**Monitoring SQL Server Agent with Powershell**

By [**Mike Femenella**](http://www.sqlteam.com/author/mike-femenella) on **24 March 2009** | [2 Comments](http://www.sqlteam.com/forums/topic.asp?TOPIC_ID=122354) | Tags: [Administration](http://www.sqlteam.com/tag/sql-server-administration), [SQL Server 2008 Features](http://www.sqlteam.com/tag/sql-server-2008-features)

This article introduces the reader to Powershell. The application that it demonstrates is one that monitors SQL Server Agent to make sure it is running.

A few months ago we had an interesting situation occur; SQLAgent shutdown and it took a while before we realized what had happened. Fortunately it wasn't a critical issue on that particular server but it could have been a major problem on other production servers. This made me ask an interesting question; how do you monitor SQL Server, specifically SQLAgent without using SQLAgent to run a job? Enter Powershell.

To work through the example presented below I am assuming you are connecting to servers on a domain that your machine is also a part of. I'm also assuming you have SMTP configured correctly. Powershell is Microsoft's new scripting language and once you work through a few examples it really is amazingly simple to use and very powerful. The current version of Powershell is version 1.0. Powershell 2.0 is currently in CTP (Community Technology Preview). The solution described below is written in 1.0 but some new features of 2.0 will probably cause me to rewrite it so I can take the new version for a test drive. In order to use Powershell against a remote server, you must have the same version of Powershell installed on that server.

One of the best parts of Powershell is that it’s free. You can download it from Microsoft at<http://www.microsoft.com/windowsserver2003/technologies/management/powershell/download.mspx>

Powershell's interface is modest at best and I found it at times not really convenient to work with. Fortunately Quest makes a tool called PowerGUI which makes things much simpler by adding intellisense and debugging support. As with Powershell itself, it is also a free download by going to<http://powergui.org/downloads.jspa>. (*Editor's Note: If you don't want to download this utility, I suggest this tutorial on* [*running powershell scripts*](http://computerperformance.co.uk/powershell/powershell_cmdlet.htm).)

Let’s start solving the problem I outlined earlier by setting up a text file that will contain a list of servers that we want to monitor. Just open up a text editor, and type in your server IP addresses or names or a mix of both and save the file to your local directory. In my case I’m using c:\servers.txt such as:

localhost
111.11.111.11
111.11.111.22

Now let’s make sure we can retrieve the contents of the file:

#get a list of servers

$servers=get-content "c:\servers.txt"

$servers

In Powershell, you define variables using the $ character rather than the @ that we’re used to in SQL Server. Variables don’t need to be defined as a particular data type, although you can cast it into other types as needed provided the cast makes sense. For example casting a hash table into a string wouldn’t work. It took me a while to get my brain around that concept so let’s look at an example. Modify your servers.txt file so that there is only 1 server name and run the following:

#get a list of servers

$servers=get-content "c:\servers.txt"

$servers.GetType().FullName

Notice that $servers is of type system.string which is expected. Now, add one or more server to your servers.txt file and run the code again. $servers is now of type system.object. Why would this be? The reason is that with multiple values it can’t set the variable to be a string anymore, it has to create an array of objects that hold the value of each one of your servers. This comes in very handy as we’ll see later in our code.

As with any other programming or scripting language we have several types of control type commands available to us in Powershell such as while, if and foreach. (*Editor's Note: Please put in actual computers in your servers.txt and not instances. Right now this is attaching to computers and not SQL Server instances.*) In this case we’re going to use a foreach statement to execute a command against each one of our servers:

$servers=get-content "c:\servers.txt"

foreach($server in $servers)

{

 get-wmiobject win32\_service -computername $server |

 select name,state |

 where {

 ($\_.name -like "SQLAGENT\*" -or $\_.name -like "SQL\*AGENT") `

 -and $\_.state -match "Stopped" } | Out-String

}

(*Editor's Note: The formatting here is poor. Powershell doesn't seem to like wrapping. The odd back tick character is the line continuation character. It's the key just to the left of the number one key. If I ever get better at Powershell I'll reformat.*)

For each object that is returned from the $servers variable we’re going to put it in variable $server and then go out and check the server for stopped SQL Server Agent instances. In Powershell, you use the | character to indicate that you want Powershell to pipe the values of the command on the left to the command on the right. Here's what the above script does:

1. First we’re asking Powershell to get the WMIobject values for all win32 services and we’re passing the name of the server into the –computername parameter.
2. Next, we’re taking the values from the get\_wmiobject call and selecting just the name and the state values that the command returns since those are the only ones we’re interested in working with in our example.
3. We’re filtering those results in 2 ways. First, we’re using a wildcard search and the –like operator on the names ($\_.name) for any name that starts with "SQLAgent" or begins with "SQL" and ends with "Agent" and that has (-match) a state of "Stopped".
4. Finally we’re taking our filtered results and telling Powershell to output that information as a string. The final step of outputting it to a string becomes important later.

Now if we stopped there, you could manually monitor any servers you wanted to and see what services were stopped. While this is moderately useful, it would be nice if this could work on a remote server or servers and email me or a distribution list of DBAs if it finds any instances of SQL Server Agent being down. Let’s work through that by slightly modifying the code above.

$servers=get-content "c:\servers.txt"

foreach($server in $servers)

{

# go to each server and return the name and state of services

# that are like "SQLAgent" and where their state is stopped

# return the output as a string

$body=get-wmiobject win32\_service -computername $server |

 select name,state |

 where {($\_.name -like "SQLAGENT\*" -or $\_.name -like "SQL\*AGENT") `

 -and $\_.state -match "Stopped"} |

 Out-String

if ($body.Length -gt 0)

{

 #Create a .net mail client

 $smtp = new-object Net.Mail.SmtpClient("yourmailserver.yourcompany.com")

 $subject="SQL Agent is down on " + $server

 $smtp.Send("from\_email", "to\_email", $subject, $body)

 "message sent"

}

}

Instead of having the output of the get-wmiobject return to the screen, we create a variable called $body which will hold the value of the services that are stopped. Remember earlier that I mentioned that "| Out-String" would be important later. If you didn’t do this you would end up returning objects and the $body would be type system.object which would not go into an email very well. Returning the output as a string means that $body becomes type system.string and so we end up with a string that can be used in the body of an email message.

If $body is a zero length string, then no matches came back and the services we are looking for were not stopped. In this case the script goes back to the top of the foreach loop and checks the next server. In Powershell we don't use =, <,>,<> like we do in SQL server, we have to use their Powershell equivalents, -lt (less than), -gt (greater than), -eq (equals) and –ne (not equals). There are more options available to you to test for case sensitivity and more advanced features but they are beyond the scope of this article.

If $body is greater than zero, we have a match in our search for stopped services. In that case we create a mail object and send out an email for each server where we find stopped services. I chose this approach because I would want one email per server that has stopped services. You could easily rearrange this example to build up $body and test for it’s value outside of the foreach loop and generate one email for all of your servers if that’s an approach you’d prefer to take. One note here is that the "To" address must be a valid address on the mail server you are referencing.

Now we have a handy script that will generate an email if we have a server that has SQLAgent stopped but we need to do our final task, scheduling it to run. The easiest way to accomplish this is Windows Scheduler. Remember we don’t want to use a SQL Server job because if agent goes down, the job will not execute. Let’s look at what we need to schedule our code to run. Save your script as ServiceCheck.ps1 and save it to your C: drive. Next, we need to create a .bat file to contain the code. Open up a new text file and enter the following:

powershell.exe -command C:\ ServiceCheck.ps1

Save this new file as c:\ServiceCheck.bat. Next, open up "Scheduled Tasks" and create a new scheduled task that will run a command line task. In the Run field you would type in: C:\ ServiceCheck.bat with a "Start In" value of C: and then schedule it as needed.

I hope this helps you get started working with Powershell and exploring the possibilities it can bring to your work with SQL Server. Below are some books and links that I found very helpful in getting started with Powershell.

**Resources**

<http://thepowershellguy.com/blogs/posh/> (Blog)

<http://blogs.msdn.com/powershell/> (Blog)

<http://www.manning.com/payette/> (Book)

<http://sqlblog.com/blogs/allen_white/default.aspx> (Blog)