Powershell nog krachtiger in nieuwe versie

Onderliggende techniek voor managementsoftware

Sander Klaassen

Powershell wordt ingezet als managementlaag voor een scala aan software. Microsoft heeft in 2005 het doel uitgesproken om powershell te gebruiken als onderliggende techniek voor alle management software. Dit is inmiddels terug te zien aan het groeiend aantal producten waar een management GUI powershell commando's uitvoert. Het is voor beheerders zaak deze techniek onder de knie te krijgen, zodat de ware kracht van het platform kan worden benut.

Het gevolg hiervan is dat in de nabije toekomst bij maatwerkapplicaties een Powershell managementlaag wordt verwacht. Met het verschijnen van Powershell versie 2 komt er nieuwe functionaliteit beschikbaar die ook voor deze maatwerkoplossingen goed van pas kan komen.
Het aanspreken van externe code is in versie 2 van Powershell ook uitgebreid. In dit artikel worden naast een aantal nieuwe key features ook de methodes besproken hoe externe code aan te spreken.

Externe code

In Powershell 2 is een nieuw cmdlet geïntroduceerd dat de mogelijkheden om externe code aan te spreken sterk uitbreidt. Zonder tussenkomst van Add-Type zijn de .net libaries al beschikbaar.

```
PS C:\PS> $ping = new-object System.Net.NetworkInformation.Ping
PS C:\PS> $ping.Send("localhost")
```

Daarnaast was het in versie 1 al mogelijk om comobjecten in een sessie te laden:

```
PS C:\PS>$ie = new-object -comobject InternetExplorer.Application
PS C:\PS>$ie.navigate2("www.microsoft.com")
PS C:\PS>$ie.visible = $true
```

Naast Internet Explorer kunnen andere applicaties ook worden aangesproken zoals Microsoft Word en Excel, mits geinstalleerd uiteraard.

In powershell versie 2 is een nieuwe cmdlet geintroduceerd die het aantal manieren om externe code aan te spreken vergroot. Om een idee te krijgen welke constructies gebruikt kunnen worden volgen er drie voorbeelden van methodes om externe code te integreren in een script. Een externe .net klasse het aanspreken, het 'on-the-fly' compileren van C# code.
Omdat met het verschijnen van Powershell versie 2 het niet overal direct beschikbaar is wordt ook stil gestaan bij het wel of niet beschikbaar zijn van een alternatief voor Powershell v1.

Add-Type -path
In dit voorbeeld wordt de functionaliteit van een eenvoudige dll beschikbaar gemaakt in een Powershell sessie. In Visual Studio is een eenvoudige dll gemaakt met de volgende broncode:

```
using System;
using System.Text;

namespace TestClass
{
    public class Class1
    {
        public void SayIt(string input)
        {
            Console.WriteLine(input);
        }
    }
}
```

Daarna is de dll als 'referentie' toegevoegd in een Powershell sessie:

```
PS C:\PS> Add-Type -path "C:\ Projects\Test2\Test2\TestClass.dll"
```

In Powershell 1 kon bovenstaande ook, maar op een minder elegante manier:

```
[Reflection.Assembly]::LoadFrom((ls c:\ Projects\Test2\Test2\Test
Class.dll))
```

Om een object te definiëren en in een Powershell variabele te zetten is niets gewijzigd ten opzichte van versie 1:

```
PS C:\PS> $myObject = new-object TestClass.Class1
```

Een Methode aanroepen gaat als volgt:

```
PS C:\PS> $myObject.SayIt("great success")
great success
```

Add-Type –assemblyname
Ook hiervoor is een Powershell versie 1 equivalent:

```
[Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
```

Het alternatief dat versie 2 biedt is een stuk fraaier:

```
Add-Type -AssemblyName System.Windows.Forms
```

Wildcards worden ondersteund, system.windows.form* werkt ook.

Add-Type en brondcode
Deze methode kan alleen uitgevoerd worden met versie 2. Er is geen alternatief beschikbaar voor gebruikers van Powershell versie 1. De broncode wordt in een variable gezet:

```
C:\PS>$source = @"
public class BasicTest
{
    public static int Add(int a, int b)
    {
        return (a + b);
    }

    public int Multiply(int a, int b)
    {
        return (a * b);
    }
}
"@
```

De broncode wordt als TypeDefinition toegevoegd in de powershell sessie:

```
C:\PS> Add-Type -TypeDefinition $source
```

Nu is de functionalteit op dezelfde manier beschikbaar als in het vorige voorbeeld:

```
C:\PS> $basicTestObject = New-Object BasicTest
C:\PS> $basicTestObject.Multiply(5, 2)
10
```

Nieuwe Features

Remoting
De meest gevierde nieuwe functionaliteit is remoting. Op machine A wordt een commando gegeven dat op machine B wordt uitgevoerd. Deze functionaliteit maakt gebruikt van een service 'Windows Remote Management (WS-Management)'. Voordat deze service Powershell commando's accepteert, moet deze eerst worden geconfigureerd. Dit is met een commando vanuit een 'elevated' DOS-prompt te doen: 'winrm –quickconfig'. Daarna is Remoting op twee manieren te gebruiken:

- het starten van een cmdlet met als parameter een server, waardoor er gegevens van die server lokaal beschikbaar komen. (zoals invoke-command;
- Het starten van een sessie, waardoor een cmdlet wordt gestart op de remote machine.

In Versie 2 zijn een aantal cmdlet's uitgebreid met een –Computername paramater.

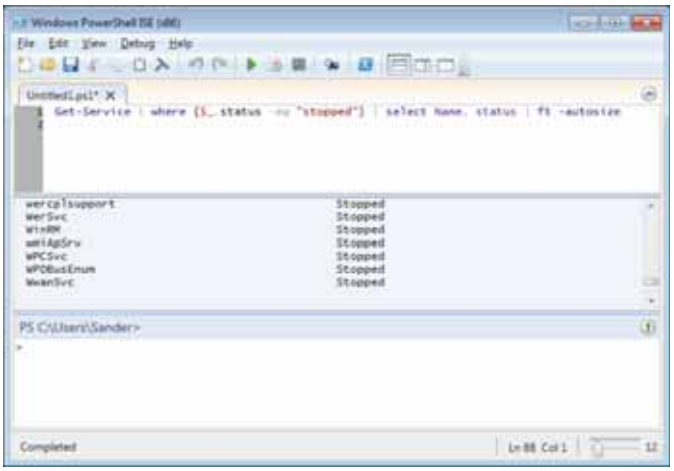
WinRM is de Microsoft implementatie van het WS-Management protocol, een openstandaard op SOAP gebaseerd, firewall-compatible manier van communiceren.

Het gebruik van een sessie is behoorlijk intuïtief en begint met New-PSSession, kijken welke sessies bestaan met Get-PSSession. Verbinden aan een bestaande sessie; Enter-PSSession en eenmaal verbonden met een sessie kan een commando worden uitgevoerd met Invoke-Command.

Background Jobs
Een nadeel van versie 1 was dat bij het sluiten van een powershell prompt ook de werkende code stopte. In versie 2 is het mogelijk een background job te starten die actief blijft, ook al wordt de Powershell sessie afgesloten. Er zijn zes cmdlets om background jobs: starten, stoppen, opvragen, uitlezen, 'wachten op' en verwijderen. (get-command *-job)

Als een Powershell background job wordt gestart, wordt het resultaat van die job niet meteen getoond. In plaats daarvan geeft het startcommando (start-Job) een object terug waarin bruikbare informatie staat, maar geen resultaat zichtbaar is. Daardoor kunnen we de powershell sessie blijven gebruiken, ook al draait er een background job op de achtergrond of is er een background job klaar.
Om het resultaat van van een background job te zien moet het 'Receive-Job' cmdlet worden gebruikt. Als dit commando wordt gegeven, wordt het resultaat tot dan toe getoond. Als op een later moment dit commando nogmaals wordt gegeven verschijnen de overige resultaten.

Windows PowerShell ISE
Een in het oog springend nieuwe feature is de grafische script editor genaamd Powershell Integrated Scripting Environment, afgekort 'Powershell ISE'. Wanneer ISE wordt gestart is het scherm opgebouwd uit drie delen (zie figuur 1) en met de standaardinstellingen alsvolgt ingedeeld:



HET SCHERM IS STANDAARD INGEDEELD IN DRIE DELEN ALS ISE WORDT GESTART.

Het bovenste deel is de script editor. Hier worden de scripts in ontworpen, geschreven, getest en gedebugged. Dit deel is verrijkt met kleurherkenning voor cmdlets, operators en sleutelwoorden. De code completion werkt naast standaard cmdlets, eigenschapen en methodes ook voor zelf gedefinieerde functies, eigenschappen en methodes. Daarnaast is hier contextgevoelige help beschikbaar, type 'invoke-command' gevolgd door F1, dan verschijnt de helpinformatie van invoke-command.
Het middelste deel is het output scherm het onderste scherm is een powershell command prompt. Het onderste deel is de com-

mand pane, een realtime powershell commandline sessie, waarmee tijdens het schrijven van het script losse commando's kunnen worden geprobeerd. Dit is dezelfde powershell sessie als het script pane. Een functie gedefinieerd in het script pane en daarna uitgevoerd, is direct beschikbaar in het command pane.
Natuurlijk blijven de standaard beveilingsinstellingen van kracht bij ISE. En voordat er zelfgeschreven scripts kunnen worden uitgevoerd, moet de execution policy (set-ExecutionPolicy) op 'RemoteOnly' worden gezet. Waardoor het toegestaan wordt lokaal geschreven scripts te starten.

Deze tool vormt een fraaie omgeving voor het schrijven van scripts, het debuggen is de moeite waard om een apparte paragraaf aan te besteden.

Script Debugger
Ook de debugmogelijkheden zijn uitgebreid in Powershell v2. In Versie 1 waren twee methodes beschikbaar, nu zijn er vier. Daarnaast is de try en catch constructie uit c# rechtreeks overgenomen. De debug methodes op een rij:

- *Write-Debug (ook in Powershell 1)*
  Dit commando schrijft de informatie naar de host mits de powershell environment daartoe is geconfigureerd. In de 'variable provider' (dir variabele:) bevindt zich een variabele die heet DebugPreference, als deze van SilentyContinue wordt aangepast in Continue zal alle informatie die met write-debug wordt aangeroepen worden getoond. Deze instelling is te veranderen met het volgende commando: $DebugPreference = "continue"

- *Set-PsDebug (ook in Powershell 1)*
  Hiermee is powershell te instrueren bij elke regel code een script te onderbreken en te vragen of deze door moet gaan met de volgende regel. Zo zijn per regel de waardes van de variabele in een script realtime uit te lezen. Het volledige commando om dit 'stap gedrag aan te zetten is Set-PSDebug –step. Uitzetten gaat met Set-PSDebug -off
  Met de –strict parameter wordt hetzelfde bereikt als in vbscript met 'Option Explicit'. Alle gebruikte variabelen moeten worden gedeclareerd. (in vbscript Dim strVariabele as String) De scoop van dit commando is globaal, net als in VBScript.

- *Set-StrictMode (nieuw in versie2)*
  Set-StrictMode - version 2.0. Dit schakelt 'strict mode' aan in de huidige scoop waarin het wordt aangeroepen (en alle onderliggende scopes). Hier wordt niet alleen gelet op declaraties maar ook op wel of niet bestaande methodes/eigenschappen en syntax. Set-StrictMode –off schakelt dit gedrag weer uit.

- *Powershell debugger (nieuw in versie2)*
  De powershell debugger is alleen te gebruiken voor het debuggen van lokale scripts. Met het cmdlet Set-PSBreakPoint (standaard alias is sbp) kan in te debuggen script een regelnummer of functie aangegeven als 'breakpoint'.

```
spb -command fnCheckInt -script OrderCheck.ps1
```

Als het script na het geven van het bovenstaan commando eenmaal dat break-point bereikt, springt powershell in de debug modus. Hierin kunnen de waardes van alle variabele realtime worden opgeroepen. Het script kan dan per regel of volledig worden uitgevoerd met commando's' of de scriptcode kan worden opgeroepen (5 regels voor en 10 regels na het breekpunt met 'l')
Met Set-Breakpoint kunnen ook acties worden gedefineerd. Zodat er geen debug modes verschijnen, maar een voorgedefineerde actie wordt uitgevoerd:

```
spb -command fnCheckInt -script OrderCheck.ps1 -action {add-
content "intOrder value = $intOrder." -path action.log}
```

Met de actie parameter kan logica worden ingebouwd, dit is zo complex te maken als je zelf witl. In hoeverre dit wenselijk is, is maar de vraag. Voordat je het weet, ben je debug logica aan het debuggen. Een volledig debug statement kan er als volgt uitzien:

```
spb -command fnCheckInt -script OrderCheck.ps1 -action {if
($intOrder -gt 42) {add-content "intOrder value = $intOrder." -path
 action.log}}
```

Als eenmaal een breakpoint gedefinieerd is, is deze niet meer aan te passen. De enige mogelijkheid om dit wel te doen is om het breakpoint in zijn geheel te verwijderen en een nieuwe te definieren. Het verwijderen kan alsvolgt:

```
get-psbreakpoint | remove-psbreakpoint
```

WMI Cmdlets
In Powershell 1 was het wmi cmdlet get-wmiobject de enige manier om een andere machine remote te benaderen. Met Get-wmiobject kwam alle managementinformatie in WMI als objectvorm in Powershell beschikbaar. Met wat gehack kunnen wmi methods ook worden gebruikt. In Powershell 2 zijn er een aantal wmi cmdlets toegevoegd, die het gebruik van wmi vergemakkelijken en uitbreiden:

Invoke-WmiMethod
Een directe vergelijking tussen de syntax van Powershell versie 1 en 2 voor het starten van notepad via WMI:

```
([wmiclass]"Win32 _ Process").Create("notepad")

invoke-wmimethod -path win32 _ process -name create -argumentlist
 notepad.exe
```

Toch zit er een addertje onder het gras. Bovenstaande constructie met invoke-wmimethod werkt alleen met static methods van een wmi class, daar zijn er maar enkele van. Voor bijvoorbeeld instant based methods word het een heel ander verhaal. Hoe dat precies zit gaat te ver om in dit artikel uit te diepen.

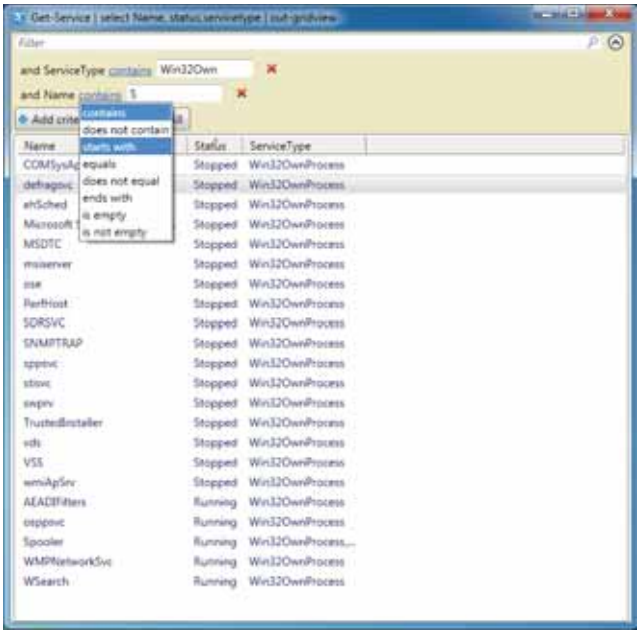
Set-WmInstance
Hiermee kunnen nieuwe wmi instances van bestaande wmi classes worden gemaakt of bestaande wmi instances worden geupdate. Waar in Powershell versie 1 alleen informatie kon worden opgehaald (of methodes konnen worden uitgevoerd) is het nu mogelijk informatie naar de wmi repository te schrijven.

The Get-WinEvent Cmdlet
Dit cmdlet is nieuw in Powershell 2. Het kan windows events ophalen van alle logs (application, system, maar ook andere logs), het kan van een lokale en remote computers worden opgehaald en worden vergeleken. Sinds het eventlog systeem is gewijzigd in Windows 2008 /Vista is de hoeveelheid beschikbare informatie enorm toegenomen. Om een idee te krijgen hoeveel het is, is het aardig om uit een elevated powershell (rechtsklik op het powerhsell icoon,'run as administrator') de volgende commandregel te starten:

```
Get-WinEvent -ListLog * | Select Logname, ProviderNames
```

The Get-Counter Cmdlet
In Powershell2 is get-Counter geïntroduceerd. Daarmee is alle performance counter informatie uit Windows (van de lokale en remote machines) beschikbaar.

The Out-Gridview Cmdlet
Een van de redenen van adhoc scripting is op korte termijn gegevens uit een omgeving te onttrekken. Hoewel met Format-Table en Format-List de informatie goed te intepreteren valt, blijft vaak de wens om de informatie meer op een excel manier te tonen. Het antwoord op deze wens is Out-Gridview. In out grid view kunnen filters te zetten op de output van het commando, en data wordt gesorteerd door op de colom naam te klikken (zie figuur2).



DE OUT-GRIDVIEW.

Conclusie

Powershell is uniek om in één scripttaal gegevens uit verschillende producten te benaderen. Dit is een ongekend krachtige functionaliteit die in de toekomst steeds vaker van pas komt naarmate de afhankelijkheid groeit tussen een toenemend aantal powershell-enabled producten. De tweede versie van Powershell brengt een aantal welkome aanvullingen met zich mee die de bruikbaarheid vergroten en het gebruik vergemakkelijken.

Sander Klaassen, is consultant bij Inovativ. Hij is te bereiken op sander.klaassen@inovativ.nl