Powershell voor de DBA

Automatiseren in een scriptingomgeving

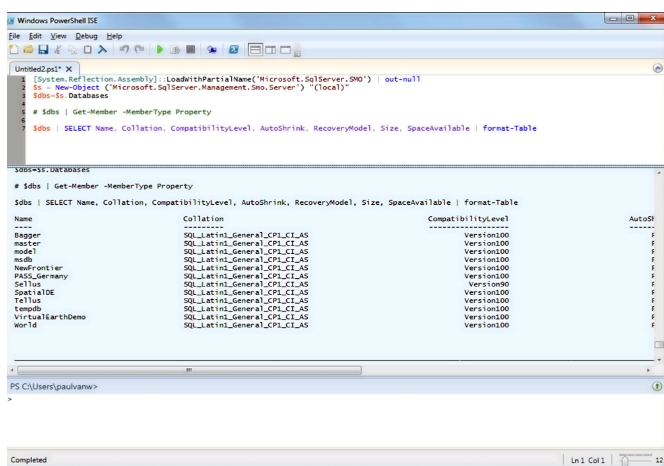
Eric ter Braake

Powershell is een scriptingomgeving voor het beheer van Windows-omgevingen. Vaak moeten handelingen met regelmaat worden uitgevoerd of is de omgeving simpelweg te groot om ad hoc handelingen efficient via de GUI uit te voeren. Automation is dan het antwoord. Powershell is de bijbehorende tool.

Dat geldt ook voor SQL Server DBA's aangezien SQL Server een onderdeel is van een Windows omgeving (alhoewel veel al via T-SQL script en de Agent is geautomatiseerd). In dit artikel begin ik met een introductie in Powershell om vervolgens te kijken naar enkele specifieke SQL Server toepasssingen. En hopelijk wordt de meerwaarde van Powrshell daarmee enigzins duidelijk.

De bouwstenen

Powershell is een scriptingomgeving waarin een aantal technieken samenkomen. Zo zullen beheerders die nog via de commandprompt hebben gewerkt (of dat nog steeds doen) bekende commando's tegenkomen zoals bijvoorbeeld cd en dir. Voor Linux-mensen is er het bekende ls in plaats van dir. Zowel dir als ls zijn aliassen voor het Powershell commando Get-Childitem. Het equivalent van cd is Set-Location. In Powershell noemen we dat een Cmdlet (command-let).
CmdLets kunnen we zien als utilities die functionaliteit geven aan de commandprompt. Ze volgen allemaal een naamgevingsconventie bestaand uit een werkwoord en een zelfstandig naamwoord met een streepje (-) ertussen. Om een lijst te krijgen van alle beschikbare cmdlets kunt u de cmdlet Get-Command uitvoeren. Dit doet u in een Powershell sessie. Om een sessie te starten, klik op start, tik powershell in het zoekvak en kies Windows Powershell als u graag via de commandprompt werkt of kies Windows Powershell ISE als u liever een grafische interface gebruikt.



FIGUUR 1.

Als u Get-Command heeft uitgevoerd ziet u dat er veel cmdlets zijn. Met behulp van Get-Help krijgt u meer informatie over hoe deze cmdlets te gebruiken en wat ze doen. De syntax is:

```
Get-Help Get-Command
```

Zoals gebruikelijk in programmeertalen is de functionaliteit uit te breiden door zelf scripts te schrijven. Vanaf Powershell 2.0 kunnen we zelf modules schrijven (ter vervanging van de snap-ins uit versie 1.0) en de functionaliteit van de module toevoegen aan de sessie waarin we werken met de cmdlet Import-Module. Een belangrijke module voor DBA's is de module SQLPSX. Deze is te downloaden via SQLPSX.Codeplex.com en bevat veel handige SQL Server specifieke cmdlets.
Zoals gezegd komen een aantal technieken samen in Powershell. Naast de commandprompt-achtige manier van werken, is Powershell een object georienteerde scripttaal. Bovendien is het gebaseerd op het .NET framework. Voor SQL Server gerelateerde scripts houdt dat in dat we zowel ADO.NET als SMO beschikbaar hebben. SMO, ofwel SQL Management Objects, is het objectmodel waarop SQL Server Management Studio (SSMS) is gebaseerd. Alle management functionaliteit die we nodig hebben, kunnen we via deze objecten bereiken. Dus ook mijn persoonlijk favoriete feature van SSMS: Scripting van objecten en acties!

Naast de verschillende .NET libraries die we kunnen gebruiken, is er de cmdlet Get-WMIObject. De naam spreekt voor zich: ook alle informatie die we via WMI (Windows Management Instrumentation) kunnen opvragen, is beschikbaar vanuit Powershell. Dit geeft ons bijvoorbeeld de mogelijkheid om met één script informatie te vergaren over ons Windows systeem (zaken als soort hardware, OS version en patchlevel, drives met hoeveelheid beschikbare ruimte, ...) als ook SQL specifieke zaken (welke databases en hun status, grootte, recovery model, ...). Zowel voor een consultant die voor het eerst bij een klant komt, als voor een DBA die met zekere regelmaat de stand van zaken opneemt, een echte timesaver.

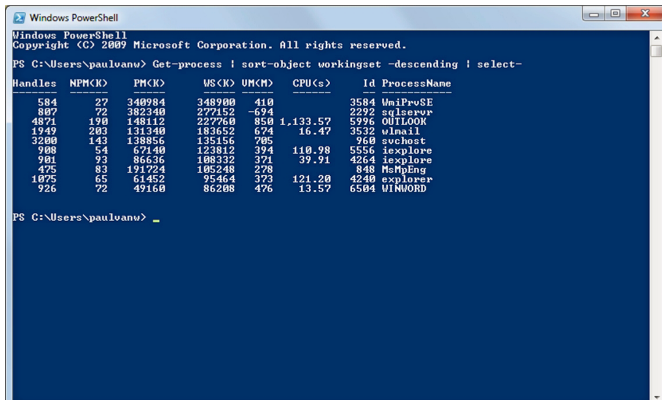
Syntax, objectorientatie en de pipe

Voordat ik enkele specifieke SQL Server scripts zal uitwerken, kijken we eerst nog naar enkele syntaxzaken. Zoals gezegd is Powershell object oriented. Dat houdt in dat het resultaat van een cmdlet altijd een object is. Andere cmdlets kunnen weer met zo'n

object werken. Bekijk als voorbeeld het onderstaande script:

```
Get-process | sort-object workingset -descending | select-object
 -first 10
```

De eerste cmdlet, get-process, levert een lijst op van alle processen op de machine (eigenlijk moet ik hier het woord collection gebruiken in plaats van lijst). Met de pipe (|) geven we het resultaat van get-process (de collection) door aan de cmdlet sort-object die alle processen sorteert op workingset (ofwel de hoeveelheid geheugen die ze gebruiken) met descending als extra parameter (parameters worden voorafgegaan door een -). De tweede pipe geeft het resultaat door aan de cmdlet select-object die in dit voorbeeld het resultaat beperkt tot de eerste tien. Met andere woorden: de output is de top 10 memory consumers op het systeem.



FIGUUR 2.

Voordat we naar de SQL Server scripts gaan kijken eerst nog enkele opmerkingen. Bij programmeren horen variabelen en control flow elementen. Variabelen beginnen in Powershell met een $ (dollarteken) en kunnen zonder expliciete deklaratie gebruikt worden. Bijvoorbeeld:

```
$MijnTekst = 'Powershell is cool'
```

Een handige feature met string variabelen is string substitution. Bekijk onderstaand script:

```
$Instance = 'SQL01'
$Database = 'AdventureWorks'
$ConnectionString = "Data Source=$Instance; Integrated
 Security=SSPI; Initial Catalog=$Database"
```

Door de dubbele quotes in de initialisatie van de variabele $ConnectionString vindt er string substitution plaats. Overal waar een $ staat, wordt de waarde van de bijbehorende variabele ingevuld. Zeer nuttig in het bovenstaande voorbeeld, zeker als we bedenken dat we onze scripts geparameteriseerd kunnen maken. Let wel op named instances met een $ in de naam! Ook de backslash kan problemen geven. Daarvoor bestaan speciale cmdlets: encode-sqlname en decode-sqlname.
Naast variabelen kent Powershell uiteraard de gebruikelijke programeercontructies zoals if, while, for, foreach en enkele varianten. Naast statements bestaan dit soort constructies ook als cmdlet, zoals bijvoorbeeld de cmdlet foreach-object.

Eindelijk SQL Server

De eerste leuke feature van Powershell is dat het ons in staat stelt SQL Server te zien als filesystem. Met de commandprompt kunnen we vanuit de root van een drive (bijvoorbeeld c:) de hele schijf bekijken. Met Powershell hebben we op vergelijkbare wijze een drive SQLSERVER:. De cmdlet get-psdrive geeft een lijst van beschikbare drives waar we naar kunnen kijken. Als u de Powershell omgeving hebt gestart via SSMS zult u SQL Server zien staan. Als u een 'gewone' sessie heeft gestart ziet u SQL waarschijnlijk niet. Er is een aparte snap-in die we moeten laden om deze functionaliteit te krijgen. Als u op internet zoekt op initialize-sqlpsenvironment vindt u het script om dit te doen. Wellicht dat u nog een security setting moet doen. Default laat Powershell het runnen van scripts niet toe. Via set-executionpolicy unrestricted kunt u dat omzeilen. Laat het gezegd zijn dat dat niet de recommended setting is.
Na het runnen van het script staat SQLServer tussen de beschikbare drives. Met set-location kunnen we nu door SQL Server browsen waarbij gebruik gemaakt wordt van de object hierarchy zoals gedefinieerd in SMO. Onderstaand script geeft een lijst van alle databases die beschikbaar zijn, gesorteerd op grote:

```
Set-location SQLSERVER:\sql\laptop\DEFAULT\Databases
Get-childitem | SELECT Name, Collation, CompatibilityLevel, Auto-
Shrink, RecoveryModel, Size, SpaceAvailable | sort-object Size
```

In de eerste regel is laptop de naam van de machine en default geeft aan dat we de default instance van SQL Server bekijken (en dus niet een named instance).
De andere benadering was geweest om rechtstreeks gebruik te maken van SMO. Onderstaand script laat dat zien.

```
$v = [System.Reflection.Assembly]::LoadWithPartialName
( 'Microsoft.SqlServer.SMO')
if ((($v.FullName.Split(','))[1].Split('='))[1].Split('.')[0] -ne '9') {
  [System.Reflection.Assembly]::LoadWithPartialName
('Microsoft.SqlServer.SMOExtended') | out-null
  [System.Reflection.Assembly]::LoadWithPartialName
('Microsoft.SqlServer.SQLWMIManagement') | out-null
  }
$s = New-Object ('Microsoft.SqlServer.Management.Smo.Server')
"(local)"
$dbs=$s.Databases
#$dbs | Get-Member -MemberType Property
$dbs | SELECT Name, Collation, CompatibilityLevel, AutoShrink,
RecoveryModel, Size, SpaceAvailable | sort-object Size|
format _ table
```

Merk bij dit script op dat we de SMO bibliotheek toevoegen. De eerste regel laadt de SMO library via de LoadWithPartialName functie. Dat is afdoende voor SQL Server 2005 instances. In SQL Server 2008 is er een extended versie bijgekomen en sommige functionaliteit is verhuisd naar deze extended assembly. In de if controleren we de versie van de zojuist geladen library. Na de komma staat in de fullname van de dll het versienummer van de dll achter het eerste =-teken. –ne staat voor not equal en 9 is het versienummer van SQL Server 2005. In het geval van SQL Server 2008 laden we de SMOExtended library. We doen hier deze moeite om het script onafhankelijk te maken van de SQL Server versie.

Merk verder op dat de voorlaatste regel begint met #. Dat maakt de gehele regel commentaar.
Eén van de voordelen van Powershell is dat het aantal servers (instances) dat u beheert er eigenlijk niet toe doet. Een script loopt net zo makkelijk 100 servers af als slechts 1. Onderstaand script gebruikt de foreach cmdlet om alle servers uit een registered server group te halen:

```
Foreach ($reg in get-childitem SQLSERVER:\SQLregistration\
Central Management Server Group\Productie Servers)
{
#Do Work
}
```

Bovenstaande code gaat ervan uit dat de locale machine de Central Management Server is waarin we een servergroep hebben aangemaakt die Productie Servers heet.
Dit script is makkelijk te combineren met bijvoorbeeld het uitvoeren van een script dat performancedata uit de verschillende servers leest. Als we deze data inlezen in een database en daar onze eigen rapporten bovenop zetten, hebben we een eenvoudige eigen monitoringtool gebouwd.

Een andere mooie toepassing voor Powrshell is het gebruik van policies. Met SQL Server 2008 werd Policy based Management geintroduceerd. Dit stelt ons in staat om onze best practices voor instances en databases vast te leggen in policies. We kunnen hierbij denken aan bijvoorbeeld naamgevingsconventies (alle viewnamen moeten beginnen met de letter v) of zaken als databasesettings (autoshrink moet op false, data en log files op verschillende drives, ...). Deze policies kunnen we vervolgens afdwingen, handmatig controleren, of gescheduled controleren. Als we in SSMS via de GUI policies maken die we gescheduled willen laten controleren, maakt SSMS voor elke policy een aparte job aan. Aangezien er waarschijnlijk een aanzienlijk aantal policies zal bestaan, worden dat erg veel jobs. Via Powershell kunnen we deze policies makkelijk maken en allemaal vanuit een en dezelfde job laten controleren. Onderstaand vindt u de basis van deze oplossing. Op codeplex kunt u een volledig uitgewerkte oplossing vinden.

```
$s = new-object ('Microsoft.SqlServer.Management.Smo.Server') $inst

$j = new-object ('Microsoft.SqlServer.Management.Smo.Agent.Job')
($s.JobServer, 'EvalDevDB')
$j.Description = 'Evaluate Dev DB Policy'
$j.Category = '[Uncategorized (Local)]'
$j.OwnerLoginName = 'sa'
$j.Create()
$jid = $j.JobID

$js = new-object ('Microsoft.SqlServer.Management.Smo.Agent.
JobStep') ($j, 'Step 01')
$js.SubSystem = 'PowerShell'
$js.Command = "C:\Demos\poleval.ps1 '$inst'"
$js.OnSuccessAction = 'QuitWithSuccess'
$js.OnFailAction = 'QuitWithFailure'
$js.Create()
$jsid = $js.ID

$j.ApplyToTargetServer($s.Name)
$j.StartStepID = $jsid
$j.Alter()

$jsch = new-object ('Microsoft.SqlServer.Management.Smo.Agent.
JobSchedule') ($j, 'Sched 01')
$jsch.FrequencyTypes = 'Daily'
$jsch.FrequencySubDayTypes = 'Once'
$startts = new-object System.Timespan(22, 0, 0)
$jsch.ActiveStartTimeOfDay = $startts
$endts = new-object System.Timespan(23, 59, 59)
$jsch.ActiveEndTimeOfDay = $endts
$jsch.FrequencyInterval = 1
$jsch.ActiveStartDate = get-date
$jsch.Create()
```

In dit script worden in volgorde via de cmdlet new-object een job, een jobstep en een schedule gemaakt. Van elk worden de juiste properties gesteld om ze vervolgens via de Create method echt aan te maken. Dit alles komt vanuit de SMO library. De truc zit voor een groot deel in de jobstep. Vanaf SQL Server 2008 kent SQL Server Powershell als jobstep type. In de gemaakte jobstep wordt een Powershell script (poleval.ps1) aangeroepen dat verschillende policies evalueert. Dit script kunt u vinden op codeplex. In alle tot nu toe getoonde scripts zit geen error handling.

> Waarom een DBA Powershell moet gebruiken? Het kan in het beheer heel veel tijd besparen!

Het spreekt (hopelijk) voor zich dat we dat in productiecode wel moeten doen. Powershell 1.0 kent daartoe de constructie Trap {error handling code} . Dit gebruikt u aan het begin van het script en zodra er een error is, wordt terug gesprongen naar dit code blok. In Powershell 2.0 hebben we het vertrouwde Try .. Catch. Dit geeft ons meer flexibiliteit aangezien we de verschillende soorten errors met verschillende catch blokken kunnen afvangen.

Tot slot

Hopelijk heeft bovenstaande snelle introductie van Powershell u een idee gegeven van de kracht van Powershell. De vraag die rest is waarom en vooral wanneer een DBA Powershell moet gebruiken. Het antwoord is dat het scripten van beheertaken veel (heel veel) tijd kan besparen. Vaak zullen T-SQL scripts ook volstaan, bijvoorbeeld voor zaken als backup van databases en index rebuilds als de maintenance plans niet meer genoeg flexibiliteit geven. Maar vaak ook gaan onze beheertaken verder dan wat met T-SQL bereikt kan worden. Verzamelen van systeemconfiguraties en system health information op OS- en/of hardwarevlak zijn goede voorbeelden. En vaak gaat beheer verder dan alleen de SQL Server en willen we onze DBA taken kunnen integreren met meer generieke Windows beheer taken. In alle gevallen kon Powershell weleens de redder in nood blijken te zijn.

Eric ter Braake, is zelfstandig trainer/consultant. Hij is te bereiken via email. trainsql@live.nl.