLETS

Invoke-ASCmd	SQLASCMDLETS
Invoke-ProcessCube	SQLASCMDLETS
Invoke-ProcessDimension	SQLASCMDLETS
Invoke-ProcessPartition	SQLASCMDLETS
Merge-Partition	SQLASCMDLETS
New-RestoreFolder	SQLASCMDLETS
New-RestoreLocation	SQLASCMDLETS
Remove-RoleMember	SQLASCMDLETS
Restore-ASDatabase	SQLASCMDLETS
Add-SqlAvailabilityDatabase	SQLPS
Add-SqlAvailabilityGroupListenerStaticIp	SQLPS
Backup-SqlDatabase	SQLPS
Convert-UrnToPath	SQLPS
Decode-SqlName	SQLPS
Disable-SqlAlwaysOn	SQLPS
Enable-SqlAlwaysOn	SQLPS
Encode-SqlName	SQLPS
Invoke-PolicyEvaluation	SQLPS
Invoke-Sqlcmd	SQLPS
Join-SqlAvailabilityGroup	SQLPS
New-SqlAvailabilityGroup	SQLPS
New-SqlAvailabilityGroupListener	SQLPS
New-SqlAvailabilityReplica	SQLPS
New-SqlHADREndpoint	SQLPS
Remove-SqlAvailabilityDatabase	SQLPS
Remove-SqlAvailabilityGroup	SQLPS
Remove-SqlAvailabilityReplica	SQLPS
Restore-SqlDatabase	SQLPS
Resume-SqlAvailabilityDatabase	SQLPS
Set-SqlAvailabilityGroup	SQLPS
Set-SqlAvailabilityGroupListener	SQLPS
Set-SqlAvailabilityReplica	SQLPS
Set-SqlHADREndpoint	SQLPS
Suspend-SqlAvailabilityDatabase	SQLPS
Switch-SqlAvailabilityGroup	SQLPS
Test-SqlAvailabilityGroup	SQLPS
Test-SqlAvailabilityReplica	SQLPS
Test-SqlDatabaseReplicaState	SQLPS

Invoke-SqlCmd

```
$instanceName = "KERRIGAN"
$dbName = "AdventureWorks2008R2"
$query = "SELECT TOP 10 * FROM Person.Person"
$fileName = "C:\Temp\ResultsFromPassThrough.csv"

#export query results to CSV
Invoke-Sqlcmd -Query $query -ServerInstance $instanceName -Database
$dbName |
Export-Csv -LiteralPath $fileName -NoTypeInfo
```

Create SMO Server Object

An SMO object, or SQL Server Management Object, allows you to programmatically access and manipulate SQL Server:

```
Import-Module SQLPS -DisableNameChecking

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

Create SSRS Proxy Object

```
$ReportServerUri = "http://localhost/ReportServer/ReportService2010.
asmx"
$proxy = New-WebServiceProxy -Uri $ReportServerUri -UseDefaultCredential

#list all children
$proxy.ListChildren("/", $true)
```

Create SSIS Object (SQL Server 2005/2008/2008R2)

For most SSIS objects included in Package Deployment Model:

```
Add-Type -AssemblyName "Microsoft.SqlServer.ManagedDTS, Version=11.0.0.0,
Culture=neutral, PublicKeyToken=89845dcd8080cc91"
```

```
$app = New-Object Microsoft.SqlServer.Dts.Runtime.Application
```

Create an SSIS Object (SQL Server 2012)

For most SSIS objects used in the new SQL Server 2012 Project Deployment Model:

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

```
Add-Type -AssemblyName "Microsoft.SqlServer.Management.
IntegrationServices, Version=11.0.0.0, Culture=neutral, PublicKeyToken=89
845dcd8080cc91"
```

```
$instanceName = "KERRIGAN"
```

```
$connectionString = "Data Source=$instanceName;Initial
Catalog=master;Integrated Security=SSPI;"
```

```
$conn = New-Object System.Data.SqlClient.SqlConnection $connectionString
```

```
$SSISServer = New-Object Microsoft.SqlServer.Management.
IntegrationServices.IntegrationServices $conn
```

Create SSAS Object

```
Import-Module SQLASCMDLETS -DisableNameChecking
```

```
#Connect to your Analysis Services server
```

```
$$SSASServer = New-Object Microsoft.AnalysisServices.Server
```


B

PowerShell Primer

In this appendix, we will cover:

- ▶ What is PowerShell, and why learn another language
- ▶ Setting up the environment
- ▶ Running PowerShell scripts
- ▶ Basics—points to remember
- ▶ Scripting syntax
- ▶ Converting script into functions

Introduction

This appendix is a very short primer to get you up and running with PowerShell. We cover the basics of the language and the syntax; however, we will not go into in-depth details and variations. A host of recommended resources is available in *Appendix C, Resources* to augment what you learn from this book.

What is PowerShell, and why learn another language

PowerShell is both a scripting environment and a scripting language meant to support administrators and developers alike in automating and integrating processes and environments.

You may already be familiar with other tools or languages that help accomplish your task, and you may be asking why you should even bother learning PowerShell. It is important to note that PowerShell is just another tool, but could be a very powerful one if used in the appropriate situations.

There are different reasons for using PowerShell:

1. Running a script is faster than clicking around the UI:
If we minimize clicks, or eliminate them in some cases, the task can potentially be done so much faster. Think about compressing, copying, archiving, and renaming multiple files. If we had to rely on the UI, this task may take much longer. However, if we can bake the logic into a script, and run the script once, then the task can be accomplished much faster and more efficiently.
2. Learning, and mastering, one language instead of five or ten:
Instead of using a duct-taped mishmash of scripting languages (batch file for some items, VBScript, Perl, COM), we can now use one single language to handle most tasks.
3. Leveraging the .NET library:
The .NET library provides a rich collection of classes that pretty much covers most programmatic items you can think of such as forms, database connectivity, networking, and the like.
4. Taking advantage of the fact that PowerShell is baked into different products:
More and more Microsoft products are being shipped with a growing number of PowerShell cmdlets because PowerShell scripting is part of Microsoft's Common Engineering Criteria program (<http://www.microsoft.com/cec/en/us/cec-overview.aspx#man-windows>). Windows Server, Exchange, Active Directory, SharePoint, SQL Server, to name a few, all have some PowerShell support.

Setting up the Environment

Before we can start talking about PowerShell, we first need to make sure you have access to an environment that has PowerShell.

PowerShell V3 comes natively with the following operating systems: Windows 8 and Windows Server 2012.

Although Windows 7, Windows Server 2008, and Windows Server 2008 R2 come with PowerShell V2, you can also install PowerShell V3 on these operating systems. You can download the Windows Management Framework 3.0 (WMF 3.0), which contains PowerShell V3 from <http://www.microsoft.com/en-us/download/details.aspx?id=34595>. If you have been testing the Beta or CTP versions of PowerShell V3, you will need to uninstall these previous versions prior to installing the Released to Manufacturing (RTM) version, which is the official publicly available version.

Running PowerShell scripts

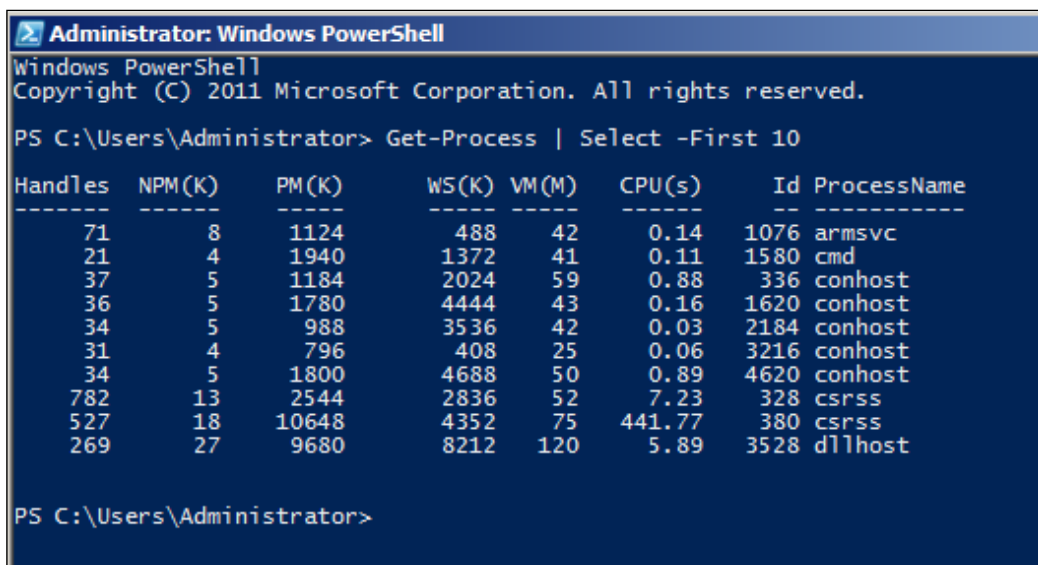
It is now time to run your first script!

Through shell or through ISE

You can run ad hoc commands through the shell or through the **Integrated Scripting Environment (ISE)**.

To use the PowerShell console, you can launch the shell by opening **Start | All Programs | Accessories | Windows PowerShell | Windows PowerShell**. Often when managing your servers, you may need to run this as Administrator (right-click on the PowerShell icon and select **Run as Administrator**).

Once the console is ready, you can type your commands and press *Enter* to see the results. For example, to display ten (10) running processes, you can use the `Get-Process` cmdlet, as shown in the following screenshot:



```

Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2011 Microsoft Corporation. All rights reserved.

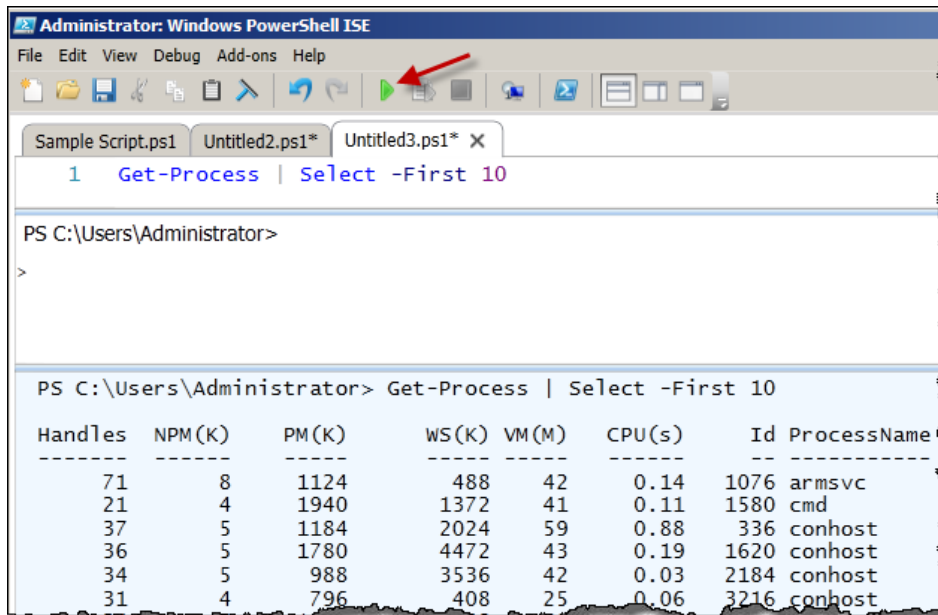
PS C:\Users\Administrator> Get-Process | Select -First 10


Handles  NPM(K)  PM(K)  WS(K) VM(M)  CPU(s)  Id ProcessName
-----  -
71       8       1124   488    42     0.14    1076 armsvc
21       4       1940   1372   41     0.11    1580 cmd
37       5       1184   2024   59     0.88    336  conhost
36       5       1780   4444   43     0.16    1620 conhost
34       5       988    3536   42     0.03    2184 conhost
31       4       796    408    25     0.06    3216 conhost
34       5       1800   4688   50     0.89    4620 conhost
782     13       2544   2836   52     7.23    328  csrss
527     18      10648  4352   75    441.77   380  csrss
269     27       9680   8212  120     5.89   3528 dllhost

PS C:\Users\Administrator>

```

You can also use the ISE, and to launch the ISE, go to **Start | All Programs | Accessories | Windows PowerShell | Windows PowerShell ISE**. Similar to the shell, you can type your command and press the **Run** button (green arrow icon).



 More details about the ISE are covered in *Chapter 1, Getting Started with SQL Server and PowerShell*.

Typically, you would save your commands in a script file with the `.ps1` extension, and run them from the shell in few different ways:

1. From PowerShell console, using the call operator (&):


```
PS C:\ > & "C:\PowerShell\My Script.ps1"
```
2. From PowerShell console, using dot sourcing. Dot sourcing simply means you prepend a dot and space to your invocation. You would invoke your script by using dot sourcing to persist variables and functions in your session:


```
PS C:\PowerShell > . ".\My Script.ps1"
PS C:\> . "C:\PowerShell\My Script.ps1"
```
3. From a command prompt:


```
C:\>powershell.exe -ExecutionPolicy RemoteSigned -File
"C:\PowerShell\My Script.ps1"
```

Execution policy

PowerShell scripts are not authorized to *just run*.

Remember the "I Love You" virus? It took off because it was so easy to launch a script just by double-clicking the .vbs file.

To avoid problems such as this, PowerShell scripts by default are blocked from running. This means you cannot just accidentally double-click a PowerShell script and execute it.

The rules that determine which PowerShell scripts can run are contained in the Execution Policy. This will need to be set ahead of time. The different settings are:

Execution Policy	Description
Restricted	Default execution policy PowerShell will not run any scripts
AllSigned	PowerShell will run only signed scripts
RemoteSigned	PowerShell will run signed scripts, or locally created scripts
Unrestricted	PowerShell will run any scripts, signed or not
Bypass	PowerShell will not block any scripts, and will prevent any prompts or warnings
Undefined	PowerShell will remove set execution policy in current user scope

To determine what your current setting is, you can use `Get-ExecutionPolicy`:

```
PS C:\>Get-ExecutionPolicy
```

If you try to run a script without setting the proper execution policy, you may get an error similar to this:

```
File C:\Sample Script.ps1 cannot be loaded because the execution
of scripts is disabled on this system. For more information, see
about_execution_policies.
```

To change the execution policy, use `Set-ExecutionPolicy`:

```
PS C:\>Set-ExecutionPolicy RemoteSigned
```

Typically, if you need to run a script that does a lot of administrative tasks, you will need to run the script as administrator.

To learn more about execution policies, run:

```
help about_execution_policies
```

For more information about how to sign your script, use:

```
help about_signing
```

Basics—points to remember

Let's explore some PowerShell basic concepts.

Cmdlets

Cmdlets, pronounced as "commandlets", are the foundation of PowerShell. Cmdlets are *small commands*, or specialized commands. The naming convention for cmdlets follows the Verb-Noun format, such as `Get-Command` or `Invoke-Expression`.

PowerShell V3 boasts a lot of new cmdlets, including cmdlets to manipulate JSON (`ConvertFrom-Json`, `ConvertTo-Json`), web services (`Invoke-RestMethod`, `Invoke-WebRequest`), and background jobs (`Register-JobEvent`, `Resume-Job`, `Suspend-Job`). In addition to built-in cmdlets, there are also downloadable community PowerShell extensions such as SQLPSX, which can be downloaded from <http://sqlpsx.codeplex.com/>.

Many cmdlets accept parameters. Parameters can either be specified by name or by position. Let's take a look at a specific example. The syntax for the `Get-ChildItem` cmdlet is:

```
Get-ChildItem [[-Path] <string[]>] [[-Filter] <string>]
[-Include <string[]>] [-Exclude <string[]>]
[-Recurse] [-Force] [-Name]
[-UseTransaction] [<CommonParameters>]
```

The `Get-ChildItem` cmdlet gets all the "children" in a specified path. For example, to get all files with a `.txt` extension in the `C:\Temp` folder, we can use `Get-ChildItem` with the `-Path` and `-Filter` parameters:

```
Get-ChildItem -Path "C:\Temp" -Filter "*.csv"
```

We can alternatively omit the parameter names by passing the parameter values by position. When passing parameters by position, the order in which the values are passed matters. They need to be in the same order in which the parameters are defined in the `Get-ChildItem` cmdlet:

```
Get-ChildItem "C:\Temp" "*.csv"
```

To learn the order in which parameters are expected to come, you can use the `Get-Help` cmdlet:

```
Get-Help Get-ChildItem
```

Learning PowerShell

The best way to learn PowerShell is to explore the cmdlets, and try them out as you learn them. The best way to learn is to explore. Young Jedi, you need to get acquainted with these three (3) cmdlets: `Get-Command`, `Get-Help`, and `Get-Member`.

Get-Command

There are many cmdlets. And that list is just going to get bigger. It will be hard to remember all the cmdlets except for the handful you use day in and day out. Besides using the search engine, you can use the `Get-Command` cmdlet to help you look for cmdlets.

Here are a few helpful cmdlets:

- ▶ To list all cmdlets:
`Get-Command`
- ▶ To list cmdlets with names that match some string patterns, you can use the `-Name` parameter and the asterisk (*) wildcard:
`Get-Command -Name "**Event*"`
- ▶ To get cmdlets from a specific module:
`Get-Command -Module SQLASCMDLETS`

Get-Help

Now that you've found the command you're looking for, how do you use it? The best way to get help is `Get-Help` (no pun intended). The `Get-Help` cmdlet provides the syntax of a cmdlet, examples, and some additional notes or links where available.

```
Get-Help Backup-SqlDatabase
Get-Help Backup-SqlDatabase -Examples
Get-Help Backup-SqlDatabase -Detailed
Get-Help Backup-SqlDatabase -Full
Get-Help Backup-SqlDatabase -Online #opens browser
```

The different parameters—`Examples`, `Detailed`, `Full`, and `Online`—will determine the amount of information that will be displayed. The `Online` parameter opens up the online help in a browser.

Get-Member

To really understand a command or an object and explore what's available, you can use the `Get-Member` cmdlet. This will list all the properties, methods of an object, or anything incoming from the pipeline.

```
$dt = (Get-Date)
$dt | Get-Member
```


Starter notes

We are almost ready to start learning the syntax. However, here are a few last notes, some points to keep in mind about PowerShell as you learn it. Keep a mental note of these items, and you are ready to go full steam ahead.

PowerShell is object oriented, and works with .NET

PowerShell works with objects, and can take advantage of the objects' methods and properties. PowerShell can also leverage the ever-growing .NET framework library. It can import any of the .NET classes, and reuse any of the already available classes.

You can find out the base class of an object by using the `GetType` method, which comes with all objects.

```
$dt = Get-Date
$dt.GetType() #DateTime is the base type
```

To investigate an object, you can always use the `Get-Member` cmdlet.

```
$dt | Get-Member
```

To leverage the .NET libraries, you can import them in your script. A sample import of the .NET libraries follows:

```
#load the Windows.Forms assembly
Add-Type -AssemblyName "System.Windows.Forms"
```

There will be cases when you may have multiple versions of the same assembly name. In these cases, you will need to specify the strong name of the assembly with the `Add-Type` cmdlet. This means you will need to supply the `AssemblyName`, `Version`, `Culture`, and `PublicKeyToken`:

```
#load the ReportViewer WinForms assembly
Add-Type -AssemblyName "Microsoft.ReportViewer.WinForms,
Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91"
```

To determine the strong name, you can open up `C:\Windows\assembly` and navigate to the assembly you want to load. You can either check the displayed properties, or right-click on the particular assembly and select **Properties**.

Assembly Name	Version	Culture	Public Key Token	Process...
Microsoft.ReportViewer.WebForms.resources	11.0.0.0	zh-CHS	89845dcd8080cc91	MSIL
Microsoft.ReportViewer.WebForms.resources	11.0.0.0	zh-CHT	89845dcd8080cc91	MSIL
Microsoft.ReportViewer.WinForms	11.0.0.0		89845dcd8080cc91	MSIL
Microsoft.ReportViewer.WinForms	10.0.0.0		b03f5f7f11d50a3a	MSIL
Microsoft.ReportViewer.WinForms.resources	11.0.0.0	de	89845dcd8080cc91	MSIL

Cmdlets may have aliases or you can create one

We may already know some scripting or programming languages, and may already have preferences on how we do things. For example, when listing directories from the Command Prompt, we may be on autopilot when we type `dir`. In PowerShell, listing directories can be accomplished by the `Get-ChildItem` cmdlet. Fear not, you can still use `dir` if you prefer. If there is another name you want to use instead of `Get-ChildItem`, you can create your own alias.

To find out aliases of a cmdlet, you can use `Get-Alias`. For example, to get the aliases of `Get-ChildItem`, you can execute:

```
Get-Alias -Definition "Get-ChildItem"
```

To create your own alias, you can use `New-Alias`:

```
New-Alias "list" Get-ChildItem
```

Here are some of the common aliases already built-in with PowerShell:

Cmdlet	Alias
Foreach-Object	%, Foreach
Where-Object	?, Where
Sort-Object	Sort
Compare-Object	compare, diff
Write-Output	echo, write
help	man
Get-Content	cat, gc, type
Get-ChildItem	dir, gci, ls
Copy-Item	copy, cp, cpi
Move-Item	mi, move, mv
Remove-Item	del, erase, rd, ri, rm, rmdir
Get-Process	gps, ps
Stop-Process	kill, spps

Cmdlet	Alias
Get-Location	gl, pwd
Set-Location	cd, chdir, sl
Clear-Host	clear, cls
Get-History	h, ghy, history

You can chain commands

You can take the result from one command and use it as an input to another command. The operator to chain commands is a vertical bar (|) called pipe. This feature makes PowerShell really powerful. This can also make your statements more concise.

If you are familiar with the Unix/Linux environment, pipes are a must-have and are incredibly valuable tools.

Let's take an example. We will export the newest log entries (time and source fields only) to a text file in JSON format:

1. We need to get the newest log entries:

```
Get-EventLog -LogName Application -Newest 10
```
2. We need only the time and source fields. Based on what we get from step 1, we need to execute the following query:

```
Select Time, Source
```
3. We need to convert to JSON. Using step 2 results as input, we need to execute the following query:

```
ConvertTo-Json
```
4. We need to save to a file. We now want to take what we have in step 3 and put it into a file:

```
Out-File -FilePath "C:\Temp\json.txt" -Force
```

5. The full command will be:

```
Get-EventLog -LogName Application -Newest 10 |  
Select Time, Source |  
ConvertTo-Json |  
Out-File -FilePath "C:\Temp\json.txt" -Force
```

This is just a simple example of how you can chain commands, but should give you an idea how it can be done.

Filter left, format right

When you chain commands, especially when your last actions are for formatting the result, you want to do this as efficiently as possible. Otherwise, you may use a lot of resources to format data, and end up only needing to display a few. It is best to trim your data first, before you pass them down the pipeline for formatting.

Package and reuse

Functions and modules allow you to package up the logic you built in your scripts, and put it in reusable structures. A function can be simply described as a *callable* code block. A module allows you to put together a library of variables and functions that can be loaded into any session, and allow the use of these variables and functions.

Your goal should be to package up most of what you've already built in scripts, and put it into functions, and later compile them into a module. Note that you can also create your functions so they behave like cmdlets.



Converting your scripts into functions is tackled at a later section in this appendix.

Common Cmdlets

Typically, cmdlets are categorized to their main purpose or functionality based on the verb used in their name. Here is a partial list of cmdlets to explore. Note that many cmdlet names are self-documenting:

Category	Cmdlet
Utility	ConvertFrom-Csv
	ConvertFrom-Json
	ConvertTo-Csv
	ConvertTo-Html
	ConvertTo-Json
	ConvertTo-Xml
	Export-Clixml
	Export-Csv
	Format-List
	Format-Table
	Get-Alias
	Get-Date
	Get-Member
	Import-Clixml
	Import-Csv
	Read-Host
	Management
	Get-Content
	Get-EventLog
	Get-HotFix
	Get-Process
	Get-Service
	Get-WmiObject
	New-WebServiceProxy
	Start-Process
	Start-Service

Category	Cmdlet
Security	ConvertFrom-SecureString
	ConvertTo-SecureString
	Get-Credential
	Get-ExecutionPolicy
	Set-ExecutionPolicy

Scripting syntax

We will now dive into the specifics of PowerShell syntax.

Statement terminators

A semicolon is typically a mandatory statement terminator in many programming and scripting languages. PowerShell considers both a newline and a semicolon as statement terminators, although using the newline is more common, that's why you won't see a lot of semicolons in most PowerShell scripts. There is a caveat for using the newline; that is, the previous line must be a complete statement before it gets executed.

Escape and line continuation

The backtick (`) is a peculiar character in PowerShell, and it has double meaning. You can typically find this character in your keyboard above the left *Tab* key, and is in the same key as the tilde (~) symbol.

The backtick is the escape character in PowerShell. Some of the common characters that need to be escaped are:

Escaped Character	Description
`n	Newline
`r	Carriage return
`'	Single quote
`"	Double quote
`0	Null

PowerShell also uses the backtick as a line continuation character. You may find yourself writing a long chain of commands and may want to put different parts of the command onto different lines to make the code more readable. If you do, you need to make sure to put a backtick at the end of each line you are continuing, otherwise PowerShell treats the newline as a statement terminator. You also need to make sure there are not any extra spaces after the backtick:

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

Variables

Variables are placeholders for values. Variables in PowerShell start with a dollar (\$) sign.

```
$a = 10
```

By default, variables are loosely and dynamically typed—meaning the variable assumes the data type based on the value of the content:

```
$a = 10
$a.GetType() #Int32

$a = "Hello"
$a.GetType() #String

$a = Get-Date
$a.GetType() #DateTime
```

Note how the data type changes based on the value we assign to the variable. You can however create strongly typed variables.

```
[int]$a = 10
$a.GetType() #Int32
```

When we have strongly typed variables, we can no longer just haphazardly assign it any value. If we do, we will get an error:

```
$a = "Hello"

<# Error
Cannot convert value "Hello" to type "System.Int32". Error: "Input
string was not in a correct format."
At line:3 char:1
+ $a = "Hello"
+ ~~~~~
```

```

+ CategoryInfo          : MetadataError: (:) [],
ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException
#>

```

We have also mentioned in the previous section that PowerShell is object oriented. Variables in PowerShell are automatically created as objects. Depending on the data type, variables are packaged with their own attributes and methods. To explore what properties and methods are available with a data type, use the `Get-Member` cmdlet:

```

1 $a = Get-Date
2 $a | Get-Member

```

ToLongTimeString	Method	string ToLongTimeString()
ToOADate	Method	double ToOADate()
ToSByte	Method	System.SByte ToSByte(System.IFormatProvider)
ToShortDateString	Method	string ToShortDateString()
ToShortTimeString	Method	string ToShortTimeString()
ToSingle	Method	float ToSingle(System.IFormatProvider)
ToString	Method	string ToString(), string ToString(System.IFormatProvider)
ToType	Method	System.Object ToType(type conversion)
ToUInt16	Method	System.UInt16 ToUInt16(System.IFormatProvider)
ToUInt32	Method	System.UInt32 ToUInt32(System.IFormatProvider)
ToUInt64	Method	System.UInt64 ToUInt64(System.IFormatProvider)
ToUniversalTime	Method	System.DateTime ToUniversalTime()
Date	Property	System.DateTime Date {get;}
Day	Property	System.Int32 Day {get;}
DayOfWeek	Property	System.DayOfWeek DayOfWeek {get;}
DayOfYear	Property	System.Int32 DayOfYear {get;}
Hour	Property	System.Int32 Hour {get;}
Kind	Property	System.DateTimeKind Kind {get;}
Millisecond	Property	System.Int32 Millisecond {get;}
Minute	Property	System.Int32 Minute {get;}

Here-string

There may be times when you need to create a string variable that will contain multiple lines of code. You should create these as `here-string`.

A `here-string` is a string that often contains large blocks of text. It starts with `@` and must end with a line that contains only `@`. For the `here-string` terminating character pair, make sure this is placed in its own line, and there are no other characters and no spaces before or after it.

```

$query = @"
INSERT INTO SampleXML
(fileName, XMLStuff, FileExtension)
VALUES ('$xmlfile', '$xml', '$fileextension')
"@

```


String interpolation

When working with strings, you need to remember that using a double quote evaluates enclosed variables, that is variables are replaced with their values. For example:

```
$today = Get-Date
Write-Host "Today is $today"

#result
#Today is 06/12/2012 19:48:24
```

This behavior may sometimes cause issues especially if you need to use multiple variables in continuation, as in the following case where we want to combine `$name`, and underscore (`_`), `$ts` and `.txt` to create a timestamped filename.

```
$name = "belle"
$ts = Get-Date -Format yyyy-MMM-dd
$filename = "$name_ $ts.txt"
```

This will give an incorrect result, because it will look for `$name_` and `$ts`, but since it cannot find `$name_`, the final filename we get is `2012-Jun-06.txt` and not `belle_2012-Jun-06.txt`.

To resolve this issue, we can use any of the following to ensure proper interpolation:

```
$filename = "$($name)_ $($ts).txt"
Write-Host $filename

$filename = "${name}_ ${ts}.txt"
Write-Host $filename

$filename = "{0}_{1}.txt" -f $name, $ts
Write-Host $filename
```

A single quote, on the other hand, preserves the actual variable name and does not interpolate the value:

```
$today = Get-Date
Write-Host 'Today is $today'

#result
#Today is $today
```

You can also store actual commands in a string. However, this is treated as a string unless you prepend it with an ampersand (&)-which is PowerShell's invoke or call operator.

```
$cmd = "Get-Process"

$cmd    #just displays Get-Process, treated as string
&$cmd   #actually executes Get-Process
```

Operators

The operators used in PowerShell may not be readily familiar to you even if you have already done some programming before. This is because the operators in PowerShell do not use the common operator symbols.

PowerShell	Traditional Operator	Description
-eq	==	Equal to
-ne	<> or !=	Not equal to
-match		Match using regex; searches anywhere in string
-notmatch		
-contains		Collection match. Does item exist in array or collection?
-notcontains		
-like		Wildcard match
-notlike		* (asterisk) for zero or more characters ? (question mark) for any single character
-clike		
-cnotlike		Case-sensitive wildcard match
-not	!	Negation
-lt	<	Less than
-le	<=	Less than or equal to
-gt	>	Greater than
-ge	>=	Greater than or equal to
-and	&&	Logical and
-or		Logical or
-bor		Bitwise or
-band	&	Bitwise and
-xor	^	Exclusive or

Note that many operators perform case-insensitive string comparisons by default. If you want to do case-sensitive matching, prepend with *c*. For example, `-ceq`, `-clike`, `-cnotlike`.

Displaying messages

Often we will need to display or log messages as our scripts execute. PowerShell provides a few cmdlets to help us accomplish this.

```
Get-Command -Name "*Write*" -CommandType Cmdlet
```

This should give a list of our `Write-` related cmdlets:

Cmdlet	Description
<code>Write-Debug</code>	Display debug message to console Typically used with <code>\$DebugPreference = "Continue"</code>
<code>Write-Error</code>	Display non-terminating error message to console
<code>Write-EventLog</code>	Write message to Windows Event Log
<code>Write-Host</code>	Display string message to host
<code>Write-Output</code>	Write an object to pipeline
<code>Write-Progress</code>	Display a progress bar
<code>Write-Verbose</code>	Display verbose message to console Typically used with <code>\$VerbosePreference = "Continue"</code>
<code>Write-Warning</code>	Display warning message to console

Although some of these cmdlets seem similar, there are some fundamental differences. For example, `Write-Host` and `Write-Output` seem to display the same messages on screen. `Write-Host` however simply displays a string, but `Write-Output` writes objects that have properties that can be queried, and can eventually be used in the pipeline.

We use `Write-Verbose` a fair bit in the recipes in this book. `Write-Verbose` does not automatically display messages on the host. It relies on the `$VerbosePreference` setting. By default, `$VerbosePreference` is set to `SilentlyContinue`, but it can also be set to `Continue`, which allows us to display messages used with `Write-Verbose` to screen.

```
$VerbosePreference = "Continue"
$folderName = "C:\BLOB Files\"

#using PowerShell V2 style Where-Object syntax
Get-ChildItem $folderName |
Where-Object {$_.PSIsContainer -eq $false} |
ForEach-Object {
    $blobFile = $_
```

```

    Write-Verbose "Importing file $($blobFile.FullName)..."
}
$VerbosePreference = "SilentlyContinue"

```

This is an elegant way of turning all messages on or off, without needing to change the script. This can also be used as a switch and can be passed to the script or a function.

Comments

Comments are important in any programming or scripting language. Comments are often used to document logic, and sometimes a chain of changes to the script.

Single line comments start with a hash sign (#):

```
#this is a single line comment
```

Block comments start with <# and end with #>:

```

<#
this is a block comment
#>

```

PowerShell also supports what's called *Comment Based Help*. This feature allows you to put a special comment block at the start of your script, or in the beginning of your function, that allows the script or function to be looked up using `Get-Help`. A sample of this type of comment block follows:

```

<#
.SYNOPSIS
    Creates a full database backup
.DESCRPTION
    Creates a full database backup using specified instance name and
    database name
    This will place the backup file to the default backup directory of
    the instance
.PARAMETER instanceName
    instance where database to be backed up resides
.PARAMETER databaseName
    database to be backed up
.EXAMPLE
    PS C:\PowerShell> .\Backup-Database.ps1 -instanceName "QUERYWORKS\
SQL01" -databaseName "pubs"
.EXAMPLE
    PS C:\PowerShell> .\Backup-Database.ps1 -instance "QUERYWORKS\
SQL01" -database "pubs"
.NOTES

```

```
To get help:
Get-Help .\Backup-Database.ps1
.LINK
http://msdn.microsoft.com/en-us/library/hh245198.aspx
#>
```

To look up the help, you can simply type a `Get-Help` followed by the script filename, or the function name:

```
PS>Get-Help .\Backup-Database.ps1
```

Special variables

PowerShell also has some special variables. These special variables do not need to be created ahead of time, they are already available. Some of the special variables are:

Special Variable	Description
<code>\$_</code>	Current pipeline object
<code>\$args</code>	Arguments passed to a function
<code>\$error</code>	Array that stores all errors
<code>\$home</code>	User's home directory
<code>\$host</code>	Host information
<code>\$match</code>	Regex matches
<code>\$profile</code>	Path to profile, if available
<code>\$PSHome</code>	Install directory of PowerShell
<code>\$PSISE</code>	PowerShell Scripting Environment object
<code>\$pid</code>	Process ID (PID) of PowerShell process
<code>\$pwd</code>	Present working directory
<code>\$true</code>	Boolean true
<code>\$false</code>	Boolean false
<code>\$null</code>	Null value

Conditions

PowerShell supports conditional logic using `if/else` statements or `switch` statements. These two constructs allow you to check for a condition, and consequently execute different blocks of code if the condition is met or not.

Let's look at an example of an `if/else` block:

```
$answer = Read-Host "Which course are you taking?"
if ($answer -eq "COMP 4677")
```

```
{
  Write-Host "That's SQL Server Administration"
}
elseif ($answer -eq "COMP 4678")
{
  Write-Host "That's SQL Server Development"
}
else
{
  Write-Host "That's another course"
}
```

Note that the `elseif` and `else` blocks are optional. They don't need to be defined if you do not have a separate code to execute if the condition is not met.

An equivalent `switch` block can be written for the above code:


```
$answer = Read-Host "Which course are you taking?"
switch ($answer)
{
  "COMP 4677"
  {
    Write-Host "That's SQL Server Administration"
  }
  "COMP 4678"
  {
    Write-Host "That's SQL Server Development"
  }
  default
  {
    Write-Host "That's another course"
  }
}
```

Note that these two constructs can be functionally equivalent for simple comparisons. The choice to use one over the other hinges on preference and readability. If there are many choices, the `switch` can definitely make the code more readable.

Regular Expressions

Regular expressions, more commonly referred to as **regex**, specify a string pattern to match. Regex can be extremely powerful, and is often used when dealing with massive amounts of text. The area of bioinformatics, for example, tends to rely heavily on regular expressions for gene pattern matching.

Regex can also be quite confusing especially for beginners. It has its own set of patterns and wildcards, and it is up to you to put these together to ensure you are matching what you need to be matched.

 See the recipe *Testing Regular Expressions* in *Chapter 9, Helpful PowerShell Snippets*.

Arrays

Arrays are collections of items. Often we find ourselves needing to store a group of items, either for further processing, or for exporting.

```
#ways to create an array
$myArray = @() #empty
$myArray = 1,2,3,4,5
$myArray = @(1,2,3,4,5)

#array of processes consuming >30% CPU
$myArray = (Get-Process | Where CPU -gt 30 )
```

Arrays can either be of a fixed size or not. Fixed-size arrays are instantiated with a fixed number of items. Some of the typical methods such as `Add` or `Remove` cannot be used with fixed-size arrays:

```
$myArray = @()
$myArray += 1,2,3,4,5
$myArray += 6,7,8
$myArray.Add(9) #error because array is fixed size
```

Removing an item from a fixed array is a little bit tricky. Although arrays have `Remove` and `RemoveAt` methods—to remove based on value and index respectively—we cannot use these with fixed-size arrays. To remove an item from a fixed-size array, we will need to reassign the new set of values to the array variable.

```
#remove 6
$myArray = $myArray -ne 6

#remove 7
$myArray = $myArray -ne 7
```

To create a dynamic-sized array, you will need to declare the array as an array list, and add items using the `Add` method. This also supports removing items from the list using the `Remove` method.

```
$myArray = New-Object System.Collections.ArrayList
$myArray.Add(1)
```

```
$myArray.Add(2)
$myArray.Add(3)
$myArray.Remove(2)
```

We can use indices to retrieve information from the array:

```
#retrieve first item
$myArray[0]

#retrieve first 3 items
$myArray[0..2]
```

We can also retrieve based on some comparison or condition:

```
#retrieving anything > 3
$myArray -gt 3
```

Hashes

A hash is also a collection. This is different from an array, however, because hashes are collections of key-value pairs. Hashes are also called associative arrays, or hash tables.

```
#simple hash
$simplehash = @{
    "BCIT" = "BC Institute of Technology"
    "CST" = "Computer Systems Technology"
    "CIT" = "Computer Information Technology"
}
$simplehash.Count

#hash containing process IDs and names
$hash = @{}
Get-Process | Foreach {$hash.Add($_.Id, $_.Name)}
$hash.GetType()
```

To access items in a hash, we can refer to the hash table variable, and retrieve based on the stored key:

```
$simplehash["BCIT"]
$simplehash.BCIT
```

Loop

A loop allows you to repeatedly execute block(s) of code based on some condition. There are different types of loop support in PowerShell. For all intents and purposes, you may not need to use all of these types, but it's always useful to be aware of what's available and doable.

There is a `while` loop, where the condition is tested at the beginning of the block:

```
$i = 1;
while($i -le 5)
{
    #code block
    $i
    $i++
}
```

There is also support for the `do while` loop, where the condition is tested at the bottom of the block:

```
$i = 1
do
{
    #code block
    $i
    $i++
}while($i -le 5)
```

The `for` loop allows you to loop a specified number of times, based on a counter you create at the `for` header.

```
for($i = 1; $i -le 5; $i++)
{
    $i
}
```

There is yet another type of loop, a `foreach` loop. This loop is a little bit different because it works with arrays or collections. It allows a block of code to be executed for each item in a collection.

```
$backupcmds = Get-Command -Name "*Backup*" -CommandType Cmdlet
foreach($backupcmd in $backupcmds)
{
    $backupcmd | Get-Member
}
```

If you're a developer, this code looks very familiar to you. In PowerShell, however, you can use pipelining to make your code more concise.

```
Get-Command -Name "*Backup*" -CommandType Cmdlet |
Foreach { $_ | Get-Member }
```

Error Handling

When developing functions or scripts, it is important to think beyond just the functionality you are trying to achieve. You also want to handle exceptions, or errors, when they happen. We all want our scripts to gracefully exit if something goes wrong, rather than display some rather intimidating or cryptic error messages.

Developers in the house will be familiar with the concept of `try/catch/finally`. This is a construct that allows us to put the code we want to run in one block (`try`), exception handling code in another (`catch`), and any must-execute housekeeping blocks in a final block (`finally`).

```
$dividend = 20
$divisor = 0

try
{
    $result = $dividend/$divisor
}
catch
{
    Write-Host ("=====" * 20)
    Write-Host "Exception $error[0]"
    Write-Host ("=====" * 20)
}
finally
{
    Write-Host "Housekeeping block"
    Write-Host "Must execute by hook or by crook"
}
```

Converting script into functions

A `function` is a reusable, callable code block(s). A function can accept parameters, and can produce different results based on values that are passed to it.

A typical anatomy of a PowerShell function looks like:

```
function Do-Something
{
    <#
        comment based help
    #>
```

```
param
(
    #parameters
)
#blocks of code
}
```

To illustrate, let's create a very simple function that takes a report server URL and lists all items in that report server. This function will take in a parameter for the report server URL, and another switch called `$ReportsOnly`, which can toggle displaying between all items, or only report items.

```
function Get-SSRSItems
{
    <#
        comment based help
    #>
    param
    (
        [Parameter(Position=0,Mandatory=$true)]
        [alias("reportServer")]
        [string]$ReportServerUri,
        [switch]$ReportsOnly
    )

    Write-Verbose "Processing $($ReportServerUri) ..."
    $proxy = New-WebServiceProxy `
        -Uri $ReportServerUri `
        -UseDefaultCredential
    if ($ReportsOnly)
    {
        $proxy.ListChildren("/", $true) |
        Where TypeName -eq "Report"
    }
    else
    {
        $proxy.ListChildren("/", $true)
    }
}
```

To call this function, we can pass in the value for `-ReportServerUri` and also set the `-ReportsOnly` switch:

```
$server = "http://server1/ReportServer/ReportService2010.asmx"

Get-SSRSItems -ReportsOnly -ReportServerUri $server |
Select Path, TypeName |
Format-Table -AutoSize
```

To allow your function to behave more like a cmdlet and work with the pipeline, we will need to add the `[CmdletBinding()]` attribute. We can also change the parameters to enable values to come from the pipeline by using `ValueFromPipeline=$true`. Inside the function definition, we will need to add three blocks:

- ▶ `BEGIN`
Preprocessing; anything in this block will be executed once when the function is called.
- ▶ `PROCESS`
Actual processing that is done for each item that is passed in the pipeline.
- ▶ `END`
Post-processing; this block will be executed once before the function terminates executing.

We will also need to specify in the parameter block that we want to accept input from the pipeline.

A revised function follows:

```
function Get-SSRSItems
{
    <#
        comment based help
    #>
    [CmdletBinding()]
    param
    (
        [Parameter(Position=0,Mandatory=$true,
            ValueFromPipeline=$true,
            ValueFromPipelineByPropertyName=$true)]
        [alias("reportServer")]
        [string]$ReportServerUri,
        [switch]$ReportsOnly
    )
    BEGIN
    {
    }
    PROCESS
    {
        Write-Verbose "Processing $($ReportServerUri) ..."
        $proxy = New-WebServiceProxy `
            -Uri $ReportServerUri -UseDefaultCredential
        if ($ReportsOnly)
        {
```

```
        $proxy.ListChildren("/", $true) |
        Where TypeName -eq "Report"
    }
    else
    {
        $proxy.ListChildren("/", $true)
    }
}
END
{
    Write-Verbose "Finished processing"
}
}
```

To invoke, we can pipe an array of servers to the `Get-SSRSItems` function, and this automatically maps the servers to our `-ReportServerUri` parameter since we specified `ValueFromPipeline=$true`. Note that `Get-SSRSItems` will get invoked for each value in our array:

```
$servers = @"http://server1/ReportServer/ReportService2010.asmx",
"http://server2/ReportServer/ReportService2010.asmx"

$servers |
Get-SSRSItems -Verbose -ReportsOnly |
Select Path, TypeName |
Format-Table -AutoSize
```

More about PowerShell

We have barely touched PowerShell basics, but this appendix should give you an idea how to use PowerShell. To learn more about PowerShell, check out *Appendix C, Resources*, which lists a number of other resources you might find useful with your PowerShell adventure.

C

Resources

Resources

There are a lot of good websites, articles, webcasts, blogs, and bloggers on PowerShell. This is by no means an exhaustive list of resources. The list below is simply meant to help you jumpstart your adventure with PowerShell and SQL Server. Bear in mind that beyond this list, there are a lot more to explore! Enjoy the adventure!

PowerShell Books

PowerShell V3

- ▶ PowerShell in Depth: An administrator's guide
by Don Jones, Richard Siddaway, and Jeffery Hicks
- ▶ PowerShell and WMI
by Richard Siddaway
- ▶ Windows PowerShell for Developers
by Doug Finke

PowerShell V2

- ▶ Learn Windows PowerShell in a Month of Lunches
by Don Jones
- ▶ PowerShell 2.0 TFM
by Don Jones

Resources

- ▶ PowerShell and WMI
by Richard Siddaway
- ▶ PowerShell in Practice
by Richard Siddaway
- ▶ Windows PowerShell 2.0 Administrator's Pocket Consultant
by William R. Stanek
- ▶ Windows PowerShell Cookbook
by Lee Holmes
- ▶ Windows PowerShell in Action, Second Edition
by Bruce G. Payette

PowerShell V2 Free E-books

- ▶ Administrator's Guide to PowerShell Remoting
by Dr. Tobias Weltner and Aleksandar Nikolic
<http://powershell.com/cs/media/p/4908.aspx>
- ▶ Effective Windows PowerShell
by Keith Hill
<http://rkeithhill.wordpress.com/2009/03/08/effective-windows-powershell-the-free-ebook/>
- ▶ Layman's Guide to PowerShell Remoting
by Ravikanth Chaganti
<http://www.ravichaganti.com/blog/?p=1305>
- ▶ Mastering PowerShell (free e-book)
by Dr. Tobias Weltner
<http://powershell.com/cs/blogs/ebookv2/default.aspx>
- ▶ PowerShell 2.0 One Cmdlet at a Time
by Jonathan Medd
http://www.jonathanmedd.net/wp-content/uploads/2010/09/PowerShell_2_One_Cmdlet_at_a_Time.pdf

- ▶ Secrets of PowerShell Remoting
by Don Jones and Dr. Tobias Weltner
<http://powershellbooks.com/SecretsOfPowerShellRemoting.zip>
- ▶ WMI Query Language (WQL) via PowerShell
by Ravikanth Chaganti
http://www.ravichaganti.com/blog/?page_id=2134

PowerShell Blogs and Sites

- ▶ PowerShell.com
<http://www.powershell.com>
- ▶ PowerShell Team—Windows PowerShell Blog
<http://blogs.msdn.com/powershell/>
- ▶ PowerShell Magazine
<http://www.powershellmagazine.com/>
- ▶ PoshCode—PowerShell Code Repository
<http://poshcode.org/>
- ▶ PowerShell Survival Guide
<http://social.technet.microsoft.com/wiki/contents/articles/183-windows-powershell-survival-guide-en-us.aspx>
- ▶ Technet Script Repository
<http://gallery.technet.microsoft.com/scriptcenter/>
- ▶ PowerShell V3 Guide
<http://social.technet.microsoft.com/wiki/contents/articles/4725.powershell-v3-guide-en-us.aspx>
- ▶ Hey Scripting Guy! Blog
<http://blogs.technet.com/b/heyscriptingguy/>
- ▶ Scripting with PowerShell—5 Part Webcast
<http://technet.microsoft.com/en-US/scriptcenter/dd742419.aspx>

PowerShell Bloggers

- ▶ Don Jones
<http://www.windowsitpro.com/topics/powershell-scripting/don-jones-on-powershell>
- ▶ Richard Siddaway
<http://richardspowershellblog.wordpress.com/>
- ▶ Jeffery Hicks
<http://jdhitsolutions.com/blog/>
- ▶ Shay Levy
<http://blogs.microsoft.co.il/blogs/scriptfanatic/>
- ▶ Lee Holmes
<http://www.leeholmes.com/blog/>
- ▶ Ed Wilson (Hey Scripting Guy!)
<http://blogs.technet.com/b/heyscriptingguy/>
- ▶ Ravikanth Chaganti
<http://www.ravichaganti.com/blog/>
- ▶ Dr. Tobias Weltner
<http://powershell.com/cs/blogs/tobias/>
- ▶ Niklas Goude
<http://www.powershell.nu/>
- ▶ Keith Hill
<http://rkeithhill.wordpress.com/>
- ▶ Doug Finke
<http://www.dougfinke.com>
- ▶ Thomas Lee
<http://tfl09.blogspot.ca/>
- ▶ Joel Bennett
<http://huddledmasses.org/>

- ▶ Jonathan Medd
<http://www.jonathanmedd.net/>
- ▶ Steven Murawski
<http://blog.usepowershell.com>
- ▶ Aleksandar Nikolic
<http://powershellers.blogspot.ca/>

SQL Server and PowerShell Bloggers

- ▶ Allen White
http://sqlblog.com/blogs/allen_white/
- ▶ Laerte Junior
<http://shellyourexperience.wordpress.com/>
- ▶ Aaron Nelson
<http://www.sqlvariant.com>
- ▶ Edwin Sarmiento
<http://bassplayerdoc.blogspot.ca/>
- ▶ Chad Miller
<http://sev17.com>
- ▶ Max Trinidad
<http://www.maxtblog.com/2011/10/sql-server-powershell-smo-simple-way-to-change-sql-user-passwords/>
- ▶ Donabel Santos
<http://www.sqlmusings.com>

PowerShell Webcasts and Podcasts

- ▶ MSDN Channel 9 PowerShell Webcasts
<http://channel9.msdn.com/tags/PowerShell/>
- ▶ PowerScripting Podcasts
<http://powerscripting.wordpress.com/>

PowerShell Tools

- ▶ Idera PowerShell Plus
<http://www.idera.com/PowerShell/powershell-plus/>
- ▶ Quest PowerGUI
<http://powergui.org/index.jspa>
- ▶ Sapien PrimalScript
<http://www.sapien.com/software/primalscript>

SQLPSX

SQLPSX is a set of modules that wrap SMO objects into easier-to-use functions. This is a project maintained and contributed to by Chad Miller, Mike Shepard, Laerte Junior, Steve Murawski, Bernd Kriszio, and Max Trinidad.

<http://sqlpsx.codeplex.com/>

PSCX

PowerShell Community Extensions is a set of modules that extend PowerShell with additional cmdlets, functions, aliases, and filters. This project is maintained by R Keith Hill.

<http://pscx.codeplex.com/>

D

Creating a SQL Server VM

In this appendix we will cover:

- ▶ Terminology
- ▶ Downloading software
- ▶ VM details and accounts
- ▶ Creating an empty virtual machine
- ▶ Installing Windows Server 2008 R2 as Guest OS
- ▶ Configuring a domain controller
- ▶ Creating domain accounts
- ▶ Installing SQL Server 2012 on a VM
- ▶ Installing PowerShell V3

Introduction

I find the best way to learn new software or application is by creating a virtual machine that has the new software on it. I typically use SQL Server VMs for my development and administration classes. I want the students to have full autonomy over the machines they are using, so that they can try different features and configurations without worrying about wrecking their own machines.



Creating and working with virtual machines may seem confusing at first for the novice. I originally wrote a step-by-step guide for my students at the following URL:

<http://www.sqlmusings.com/wp-content/uploads/2009/09/Step-by-Step-Guide-to-Creating-a-SQL-Server-VM-Using-VMWare.pdf>

This original document uses VMWare Server, which is no longer available and supported.

What you see in this appendix is an updated version, specifically using VMWare Player and SQL Server 2012.

Terminology

Let's start off with some terminologies:

Terminology	Description
Virtual Machine, or VM	<p>This is essentially a standalone computer installed within another platform/OS.</p> <p>A virtual machine is also sometimes called a guest machine. This typically provides a complete system platform with its own set of operating system, hardware configurations, and installed software packages, but still runs on top of a host machine that has the main OS (operating system), and the physical hardware.</p> <p>There are different applications that can create and run virtual machines. A partial list includes:</p> <ul style="list-style-type: none">▶ VMWare Player (free): http://www.vmware.com/products/player/▶ VMWare Workstation, or other products: http://www.vmware.com/products/workstation/overview.html▶ Windows Server Hyper-V Server 2008 R2: http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx▶ MS Virtual PC: http://www.microsoft.com/windows/virtual-pc/▶ Virtual Box: https://www.virtualbox.org/wiki/Downloads

Terminology	Description
ISO File	<p>This is a disk image—an archive file of an optical disc in a format defined by the International Organization for Standardization (ISO). This contains archived CD/DVD content.</p> <p>In a VM, an ISO file can be treated as a real CD/DVD. All you need to do is to point the CD/DVD settings to the ISO file path.</p> <p>If you need to, you can also burn the ISO to CD/DVD or create ISO files using any CD/DVD image file processing tool, such as:</p> <ul style="list-style-type: none"> ▶ PowerISO: http://www.poweriso.com/ ▶ MagicISO: http://www.magiciso.com/ ▶ FreeISO Creator: http://www.minidvdsoft.com/isocreator/ ▶ Nero Burning Software: http://www.nero.com/enu/
Service Account	This is the account used to run services running on a Windows operating system.

Downloading software

We will use VMWare Player, a free virtual machine application, and the trial versions for Windows Server 2008 R2, SQL Server 2012, Windows Management Framework, and optionally Visual Studio 2010.

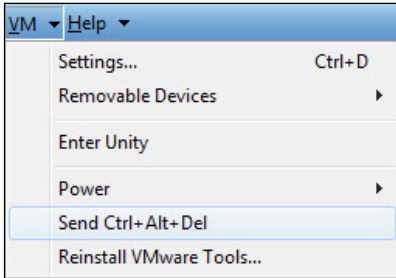
- ▶ Download and install VMWare Player from the following URL:
<http://www.vmware.com/products/player/>
 You can find the VMWare Player documentation at http://www.vmware.com/support/pubs/player_pubs.html.
- ▶ Download Windows Server 2008 R2 with SP1 trial version ISO file (or if you have a licensed copy, use that) from the following URL:
<http://technet.microsoft.com/en-us/evalcenter/dd459137.aspx>
- ▶ Download SQL Server 2012 trial version ISO file (or if you have a licensed copy, use that) from:
<http://www.microsoft.com/en-us/download/details.aspx?id=29066>

- ▶ Download the Windows Management Framework. As we will be running Windows Server 2008 R2, you will need to download the 64-bit version of the `WINDOWS6.1-KB2506143-x64.MSU` file from the following URL:
<http://www.microsoft.com/en-us/download/details.aspx?id=34595>
(Optional) If you are planning to create some SQLCLR assemblies, you will need at least Visual Studio 2010 Professional. SQL Server Data Tools (previously known as BIDS, or Business Intelligence Development Studio) is not sufficient for creating the assemblies.
- ▶ Download the Visual Studio 2010 trial version ISO (or if you have a licensed copy, use that) from the following URL:
<http://www.microsoft.com/en-us/download/details.aspx?id=12187>

VM details and accounts

The following table lists accounts and VM details that we will use in this appendix:

Item	Description
Virtual Machine Name	SQL2012VM
Virtual Machine Computer Name	KERRIGAN
Virtual Machine Computer Administrator Account	UserName: Administrator Password: P@ssword
SQL Server Instances	Default: KERRIGAN Named: KERRIGAN\SQL01
SQL Server Service Account	UserName: QUERYWORKS\sqlservice Password: P@ssword
SQL Server Agent Account	UserName: QUERYWORKS\sqlagent Password: P@ssword
Additional domain accounts	QUERYWORKS\aterra QUERYWORKS\jraynor QUERYWORKS\mhorner

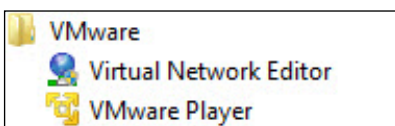
One more note to log in to the VM when it's ready:	
Item	Description
Logging in to the VM	<p>Click on VM Send Ctrl + Alt + Del</p>  <p>Additional VMWare shortcuts can be found here: http://www.vmware.com/support/ws55/doc/ws_learning_keyboard_shortcuts.html</p>

Creating an empty virtual machine

Determine first if your host is a 64-bit machine. You can go to **Start** | **All Programs** | **Accessories** | **System Tools** | **System Information**. You should see it under **System Type**. If you see **x86**, then you will need to use the 32-bit versions of the software.

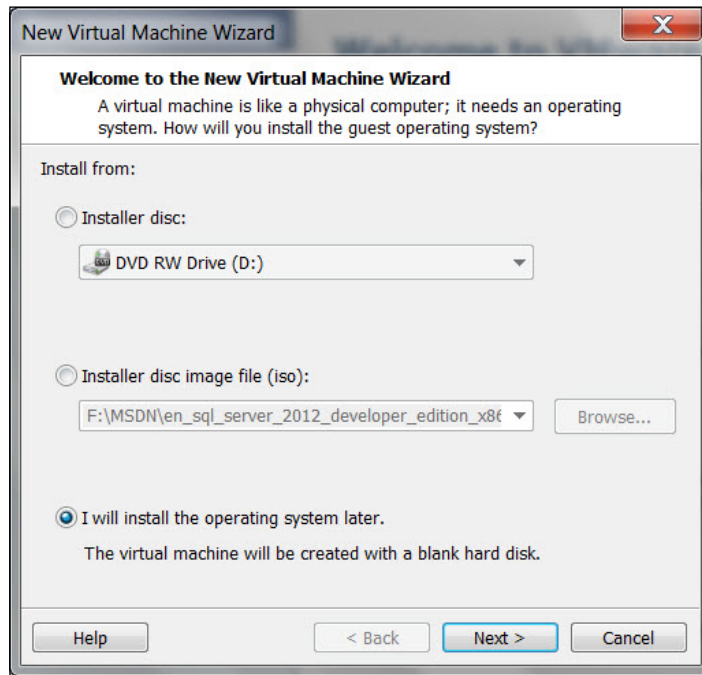
Once ready, we will create our empty virtual machine. We will call our virtual machine SQL2012VM:

1. Launch **VMWare Player**. To do so, go to **Start** | **VMWare** | **VMWare Player**.

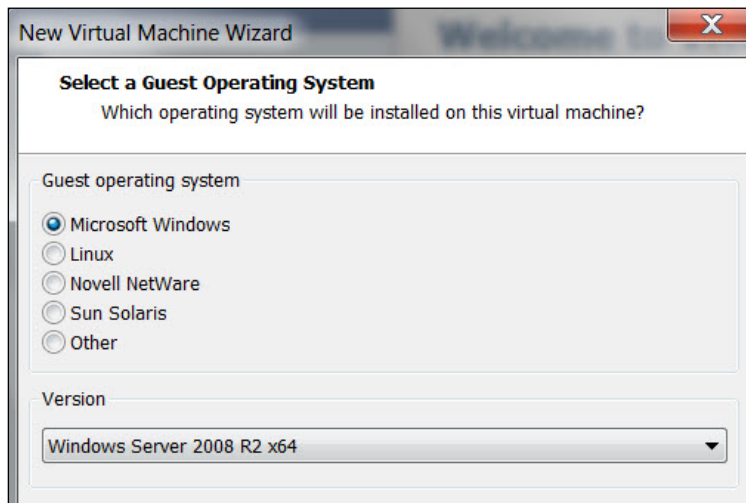


2. On the initial screen, click on the **Create New Virtual Machine** button.
You can also do this by going to **File** | **Create New Virtual Machine**.

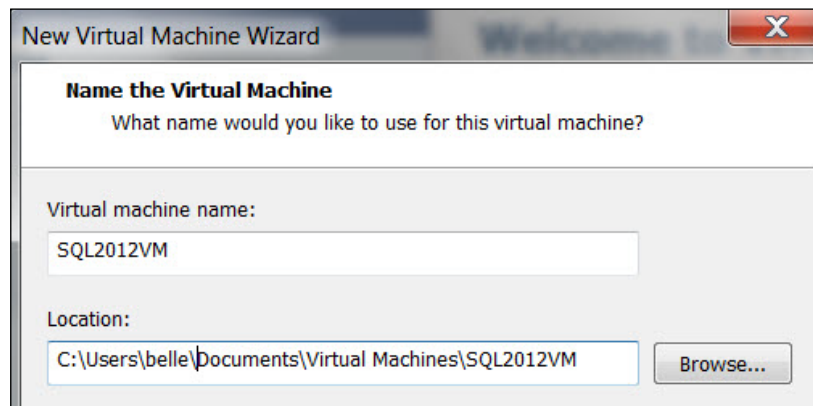
3. In the Welcome screen, select **I will install operating system later** as shown in the following screenshot:



4. On the **Select a Guest Operating System** screen, choose **Microsoft Windows** for the guest operating system, and choose **Windows Server 2008 R2 x64** from the **Version** drop-down menu.



- We will name our virtual machine `SQL2012VM` as shown in the following screenshot. If you prefer, you can also change the location of your VM.



- For our virtual machine, we will allocate 40 GB disk space. Feel free to adjust it as you see fit for your own use. You will want to allocate a bigger disk space if you want to use this VM for some data warehouses and cubes.

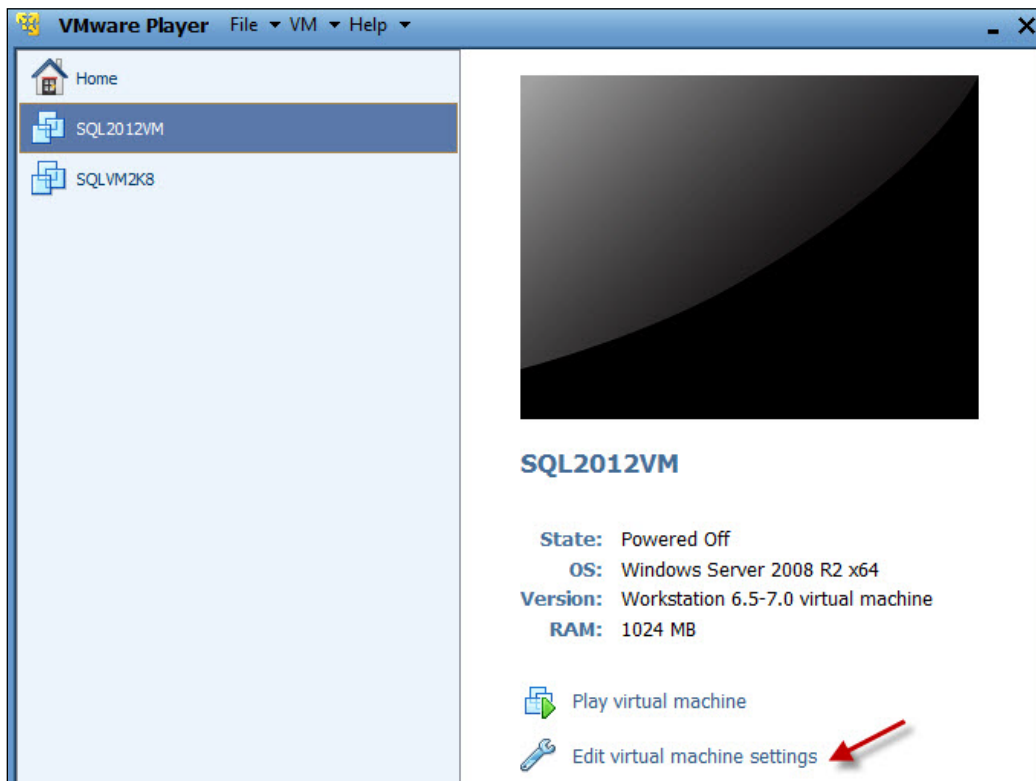


- On the **Ready to Create** screen, click on **Finish**.

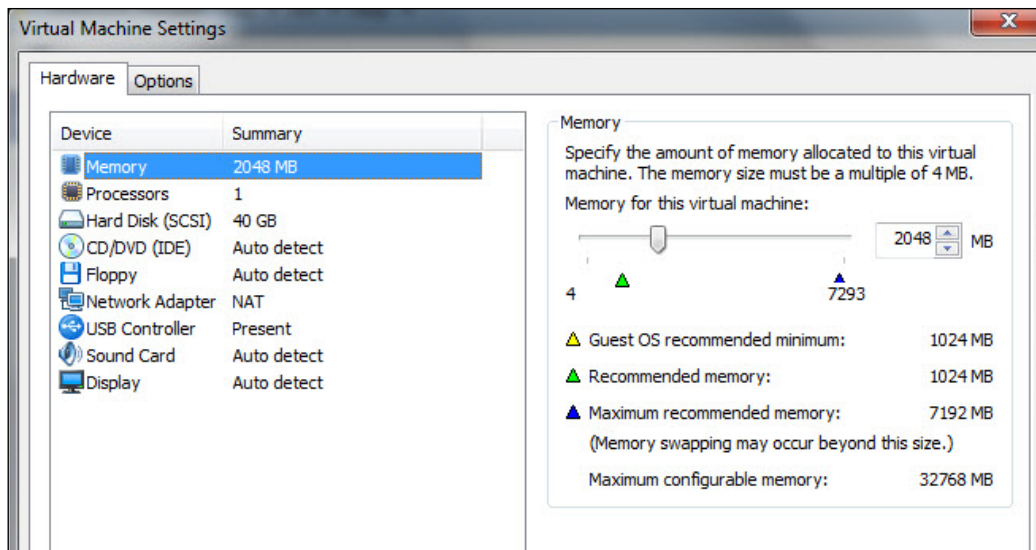
Installing Windows Server 2008 R2 as Guest OS

To install the operating system, we first need to mount the Windows Server ISO and play the virtual machine. After that, we can follow the installation wizard.

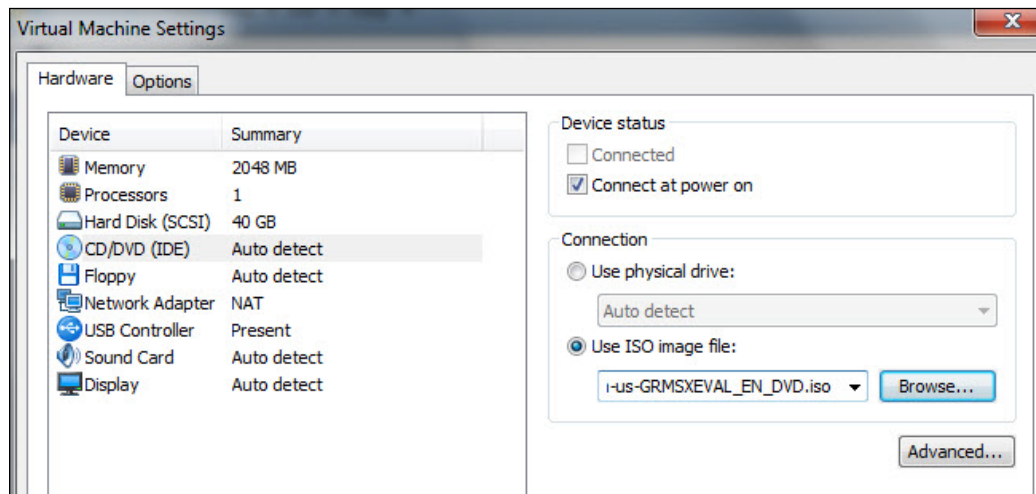
1. Launch VMWare Player. Go to **Start | VMWare | VMWare Player**.
2. Select **SQL2012VM** and then select **Edit virtual machine settings**, as shown in the following screenshot:



3. Let's increase the memory settings—adjust this based on your available hardware configurations. For our purposes we will increase memory to 2 GB (or **2048 MB**), but you can definitely set this higher if you wish. Just make sure you have enough memory still left for your host OS, and other VMs that you may be running simultaneously.

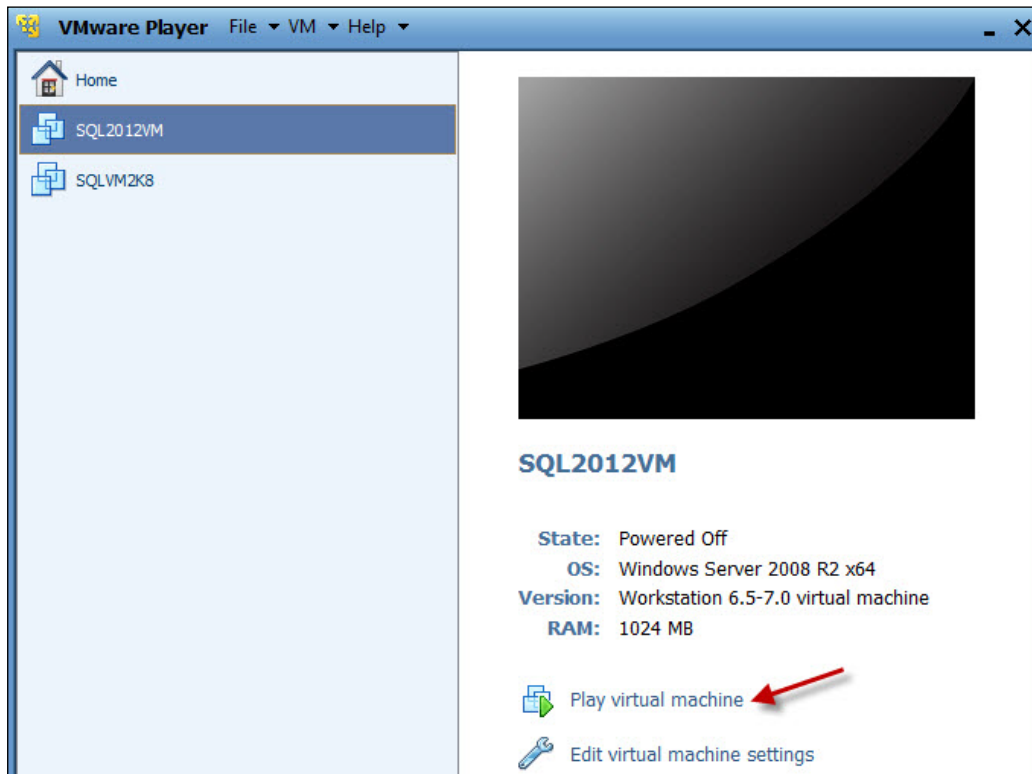


4. Select **CD/DVD**, and choose the **Use ISO image file** radio button. Navigate to Windows Server 2008 R2 ISO, and click on **OK**.



Creating a SQL Server VM

5. Go back to the main VMWare Player screen and, while **SQL2012VM** is selected, click on **Play Virtual Machine**.

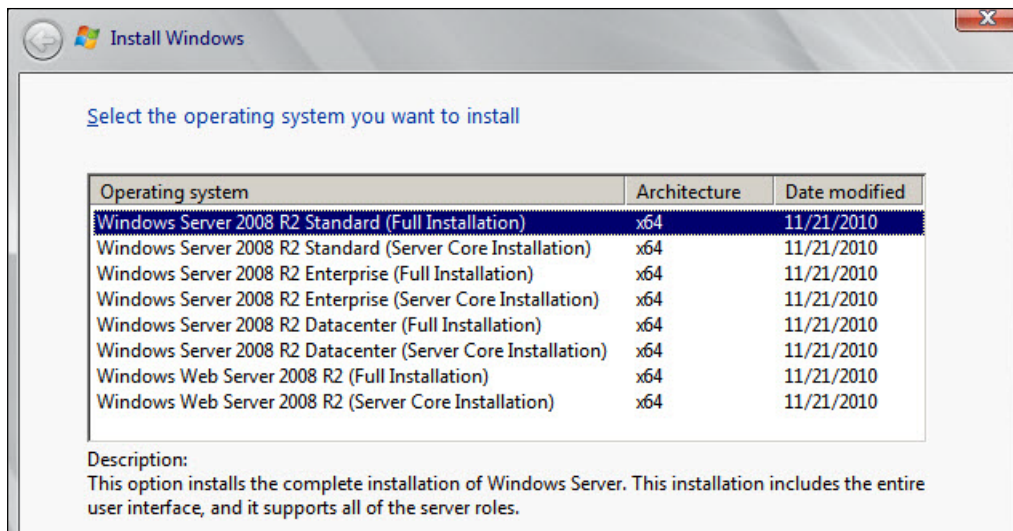


6. Since we have mounted the ISO, the Windows Server installation screen will be displayed when the VM starts. Now we will need to follow the installation for Windows Server 2008 R2.

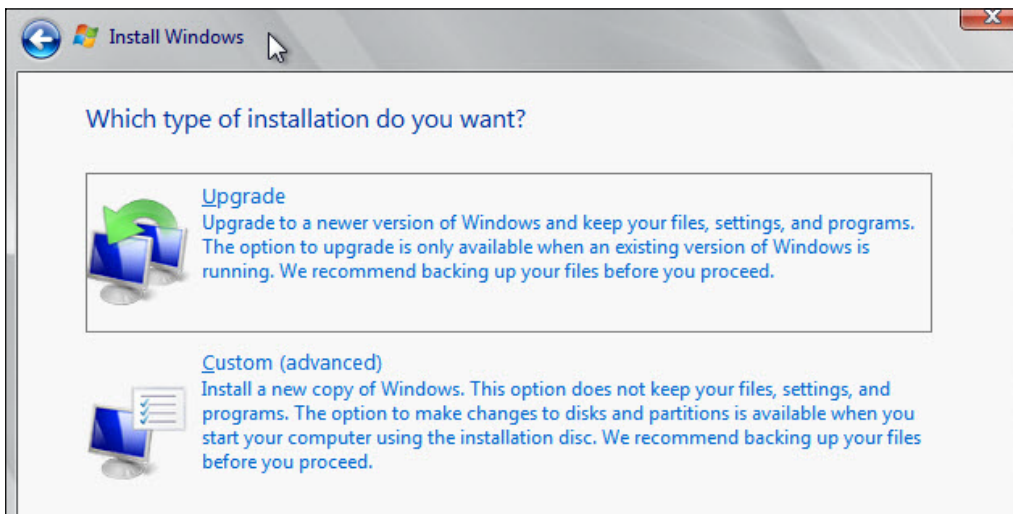
For the installation language we will use **English**, and keyboard will be **US**.

When prompted to install, select **Install Now**.

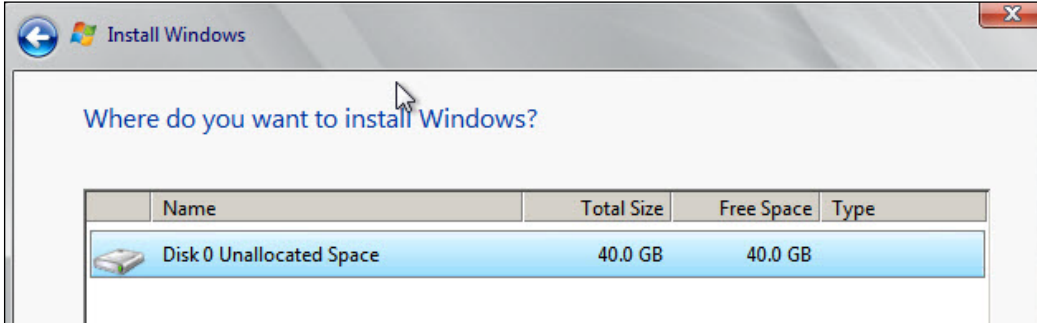
7. When asked about the operating system to install we will choose **Windows Server 2008 R2 Standard x64**, but feel free to choose a different edition that you want to explore.



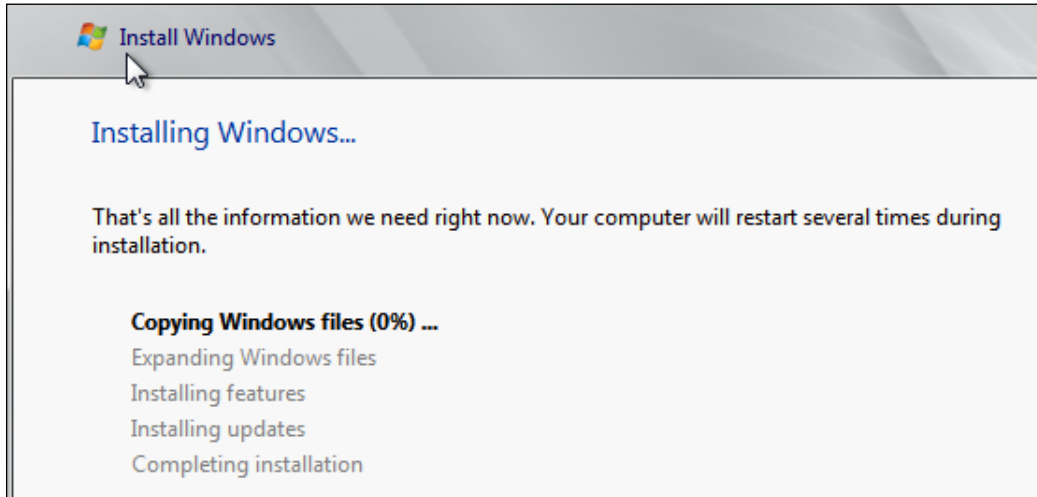
8. Accept the license terms, and click on **Next**.
9. When prompted for the type of installation, select **Custom**.



10. In the **Install Windows** dialog, select **Disk 0 Unallocated space**.



11. Let the installation complete. Note that the VM will be restarted a few times by the installation process.



12. In one of the restarts you will be prompted to change the password.

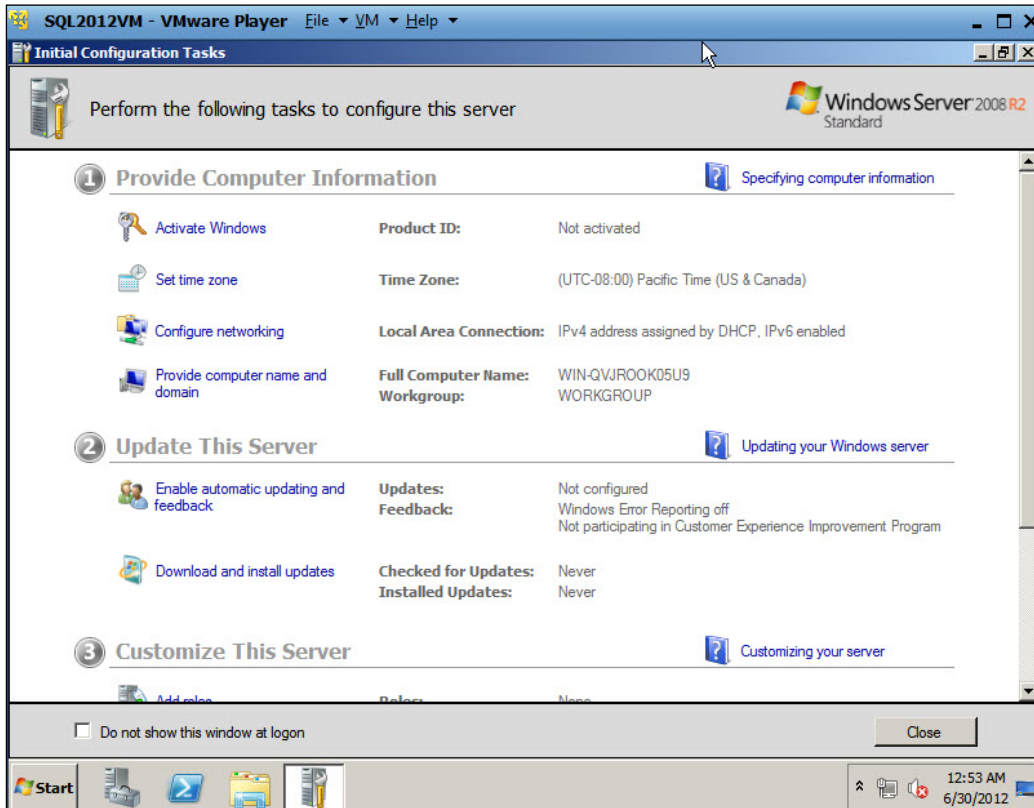


Provide the administrator password as shown in the following screenshot:

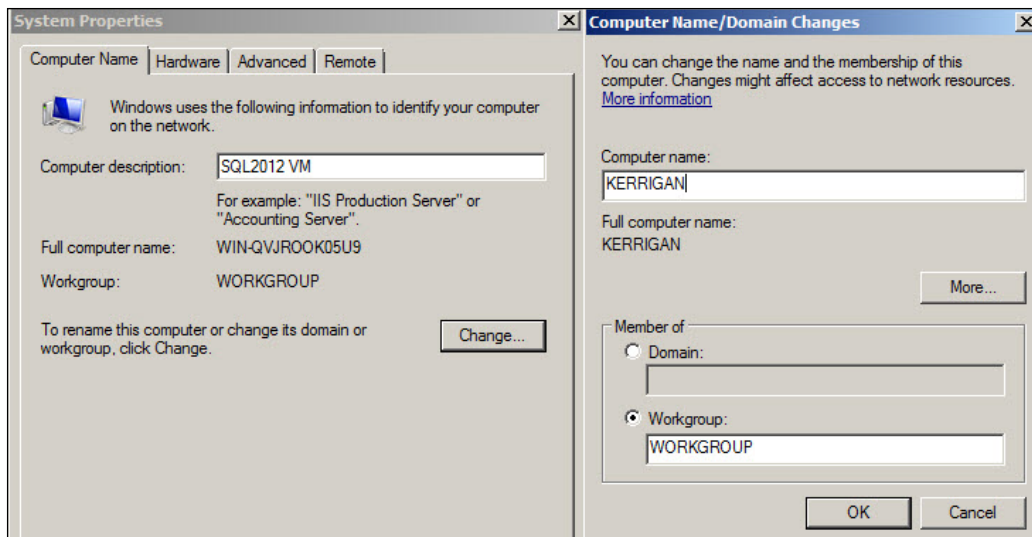


When done, click on the arrow. This will log you in to your new VM.

13. By default, the **Initial Configuration Tasks** screen will be displayed when you first log in.

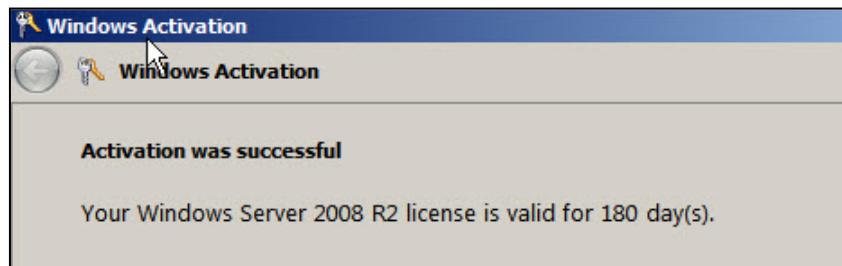


14. Under the **Provide Computer Information** section, click on **Provide computer name and domain** and set the following options:
 - a. In the **Computer description**, type SQL2012 VM.
 - b. Click on the **Change** button.
 - c. In the **Computer name** textbox, type KERRIGAN.



Click on **OK**, and then **Apply**. You will be prompted to restart the VM; choose **Restart Later**.

15. Activate windows. Leave the **Serial Number** textbox blank, and click **Activate**.



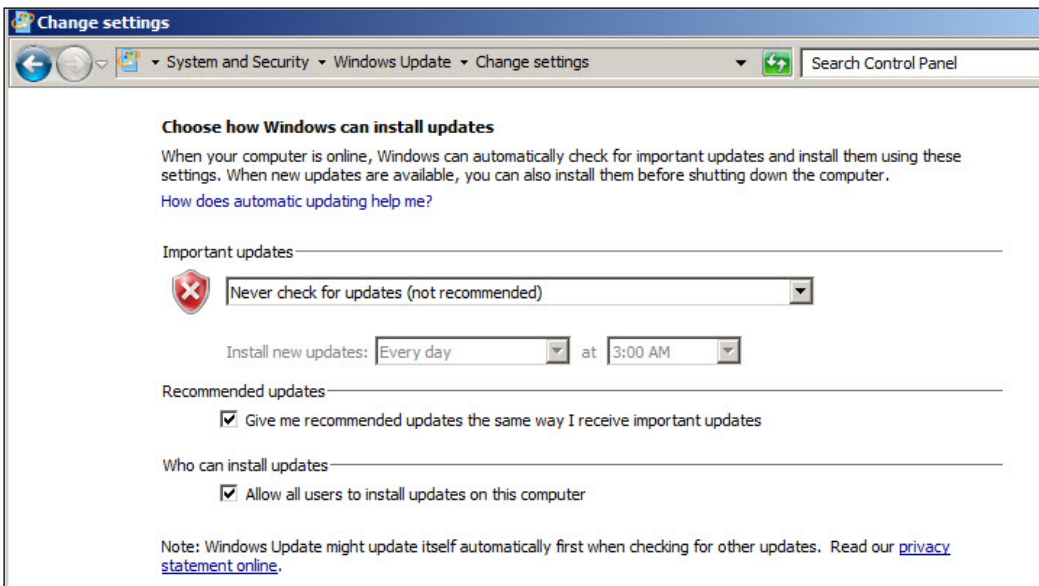
16. Restart the VM.
17. We are almost ready. We just need to install the updates. Go to **Start | Windows Update**.



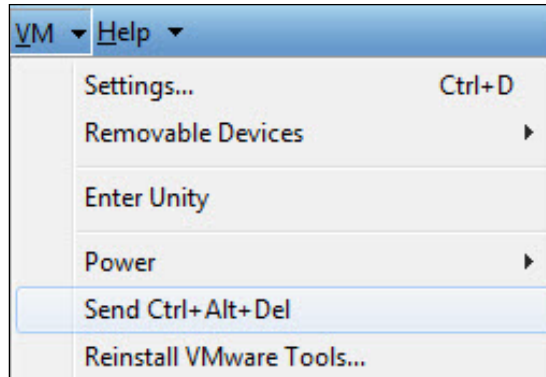
18. Click on **Let me choose my settings**, the link below **Turn on automatic updates**. By doing this, we will disable automatic, ongoing updates for this VM.



19. Under **Important Updates**, choose **Never check for updates** and click on **OK**.



20. Click on **check for updates** and install all the critical and relevant updates.
When asked to install Internet Explorer 9, install it.
21. When prompted to restart, click on **OK**. Once the VM has restarted, log in using **VM | Send Ctrl + Alt + Delete**.

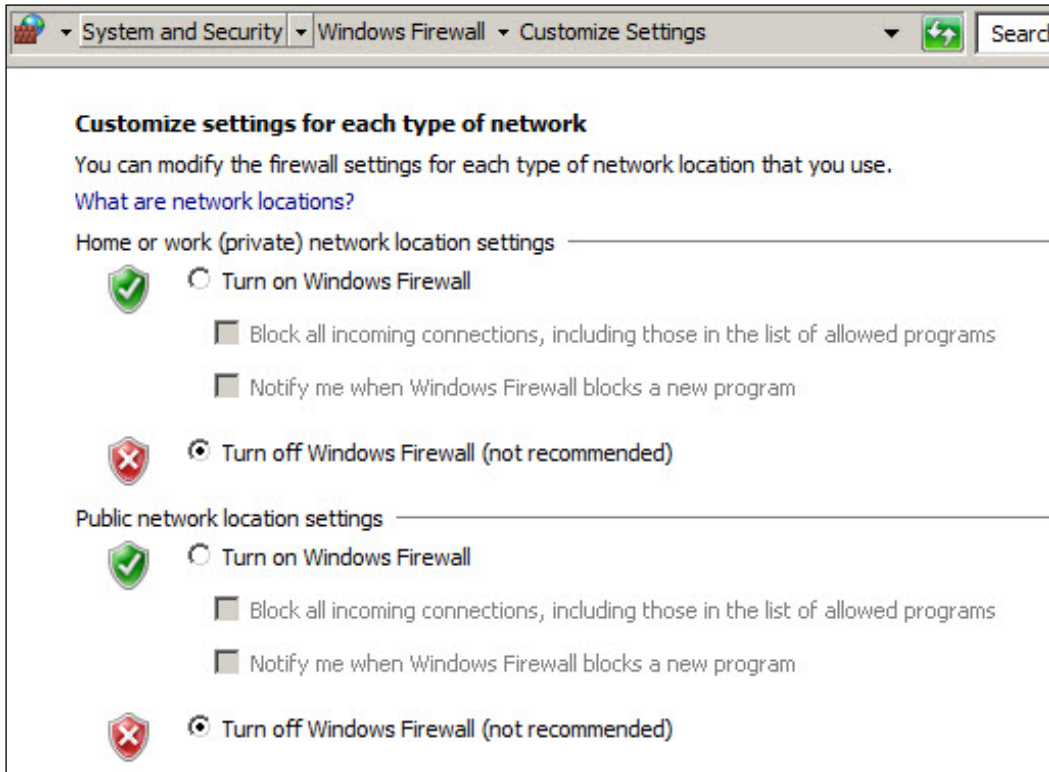


22. Now let's disable the firewall. We are only doing this for our development VM. Go to **Administrative Tools | Server Manager**.
23. Under the section **Customize This Server** go to **Configure Windows Firewall** and disable the firewall.



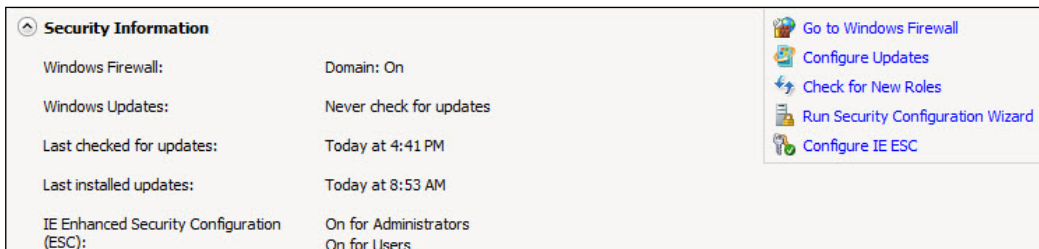
24. On the left-hand side pane, select **Turn Windows Firewall on or off**.

25. In the **Customize Settings** dialog, choose to turn off Windows firewall for both private and public networks.

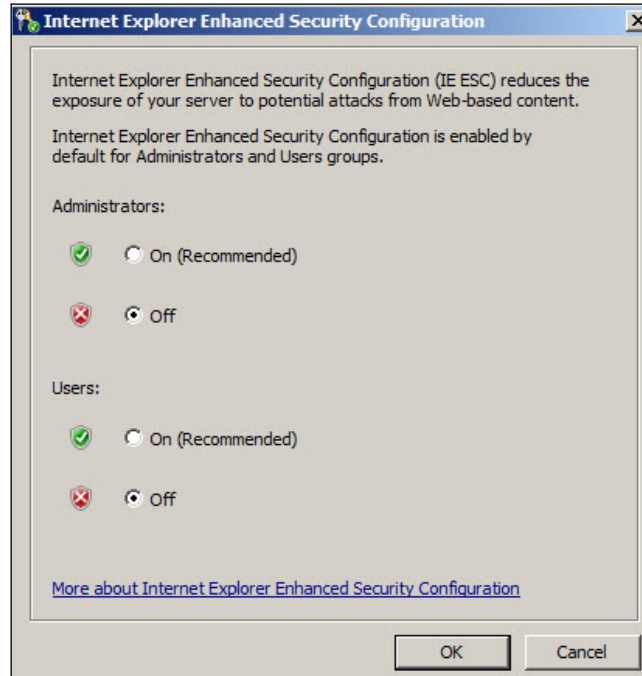


26. Now let's disable **IE Enhanced Security Configuration**. Go to **Start | Administrative Tools | Server Manager**.

27. Under the **Security Information** section, click on **Configure IE ESC** located on the top-right box, as shown in the following screenshot:



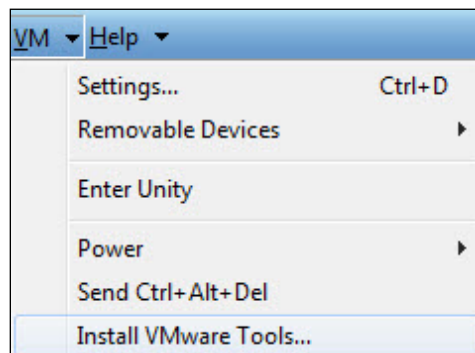
28. Select **Off** for both **Administrators** and **Users**. Click on **OK**.



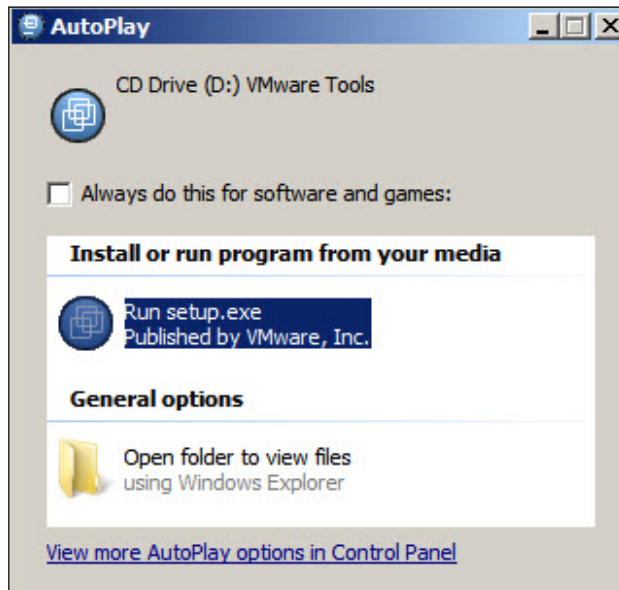
Installing VMWare tools

For an enhanced VM experience, we want to install VMWare tools.

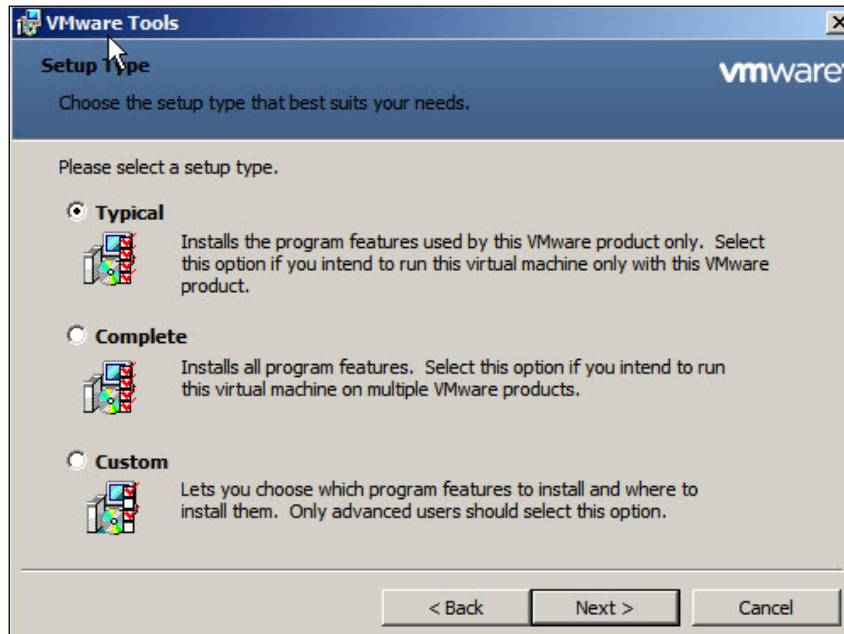
1. Launch **VMWare Player**.
2. In the **VM** menu, select **Install VMWare Tools**.



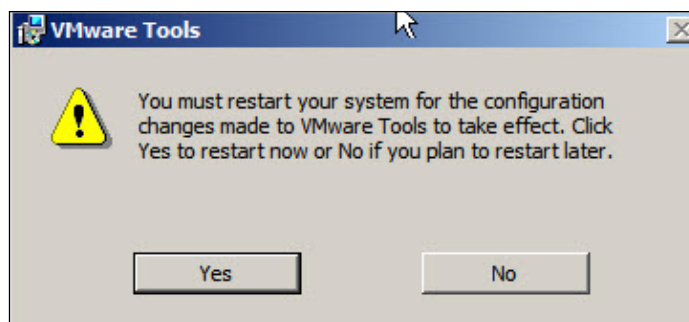
- When the **AutoPlay** dialog comes up, click on **Run setup.exe** as shown in the following screenshot:



- Select the **Typical** setup type, and click on **Next**.



5. Follow the wizard to completion.
6. Once the installation is done, you will be prompted to restart. Click on **Yes**.



Configuring a domain controller

In a production environment, it is not recommended to install the domain controller with any of your other server software.



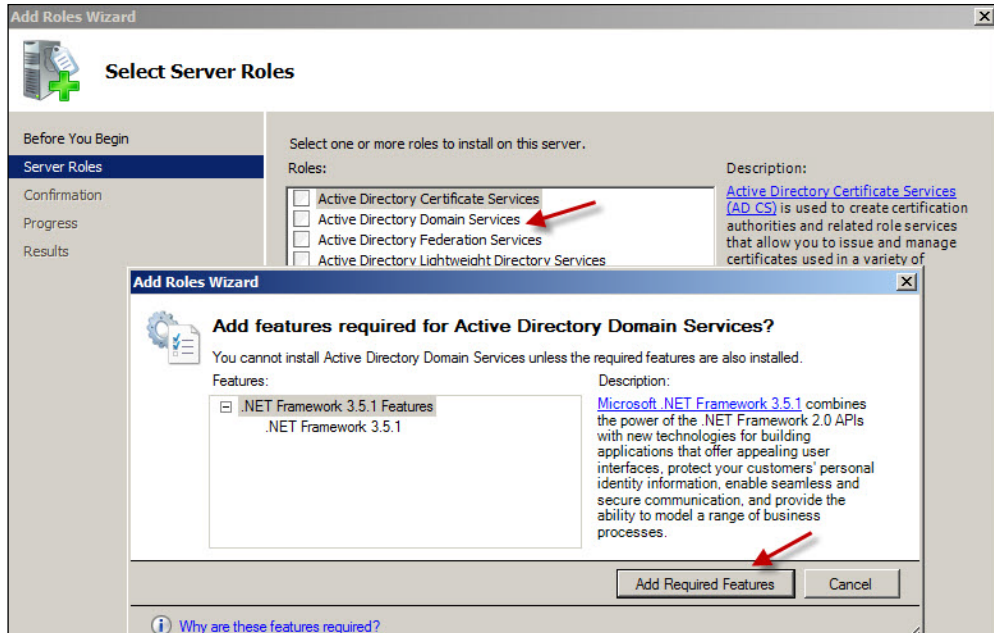
Note that this section is optional. You do not need to configure a domain controller to start using the recipes in this book.

For development and testing purposes (such as ours), however, we will install them on the same machine. Should you want to mimic a production setup, you can create another Windows Server 2008 R2 VM with a different computer name, and follow the steps given:

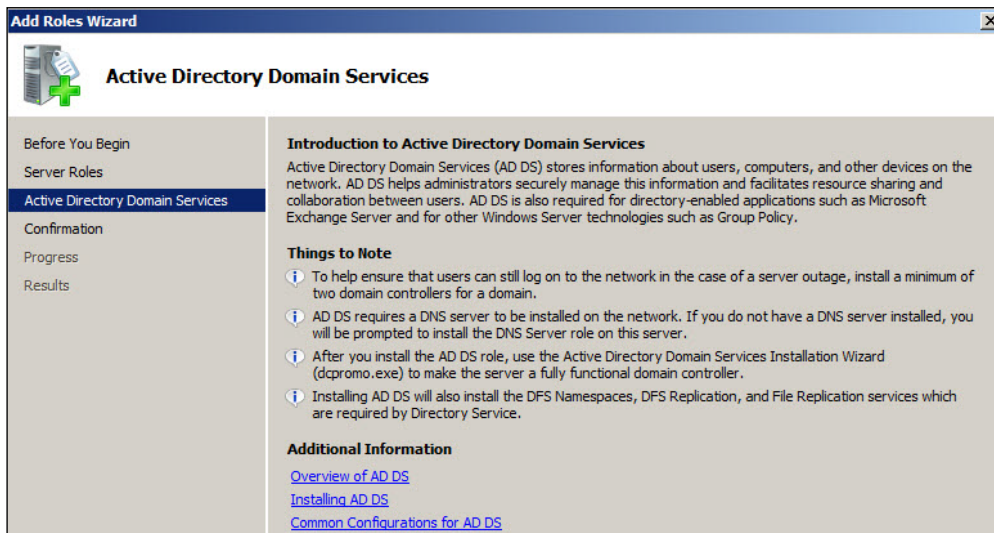
1. Launch **VMWare Player**.
2. Play **SQL2012 VM**.
3. Log in to our VM, **KERRIGAN**.
4. Go to **Administrative Tools | Server Manager**.



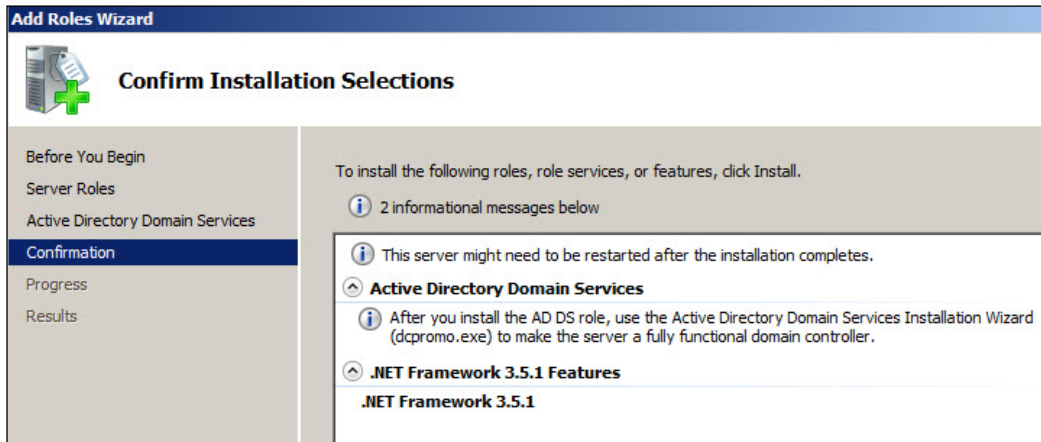
5. Click on **Add roles**. This will launch the **Add Roles Wizard** dialog.
6. In the **Select Server Roles** screen, choose **Active Directory Domain Services**. This action will trigger the display of another window prompting if you want to install required features. Click on **Add Required Features**.



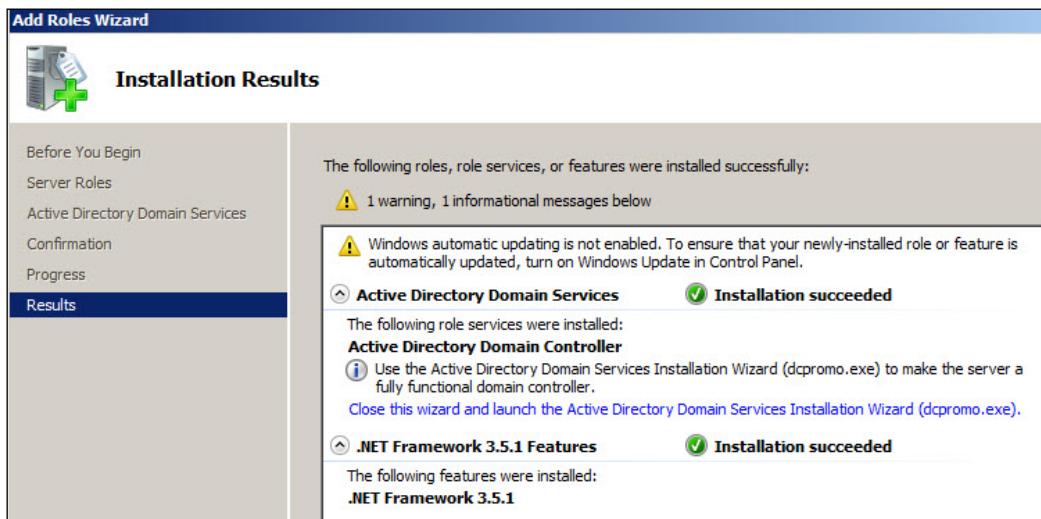
7. Click on **Next** in the **Introduction to Active Directory Domain Services** window.



8. Click on **Next** in the **Confirm Installation Selections** window.



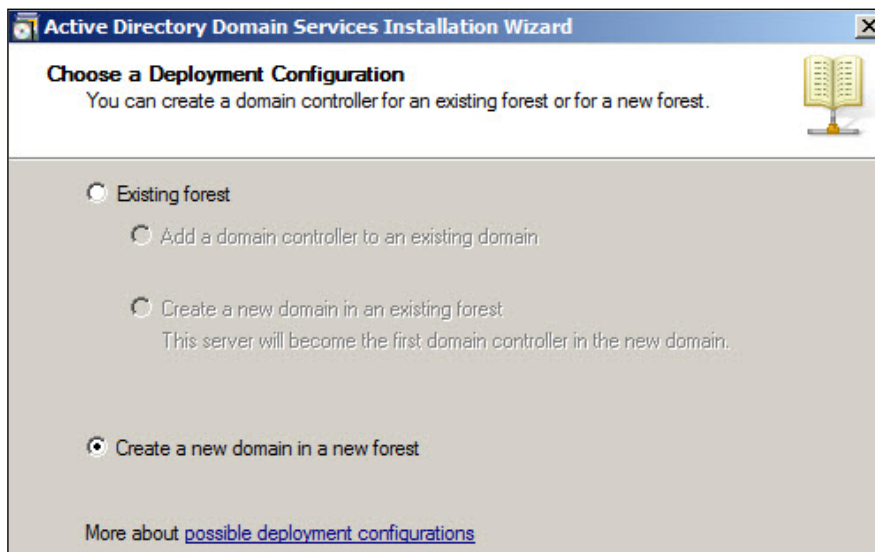
9. When done, view the results. Note that you will get a warning because we disabled automatic updates. For our purposes, this is acceptable.



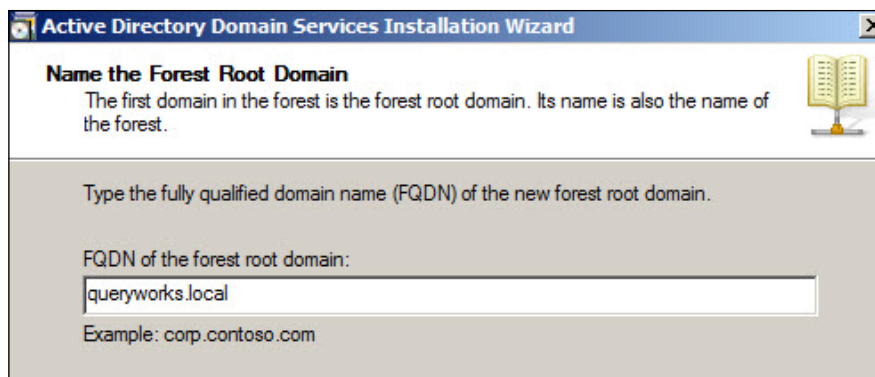
10. To configure the domain, go to **Start** and in the textbox type `dcpromo` and press *Enter*. This will start the **Active Directory Domain Services Installation Wizard** application.



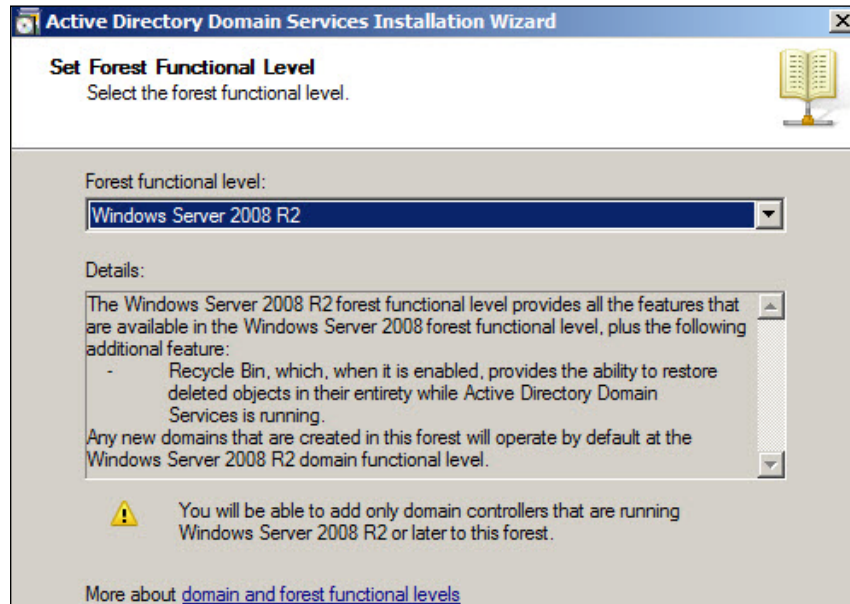
11. Click on **Next** on the wizard until you hit the **Deployment Configuration** window. Select **Create a new domain in a new forest**.



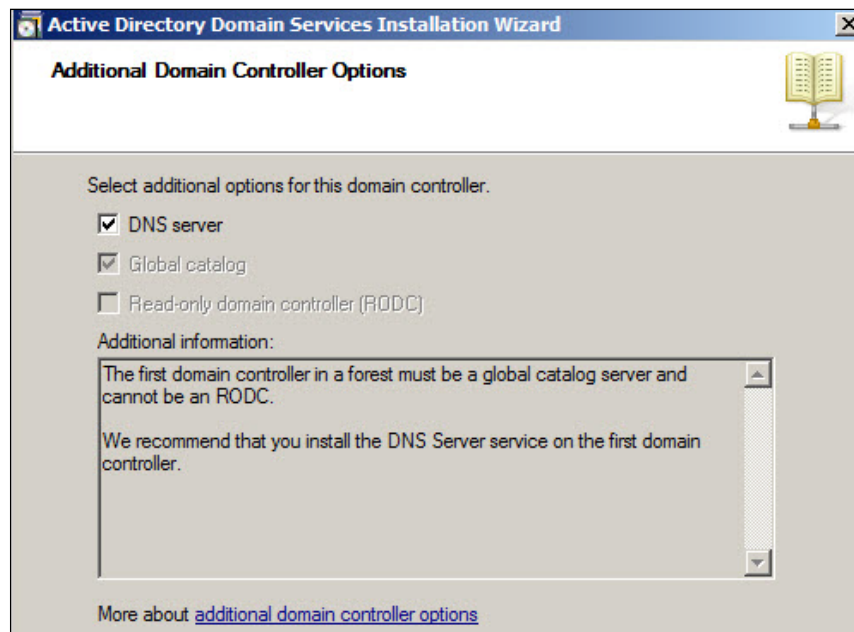
12. In the **Name the Forest Root Domain** window, type `queryworks.local` in the **FQDN of the forest root domain** textbox.



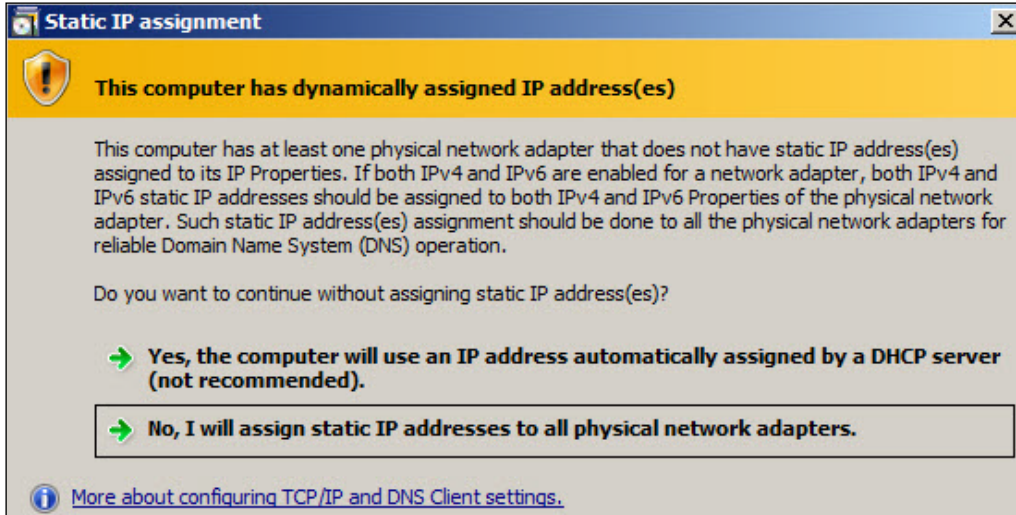
13. For **Forest functional level**, select Windows Server 2008 R2 and click on **Next**.



14. In the **Additional Domain Controller Options** window, select **DNS server** as shown in the following screenshot:



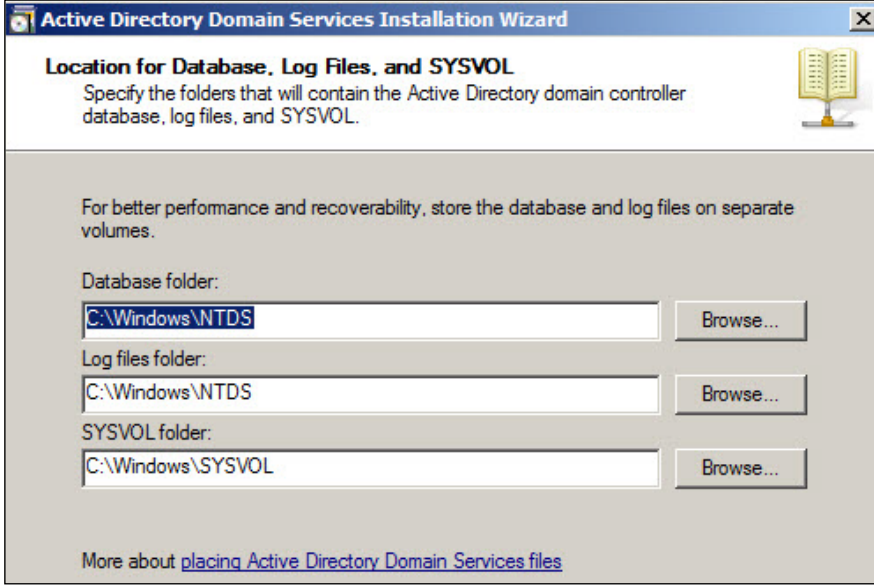
15. You will receive a warning about the computer having a dynamically assigned IP address. Click on **Yes, the computer will use an IP address automatically assigned by a DHCP server**. For our purposes, this is acceptable. Note that if this option is chosen, it is assumed that a DHCP server is already available on the network. Otherwise, the server will use an **APIPA** scheme, or **Automatic Private IP Addressing**, which does not work well with domain controllers.



16. Another warning dialog will appear, informing you that delegation for the DNS server cannot be created. Click on **Yes** to continue.

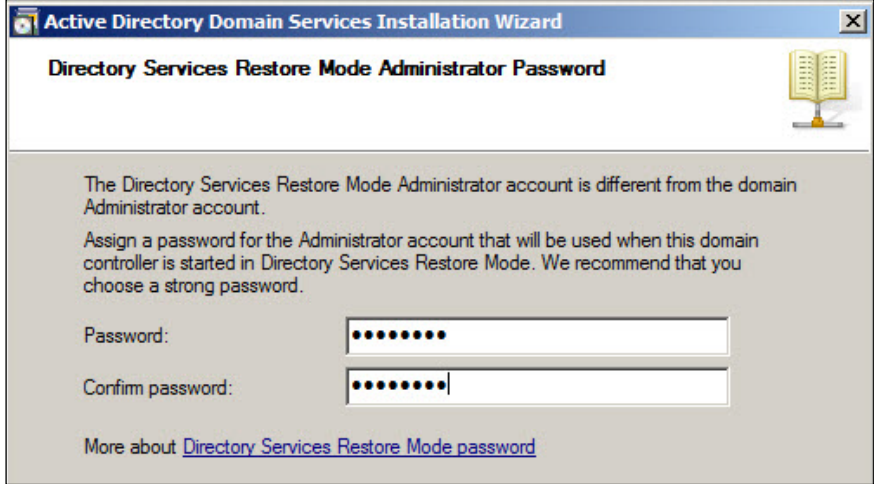


17. Accept the default folders for **Location for database, log files, and SYSVOL**.



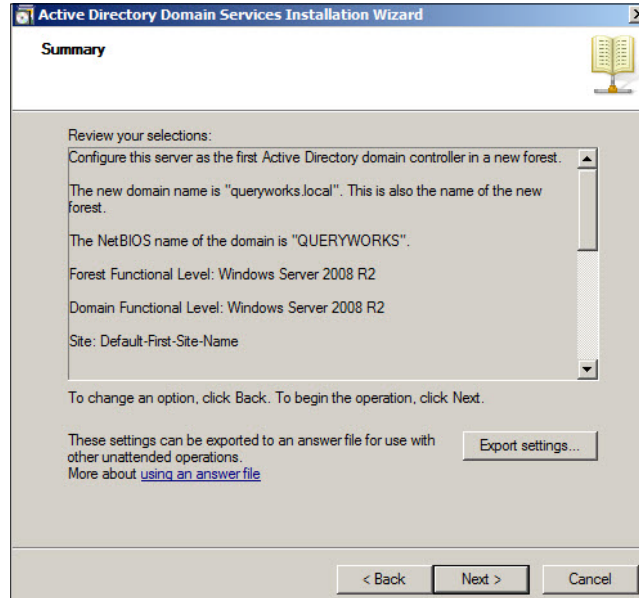
The screenshot shows the 'Active Directory Domain Services Installation Wizard' window. The title bar reads 'Active Directory Domain Services Installation Wizard'. The main heading is 'Location for Database, Log Files, and SYSVOL'. Below the heading, it says 'Specify the folders that will contain the Active Directory domain controller database, log files, and SYSVOL.' There is a small icon of an open book on the right. The main content area has a grey background and contains the following text: 'For better performance and recoverability, store the database and log files on separate volumes.' Below this, there are three rows of input fields, each with a 'Browse...' button to its right. The first row is 'Database folder:' with the text 'C:\Windows\NTDS'. The second row is 'Log files folder:' with the text 'C:\Windows\NTDS'. The third row is 'SYSVOL folder:' with the text 'C:\Windows\SYSVOL'. At the bottom of the window, there is a link: 'More about [placing Active Directory Domain Services files](#)'.

18. Type the password.

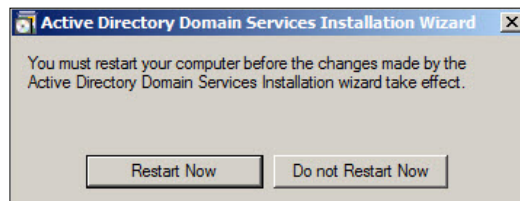


The screenshot shows the 'Active Directory Domain Services Installation Wizard' window. The title bar reads 'Active Directory Domain Services Installation Wizard'. The main heading is 'Directory Services Restore Mode Administrator Password'. Below the heading, it says 'The Directory Services Restore Mode Administrator account is different from the domain Administrator account.' and 'Assign a password for the Administrator account that will be used when this domain controller is started in Directory Services Restore Mode. We recommend that you choose a strong password.' There is a small icon of an open book on the right. Below the text, there are two rows of input fields, each with a 'Password:' or 'Confirm password:' label to its left. Both fields contain a series of dots. At the bottom of the window, there is a link: 'More about [Directory Services Restore Mode password](#)'.

19. Review the **Summary** of your selections.



20. Finish the installation process of the wizard. When done, you will be prompted to restart as shown in the following screenshot:



21. Once the Virtual Machine is back online, you will notice that the login screen no longer shows **KERRIGAN\Administrator**. It should now show **QUERYWORKS\Administrator**.



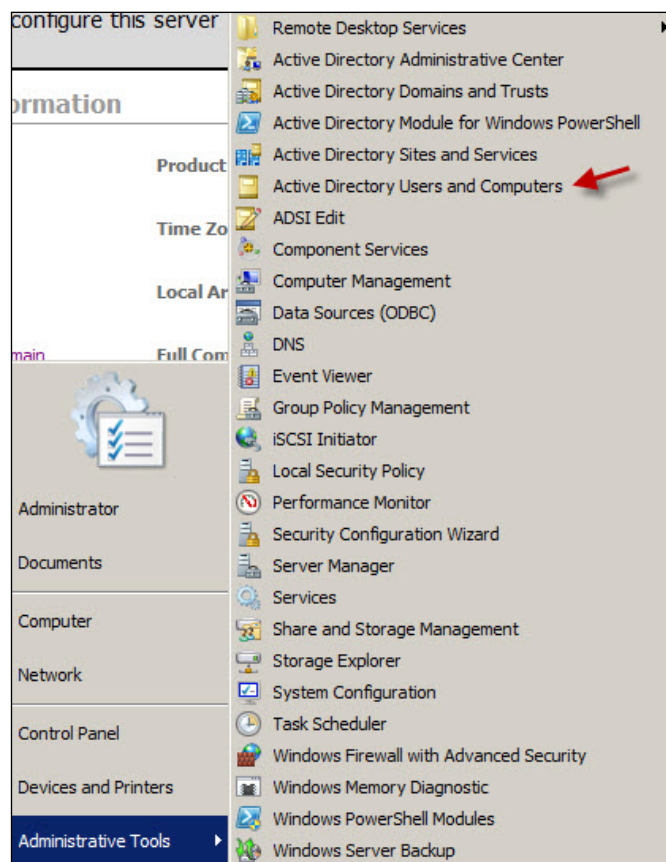
Creating domain accounts

Next, we will create some domain accounts to be used for our exercises. We will create the following:

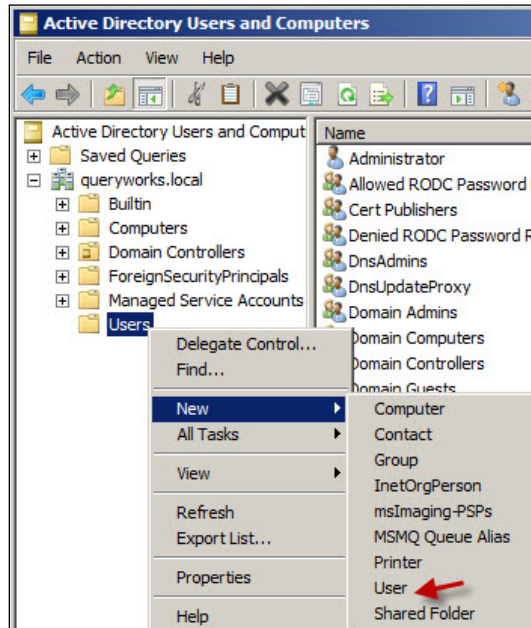
- ▶ QUERYWORKS\sqlservice
- ▶ QUERYWORKS\sqlagent
- ▶ QUERYWORKS\aterra
- ▶ QUERYWORKS\jraynor
- ▶ QUERYWORKS\mhorner

To add these accounts, follow the steps listed below:

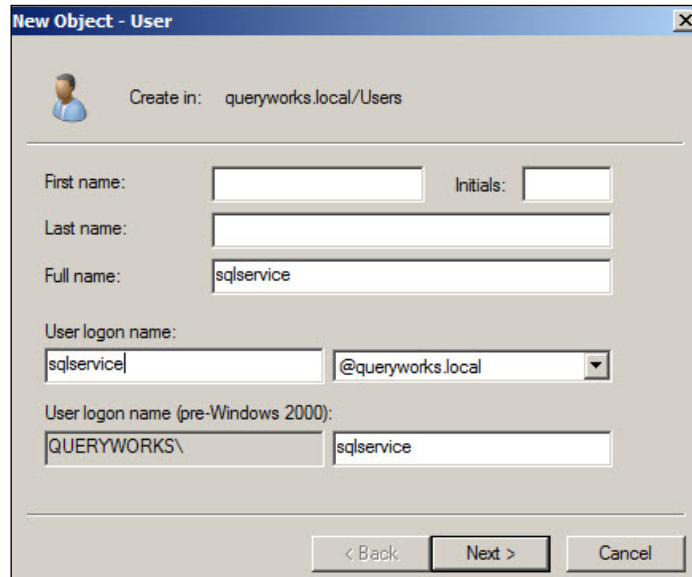
1. Launch **SQL2012VM** and log in.
2. Go to **Start | Administrative Tools | Active Directory Users and Computers**.



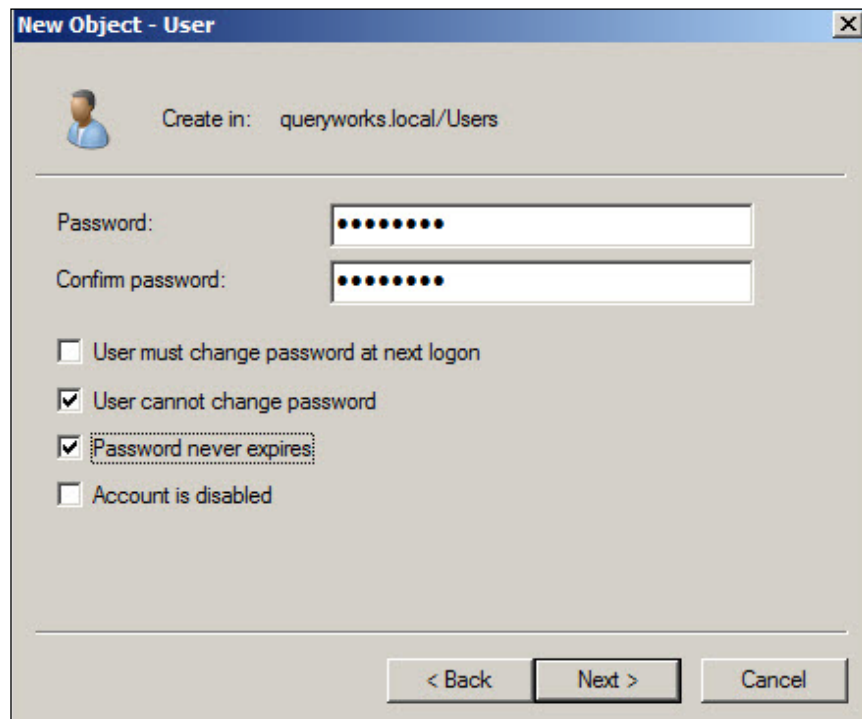
3. In the **Active Directory Users and Computers** window, expand **queryworks.local**. Right-click on **Users** and select **New | User**, as shown in the following screenshot:



4. In **Full name** and **User logon name**, type `sqlservice` and click on **Next** as shown in the following screenshot:



5. Type the password, then check **User cannot change password** and **Password never expires**, as shown in the following screenshot:



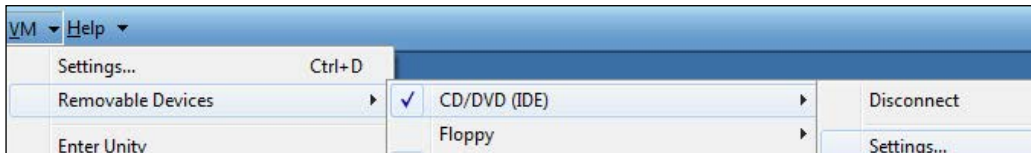
The screenshot shows a dialog box titled "New Object - User". At the top left is a user icon. To its right, it says "Create in: queryworks.local/Users". Below this is a horizontal line. Underneath, there are two password input fields. The first is labeled "Password:" and the second is labeled "Confirm password:". Both fields contain a series of dots. Below the password fields are four checkboxes with the following labels: "User must change password at next logon" (unchecked), "User cannot change password" (checked), "Password never expires" (checked), and "Account is disabled" (unchecked). At the bottom of the dialog are three buttons: "< Back", "Next >", and "Cancel".

6. Click on **Next** and then **Finish**.
7. Repeat steps 3 to 6 for creating the rest of the users:
 - ❑ sqlagent
 - ❑ aterra
 - ❑ jraynor
 - ❑ mhorner

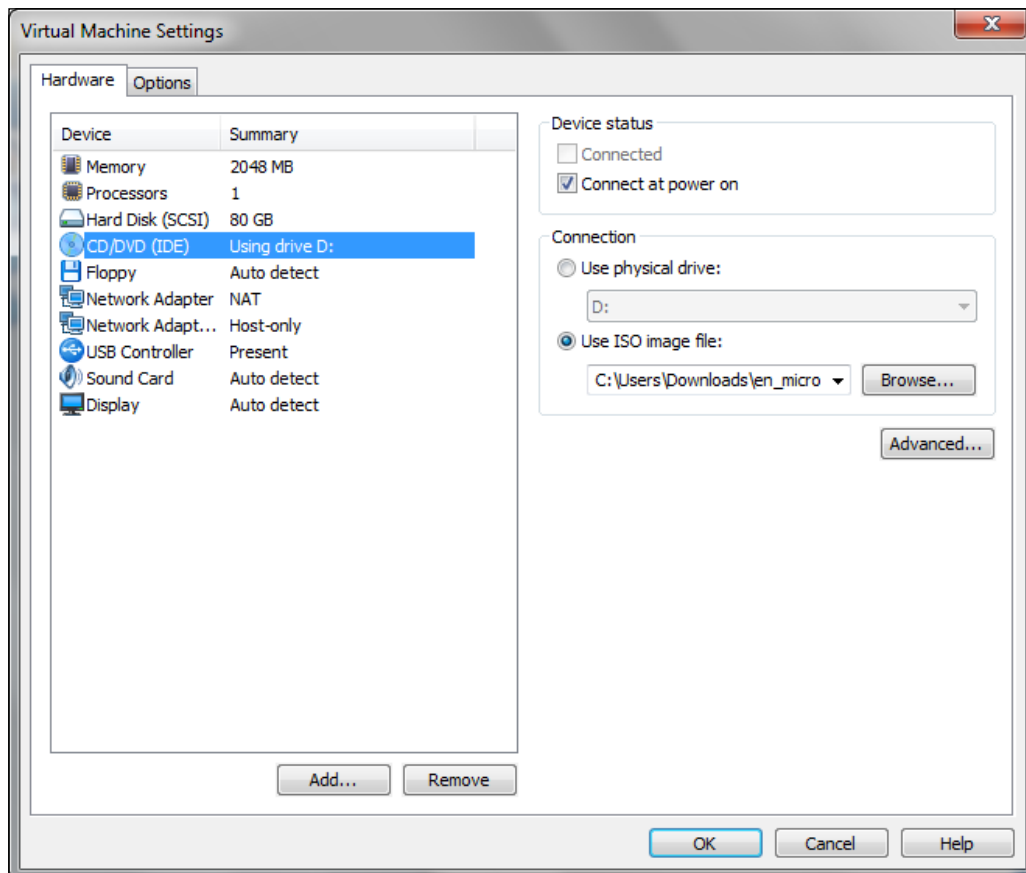
Installing SQL Server 2012 on a VM

We are now ready to install SQL Server 2012. Carry out the following steps:

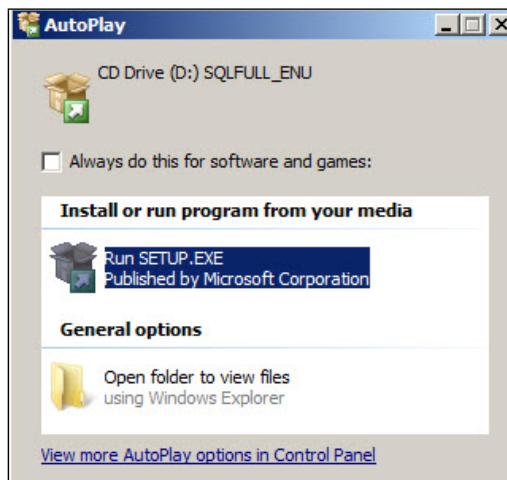
1. Launch **VMWare Player**.
2. Play **SQL2012VM** and login.
3. Go to **VM | Removable Devices | CD/DVD | Settings**.



4. Change the ISO image file path to the SQL Server 2012 ISO file, and click on **OK**.



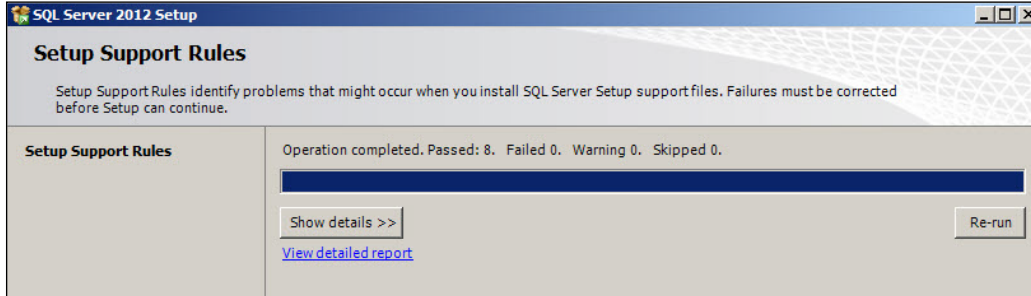
- Once you click on **OK**, the **Autoplay** window should appear. Click on **Run SETUP.EXE** as shown in the following screenshot:



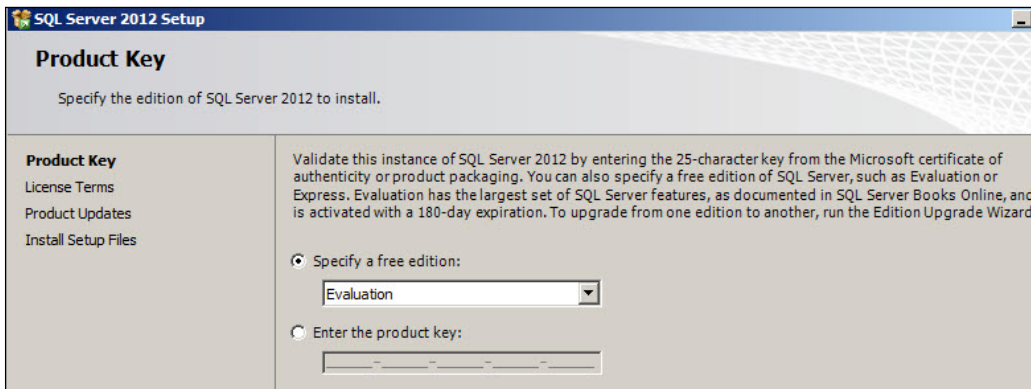
- Select **Installation** from the left-hand pane, and choose **New SQL Server stand-alone installation or add features to an existing installation**, as shown in the following screenshot:



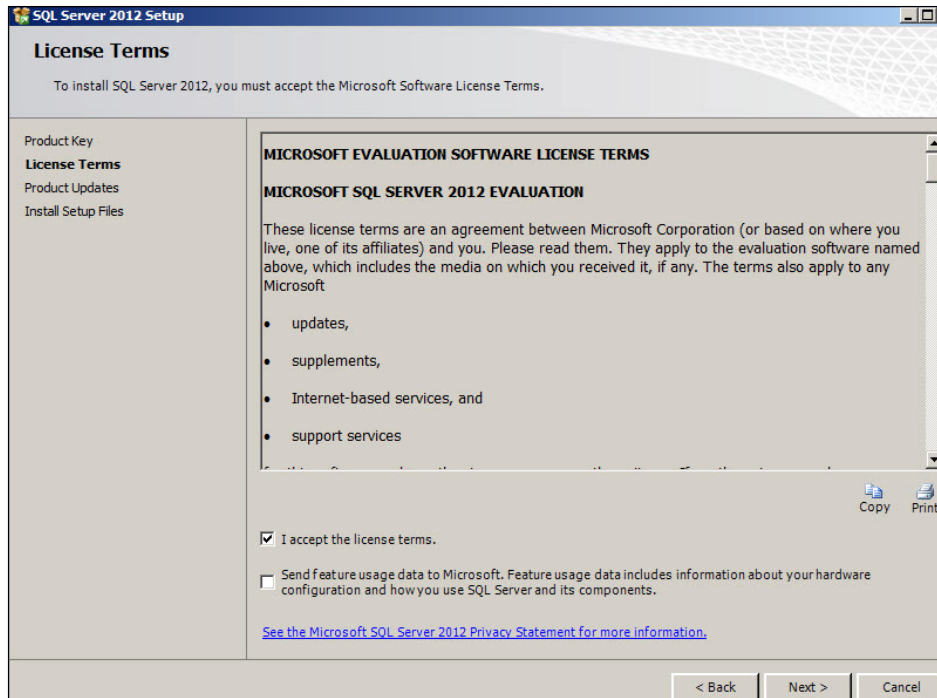
7. Run **Setup Support Rules** and click on **Next**.



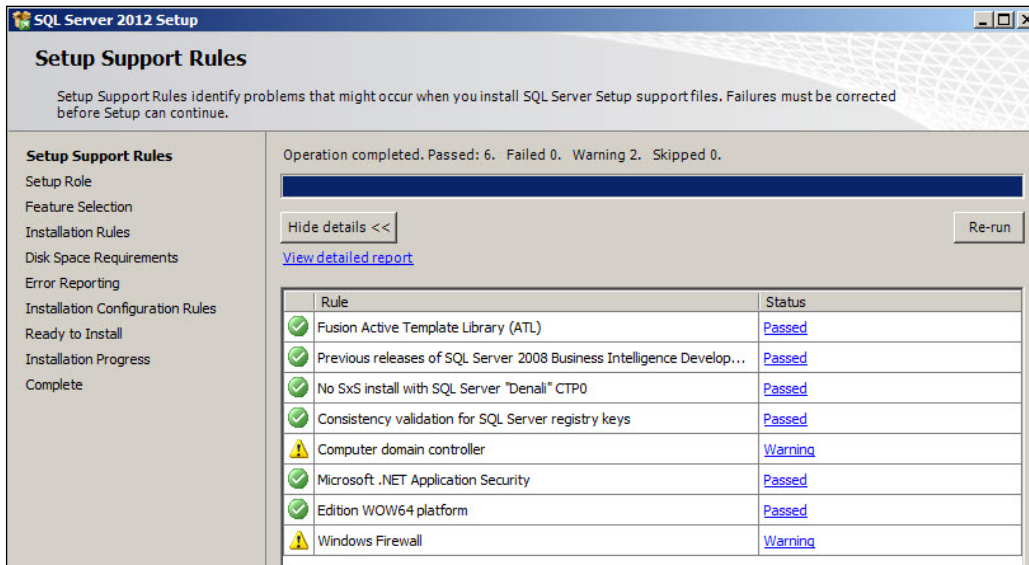
8. For the trial edition, we want to keep the default **Evaluation** product key selected.



9. In the **License Terms** window, select **I accept the license terms**, and click on **Next** as shown in the following screenshot:



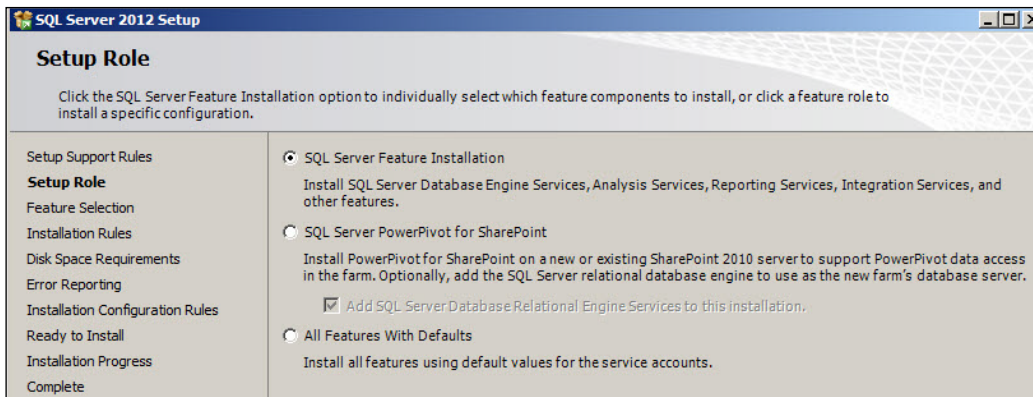
10. Possible issues will be flagged in the next window. For our purposes, we expect to see a warning because of the domain controller and the firewall, as shown in the following screenshot:



For security reasons, it is recommended to not install SQL Server on top of the domain controller, as discussed in the article at <http://msdn.microsoft.com/en-us/library/ms143506.aspx>. For our purposes, this is just a test machine that will not be used as a production box, we can ignore this warning.



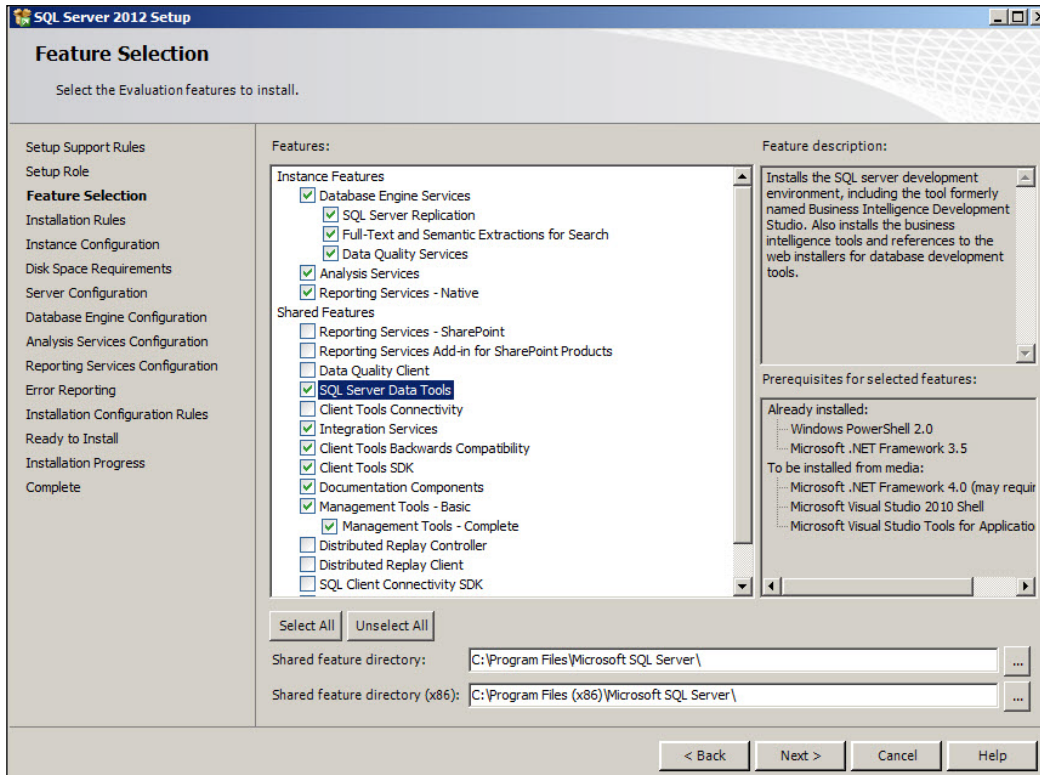
11. In the **Setup Role** screen, select **SQL Server Feature Installation** as shown in the following screenshot:



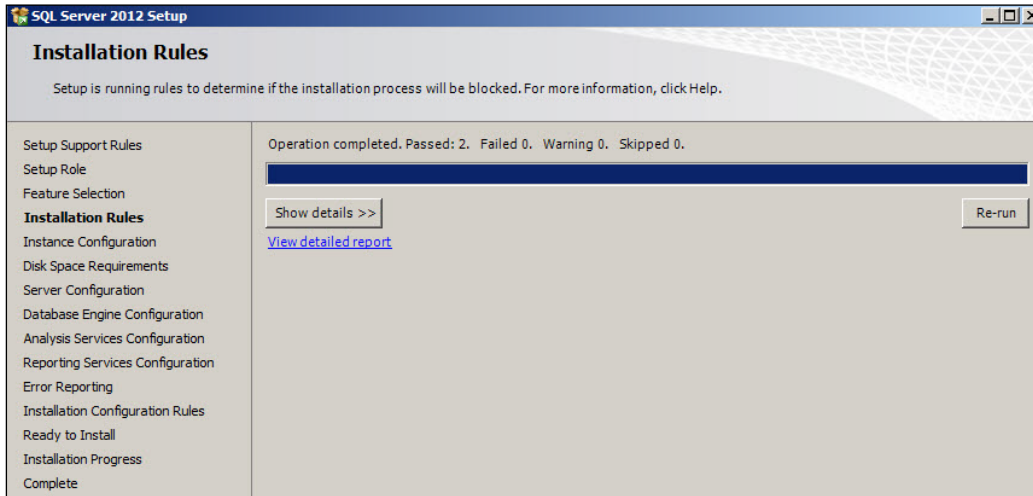
12. In the feature selection, be sure to choose:
- Database Engine Services** (all components)
 - Analysis Services**
 - Reporting Services - Native**
 - SQL Server Data Tools**

- Integration Services**
- Documentation Components**
- Management Tools - Basic**
- Management Tools - Complete**

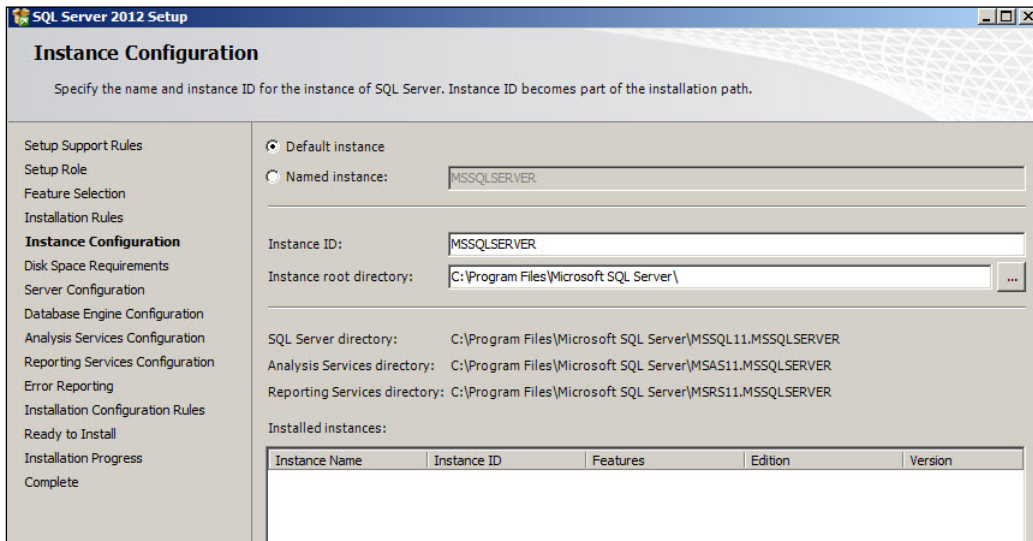
Feel free to choose additional features you want to try and then click on **Next**.



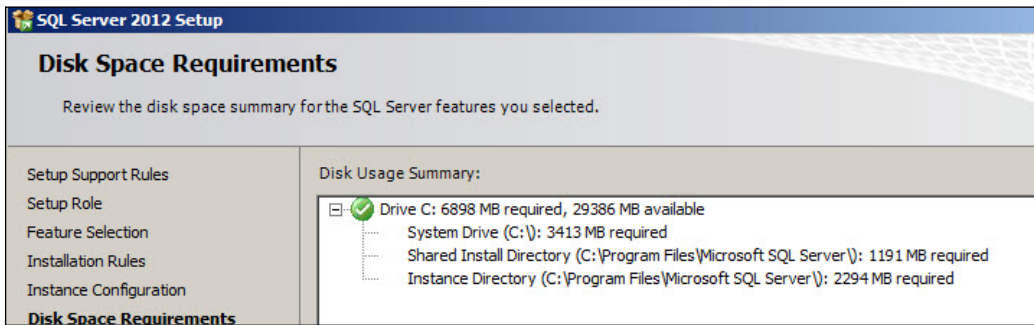
13. The install wizard will now check **Installation Rules**. If there are any errors reported, be sure to resolve them before continuing with the installation. Click on **Next**.



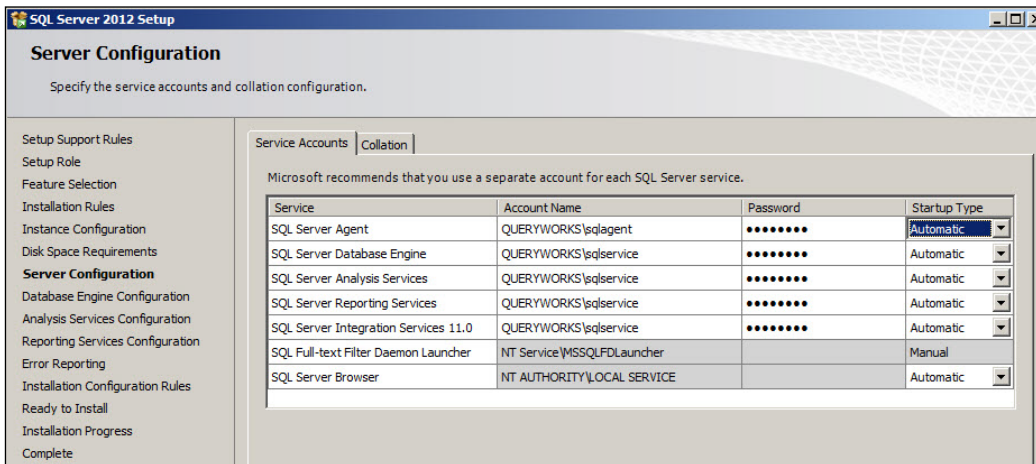
14. In the **Instance Configuration** window, choose to install **Default instance**.



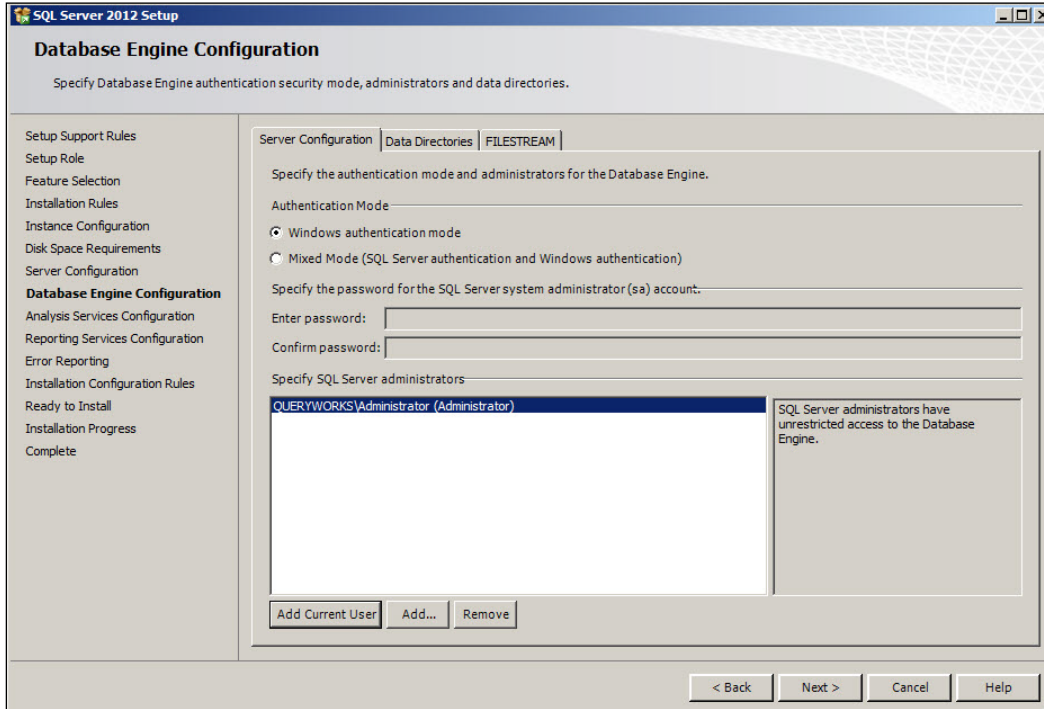
15. The next screen shows the disk space summary for all the features we've chosen so far; click on **Next**.



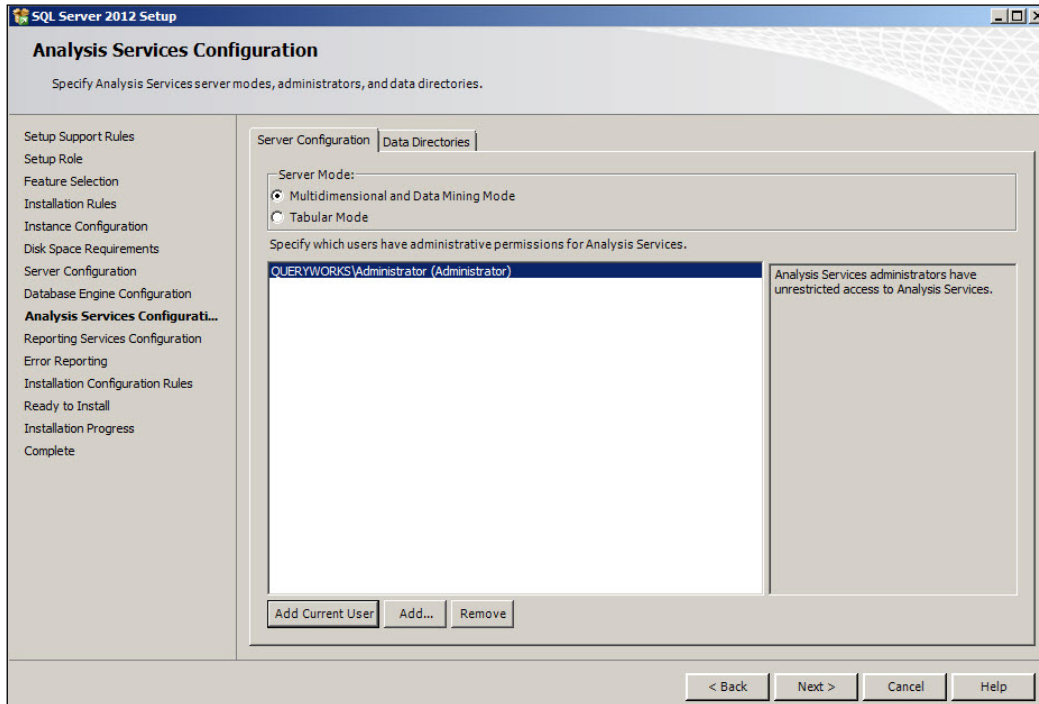
16. In the **Server Configuration** screen, change the account settings for all services except **Full-text Filter Daemon Launcher** and **SQL Server browser**; for other services:
- ❑ Use `QUERYWORKS\sqlagent` and corresponding password for **SQL Server Agent**.
 - ❑ Use `QUERYWORKS\sqlservice` and corresponding password for remaining services.



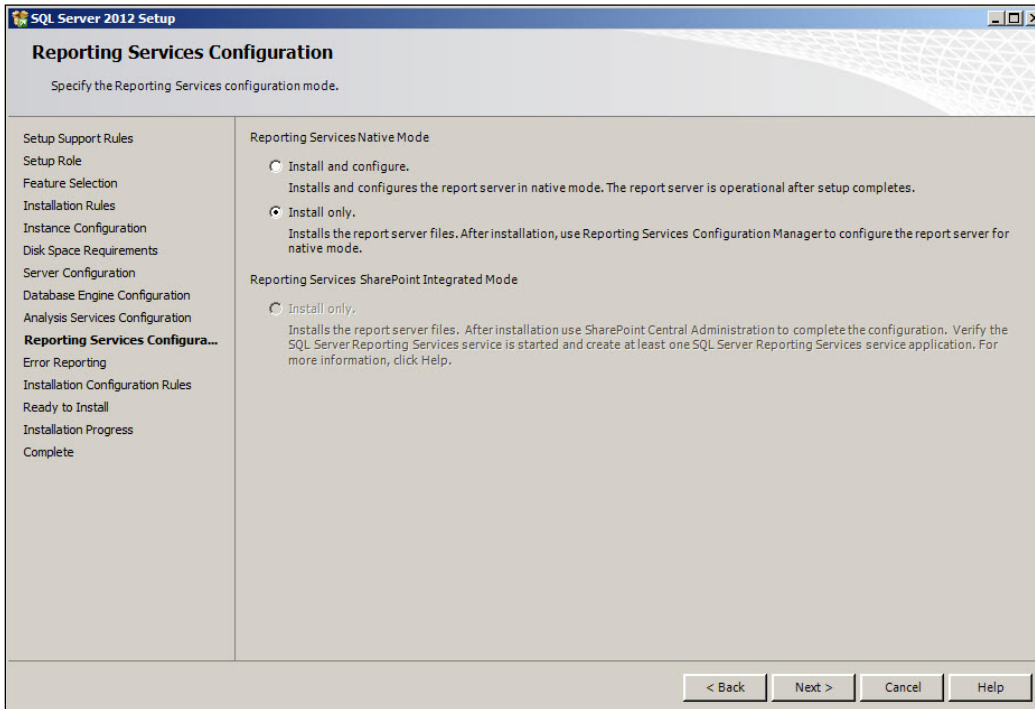
17. For the database engine configuration, choose **Windows authentication mode**. Be sure to click on the **Add Current User** button to add **QUERYWORKS\Administrator** as sysadmin to your default instance.



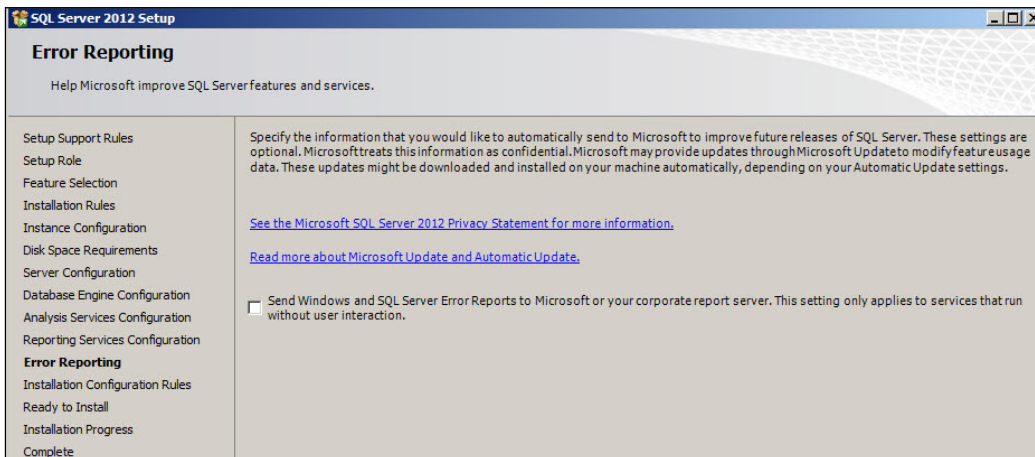
18. In **Analysis Services Configuration**, select **Multidimensional and Data Mining mode**. Click on **Add Current User** again for users that will have administrative privileges over Analysis Services.



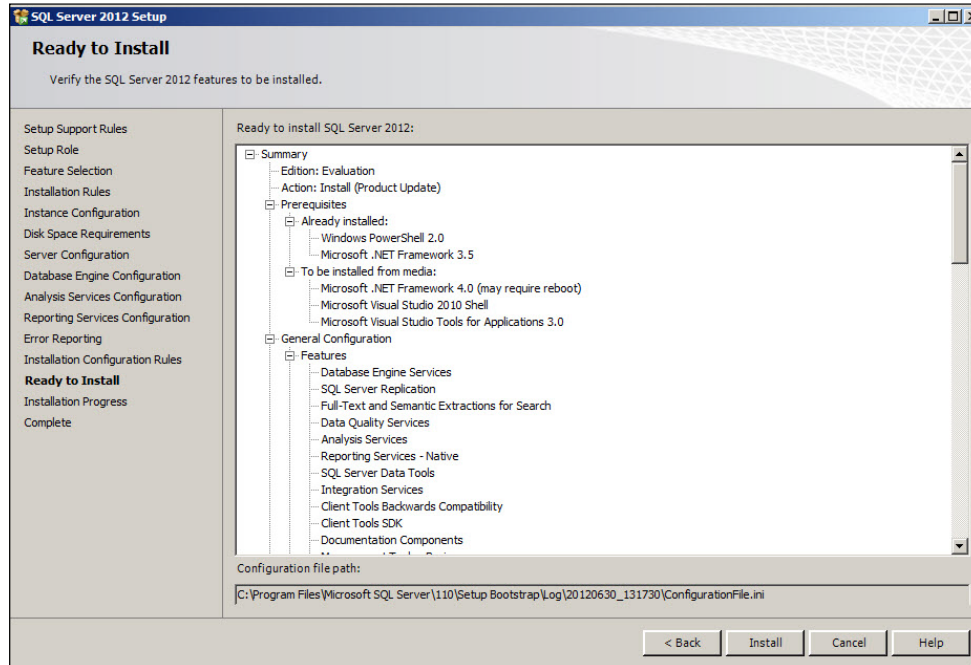
19. For **Reporting Services Configuration**, select **Install only**. We will configure reporting services later.



20. On the **Error Reporting** window, leave the checkbox unchecked. Click on **Next**.

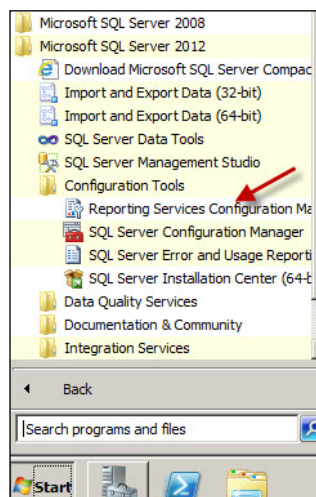


21. Review your settings in the **Ready to Install** screen, and click on **Install**.

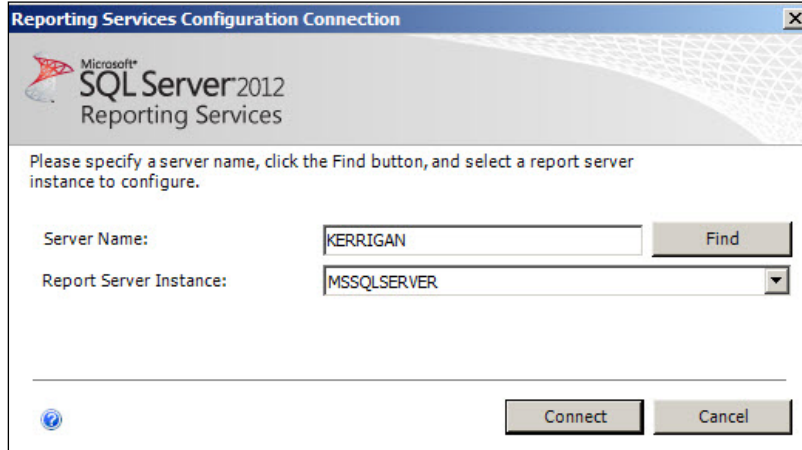


22. Once installation is finished, click on **Close**.

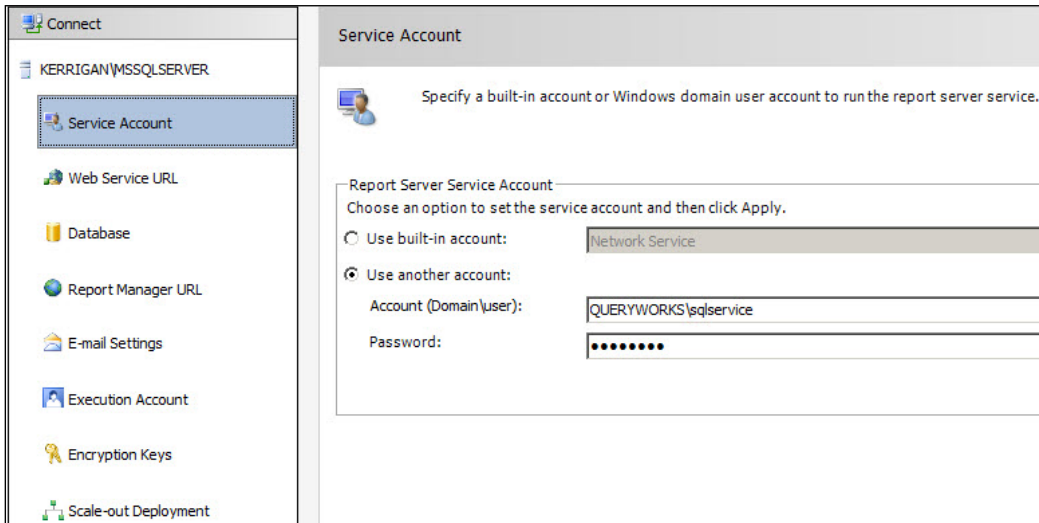
23. Now, we need to configure Reporting Services in the native mode. Go to **Start | All Programs | SQL Server 2012 | Configuration Tools | Reporting Services Configuration Manager**.



24. Connect to the default instance.



25. Click on **Service Account**. Double-check the service account assigned to run Reporting Services. If it is properly set, we do not need to make any changes in this window.



26. Click on **Web Service URL**; we will accept the default values. Click on **Apply**.

Web Service URL

Configure a URL used to access the Report Server. Click Advanced to define multiple URLs for a single Report Server instance, or to specify additional parameters on the URL.

Report Server Web Service is not configured. Default values have been provided to you. To accept these defaults simply press the Apply button, else change them and then press Apply.

Report Server Web Service Virtual Directory

Virtual Directory:

Report Server Web Service Site Identification

IP Address:

TCP Port:

SSL Certificate:

SSL Port:

Report Server Web Service URLs

URLs:

27. Click on **Database** and then click on the **Change Database** button. This will launch another set of windows.

Report Server Database

Reporting Services stores all report server content and application data in a database. Use this page to create or change the report server database or update database connection credentials.

Current Report Server Database

Click Change database to select a different database or create a new database in native or SharePoint integrated mode.

SQL Server Name:
Database Name:
Report Server Mode:

Current Report Server Database Credential

The following credentials are used by the report server to connect to the report server database. Use the options below to choose a different account or update a password.

Credential:
Login:
Password:

28. Select **Create a new report server database**.

Report Server Database Configuration Wizard

Change Database

Choose whether to create or configure a report server database.

Action	Select one of the following options to create an empty report server database or select an existing report server database that has content you want to use.
Database Server	
Database	
Credentials	
Summary	
Progress and Finish	

Select a task from the following list:

- Create a new report server database.
- Choose an existing report server database.

29. Leave the default database name as **ReportServer**, and click on **Next**.

Report Server Database Configuration Wizard

Change Database

Choose whether to create or configure a report server database.

Action	Enter a database name, select the language to use for running SQL scripts, and specify whether to create the database in native or SharePoint mode.
Database Server	
Database	
Credentials	
Summary	
Progress and Finish	

Database Name:

Temp Database Name:

Language:

Report Server Mode:

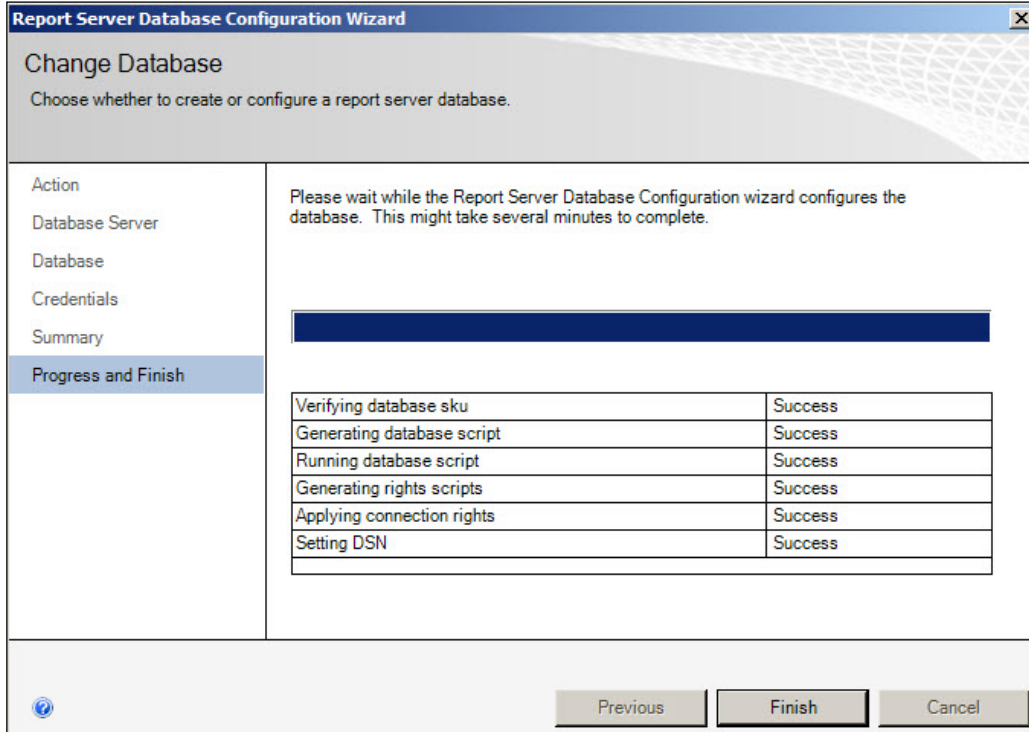
30. For **Authentication Type**, use **Service Credentials**. Click on **Next**.

Report Server Database Configuration Wizard	
<h3>Change Database</h3> <p>Choose whether to create or configure a report server database.</p>	
<ul style="list-style-type: none"> Action Database Server Database Credentials Summary Progress and Finish 	<p>Specify the credentials of an existing account that the report server will use to connect to the report server database. Permission to access the report server database will be automatically granted to the account you specify.</p> <p>Credentials:</p> <p>Authentication Type: <input type="text" value="Service Credentials"/></p> <p>User name: <input type="text" value="QUERYWORKS\sqlservice"/></p> <p>Password: <input type="password"/></p>

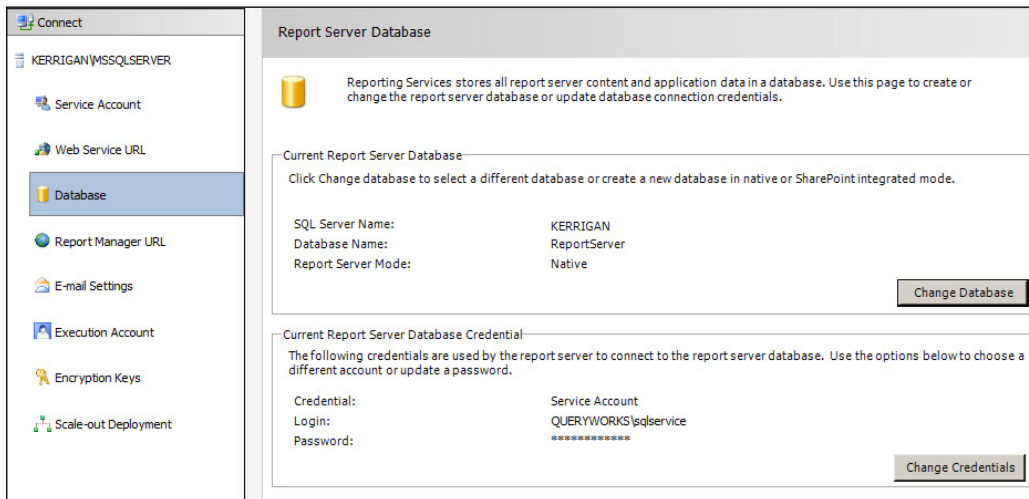
31. Review your **Summary** and click on **Next**.

Report Server Database Configuration Wizard	
<h3>Change Database</h3> <p>Choose whether to create or configure a report server database.</p>	
<ul style="list-style-type: none"> Action Database Server Database Credentials Summary Progress and Finish 	<p>The following information will be used to create a new report server database. Verify this information is correct before you continue.</p> <p>SQL Server Instance: KERRIGAN</p> <p>Report Server Database: ReportServer</p> <p>Temp Database: ReportServerTempDB</p> <p>Report Server Language: English (United States)</p> <p>Report Server Mode: Native</p> <p>Authentication Type: Service Account</p> <p>Username: QUERYWORKS\sqlservice</p> <p>Password: *****</p>

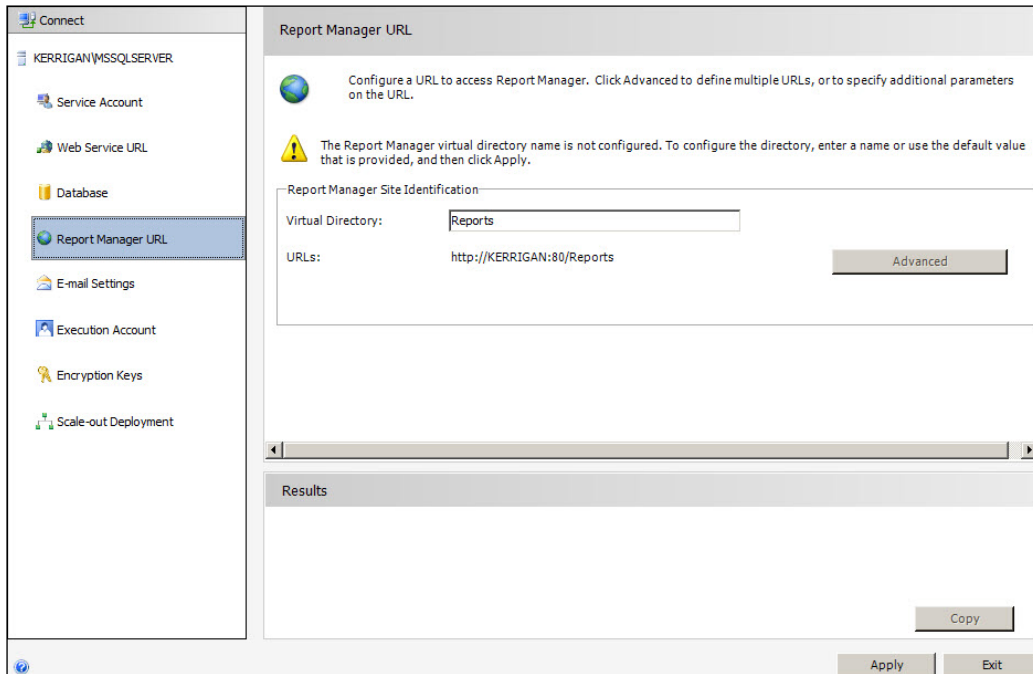
32. When the configurations have been successfully applied, click on **Finish**.



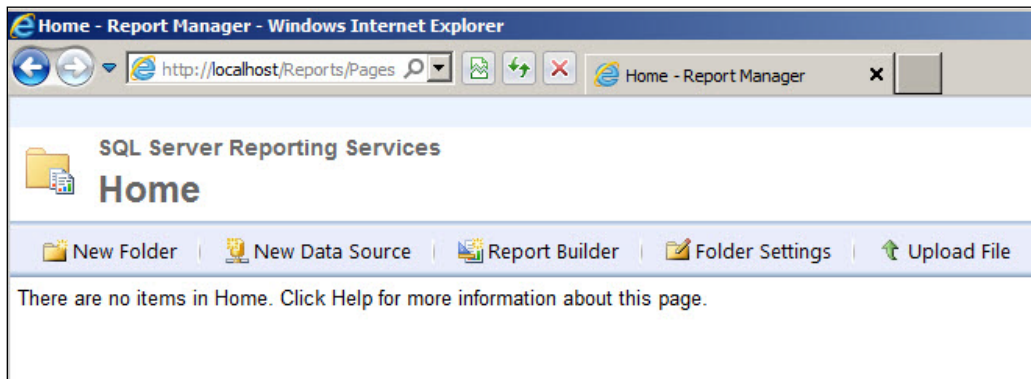
Note that the original **Report Server Database** screen will now be populated with the newly configured values.



33. Click on **Report Manager URL**, and click on **Apply**.



34. Test the URL by launching Internet Explorer. In the browser, type `http://localhost/Reports` (this is the same URL as `http://KERRIGAN:80/Reports`):



Installing sample databases

The SQL Server sample databases can be found here:

<http://msftdbprodsamples.codeplex.com/>

You can choose to install both SQL Server 2012 OLTP and DW samples. If you are going to try the recipes that involve Analysis Services cubes, then you definitely have to install the DW samples.

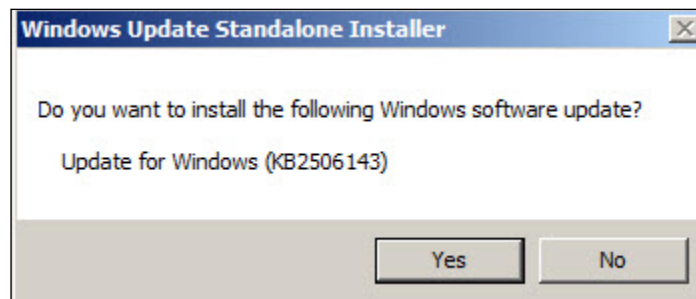
Complete instructions on how to install the sample databases can be found at:

http://social.technet.microsoft.com/wiki/contents/articles/3735.sql-server-samples-readme-en-us.aspx#Readme_for_Adventure_Works_Sample_Databases

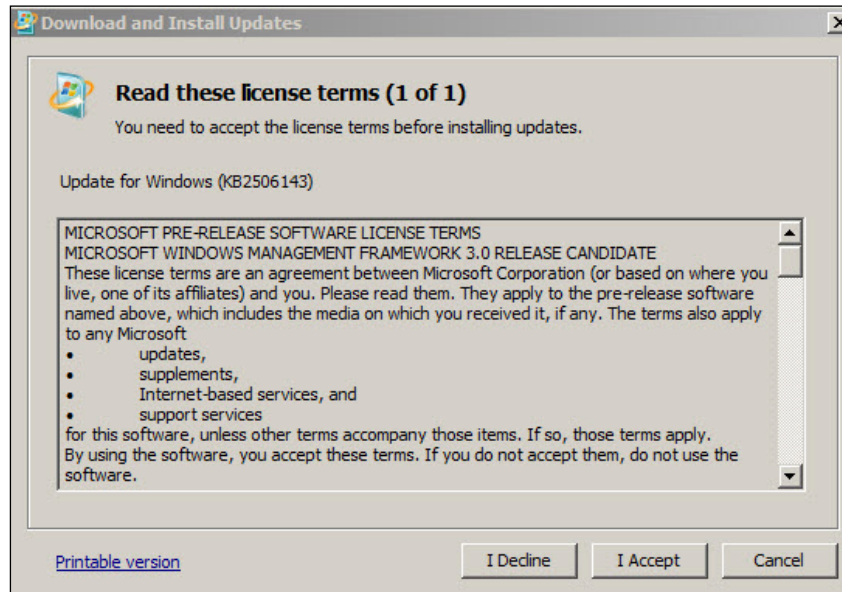
Installing PowerShell V3

As Windows Server 2008 R2 does not natively come with PowerShell V3, we will need to install it separately.

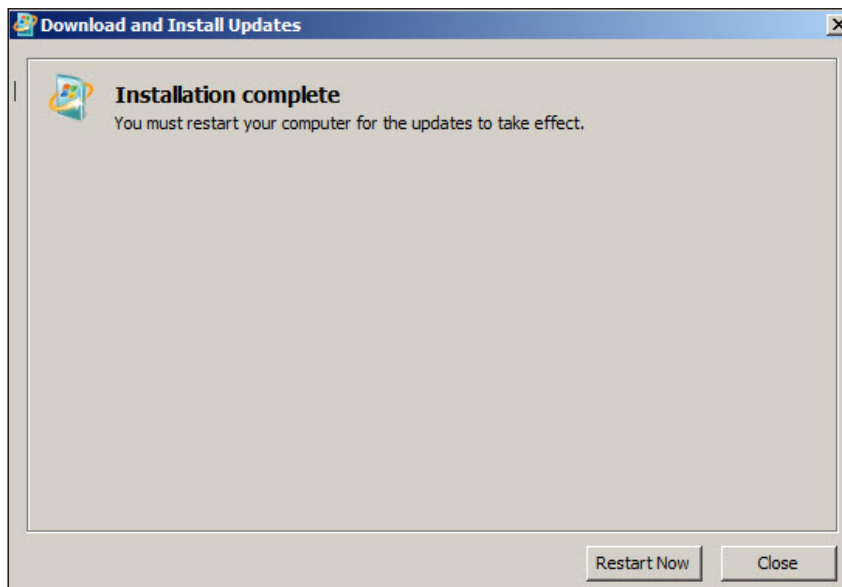
1. Launch **VMWare Player**.
2. Play **SQL2012VM** and login.
3. Copy the Windows Management Framework file from your host to your VM.
If you have installed VMWare Tools, you should be able to either drag-and-drop or copy-and-paste the file from the host to the guest VM. At the time of writing, the filename was `Windows6.1-KB2506143-x64.msu`.
4. Double-click on the executable file.
5. You will be prompted to confirm that you want to install this update. Click on **Yes** as shown in the following screenshot:



6. Agree to the license terms. Click on **I Accept**.



7. Wait for the installation to complete.



8. Restart your VM.

9. Confirm that PowerShell V3 is installed by launching the PowerShell console. Click on the console icon as shown in the following screenshot:



10. Type `$host.version` in the console and you should see the value **3** for the **Major** build, as shown in the following screenshot:

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2012 Microsoft Corporation. All rights reserved.

PS C:\Users\Administrator> $host.version

Major  Minor  Build  Revision
-----
3      0      -1     -1

PS C:\Users\Administrator>
```

Index

Symbols

7-Zip

URL, for downloading 468

.dtsx file

SSIS package, downloading to 433-435

.ispac file

about 442

deploying 443

-LogTruncationType parameter 329

.NET language

C# 18

PowerShell 18

VB.NET 18

.NET library 516

-NoRecovery parameter 329

.rdl file 412

A

ActiveDirectory 7

Add-Content cmdlet 477

AFTER trigger

creating, in PowerShell 90-94

aliases

testing, in PowerShell 466, 467

APIPA scheme 574

arrays 505, 536

assembly

creating, in SQL Server 374-376

asymmetric keys

creating 293-298

attachment

e-mail, sending with 482-484

authentication modes

listing, PowerShell used 210, 211

listing, SMO used 210, 211

modifying 211-214

Automatic Private IP Addressing.

See APIPA scheme

B

backtick 527, 528

Backup-ASDatabase cmdlet 450

backup device

creating, PowerShell used 310, 311

backup file

backup header information, listing

from 312-315

gathering 337, 338

backup header information

listing, from backup file 312-315

backup history

listing, for SQL Server instance 309

BackupRestoreBase 321

Backup-SqlDatabase cmdlet

about 319

used, for creating filegroup backup 329-331

bcp

about 102

used, for performing bulk export 102-104

used, for performing bulk import 110-113

binary data

extracting, from SQL Server 370-373

storing, into SQL Server 366-369

block comments 504, 533

blocking process

killing, in SQL Server 131, 132

listing, in SQL Server instance 128-130

bloggers, PowerShell 546

bulk export, performing
 bcp used 102-104
 Invoke-Sqlcmd used 100-102

bulk import, performing
 bcp used 110-113
 BULK INSERT used 105-109

BULK INSERT
 used, for performing bulk import 105-109

BulkLogged recovery model 308

Business Intelligence (BI) 386

Business Intelligence Development Studio (BIDS) 442

C

callable code block 525

Cascading Style Sheets (CSS) 364

C# code 8
 embedding, in PowerShell 484-486

cell-level encryption
 setting up, steps 298

certificate
 about 293
 creating, PowerShell used 291, 292
 creating, SMO used 292
 URL, for info 293

change tracking
 disabling, to target database 275, 276
 enabling, to target database 275, 276

change tracking (CT) 276

ChangeTrackingEnabled property 276

Cisco 7

Citrix 7

Clear-Host cmdlet 500

cmdlets 8 23, 520, 526

cmdlets, file manipulation
 Add-Content 477
 Copy-Item 477
 Get-Content 477
 Join-Path 477
 Move-Item 478
 New-Item 477
 Remove-Item 478
 Test-Path 477

cmdlets, folder manipulation
 Copy-Item 475
 Get-ChildItem 475

 Move-Item 475
 New-Item 475
 Remove-Item 475
 Test-Path 475

Code Access Security (CAS) 377

commands
 chaining 524, 525

Comma-Separated Value (CSV) 118

comment-based help
 about 458, 503
 enabling 503
 URL, for info 459

comments 504, 533

Common Language Runtime (CLR) 351, 376

common operators, PowerShell 502

common patterns, regular expressions 472

Compare-Object cmdlet 499

concepts, PowerShell
 aliases, of cmdlets 523, 524
 cmdlets 520
 commands, chaining 524, 525
 .NET 522
 object oriented 522
 package 525

condition
 about 264
 creating 264-268

conditional statements 535

configuration settings, SQL Server
 listing 51-54

configuring
 domain controller 569-576

ConnectionInfoBase class 281

ConnectionInfoExtended library 280, 286

ConnectionInfo library 280, 286

contents
 extracting, of trace file 284-288

Continue() method 50

ConvertFrom-Csv cmdlet 526

ConvertFrom-Json cmdlet 526

ConvertFrom-SecureString cmdlet 527

ConvertTo-Csv cmdlet 526

ConvertTo-Html cmdlet 526

ConvertTo-Json cmdlet 526

ConvertTo-SecureString cmdlet 527

ConvertTo-Xml cmdlet 526

Copy-Item cmdlet 475, 477, 499

CreateCatalogItem method 414
CreateFolderOnSqlServer method 427
Create method 290
credential
 about 510
 creating 244, 245
CSV
 Get-Process cmdlet, exporting to 467, 468
Cumulative Update (CU) 127

D

data
 extracting, from web service 490-492
database
 attaching, with data files 145-149
 copying, PowerShell used 149, 150
 copying, SMO used 150, 151
 creating, with default properties 67, 68
 detaching, programmatically 143, 144
 differential backup, creating on 324-326
 dropping, PowerShell used 72, 73
 dropping, SMO used 72, 73
 filegroup, adding to 154
 restoring 332-340
database backup
 creating, on mirrored backup files 321-323
Database Mail
 setting up, programmatically 168-177
 working 176, 177
database mappings
 listing 222-224
database master key
 about 289
 creating 289, 290
database objects
 searching, PowerShell used 60-66
database owner
 changing 73-75
database properties
 altering, PowerShell used 68-71
 altering, SMO used 68-71
database recovery model
 modifying 306, 307
database, restoring
 backup files, gathering 337, 338
 differential backup, restoring 340

 full backup, restoring with NORECOVERY
 option 339
 transaction logs, restoring 340, 341
database role
 creating 237-240
database user
 creating, PowerShell used 232-234
 creating, SMO used 232-234
 permissions, assigning to 234-236
data files
 database, attaching with 145-149
data source
 updating, for SSRS report 409-412
DataSourceDefinition class
 about 407
 properties 407, 408
DateTimeFormatInfo class
 URL, for info 460
date-time format strings, PowerShell 502
DBCC commands
 running, PowerShell used 167
DDL events
 WMI Server event alerts, setting
 up for 136-142
DependentServices property 47
DeployProject method 443
development environment 36
differential backup
 creating, on database 324-326
DisableNameChecking parameter 22
disk space usage
 checking, for SQL Server instance 133-135
domain accounts
 creating 577-579
domain controller
 configuring 569-576
DropAndMove method 161
Dynamically Linked Libraries (DLL) 12, 376

E

e-mail
 sending, with attachment 482-484
empty virtual machine
 creating 553-555
EnumFragmentation method 163, 166
EnumObjects method 66

EnumProcesses method 129

error handling 539

error messages
 displaying, in PowerShell 461, 462

escape and line continuation 527, 528

evaluation modes, policy
 CheckOnChanges 271
 CheckOnSchedule 271
 Enforce 271
 none 271

event log
 reading 481, 482

Excel format
 SSRS report, downloading in 396-400

Exchange 7

execution policies
 about 11
 AllSigned 498
 Bypass 498
 modifying 498
 RemoteSigned 498
 Restricted 498
 Undefined 498
 Unrestricted 498

execution policy, PowerShell scripts
 about 519
 AllSigned 519
 Bypass 519
 RemoteSigned 519
 Restricted 519
 Undefined 519
 Unrestricted 519

Export-Clixml cmdlet 468, 526

Export-Csv cmdlet 468, 526

ExpressionNodeOperator
 URL 268

ExpressionNodes
 URL 268

Extensible Stylesheet Language. *See* XSL

F

facet properties
 listing 252-254

facets
 listing 252-254

failed login attempts
 listing, in SQL Server instance 220, 221

features, PowerShell v3
 Improved Integrated Scripting Environment (ISE) 10
 Module AutoLoading 8
 robust sessions 8
 scheduled jobs 8
 simplified language syntax 9
 web service support 9
 workflows 8

filegroup
 adding, to database 154
 indexes, moving to 158-161
 secondary data files, adding to 156, 157

filegroup backup
 creating, Backup-SqlDatabase cmdlet used 329-331

file list information
 listing 312-315

file manipulation
 URL, for info 478

files
 manipulating, in PowerShell 476, 477
 searching, in PowerShell 478-480
 SSIS package, downloading to 433-435

FolderExistsOnSqlServer method 427

folders
 creating, in SSIS instance 425-427
 creating, in SSIS package store 425-427
 managing, in PowerShell 474-476

Foreach loop 506

Foreach-Object cmdlet 47, 354, 369, 419, 499

for loop 506

Format-List cmdlet 526

format patterns, timestamp
 about 460
 dd 460
 dddd 460
 hh 460
 HH 460
 mm 460
 MM 460
 MMM 460
 MMMM 460

ss 460
tt 460
yy 460
yyyy 460

Format-Table cmdlet 526

FreeISO Creator

URL 551

full backup

restoring, with NORECOVERY 339

full database backup

creating, PowerShell used 316-319

Full recovery model 308

function

about 8, 507, 539

script, converting into 540-542

G

Get-Alias cmdlet 466, 526

Get-ChildItem cmdlet 369, 475, 478, 499, 520, 526

Get-Command cmdlet 497, 521

Get-Content cmdlet 153, 414, 477, 499, 526

Get-Credential cmdlet 527

Get-Date cmdlet 526

GetDtsServerPackageInfos method 434

Get-EventLog cmdlet 526

Get-ExecutionPolicy cmdlet 519, 527

Get-Help cmdlet

about 497, 521

PowerShell script, documenting for 456-459

Get-History cmdlet 500

Get-Hotfix cmdlet 126

Get-HotFix cmdlet 526

GetItemDataSources method 411

GetItemDefinition method 419

Get-Location cmdlet 500

Get-Member cmdlet 71 497, 521, 522, 526

GetName method 282

GetOrdinal method 282, 287

GetPolicies method 423

Get-Process cmdlet

about 467, 500, 526

exporting, to CSV 467, 468

exporting, to XML 467, 468

URL, for info 465

Get-Service cmdlet 45, 48, 526

Get-SSRSItems fun 542

GetValue method 282, 287

Get-WmiObject cmdlet 41, 526

Global Assembly Cache (GAC) 21

Guest OS

Windows Server 2008 R2,
installing as 556-566

H

hash 537

hash table 505

help cmdlet 499

here-string 504, 529

here-string variable 354

hMailServer 177

URL 177

Hotfix 127

HTML report

creating, PowerShell used 486, 487

I

Import-Clixml cmdlet 526

Import-Csv cmdlet 526

ImportPolicy method

about 263, 264

parameters 264

ImportPolicy method, parameters

ImportEnabledState 264

overwriteExistingCondition 264

overwriteExistingPolicy 264

XMLReader 264

Improved Integrated Scripting Environment (ISE) 10

index

creating, PowerShell used 95-98

creating, SMO used 95-98

moving, to filegroup 158-161

rebuilding 164-166

reorganizing 164-166

index fragmentation

investigating, PowerShell used 162,-164

investigating, SMO used 162-164

index type

filtered 98

FullText 98

- spatial 98
- XML 98
- INSERT statement 354**
- installed hotfixes**
 - listing, in SQL Server 124-127
- installing**
 - PowerShell V3 598, 599
 - SQL Server 2012, on VM 580-597
 - SQL Server sample databases 598
 - VMWare tools 567, 569
 - Windows Server 2008 R2, as
 - Guest OS 556-566
- instance classes 18**
- instance properties**
 - exporting, to text file 116-120
- Integrated Scripting Environment (ISE) 517**
- Intellisense 10**
- International Organization for Standardization (ISO) 551**
- Invoke-Command cmdlet 494**
- Invoke-Expression cmdlet**
 - about 283, 288, 468
 - using 469, 470
- Invoke-Expression command 373**
- Invoke-PolicyEvaluation cmdlet 274**
- Invoke-Sqlcmd cmdlet**
 - about 100, 153, 369, 512
 - used, for performing bulk export 100-102
- Invoke-SqlCmd cmdlet 354**
- Invoke-WebRequest cmdlet 9**
- ISO File 551**
- ISPAC file**
 - deploying, to SSISDB 441-443
- items**
 - listing, in SSRS report server 386-388

J

Join-Path cmdlet 477

K

KillDatabase method 73

KillProcess method 132

L

ListChildren method 388, 390

- ListChildren method 387**
- LoadFromDtsServer method 434**
- LoadFromSqlServer method 432**
- Load() method 21, 366**
- LoadWithPartialName() method 21**
- local network**
 - SQL Server instances, listing in 39-41
- log errors**
 - listing, in SQL Server 215-219
- logics 506**
- login**
 - creating, PowerShell used 227, 228
 - creating, SMO used 227, 228
 - listing 222-224
 - permissions, assigning 229-231
 - roles, assigning 229-231
- LoginMode property 213**
- login/user roles**
 - listing 225, 226
- loops**
 - about 506, 537, 538
 - for 506
 - Foreach 506
 - while 506

M

MagicISO

- URL 551

ManagedComputer object 40-44

management cmdlets

- about 508
- Get-ChildItem 508, 526
- Get-Content 508, 526
- Get-EventLog 508, 526
- Get-HotFix 508, 526
- Get-Process 508, 526
- Get-Service 508, 526
- Get-WmiObject 508, 526
- New-WebServiceProxy 508, 526
- Start-Process 508, 526
- Start-Service 508, 526

MasterKey class 290

MasterKey object 290

messages

- displaying 532

Microsoft .NET Framework 4.0
installing 12

mirrored backup files
database backup, creating on 321-323

mixed assembly error 28

module 24

Module AutoLoading 8

modules and snap-ins 11

Move-Item cmdlet 475, 478, 499

MSDN Add-Type
URL 486

MSDN ConvertTo-HTML
URL 488

MSDN Get-Date
URL, for info 461

MSDN Get-Process
URL, for info 465

MSDN Invoke-Expression
URL 470

MSDN Send-MailMessage
URL 484

MS Virtual PC
URL 550

multiple servers
SQL query, executing to 152, 153

N

Nero Burning Software
URL 551

New-Item cmdlet 356, 475, 477

New-WebServiceProxy cmdlet 387 526

NORECOVERY option
about 339
full backup, restoring 339

O

online piecemeal restore
performing 342-349

OPENROWSET method
URL, for info 369

operators 531

orphaned users
fixing 241, 242, 243

OverrideIfAlreadyUser option 75

P

Pause() method 50

PDF format
SSRS report, downloading in 396-400

PercentCompleteEventHandler 321

permissions
assigning, to database user 234-236
assigning, to login 229-231
listing 225, 226

pipe operator 524

policies
about 252-254
creating, programmatically 268-271
evaluating, against SQL Server
instance 272-274
evaluation modes 271
exporting, from SQL Server
Management Studio 261
exporting, to XML file 257-261
importing, into SQL Server 261-263
listing 255-257

Policy Based Management (PBM) 253

PolicyStore parameter 256, 257

PowerISO
URL 551

PowerShell
about 7, 36, 515
administrative tasks 116
administrator, scripting 36
advantages 516
AFTER trigger, creating 90-94
aliases 499, 500
aliases, testing 466, 467
arrays 505
asymmetric keys, creating 293-298
attachment, sending with e-mail 482-484
backup header information, listing from
backup file 312-315
backup history, listing for SQL Server
instance 309
capabilities 386
C# code, embedding 484-486
certificate, creating 291, 292
change tracking, disabling to target
database 275, 276

- change tracking, enabling to target
 - database 275, 276
- cmdlets, for displaying output 500
- comments 504
- common operators 502
- comment based help, enabling 503
- condition, creating 264-268
- credentials 510
- database backup, creating on mirrored
 - backup files 321-323
- database, creating with default
 - properties 67, 68
- database, dropping 72, 73
- database master key, creating 289, 290
- database recovery model,
 - modifying 306, 307
- database, restoring 332-341
- database role, creating 237-340
- data, extracting from web service 490-492
- date-time format strings 502
- differential backup, creating on
 - database 324-326
- environment, setting up 516
- error messages, displaying 461, 462
- execution policy, getting 498
- execution policy, modifying 498
- execution policy, setting 498
- facet properties, listing 252-254
- facets, listing 252-254
- file list information, listing 312-315
- files, manipulating 476, 477
- files, searching 478-480
- folders, managing 474-476
- functions 507
- hardcoded query, executing 99, 100
- hash table 505
- here-string 504
- index, rebuilding 164-166
- index, reorganizing 164-166
- instance configuration settings,
 - modifying 55-60
- Invoke-Expression, using 469, 470
- learning 497
- logics 506
- loops 506
- management cmdlets 508
- online piecemeal restore,
 - performing 342-349
- orphaned users, fixing 241-243
- permissions, assigning to database user 234-236
- permissions, assigning to login 229-231
- podcasts 547
- policies, listing 255-257
- policy, evaluating 272-274
- policy, exporting to XML file 257-261
- policy, importing into SQL Server 261-263
- profiler trace event, running 276-283
- profiler trace event, saving 276-283
- regex characters 504, 505
- regex patterns 504, 505
- regular expressions, testing 470-474
- resources 543
- roles, assigning to login 229-231
- script, running 499
- scripts, running 38, 517, 518
- security cmdlets 508
- special characters 500
- special variables 501
- SQL script, executing 99, 100
- SQL Server Assemblies, adding 509
- SQL Server event alert, adding 187, 188
- SQL Server job, creating
 - programmatically 183-186
- SQL Server job, running
 - programmatically 190, 191
- SQL Server Snapins, adding 509
- symmetric keys, creating 293-298
- timestamp, getting in 459, 460
- transaction log backup, creating 327, 328
- Transparent Data Encryption (TDE),
 - setting up 299-303
- used, for adding secondary data files to filegroup 156, 157
- used, for altering database properties 68-71
- used, for copying database 149, 150
- used, for creating backup device 310, 311
- used, for creating database user 232-234
- used, for creating filegroup 154
- used, for creating full database backup 316-319
- used, for creating HTML report 486, 487
- used, for creating index 95-98

- used, for creating login 227, 228
- used, for creating policy 268-271
- used, for creating RSS feed 358-363
- used, for creating SQL Server operator 181, 182
- used, for creating stored procedure 85-89
- used, for creating table 76, 78
- used, for creating view 82-84
- used, for executing SSIS package 430-432
- used, for extracting content of trace file 284-288
- used, for investigating index fragmentation 162-164
- used, for listing authentication modes 210, 211
- used, for listing processes 462-465
- used, for listing SQL Server jobs 178-180
- used, for parsing XML 488, 489
- used, for running DBCC commands 167
- used, for scheduling SQL Server job 192, 193-201
- used, for searching database objects 60-66
- used, for setting up Database Mail 168-177
- utility cmdlets 508
- webcasts 547
- Windows event log, reading from 481, 482

PowerShell, books

- PowerShell V2 543
- PowerShell V2 Free E-books 544, 545

PowerShell ISE

- about 37, 276
- Command Pane 37
- Script Pane 37

PowerShell Remoting

- about 492
- URL, for info 495
- using 493, 494

PowerShell, resources

- bloggers 546, 547
- blogs 545
- books 543-545
- sites 545

PowerShell script

- documenting, for Get-Help cmdlet 456-459

PowerShell scripts

- execution policy 519
- running 27, 517, 518

PowerShell syntax

- about 527
- arrays 536
- comments 533
- conditional statements 535
- error handling 539
- escape and line continuation 527, 528
- hashes 537
- loop 537, 538
- message display 532
- operators 531
- regular expressions 535
- special variables 534
- statement terminators 527
- variables 528, 529

PowerShell V2

- books 543, 544
- SQLPS module, importing 509
- Where-Object syntax 498

PowerShell V2 Free E-books 544, 545

PowerShell V3

- books 543
- features 8
- installing 12, 598, 599
- Where-Object syntax 498

PowerShell v3 ISE

- enabling 13

PowerShell v3 Sneak Peek Screenshot

- URL 13

processes

- listing, PowerShell used 462-465

Process ID (PID)

- about 501

profiler trace event

- running 276-283
- saving 276-283

properties

- listing, for SSRS report 388-390

proxy

- creating, in SQL Server 246-248

PSCX

- about 548
- URL 548

PSDrive 11

PSProvider 10

PSScheduledJob module 8

Q

query

executing, from PowerShell 99, 100

Quest 7

Quick Fix Engineering (QFE) 127

R

RDL files

downloading, from SSRS report server 416-420

ReadErrorLog method 219

Read-Host cmdlet 526

Really Simple Syndication. *See* RSS

RecoveryModel property

about 308
modifying 307

Recovery Point Objective (RPO) 308

Recovery Time Objective (RTO) 308

Refresh() method 50

regex characters, PowerShell 504, 505

regex methods

URL, for info 474

regex patterns, PowerShell 504, 505

Register-WmiEvent cmdlet 142

regular expressions

about 472, 535
common patterns 472
testing, in PowerShell 470-474

Release Candidate (RC) 13

Released to Manufacturing (RTM) 516

RemoteSigned 11

Remove-Item cmdlet 475, 499

Render method 399

Report Manager

SSRS report, uploading to 412-415

ReportService2010 web service 387

ReportViewer

used, for displaying SSRS report 391-395
working 393

ReportViewer control 396

Restore-ASDatabase cmdlet 452

robust sessions, PowerShell v3 8

roles

assigning, to login 229-231

RSS 363

RSS feed

creating, from SQL Server content 358-363
creating, PowerShell used 358-363
creating, T-SQL used 358-363
URL, for info 363
XSL, applying to 363, 365

RTM 126

running processes

listing, in SQL Server instance 128-130

S

sample code

working with 12

SaveToDtsServer method 430

SaveToSqlServer method 430

SaveToXml method 434

script

converting, into function 540-542
running 38, 499

secondary data files

adding, to filegroup 156, 157

security cmdlets

about 508

ConvertFrom-SecureString 508, 527

ConvertTo-SecureString 508, 527

Get-Credential 508, 527

Get-ExecutionPolicy 508, 527

Set-ExecutionPolicy 508, 527

security levels

EXTERNAL_ACCESS 377

SAFE 377

UNSAFE 377

semicolon 527

ServerInstance property 40

service accounts

about 551

listing 204, 205

modifying 206-209

service packs

listing, in SQL Server 124-127

Set-Alias cmdlet 366

Set-ExecutionPolicy cmdlet 519, 527

SetItemDataSources method 412 415

Set-Location cmdlet 500

set_Login method 31

SetOwner method 75

SetPolicies method 425

SharePoint 7

Simple recovery model 308

simplified language syntax 9

single line comments 504, 533

SMO

about 18, 36

certificate, creating 292

database, dropping 72, 73

installing 18, 19

instance classes 18

permissions, assigning to database
user 234-236

permissions, assigning to login 229-231

roles, assigning to login 229-231

Transparent Data Encryption (TDE),
setting up 299-303

used, for adding secondary data files
to filegroup 156, 157

used, for altering database
properties 68-71

used, for copying database 150, 151

used, for creating database user 232-234

used, for creating filegroup 154

used, for creating index 95-98

used, for creating login 227, 228

used, for creating SQL Server
operator 181, 182

used, for creating stored procedure 85-89

used, for creating table 79-81

used, for creating view 82-84

used, for investigating index fragmentation
162-164

used, for listing authentication
modes 210, 211

used, for scheduling SQL Server job 192-201

utility classes 18

SMO assemblies

loading 20, 21

working 21

SMO Certificate object 292

SMO libraries

gaining 18

SMO Server Object

creating 512

exploring 32

working 33

snap-in 24

Sort-Object cmdlet 499

special characters 500

special variables 501, 534

SQLCLR assemblies

about 381

URL, for info 379

SqlConnectionInfo connection object 281

SqlConnection object 438

SQL Login 225

SQL Management Objects. *See* **SMO**

SQLPS and SQLASCMDLETS cmdlets

list 25, 26

SQLPS module

importing 509

SQLPSX

about 520, 548

URL 548

SQL query

executing, to multiple servers 152, 153

SQL-related cmdlets

discovering 22

working 23

SQL-related modules

discovering 23

SQL script

executing, from PowerShell 99, 100

SQL Server

about 36, 252, 386

assembly, creating 374-376

authentication modes, modifying 211-214

binary data, extracting from 370-373

binary data, storing into 366-369

blocking process, killing 131, 132

bloggers 547

configuration settings, listing 51-54

credential, creating 244, 245

database, detaching

programmatically 143, 144

database inventory, creating 120-123

database owner, changing 73, 75

installed hotfixes, listing 124-127

instance configuration settings 55-60

instance properties, exporting to
text file 116-120

log errors, listing 215-219

policy, importing into 261-263

- proxy, creating 246-248
- service accounts, listing 204, 205
- service accounts, modifying 206-209
- service packs, listing 124-127
- services, discovering 43, 44
- services, starting 46-50
- services, stopping 46-50
- trigger, creating 90-94
- working, with PowerShell 10, 11
- XML content, extracting from 355-357
- XML content, inserting into 352-354
- SQL Server 2012**
 - installing 12
 - installing, on VM 580-597
 - SSAS cmdlets, listing 447, 448
- SQL Server 2012 trial version ISO file**
 - URL, for downloading 551
- SQL Server Agent Account 552**
- SQL Server Analysis Services.** *See* **SSAS**
- SQL Server and PowerShell**
 - working with 10
- SQL Server Assemblies**
 - adding 509
- SQL Server content**
 - RSS feed, creating from 358-363
- SQL Server database**
 - user-defined assemblies, creating in 378
- SQL Server database inventory**
 - creating 120-123
- SQL Server Data Tools (SSDT) 442**
- SQL Server event alert**
 - adding 187, 188
- SQL Server instance**
 - backup history, listing for 309
 - blocking processes, listing 128-130
 - disk space usage, checking for 133-135
 - failed login attempts, listing in 220, 221
 - running processes, listing 128-130
- SQL Server instance inventory**
 - about 116
 - creating 116-120
- SQL Server instance object**
 - creating 29
 - working 30
- SQL Server instances**
 - about 39, 552
 - listing, in local network 39-41
- SQL Server Integration Services.** *See* **SSIS**
- SQL Server job**
 - creating, programmatically 183-186
 - running, programmatically 190, 191
 - scheduling, PowerShell used 192-201
 - scheduling, SMO used 192-201
- SQL Server jobs**
 - listing, PowerShell used 178-180
- SQL Server Management Objects.** *See* **SMO**
- SQL Server Management Studio**
 - policy, exporting from 261
- SQL Server operator**
 - creating, PowerShell used 181, 182
 - creating, SMO used 181, 182
- SQL Server PowerShell documentation**
 - URL 13
- SQL Server PowerShell hierarchy**
 - exploring 14-16
 - working 17, 18
- SQL Server processes**
 - blocking 510
 - running 510
- SQL Server proxy**
 - creating 246, 248
- SQL Server Reporting Services.** *See* **SSRS**
- SQL Server sample databases**
 - installing 598
- SQL Server Service Account 552**
- SQL Server services**
 - discovering 43, 44
 - starting 46-50
 - stopping 46-50
- SQL Server Snapins**
 - adding 509
- SSAS**
 - about 386, 448
 - database, backing up 450
 - database, restoring 451, 452
 - instance properties, listing 448, 449
- SSAS cmdlets**
 - listing 447, 448
- SSAS cube**
 - about 452
 - processing 452-454
- SSAS database**
 - backing up 450
 - restoring 451, 452

SSAS database backup

creating 450

SSAS instance properties

listing 448, 449

SSAS Object

creating 513

SSIS 386**SSISDB**

catalog, creating 435-438

folder, creating 439, 440

ISPAC file, deploying to 441-443

SSIS package 444-446

SSISDB catalog

creating 435-438

SSISDB folder

creating 439, 440

SSIS instance

folders, creating in 425-427

SSIS Object (SQL Server 2005/2008/2008R2)

creating 513

SSIS Object (SQL Server 2012)

creating 513

SSIS package

deploying, to package store 428-430

downloading, to .dtsx file 433-435

executing 444-446

executing, PowerShell used 430-432

SSIS package store

folders, creating in 425-427

SSIS package, deploying to 428-430

SSRS

about 386

data source, creating 404-408

folder, creating 400-404

SSRS data source

about 404

creating 404-408

SSRS folder

creating 400-404

SSRS Proxy Object

creating 513

SSRS report

data source, updating 409-412

displaying, ReportViewer used 391-395

downloading, in Excel format 396-400

downloading, in PDF format 396-400

properties, listing 388-390

uploading, to Report Manager 412-415

user, adding to 421-425

SSRS report properties

listing 388-390

SSRS report server

items, listing in 386-388

RDL files, downloading from 416-420

standard Date and Time format strings

URL, for info 461

Start() method 50**Start-Process cmdlet 526****Start-Service cmdlet 526****statement terminators 527****Stop() method 50****Stop-Process cmdlet 500****stored procedure**

creating, PowerShell used 85-89

creating, SMO used 85-89

string interpolation 530**switch statement 473****symmetric keys**

creating 293-298

T**table, creating**

PowerShell used 76-78

SMO used 79-81

tasks, modules

list installed modules 24

list loaded modules 24

load a specific module 24

show commands in module 24

tasks, snap-ins

list installed snap-ins 24

list loaded snap-ins 24

load specific snap-in 24

show commands in snap-in 24

Template Definition File (TDF) 277**Test-Path cmdlet 475, 477****text file**

instance properties, exporting to 116-120

TextHeader property 84**TextMode property 84****timestamp**

about 459

format patterns 460

- getting, in PowerShell 459, 460
- trace file**
 - content, extracting 284-288
- TraceFile class 286, 287**
- TraceFile object 281, 282**
- TraceServer class 280**
- TraceServer object 282**
- transaction log backup**
 - creating 327, 328
- transaction logs**
 - restoring 340, 341
- Transparent Database Encryption (TDE)**
 - about 252, 299
 - setting up, programmatically 299-303
- trigger**
 - creating, in SQL Server 90-94
- TRUNCATE TABLE command 112**
- T-SQL**
 - about 36
 - used, for creating RSS feed 358-363

U

- Unregister-Event cmdlet 142**
- user-defined assemblies**
 - extracting 379-383
 - listing, in SQL Server database 378
 - resaving, back to filesystem as DLLs 379-383
- users**
 - adding, to SSRS report 421-425
 - listing 222-224
- utility classes 18**
- utility cmdlets**
 - about 508, 526
 - ConvertFrom-Csv 508, 526
 - ConvertFrom-Json 508, 526
 - ConvertTo-Csv 508, 526
 - ConvertTo-Html 508, 526
 - ConvertTo-Json 508, 526
 - ConvertTo-Xml 508, 526
 - Export-Clixml 508, 526
 - Export-Csv 508, 526
 - Format-List 508, 526
 - Format-Table 508, 526
 - Get-Alias 508, 526
 - Get-Date 508, 526

- Get-Member 508, 526
- Import-Clixml 508, 526
- Import-Csv 508, 526
- Read-Host 508, 526

V

- variables 528, 529**
- view**
 - creating, PowerShell used 82-84
 - creating, SMO used 82-84
- Virtual Box**
 - URL 550
- Virtual Machine Computer Administrator Account 552**
- Virtual Machine Computer Name 552**
- Virtual Machine Name 552**
- Virtual Machine (VM)**
 - about 550
 - accounts 552
 - logging in 553
 - SQL Server 2012, installing on 580-597
- Visual Studio 2010 trial version ISO**
 - URL, for downloading 552
- VMWare 7**
 - tools, installing 567-569
 - URL, for shortcuts 553
- VMWare Player**
 - about 551
 - URL 550
 - URL, for documentation 551
 - URL, for downloading 551
- VMWare tools**
 - installing 567, 569
- VMWare Workstation**
 - URL 550

W

- Web-based Enterprise Management (WBEM) 40**
- web service**
 - data, extracting from 490-492
- web service proxy 387**
- Web Services for Management (WSMan) 494**
- web service support 9**
- Where-Object cmdlet 219, 390, 499**

Where-Object syntax 117

for PowerShell V2 498

for PowerShell V3 498

Windows event log

reading, from PowerShell 481, 482

Windows Login 225**Windows Management Framework**

URL, for downloading 552

Windows Management Framework 3.0

installing 13

Windows Management Instrumentation**(WMI) 7, 40, 134, 140****Windows PowerShell Tip of the****Week- Formatting Dates and Times**

URL, for info 461

Windows PowerShell Workflow (PSWF) 8**Windows Remote Management (WinRM) 494****Windows Server 7****Windows Server 2008 R2**

installing, as Guest OS 556-566

URL, for downloading 551

Windows Server Hyper-V Server 2008 R2

URL 550

WMI Query Language (WQL) 140, 142**WMI Server event alerts**

setting up, for DDL events 136-142

workflows, PowerShell v3 8**Write-Debug cmdlet 500, 532****Write-Error cmdlet 500, 532****Write-EventLog cmdlet 500, 532****Write-Host cmdlet 500, 532****Write-Output cmdlet 499, 500, 532****Write-Progress cmdlet 500, 532****Write-Verbose cmdlet 500, 532****Write-Warning cmdlet 500, 532****X****XML**

about 351

Get-Process cmdlet, exporting to 467, 468

parsing, PowerShell used 488, 489

XML content

extracting, from SQL Server 355-357

inserting, into SQL Server 352-354

XML file

policy, exporting to 257-261

XSL

applying, to RSS feed 363-365

XslCompiledTransform variable 366



Thank you for buying SQL Server 2012 with PowerShell V3 Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.PacktPub.com.

About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

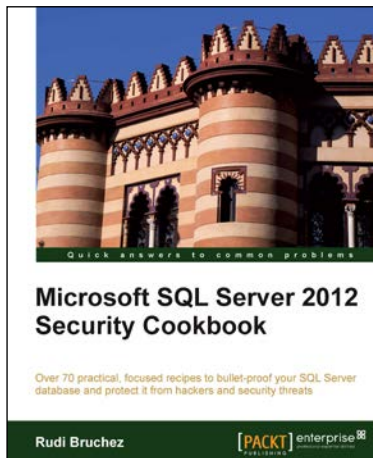


Microsoft Windows PowerShell 3.0 First Look

ISBN: 978-1-849686-44-0 Paperback: 200 pages

A quick, succinct guide to the new and exciting features in PowerShell 3.0

1. Explore and experience the new features found in PowerShell 3.0
2. Understand the changes to the language and the reasons why they were implemented
3. Quickly get up to date with the latest version of Powershell with concise descriptions and simple examples

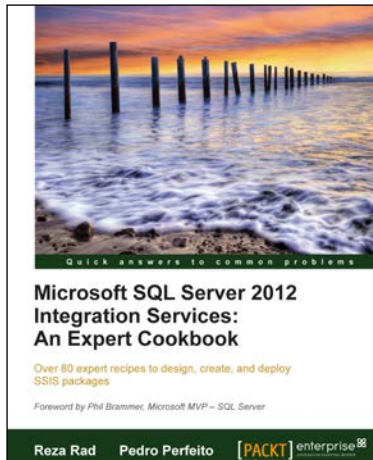


Microsoft SQL Server 2012 Security Cookbook

ISBN: 978-1-849685-88-7 Paperback: 322 pages

Over 70 practical, focused recipes to bullet-proof your SQL Server database and protect it from hackers and security threats

1. Practical, focused recipes for securing your SQL Server database
2. Master the latest techniques for data and code encryption, user authentication and authorization, protection against brute force attacks, denial-of-service attacks, and SQL Injection, and more
3. A learn-by-example recipe-based approach that focuses on key concepts to provide the foundation to solve real world problems

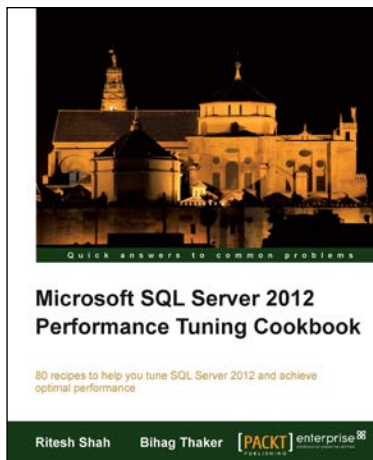


Microsoft SQL Server 2012 Integration Services: An Expert Cookbook

ISBN: 978-1-849685-24-5 Paperback: 564 pages

Over 80 expert recipes to design, create, and deploy SSIS packages

1. Full of illustrations, diagrams, and tips with clear step-by-step instructions and real time examples
2. Master all transformations in SSIS and their usages with real-world scenarios
3. Learn to make SSIS packages re-startable and robust; and work with transactions



Microsoft SQL Server 2012 Performance Tuning Cookbook

ISBN: 978-1-849685-74-0 Paperback: 478 pages

80 recipes to help you tune SQL Server 2012 and achieve optimal performance

1. Learn about the performance tuning needs for SQL Server 2012 with this book and ebook
2. Diagnose problems when they arise and employ tricks to prevent them
3. Explore various aspects that affect performance by following the clear recipes

Please check www.PacktPub.com for information on our titles