This site uses cookies for analytics, personalized content and ads. By continuing to browse this site, you agree to this use. <u>Learn more</u>

Microsoft

Scripting

Product Blogs V DevOps V Features V Languages V .NET V Platform Development V Data Development V



Summary: Richard Siddaway introduces you to Windows PowerShell jobs on remote machines.

Honorary Scripting Guy, Richard Siddaway, here today filling in for my good friend, The Scripting Guy. This is the third in a series of posts that, hopefully, will shine the spotlight on Windows PowerShell jobs, remind people of their capabilities, and encourage their greater adoption. The full series comprises:

March 4th, 2014

1. Introduction to PowerShell Jobs 2. WMI and CIM Jobs 3. Remote Jobs (this post) 4. Scheduled Jobs 5. Jobs and Workflows

6. Job Processes

7. Jobs in the Enterprise So far, you've seen how to use the core Job cmdlets and the -AsJob parameter, and we discovered

the interesting aspects of the CIM cmdlets and CDXML. One of biggest selling points for Windows PowerShell is the way it enables you to administer remote machines. You have a number of options: Cmdlets with the –ComputerName parameter

Invoke-Command

 Interactive remote sessions Windows PowerShell remoting sessions

Workflows

 CIM sessions WSMAN cmdlets

• Desired State Configuration All of these options enable you to perform at least some amount of administration on the remote

machine. But how do Windows PowerShell jobs fit into this framework? Start-Job doesn't have a -ComputerName parameter. A couple of techniques immediately come to mind. Is there any difference, for instance, between the following two commands? Start-Job -ScriptBlock {Get-Process -ComputerName server03}

Invoke-Command -ScriptBlock {Get-Process} -ComputerName server03 -AsJob

And what do they actually do?

They both start a job. They both call Get-Process. And they both run against a remote machine called

the results come back to the local machine.

server03. In reality, though, there is a difference. The difference is about where the job runs and how you get to

the data. _ | _ X Administrator 64 bit C:\scripts

Windows PowerShell Copyright (C) 2013 Microsoft Corporation. All rights reserved. £> Start-Job -ScriptBlock {Get-Process -ComputerName server03} --------------localhost True Get-Process -ComputerN... E> Invoke-Command -ScriptBlock {Get-Process} -ComputerName server03 -AsJob -----True server03 **Get-Process** Start-Job starts a normal background job. The remote connection is handled within the job script

block. The script block runs as a separate Windows PowerShell process (more on that in Part 6 of the series). The remote connection is managed by Get-Process, so it is destroyed when the data is returned. Notice that the location says localhost, which means that the job is running locally. In the second job, you're using Invoke-Command to run the command and control the connection to the remote machine. Invoke-Command uses standard Windows PowerShell remoting for connectivity

to remote machines. The -AsJob parameter brings the Windows PowerShell job engine into play.

Notice that this time the PSJobTypeName is **RemoteJob** and the location is server03 (the remote machine). The big difference is where the job runs. By using **Start-Job**, it runs locally and the code in the script block handles the connectivity. By using Invoke-Command, the job runs on the remote machine, but

Irrespective of how you started the job, you can use the standard job cmdlets with these jobs. _ D X Administrator 64 bit C:\scripts

£> Get-Job Command localhost Get-Process -ComputerN... Completed server03 **Get-Process** f> Receive-Job -Id 2 -Keep | select -First 1 Id ProcessName 428 csrss f> Receive-Job -Id 4 -Keep | select -First 1 2976 46 0.22 428 csrss Get-Job works as you would expect. You can use Receive-Job in exactly the same way on both jobs.

There are a number of broad scenarios for running jobs on remote machines: Option 1: Run a normal job and perform the remote connectivity inside the job's script block. This is

demonstrated in the first of the examples you've seen. Option 2: Start an interactive session to a remote server. If you start a job in the interactive session,

you have exactly the same experience as if you were running on the local machine, but all actions are performed on the remote machine. Option 3: Run the background job on a remote machine and have the data returned to the local machine. This is demonstrated by the second of the examples you've seen.

Option 4: Run the background job on the remote machine and keep the results on the remote machine. This very similar to Option 2, but you are using a Windows PowerShell remote session rather than an interactive session.

You've seen examples of Options 1 and 3 (although, I'll provide another example later that uses a remote session). So let's look at running jobs in an interactive session. _ D X Administrator 64 bit C:\scripts

[server03]: PS C:\Users\richard\Documents> Start-Job -ScriptBlock {Get-Process} PSJobTypeName State HasMoreData Location Command Job1 BackgroundJob Running localhost **Get-Process** [server03]: PS C:\Users\richard\Documents> Get-Job PSJobTypeName State Location Command BackgroundJob Completed localhost **Get-Process** [server03]: PS C:\Users\richard\Documents> Receive-Job -Id 1 -Keep | select -First 1 WS(K) VM(M) CPU(s) Id ProcessName 3292 26 0.02 424 conhost [server03]: PS C:\Users\richard\Documents> Get-Job | Remove-Job [server03]: PS C:\Users\richard\Documents> Exit-PSSession Interactive remote sessions have to be one of the coolest Windows PowerShell features ever. Use

\$sess, you can enter that session: \$sess = New-PSSession –ComputerName server03

Enter-PSSession to open the session. If you already have a session open referenced by the variable

Enter-PSSession -Session \$sess When you enter the session, your prompt changes to include the remote machine name. In this case

£> Enter-PSSession -ComputerName server03

£> to

[server03]: PS C:\Users\richard\Documents>

changing from

You can then use Start-Job and the other job cmdlets as you would on a local machine. Notice that the location is localhost and the job type is BackgroundJob. Also notice that the first process returned is different when compared to the previous examples. This is because of the interactive session.

Each time you create a new interactive session, the job ID counters reset. Notice that the job starts with an ID of 1 rather than the first job having an ID of 2 when running locally.

existing session, it will remain available for use until you explicitly destroy it. The other option that you haven't see yet is Option 4—running the job on the remote machine and keeping the data on the remote machine.

When you exit the session, it is destroyed if you created it with Enter-PSSession. If you are using an

_ D X Administrator 64 bit C:\scripts £> \$sess = New-PSSession -ComputerName server03 £> Invoke-Command -Session \$sess -ScriptBlock {Start-Job -ScriptBlock {Get-Process}} PSJobTypeName State BackgroundJob Running True localhost

£> Invoke-Command -Session \$sess -ScriptBlock {Get-Job} PSComputerName PSJobTypeName State BackgroundJob Completed True localhost £> Invoke-Command -Session \$sess -ScriptBlock {Receive-Job -Id 1 -keep | select -First 1} WS(K) VM(M) CPU(s) Id ProcessName 4 1368 3292 26 0.02 2724 conhost £> Invoke-Command -Session \$sess -ScriptBlock {Get-Job | Remove-Job} £> \$sess | Remove-PSSession As with any remoting work, if you are going to send multiple commands to the same machine, create a remote session. It's much more efficient compared to creating and tearing down multiple connections. You also need the continuity of the session to ensure that you can access the job when it has completed.

\$sess = New-PSSession -ComputerName server03 With the session in place, you can use it with Invoke-Command to send the Start-Job command to the remote machine.

Invoke-Command -Session \$sess -ScriptBlock {Start-Job -ScriptBlock {Get-Process}} You will get back the job start information, which may be confusing because the job is supposed to be running on the remote machine. Remember that Invoke-Command always returns any response

from an issued command. You can easily test the state of any jobs on your local machine by using:

In this example there were no local jobs. Next, test that the job has completed:

Invoke-Command -Session \$sess -ScriptBlock {Get-Job}

You may need to repeat the test a number of times if it's a long running job. After the job has completed, you can retrieve the data: Invoke-Command -Session \$sess -ScriptBlock {Receive-Job -Id 1 -keep | select -First 1}

\$sess | Remove-PSSession

PSJobTypeName State

more work for yourself in collecting the data.

Get-Job

At the end of your work, always clean up and remove unwanted jobs and the remote sessions. Invoke-Command -Session \$sess -ScriptBlock {Get-Job | Remove-Job}

You were promised a look at using Invoke-Command –AsJob (Option 3) through a Windows PowerShell remoting session. Select Administrator 64 bit C:\scripts

_ D X

Get-Process

Create the session and use the session with Invoke-Command and the -AsJob parameter. The job will run on the remote machine, but the results will be returned to the local machine as shown in the previous image. Access the results and manage the job as any other job on your local machine. So, one question remains...

Should we bring back the data to the local machine or keep the data on the remote machine? That's Options 1-3 versus Option 4. As usual, the answer is, "That it depends." If you are performing some kind of fan-out administration task, you want the data and information

about job success or failure brought back to the local machine. If you can't do this, you're making

be confidential, and it would be more secure if it was left on the remote machine until you need it. That's it for today. Tomorrow you'll learn about creating and using scheduled jobs on remote computers. Bye for now.

I'd reserve the option for keeping the data on the remote machine for situations where the data may

Ed Wilson, Microsoft Scripting Guy

~Richard Thanks again, Richard!

I invite you to follow me on <u>Twitter</u> and <u>Facebook</u>. If you have any questions, send email to me at scripter@microsoft.com, or post your questions on the Official Scripting Guys Forum. See you tomorrow. Until then, peace.

Scripter, PowerShell, vbScript, BAT, CMD Follow Dr Scripto

Dr Scripto

Posted in <u>Uncategorized</u> Tagged <u>background jobs</u>, <u>Richard Siddaway</u>, <u>scripting techniques</u>, <u>Windows PowerShell</u>

.NET | .NET Core

2009 Summer Scripting Games

to join the discussion.

PowerShell Resources PowerShell Documentation

PowerShell Forums PowerShell on TechCommunity PowerShell.org – Community Resource

Scripting Forums

Microsoft Store

0 comments

Aaron Nelson activation

Education

Topics

2011 Scripting Games 2010 Scripting Games 2013 Scripting Games 2012 Scripting Games 2014 Scripting Games 2014 Winter Scripting Games 4.0 | 70-410 2015 Holiday Series Access.Application **Active Directory** Active Directory Application Mode (ADAM)

.NET Framework

September 2018 July 2018 June 2018 May 2018 March 2018 February 2018 December 2017

Developer

Company

Privacy & cookies Terms of use Trademarks Safety & eco About our ads © Microsoft 2019

Archive

October 2019

August 2019

July 2019

September 2019

Enterprise

Stay informed

Surface Pro X Account profile Microsoft in education Microsoft Visual Studio Careers Azure Office for students Surface Laptop 3 **Download Center** AppSource Windows Dev Center **About Microsoft** Microsoft Store support Surface Pro 7 Office 365 for schools Developer Network Company news Automotive Privacy at Microsoft Windows 10 apps Returns Deals for students & parents Government TechNet Microsoft Azure in education Microsoft developer program Office apps Order tracking Healthcare Investors Store locations Manufacturing Channel 9 Diversity and inclusion Office Dev Center Buy online, pick up in store Financial services Accessibility In-store events Retail Microsoft Garage Security

English (United States)

What's new