CHAPTER 12

PowerShell Jobs

In this chapter, we will discuss PowerShell's batch-job execution features, which are implemented in the form of cmdlets. At its simplest, we have the ability to run a script as a background job. However, this requires that the PowerShell session that submitted the job remains active. To avoid this constraint, we can create a scheduled job. PowerShell has a number of cmdlets to support job scheduling. As we will see, these cmdlets are really an interface to the Windows Task Scheduler. We will discuss the cmdlets that support the three components of a scheduled job: the job definition, triggers, and options. The job definition defines what code will be executed. The triggers define when the job should be executed–i.e., the schedule. The options define *how* the job should be executed. We will wrap up the section on job cmdlets by reviewing a script that provides a simple console with which to access our PowerShell jobs. Then, we will discuss why SQL Server Agent is a better solution for job scheduling. We will step through how to create a SQL Server Agent job that executes a PowerShell script. Then, we will discuss how we can easily manipulate SQL Server Agent jobs with PowerShell. Although PowerShell's job scheduling may not be the best solution for production, there are two good reasons we need to understand it. First, PowerShell's job-execution and scheduling features can be very useful in development. Second, PowerShell's remote execution and workflows use batch-execution features, so we need to understand these in order to discuss remote execution and workflows, which will be covered in the next chapter.

PowerShell Jobs

PowerShell has built-in support to run jobs. A job can be run as an immediate background process or be scheduled. Oddly, the internals are different for each method. Jobs executed immediately execute as a background child PowerShell session. To support scheduled jobs, Microsoft decided to leverage the Windows Task Scheduler rather than add native support. To create and maintain scheduled jobs, we use a set of cmdlets from the module `PSScheduledJob` that interfaces with the Windows Task Scheduler. One benefit of this design is that we can use the Task Scheduler to view, modify, and execute PowerShell scheduled jobs. Note: There is also a `ScheduledTasks` module that is available only for Windows Server 2012, Windows 8, and Windows 8.1. This module provides cmdlets to create non-PowerShell scheduled jobs in the Windows Task Scheduler, and it will not be covered here. See Table 12-1 for a list of the cmdlets that support unscheduled and scheduled jobs, and a description of what they do. Note: PowerShell calls a schedule a *trigger*, so the cmdlets ending with *JobTrigger* refer to job scheduling.

Table 12-1. *PowerShell Job cmdlets*

CmdLet	Description	Example
`Add-JobTrigger`	Create a new job schedule	`Add-JobTrigger -Trigger (New-JobTrigger -AtStartup) -Name Bryan1PSJob`
`Disable-JobTrigger`	Disable a job schedule	`Get-ScheduledJob -Name MyDailyJob | Get-JobTrigger | Disable-JobTrigger`
`Disable-ScheduledJob`	Disable a scheduled job	`Disable-ScheduledJob -ID 5`
`Enable-JobTrigger`	Enable a job schedule	`Get-ScheduledJob -Name MyDailyJob | Get-JobTrigger | Enable-JobTrigger`
`Enable-ScheduledJob`	Enable a scheduled job	`Enable-ScheduledJob -ID 5`
`Get-Job`	Retrieve job information	`Get-Job -Name MyDailyJob`
`Get-JobTrigger`	Retrieve job schedule information	`Get-JobTrigger -Id 5`
`Get-ScheduledJob`	Get information on scheduled jobs on the computer	`Get-ScheduledJob -Name T*`
`Get-ScheduledJobOption`	Get job option information for scheduled jobs on the computer	`Get-ScheduledJobOption -Name MyDailyJob`
`New-JobTrigger`	Create a new job schedule	`New-JobTrigger -Daily -At 5PM`
`New-ScheduledJobOption`	Create a new scheduled job option	`New-ScheduledJobOption -RequireNetwork`
`Receive-Job`	Retrieve the output from a job	`Receive-Job -Name MyDailyJob -Keep`
`Register-ScheduledJob`	Create a scheduled job	`Register-ScheduledJob –Name TestJob -ScriptBlock { dir $home  }`
`Remove-Job`	Delete a job	`Remove-Job -Id 2`
`Remove-JobTrigger`	Delete a job schedule	`Remove-JobTrigger -Name MyDailyJob`
`Resume-Job`	Resume execution of a previously suspended job	`Resume-Job –Id 4`
`Set-JobTrigger`	Modify a job trigger of a scheduled job	`Get-JobTrigger -Name MyDailyJob | Set-JobTrigger -At "12:00 AM"`
`Set-ScheduledJob`	Modify a scheduled job	`Get-ScheduledJob | Set-ScheduledJob -ClearExecutionHistory`
`Set-ScheduledJobOption`	Modify a job option of a scheduled job	`Get-ScheduledJobOption -Name MyDailyJob | Set-ScheduledJobOption -WakeToRun`
`Start-Job`	Begin execution of a PowerShell background job	`Start-Job -scriptblock { Get-Date }`
`Stop-Job`	Terminate execution of a job	`Stop-Job -Name Job22`
`Suspend-Job`	Suspend execution of a job	`Suspend-Job –Name Job25`
`Unregister-ScheduledJob`	Remove the scheduled job from the local computer	`Unregister-ScheduledJob –Name TestJob`
`Wait-Job`	Wait for the background job to complete before returning to the command prompt	`Start-Job -scriptblock { Get-Date } | Wait-Job`

Running Background Jobs

Start-Job

We can use the Start-Job cmdlet to submit a PowerShell script to run in the background immediately. As mentioned previously, the job executes as a child PowerShell session. There are a number of parameters to support how the job is submitted. To simply run a script block as a job, we would enter a command like the one here:

```
Start-Job -ScriptBlock { Import-Module umd_northwind_etl; Invoke-UdfCustomerLoad }
```

We run the Start-Job cmdlet, using the ScriptBlock parameter to pass a short script to execute. Notice that because we packaged up the Northwind customer load into a module's function, we can just import the module and call the function. This provides a concise way to call programs. When we execute the statement, we should see the output that follows, which confirms the job was submitted.

The following message is sent back to the console:

```
Id     Name     PSJobTypeName   State     HasMoreData   Location    Command
--     ----     -------------   -----     -----------   --------    ------------------------
15     Job15    BackgroundJob   Running   True          localhost   Import-Module umd_nor...
```

With names like Job15, it may be hard to remember what the job does. We can use the Name parameter to give a meaningful name to the job. Recall that the product load job reads from a Microsoft Access source, which requires that we run the 32-bit version of PowerShell; we can do this by using the RunAs32 parameter. The statement that follows executes the product load as a background job and gives it a custom Name. Note: According to Microsoft PowerShell Help, on Windows 7 and Server 2008 R2, the RunAs32 parameter cannot be used when the Credential parameter is used. Let's look at the following code:

```
Start-Job  -Name ProductLoad  `
           -ScriptBlock { Import-Module umd_northwind_etl; Invoke-UdfProductLoad }  -RunAs32
```

This will generate the following messages to the console. Notice the name is now ProductLoad, as we specified:

```
Id   Name         PSJobTypeName  State    HasMoreData   Location    Command
--   ----         -------------  -----    -----------   --------    ------------------------
17   ProductLoad  BackgroundJob  Running  True          localhost   Import-Module umd_nor...
```

Another interesting variation is to pass an object into the job using the InputObject parameter, as shown here:

```
Start-Job -Name Test1 -ScriptBlock {"The input object is:"; $input }  -InputObject 1,2,3,4
```

We submitted the script block as a job named Test1 and passed the array of values 1,2,3,4 as the InputObject parameter of Start-Job. Anything passed in using the InputObject parameter can be accessed in the script block using the variable $input. So, this job will write the values of the array. We can view the job output as shown here:

```
Receive-Job -Name InputObj -Keep
```

The cmdlet, Receive-Job, displays the messages written out by the job. The Name parameter identifies the name of the job. We could use the ID parameter to identify the job by its ID. The Keep parameter tells PowerShell to retain the job's output. Without this parameter, the job output is automatically erased. We should see the following job output:

```
The input object is:
1
2
3
4
```

There are a number of optional parameters to use with Start-Job. Some are documented in Table 12-2.

Table 12-2. *Start-Job Parameters*

Parameter	Description
FilePath	Used to specify the path to a script file to be executed. This is an alternative to using the ScriptBlock parameter.
Credential	Used to specify a credential under which to execute the job.
InitializationScript	Specifies a script to run before the job starts, so a good place for set-up work.
PSVersion	If there are multiple PowerShell versions on the machine, use this parameter for 2.0. or 3.0.

Receive-Job

To view the output of a job, we use the Receive-Job cmdlet much the same way as we did for Start-Job. Let's review some of the variations on how we can view job output. Note: To see a list of jobs, just enter Get-Job.

To get a job's output by job ID, we can use the ID parameter, as shown here:

```
Receive-Job -ID 19
```

We can get a reference to the job by assigning the output of the Start-Job cmdlet to a variable. Then, we can use the variable to get the job results, as shown:

```
$myjob = Start-Job -Name InputObj -ScriptBlock { $input }  -InputObject 1,2,3,4
Receive-Job -Job $myjob
```

A nice thing about this approach is that we don't have to remember job names or IDs. It can also be a handy way for a script to retain a job reference for later use.

Since Start-Job is a cmdlet, it can be called at any time, even from within a script contained in a job. When this happens, a child job is created under the parent. Let's look at an example:

```
Start-Job -Name ParentChild -ScriptBlock { "Parent Job" ; Start-Job -ScriptBlock { "Child
Job" } }

Receive-Job -Name ParentChild -Keep
```

Start-Job submits a job named ParentChild, which contains a script block that executes another Start-Job command. To make this easy to follow, the outer job outputs the message "Parent Job" and the child job outputs the message "Child Job". The Receive-Job cmdlet displays the output of the job to the console, as shown here:

```
Parent Job

Id      Name      PSJobTypeName   State         HasMoreData     Location        Command
--      ----      -------------   -----         -----------     --------        -----------
1       Job1      BackgroundJob   Running       True            localhost       "Child Job"
```

Notice that the first line is the message Parent Job, output by the parent job. This is followed by information about the child job; in other words, Receive-Job retrieves the output of the parent job and shows the child job's information.

If we don't want the output of the child job, we should be able to use the NoRecurse parameter, as shown here:

```
Receive-Job -Name ParentChild -Keep -NoRecurse
```

However, in testing the Receive-Job statement, I discovered a bug. NoRecurse is supposed to suppress showing output of child jobs. When I tried it, I got no results back at all. According to the blog link that follows, all output goes to the child job, which renders the NoRecurse parameter useless.

https://connect.microsoft.com/PowerShell/feedback/details/771705/receive-job-with-norecurse-returns-null

Get-Job

Get-Job lists active and completed background PowerShell jobs of the current session. To view scheduled jobs, use the Get-ScheduledJob cmdlet. Let's try Get-Job with no parameters, as shown:

```
Get-Job
```

If there are any jobs in the queue, we should see output similar to what is here:

```
Id      Name    PSJobTypeName   State       HasMoreData  Location     Command
--      ----    -------------   -----       -----------  --------     ------------------------
9       Job9    BackgroundJob   Completed   True         localhost    Dir
13      Job13   BackgroundJob   Completed   True         localhost    Import-Module umd_nor...
15      Job15   BackgroundJob   Completed   False        localhost    Import-Module umd_nor...
```

Get-Job displays entries in the job queue. Although some of the columns are fairly intuitive, let's review what the columns mean; see Table 12-3.

Table 12-3. *Column Descriptions for the Output of Get-Job*

Column	Description
`Id`	Unique identifying number of the job that can be used with the other job control cmdlets to specify the intended target job
`Name`	Textual name for the job, which, if not specified by the user when the job is created, is automatically generated by PowerShell as *Jobxx*, where xx is the Job ID.
`PSJobTypeName`	This will have a value of `BackgroundJob` for jobs submitted by `Start-Job` locally. A value of `RemoteJob` means that the job was started using the `AsJob` parameter of a cmdlet such as `Invoke-Command`, i.e., the job is running on another machine. A value of `PSWorkFlowJob` means the job is a workflow started using the `AsJob` parameter of the `workflows` cmdlet.
`State`	The current state of the job. Valid values are `NotStarted`, `Running`, `Completed`, `Failed`, `Stopped`, `Blocked`, `Suspended`, `Disconnected`, `Suspending`, `Stopping`
`HasMoreData`	Indicates if the job output has been retrieved yet. `True` means it has and `False` means it has not.
`Location`	Indicates where the job is running or was run. `Localhost` means on the local machine. Remote jobs will show the host's machine name.
`Command`	The PowerShell commands in the job

Not only does `Get-Job` provide so many job details, it also supports a number of filter parameters so as to limit the list to jobs we are interested in. A listing of these with descriptions and examples is provided in Table 12-4.

Table 12-4. *Get-Job Parameters and Examples*

Parameter	Description	Example
`Command`	Filters based on a command included in the job	`Get-Job -Command "*Get-ChildItem*"`
`ID`	Filters based on the job ID unique to the session	`Get-Job -ID 10`
`IncludeChildJob`	Includes child jobs with parents	`Get-Job -IncludeChildJob`
`Name`	Filters based on the job name	`Get-Job -Name MyJob`
`State`	Filters based on the job state	`Get-Job -State Completed`
`After`	Filters out jobs that ran before the specified date/time	`Get-Job -After '2015-05-01'`
`Before`	Filters out jobs that ran after the specified date/time	`Get-Job -Before '2015-05-01'`
`ChildJobState`	Shows only the child jobs in the specified state	`Get-Job -ChildJobState Completed`
`HasMoreData`	Filters based on the `HasMoreData` column. To see only jobs with more data, use `HasMoreData:$true`. To see only jobs that do not have more data, use `HasMoreData:$false`.	`Get-Job -HasMoreData:$true`
`Newest`	Specified number of more recent jobs to list	`Get-Job -Newest 5`

Wait-Job

The Wait-Job cmdlet tells PowerShell to wait until one or more background jobs have completed before continuing.

To wait for a specific job, we can use the job ID or job name, as shown here:

```
Wait-Job -ID 5,6
Wait-Job -Name Job10 -Timeout 10
```

The first statement will wait until the jobs with IDs 5 and 6 complete. The second line will wait for the job named Job10 to complete, but limits the wait to ten seconds. Note: For both ID and Name, a list can be specified. We can also pipe a list of jobs into the Wait-Job cmdlet as shown here:

```
Get-Job | Wait-Job
```

This statement pipes all jobs into the Wait-Job cmdlet, which means the command will stop until all jobs complete. We can wait until any job in a list of jobs completes, as shown here:

```
Wait-Job -id 1,6,10,12 -Any
```

This statement will wait until any one of the listed jobs completes. Wait-Job is useful in scripts when a certain job needs to complete before the script continues.

Remove-Job

By default, job output is deleted when we use the Receive-Job cmdlet, but this behavior can be overridden with the Keep parameter. We can then use the Remove-Job cmdlet to clean up the job queue of unwanted jobs. Bear in mind, background jobs are automatically deleted when the session ends. We can use Get-Job to pipe the list of jobs to remove into the Remove-Job cmdlet. Here are some examples of this:

```
Get-Job -Name Job* | Remove-Job           # Removes all jobs that start with the letters
                                          Job.
Get-Job -HasMoreData:$false | Remove-Job  # Removes all jobs where output has been
                                          received.
Get-Job -Before '2015-05-01' | Remove-Job # Removes all jobs completed before the date
```

By using Get-Job, we can get more flexibility in what jobs we remove. However, Remove-Job has some parameters we can use. Let's see some examples:

```
Remove-Job -ID 12                     # Removes job with ID 12
Remove-Job -Name MyJob -Force         # Removes job named MyJob even if it is running.
Remove-Job -State Completed -Confirm  # Prompt for confirmation to remove all completed
jobs.
Remove-Job -Name * -WhatIf            # Writes messages about what jobs would be removed.
```

Stop-Job

To stop a running job, we use the `Stop-Job` cmdlet. Similar to other job cmdlets we've seen, `Stop-Job` supports parameters for the job ID and name. Some examples of using `Stop-Job` are as follows:

```
Stop-Job -ID 15                 #  Stops job ID 15
Stop-Job -Name MyJob            #  Stops job named MyJob
Get-Job | Stop-Job              #  Stops all jobs
Stop-Job -Name j* -WhatIf       #  Displays messages about what jobs would be deleted
Stop-Job -Name j* -Confirm      #  Prompts the user to confirm before stopping the jobs
```

Another way to end all background jobs is to exit the session.

Scheduled Jobs

PowerShell's scheduled jobs have all the characteristics of unscheduled jobs. In fact, the job commands we discussed earlier work with scheduled jobs as well. However, scheduled jobs have some things that make them different from temporary jobs.

- The job definition, including the script block or script file path, needs to be stored to disk so it can persist indefinitely and survive system reboots.
- The creation and maintenance of job schedules need needs to be supported.
- Job execution must be independent of any active PowerShell sessions, unlike with temporary jobs, which are lost when the session ends.
- A mechanism is needed to reload job definitions and schedules after a machine reboot.

As mentioned previously, Microsoft's answer to these requirements was to use the Windows Task Scheduler.

Creating a Scheduled Job

With so many cmdlets and options for scheduled jobs, it might be useful to start with a quick overview of creating and executing a simple scheduled job step by step. Let's walk through the process.

1. Create the job schedule, which is called a *trigger*, as shown next. The trigger will execute every day at 10 AM:

   ```
   $jobtrigger = New-JobTrigger -Daily -At '10:00 AM'
   ```

2. Create the job options with a statement like the one that follows, with which we are creating a `Verbose` job option:

   ```
   $joboptions = New-ScheduledJobOption –Verbose
   ```

3. Create the scheduled job using the `Register-ScheduledJob` cmdlet. The statement in this example will create a new scheduled job using the `Verbose` job option; it will run daily at 10 AM. The trigger and `ScheduledJobOption` are defined using the variables we created in steps 1 and 2.

```
Register-ScheduledJob   -Name MyDailyJob -ScriptBlock { Get-Date } `
                        -Trigger $jobtrigger -ScheduledJobOption $joboptions
```

When you run the `Register-ScheduledJob` statement to register the job, a new task is added to the Windows Task Scheduler. To view it, start the Task Scheduler by clicking on the Start menu, select All Programs, and then Administrative Tools. The Task Scheduler will be visible in the program list, as shown in Figure 12-1.

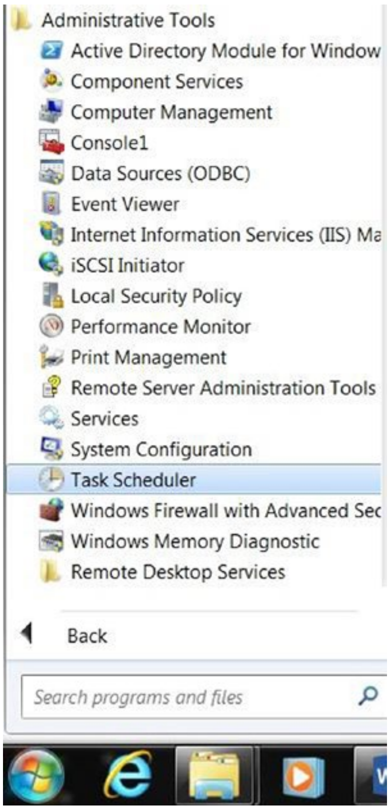
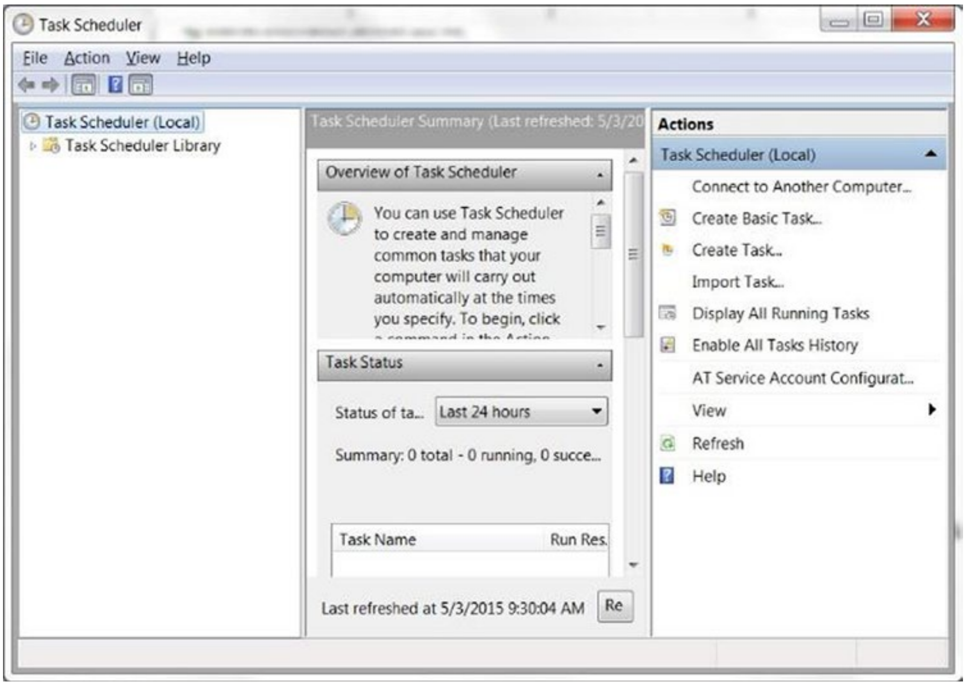


Figure 12-1. *Starting the Task Scheduler*

Click on the Task Scheduler to start it. You will then see the Task Scheduler's main screen, as shown in Figure 12-2.



You can use Task Scheduler to create and manage common tasks that your computer will carry out automatically at the times you specify. To begin, click

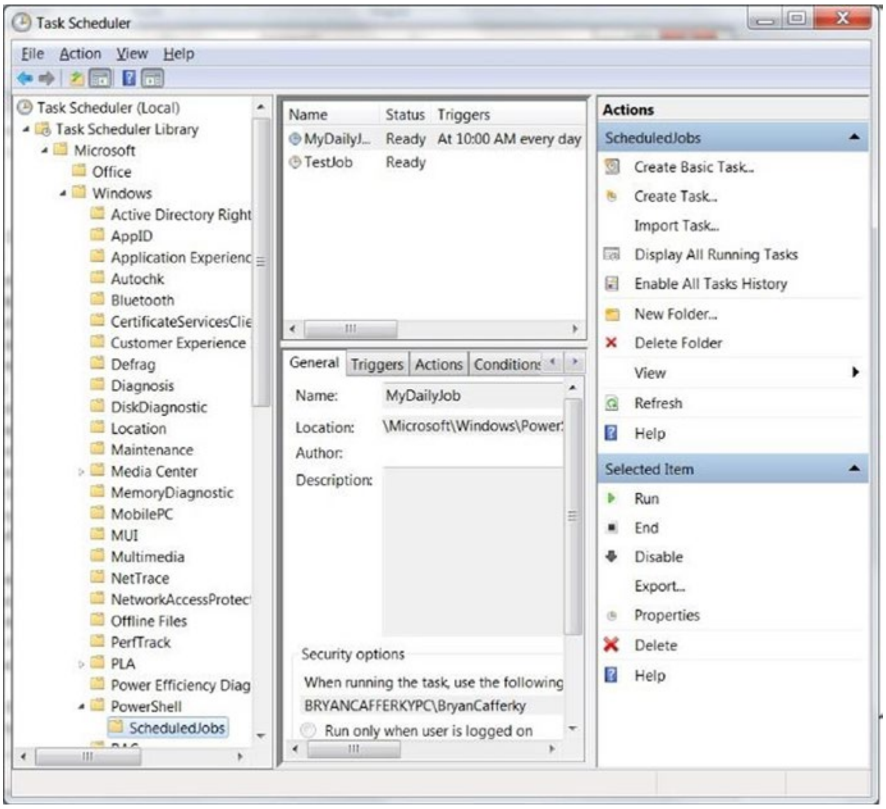
Status of ta... Last 24 hours

Summary: 0 total - 0 running, 0 succe...

Last refreshed at 5/3/2015 9:30:04 AM

Figure 12-2. *The Task Scheduler main screen*

The Task Scheduler is used by many applications to automate tasks, such as periodically checking for and installing program updates. PowerShell uses it to store and execute scheduled jobs. We can see the PowerShell scheduled jobs by expanding the Task Scheduler Library folder, followed by expanding Windows, and PowerShell. Then, click on ScheduledJobs, and you should see the PowerShell job we just created, as shown in Figure 12-3.



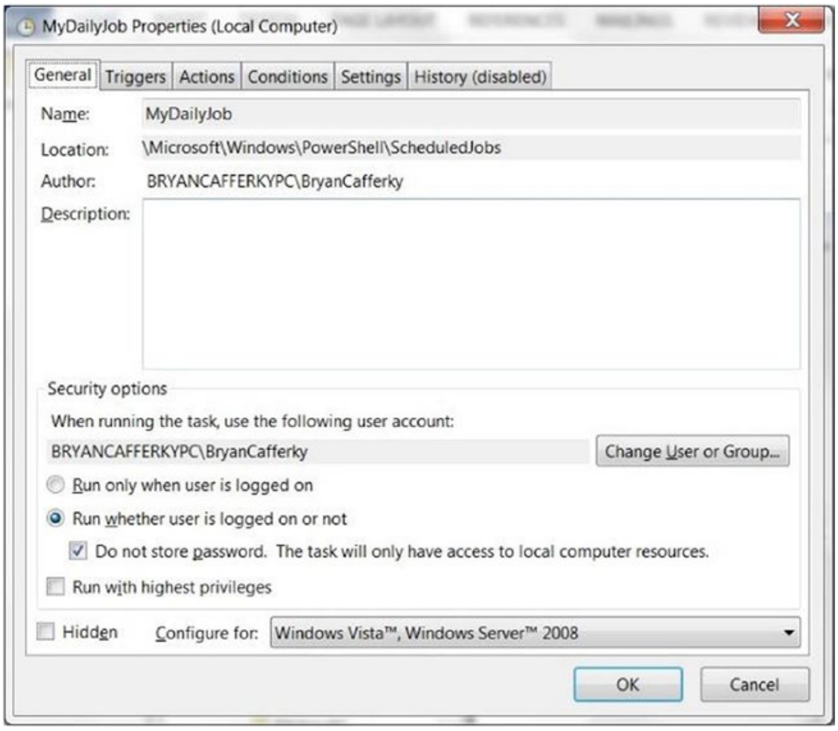
Name	Status	Triggers
MyDailyJ...	Ready	At 10:00 AM every day
TestJob	Ready	

Name: MyDailyJob
Location: \Microsoft\Windows\Power
Author:
Description:

Security options
When running the task, use the following
BRYANCAFFERKYPC\BryanCafferky
Run only when user is logged on

Figure 12-3. *PowerShell scheduled job in the Task Scheduler*

The fact that PowerShell scheduled jobs are persisted in the Task Scheduler confirms that the scheduled jobs cmdlets are really just interfaces to the Task Scheduler. If you created a job named `MyDailyJob`, you could double-click on the job to see the details, as shown in Figure 12-4.


MyDailyJob Properties (Local Computer)

General | Triggers | Actions | Conditions | Settings | History (disabled)

Name: MyDailyJob
Location: \Microsoft\Windows\PowerShell\ScheduledJobs
Author: BRYANCAFFERKYPC\BryanCafferky
Description:

Security options
When running the task, use the following user account:
BRYANCAFFERKYPC\BryanCafferky
Change User or Group...
Run only when user is logged on
[✓ Run whether user is logged on or not]
[✓ Do not store password. The task will only have access to local computer resources.]
Run with highest privileges

Hidden
Configure for: Windows Vista™, Windows Server™ 2008

OK
Cancel

Figure 12-4. *PowerShell scheduled job fetails in Task Scheduler*

Figure 12-4 shows that we can view and edit the PowerShell scheduled job details in the Task Scheduler. At first, it may seem odd that Microsoft simply used the Task Scheduler for PowerShell scheduled jobs instead of creating something specific for PowerShell. However, upon consideration, this has some advantages. One, we get a nice graphical interface with which to view, maintain, and execute jobs. Second, PowerShell scheduled jobs integrate into the Windows architecture as other applications do. Third, features like the recovery of scheduled jobs after a system reboot are built into the Task Scheduler already.

Maintaining Scheduled Jobs

Scheduled jobs, as we have seen, consist of three objects: triggers, job options, and the job definition, which includes the PowerShell code to be executed. There are several cmdlets to help us view, create, and delete job triggers as well as attach them to scheduled jobs. These are `Get-JobTrigger`, `New-JobTrigger`, `Set-JobTrigger`, `Remove-JobTrigger`, and `Add-JobTrigger`. Let's review these in detail.

New-JobTrigger

Schedules, called *triggers*, define the parameters of when a job should be executed. In some scheduling systems, SQL Agent, for example, schedules can be defined once, saved under a schedule name, and used by multiple jobs. This is not the case with PowerShell scheduled jobs. A trigger is specific to a job, which is why the `New-JobTrigger` cmdlet does not support a `Name` parameter. The same is true of job options. To create a new trigger, we use the `New-JobTrigger` cmdlet, which would be followed by one or more cmdlets that add the trigger to a scheduled job. Let's look at some examples of using the `New-JobTrigger` cmdlet.

Note: The examples show the result being assigned to the variable `$jobtrigger`, because storing the trigger and options in variables provides better readability and reusability, and shorter job definition statements. However, they can be included inline enclosed in parentheses as well. The statement that follows creates a trigger to run a job once:

```
$jobtrigger = New-JobTrigger -Once -At '2105-10-22 10:00 AM'  # Run once at date/time.
```

The previous statement will create a trigger that fires only once, at 10 AM. The following statement uses the `AtStartup` parameter to trigger an execution whenever Windows starts:

```
$jobtrigger = New-JobTrigger -AtStartup    # Run when Windows starts, i.e., after a reboot.
```

The next trigger will run weekly on Monday and Tuesday at noon:

```
$jobtrigger = New-JobTrigger -Weekly -DaysOfWeek Monday, Tuesday -At "12:00 PM"
```

The trigger that follows will execute at 1:00 AM for two days.

```
$jobtrigger = New-JobTrigger -Daily -At "1:00 AM" -DaysInterval 2
$jobtrigger = New-JobTrigger -Once -At "05/01/2015 1:00:00" `
                             -RepetitionInterval (New-TimeSpan -Hours 2) `
                             -RepetitionDuration (New-Timespan -Days 5)
```

The trigger definition syntax is odd, because although it specified the `Once` parameter, it is creating a custom schedule that will start on May 1, 2015, and repeat every two hours for a period of five days. Remember, to create a custom schedule, use the `Once` parameter. To have the schedule continue with no end, use the `RepeatIndefinitely` parameter, which can only be used when `Once` is specified.

Add-JobTrigger

We can use the `Add-JobTrigger` cmdlet to attach a trigger to a scheduled job. Some examples of using this cmdlet are provided here:

```
Add-JobTrigger -Trigger $jobtrigger -Name MyDailyJob
```

This statement assumes we created a job trigger using `New-JobTrigger` and stored it in `$jobtrigger`. The statement attaches the trigger to the job MyJob.

The statement that follows creates the trigger using the `New-JobTrigger` statement enclosed in parentheses, which means it will be executed before attaching to `MyDailyJob`:

```
Add-JobTrigger -Trigger (New-JobTrigger -AtStartup) -Name MyDailyJob
```

The next statement uses `Get-ScheduledJob` to get the job definition for `MyJob`, which is piped into the `Add-JobTrigger` cmdlet, which adds a trigger for daily execution at 1:00 AM.

```
Get-ScheduledJob MyDailyJob | Add-JobTrigger -Trigger (New-JobTrigger -Daily -At "1:00 AM")
```

The statement that follows copies the trigger from the job `SomeOtherJob` and copies it to `MyJob1` and `MyJob2`.:

```
Add-JobTrigger -Name MyJob1, MyJob2 -Trigger (Get-JobTrigger -Name SomeOtherJob)
```

The cmdlet `Add-JobTrigger` uses the `ID` parameter to specify the job by ID, or the `Name` parameter, as shown, to specify the job by name. The job definition can also be piped in, as shown, or passed in using the `InputObject` parameter.

Get-JobTrigger

The `Get-JobTrigger` cmdlet retrieves job trigger definitions from scheduled jobs. The following statement will get the triggers attached to the job `MyDailyJob`:

```
Get-JobTrigger -Name MyDailyJob
```

Multiple triggers can be attached to a single job. We can see the triggers using `Get-JobTrigger` as shown here:

```
Get-JobTrigger -Name MyWeeklyJob
```

Assuming we have multiple triggers attached to the job, the output would look similar to the output shown here:

```
Id         Frequency       Time                     DaysOfWeek             Enabled
--         ---------       ---------------------   -----------------       -------
1          Weekly          5/3/2015 12:00:00 PM    {Monday, Tuesday}       True
2          Once            10/22/2105 10:00:00 AM                          True
3          Weekly          5/3/2015 12:00:00 PM    {Monday, Tuesday}       True
```

There are some interesting things to note. First, triggers have a unique ID within a job. Second, we can attach duplicate triggers, as the job in our example has–i.e., two triggers so as to run the job weekly on Monday and Tuesday. Be careful of this. Third, since there is an enabled flag, there must be a way to correspondingly disable the trigger, which we will discuss shortly. Note: The Microsoft online documentation shows the cmdlet `Get-ScheduledJob` as listing all the triggers associated with a job. However, in my testing, only one trigger was shown.

When there are multiple triggers for a job, the trigger ID gives us a way to specify the trigger:

```
Get-JobTrigger -ID 12 -TriggerId 2
```

In this statement, we specify both the job and trigger by their IDs. We should see the trigger details displayed to the console, similar to the output here:

```
Id         Frequency       Time                     DaysOfWeek             Enabled
--         ---------       --------------------- ----------                -------
2          Once            10/22/2105 10:00:00 AM                          True
```

We can also pipe the results of `Get-ScheduledJob` into `Get-JobTrigger`, as shown:

```
Get-ScheduledJob -Name *j* | Get-JobTrigger
```

This statement will display all the triggers associated with any jobs that have the letter j in the job name. As we will see, `Get-JobTrigger` is useful, because we can pipe its results into other cmdlets like `Set-JobTrigger` to apply actions to multiple triggers.

Set-JobTrigger

The `Set-JobTrigger` cmdlet makes changes to a trigger on a scheduled job. Some examples of using `Set-JobTrigger` are seen next. Note: To create or change scheduled jobs, you must start PowerShell as administrator. The statement that follows will set `MyJob` to run when Windows starts:

```
Get-JobTrigger -Name MyJob | Set-JobTrigger -AtStartup -Passthru
```

The statement that follows pipes Trigger ID 2 from `MyWeeklyJob` into the `Set-JobTrigger` cmdlet, which sets the trigger to run the job on Tuesdays at 8 AM.

```
Get-JobTrigger -Name MyWeeklyJob -TriggerId 2 |
Set-JobTrigger -DaysOfWeek Tuesday at "8:00 AM"
```

The first statement that follows stores the trigger from `MyJob` in variable `$trigger`, which is used with the `InputObject` parameter on the next line to set the `AtLogOn` value to the user `Domain01\user1`, i.e., the job will run whenever the user logs on to the machine. See the following:

```
$trigger = Get-JobTrigger -Name MyJob
Set-JobTrigger -InputObject $trigger -AtLogOn -User Domain01\user1
```

`Set-JobTrigger` and `New-JobTrigger` share the same parameters. These are documented in Table 12-5.

Table 12-5. *Set-JobTrigger and New-JobTrigger Parameters*

Parameter	Description
`At`	Starts the job at a specified time
`AtLogon`	Starts the job when the specified user logs on
`AtStartUp`	Runs the job when Windows starts
`Daily`	Run the job daily
`DaysInterval`	The number of days between executions for a daily schedule
`DaysOfWeek`	For a weekly schedule, the days of the week on which the job should run
`InputObject`	A `ScheduledJobTrigger` object that will be modified
`Once`	Runs the job one time only or defines a custom schedule
`Passthru`	Displays the resulting job trigger details to the console
`RandomDelay`	Runs the job using a random amount of time delay with an initial start date/time and constrained by a maximum delay value. This parameter takes a `timespan` object or string value
`RepeatIndefintely`	Specifies that the job continue running per schedule with no end
`RepetitionDuration`	Sets a limit on how long the job schedule will execute
`RepetitionInterval`	Time spacing between job executions. For example, if the value is four hours, the job will run every four hours
`User`	Specifies that the job be run when the user specified by this parameter logs on
`Weekly`	Sets a weekly schedule
`WeeklyInterval`	Sets the number of weeks between executions for a weekly schedule

Getting and Setting Job Options

There are a number of options that can be set so as to control how a scheduled job executes. The cmdlet `Get-ScheduledJobOption` will display the option name and value for a job. Let's use it to list the settings for `MyDailyJob`, as shown here:

```
Get-ScheduledJobOption -Name MyDailyJob
```

When we enter this statement, we should see the following output:

```
StartIfOnBatteries     : False
StopIfGoingOnBatteries : True
WakeToRun              : False
StartIfNotIdle         : True
StopIfGoingOffIdle     : False
RestartOnIdleResume    : False
IdleDuration           : 00:10:00
IdleTimeout            : 01:00:00
ShowInTaskScheduler    : True
RunElevated            : False
RunWithoutNetwork      : True
DoNotAllowDemandStart  : False
MultipleInstancePolicy : IgnoreNew
JobDefinition          : Microsoft.PowerShell.ScheduledJob.ScheduledJobDefinition
```

As the list shows, there are quite a few options. However, defaults are provided for most of them that are usually acceptable.

As shown, `Get-ScheduledJobOption` is the way we retrieve information about job options. There are four ways to call `Get-ScheduledJobOption.` First, we can pass the job name as shown. Second, we can pass the job ID with the `ID` parameter. Third, we can pipe scheduled job objects into the cmdlet. Fourth, we can pass the job definition via the `InputObject` parameter. Examples of these are shown next. The following statement retrieves job option details using the job ID:

```
Get-ScheduledJobOption -ID 10
```

The next statement pipes the output of `Get-ScheduledJob` into `Get-ScheduledJobOption`, which will retrieve the job option details.

```
Get-ScheduledJob -Name MyDailyJob | Get-ScheduledJobOption
```

Next, we see the `Get-ScheduledJobOption` cmdlet's `InputObject` parameter being used to pass the job definition that was obtained as the result of the `Get-ScheduledJob` cmdlet that is enclosed in parentheses.

```
Get-ScheduledJobOption -InputObject (Get-ScheduledJob -Name MyDailyJob)
```

We can change job options using the `Set-ScheduledJobOption` or `New-ScheduledJobOption` cmdlets. Table 12-6 lists the parameters to these cmdlets and what they do.

Table 12-6. *Scheduled Job Options and What They Do*

Parameter	Description
ContinueIfGoingOnBattery	When True, the job will continue to run even if the computer loses AC power. The default is False.
DoNotAllowDemandStart	When True, users cannot start the job manually. Only the trigger will run the job. The default is False.
HideInTaskScheduler	When True, the job will not be visible in the Task Scheduler on the computer where the job runs. The default is False.
MultipleInstancePolicy	Defines what the Task Manager should do if the job if the job is running when another request to run the job is raised. The default, IgnoreNew, does not create another instance of the job. Parallel causes a new job instance to start immediately. Queue will have the second-run request processed only after the current job instance has finished. StopExisting causes the current job to be cancelled and a new instance to be started.
RequiredNetwork	True means that the job will not start if network services are not available. The default is False. Setting this parameter to True will automatically set RunWithoutNetwork to False.
RestartOnIdleResume	True means that a suspended job will resume when the machine becomes idle and meets any IdleDuration value if set.
RunElevated	True means to run the job elevated to the Administrators permissions. The scheduled job should be registered with the Credential parameter.
StartIfIdle/IdleDuration	This IdleDuration parameter is used with SetIfIdle to define the amount of time the computer must be idle before the job can run. If the idle timeout condition is not met, the job execution is skipped until the next scheduled time. The default is ten minutes.
StartIfIdle/IdleTimeout	These parameters work in tandem with the StartIfIdle/IdleDuration parameters to define how long to wait for the required idle state to occur. If the timeout expires before the idle duration is met, the job will not run until the next scheduled time.
StartIfOnBattery	When True, the job will start even if there is no AC power to the machine.
StopIfGoingOffIdle	True means that if the machine becomes active while the job is running, the job will be suspended. False means to keep running the job. True is the default.
WakeToRun	True means the machine will wake from hibernate or sleep modes to run the job per the schedule. False means do not wake the machine. The default is False.

For some of the options, such as StartIfOnBattery, it seems unlikely we would want to override the default, i.e., risk a system shutdown in the middle of a job execution. However, it may be useful to change some of the other options to meet requirements. Let's review using some of these options and see how they work:

```
New-ScheduledJobOption -RequireNetwork –MultipleInstancePolicy Queue
```

This statement creates a scheduled job option that uses RequiredNetwork to tell the Task Scheduler to test and confirm that the network services are running before starting the job. It also sets MultipleInstancePolicy to Queue, which requires that in the instance where multiple instances of the job are requested, they should be added to the queue to be executed sequentially.

The statement that follows stores a job option in a variable, which it then uses to apply to a scheduled job:

```
$RunAsAdmin = New-ScheduledJobOption -RunElevated
Register-ScheduledJob -Name MyJob -FilePath C:\PowerShell\Scripts\installpatches.ps1 `
        -Trigger (New-JobTrigger -Daily -At "8:00 AM") -ScheduledJobOption $RunAsAdmin
```

The first statement stores a job option into variable $RunAsAdmin to run the job as administrator. Then, the statement that follows registers a new scheduled job named MyJob that runs a script stored on disk daily at 8 AM using the job option in variable $RunAsAdmin; i.e., it will run as administrator. To run these statements, you will need to start PowerShell as administrator. This example references a script file, but the problem with that approach is that the path is hard coded, which means the job will fail if the script is moved. A less fragile approach is to encapsulate the script as a function and add the function to a module so it can automatically be loaded for us using Import-Module. Let's look at an example of this:

```
Register-ScheduledJob –Name MyOtherJobAdmin –Trigger (New-JobTrigger -Daily -At "10 AM") `
                      -InitializationScript {Import-Module umd_northwind_etl}        `
                      -ScriptBlock { Invoke-UdfStateSalesTaxLoad }  `
                      -ScheduledJobOption (New-ScheduledJobOption –RunElevated) `
                      -Credential BryanCafferky
```

This statement will register a new scheduled job named MyOtherJobAdmin to run daily at 10 AM with administrative permissions. Because the parameter Credential is specified, we are prompted for the password when we run the statement. Notice we use the InitializationScript parameter to get the umd_northwind_etl loaded before the script block executes. We could also just have included the import-module statement in the script block, i.e., Import-Module umd_northwind_etl; Invoke-UdfStateSalesTaxLoad. However, I think using the InitializationScript parameter makes the job definition easier to read. We can execute the job immediately despite the trigger by using the Start-Job cmdlet, as shown here:

```
Start-Job -DefinitionName MyOtherJobAdmin -Verbose
```

The Start-Job cmdlet starts the job named MyOtherJobAdmin immediately. An oddity of this cmdlet is that instead of using the parameter Name as the other job cmdlets do, it calls the parameter DefinitionName. We could also use the Set-ScheduledJob cmdlet with the RunNow parameter. When we run the statement, we should see output similar to the following:

```
Id  Name            PSJobTypeName  State    HasMoreData  Location   Command
--  ----            -------------  -----    -----------  --------   -----------------------
24  MyOtherJobAdmin PSScheduledJob Running  True         localhost  Import-Module umd_nor...
```

To view the job status, use the Get-Job cmdlet, and to see the output, use Receive-Job output with the Keep parameter if you do not want the output deleted.

Next, we use the Receive-Job cmdlet to view the job output. You should see messages showing connection details followed by the insert statements. See the code here:

```
Receive-Job -Name MyOtherJobAdmin –Keep
```

To change an option on a registered job, we can use the Set-ScheduledJobOption cmdlet, as shown here:

```
Get-ScheduledJobOption -Name MyOtherJob4 | Set-ScheduledJobOption -WakeToRun -Passthru
```

The Set-ScheduledJobOption does not take a job name as a parameter. Instead, as shown, we need to get a job option object, which we do by piping the output of Get-ScheduledJobOption into the Set-ScheduledJobOption cmdlet. An alternative to this is to store the scheduled job option in a variable and pass this to Set-ScheduledJobOption as the InputObject parameter. The PassThru parameter causes the option settings to be returned, as shown here:

```
StartIfOnBatteries     : False
StopIfGoingOnBatteries : True
WakeToRun              : True
StartIfNotIdle         : True
StopIfGoingOffIdle     : False
RestartOnIdleResume    : False
IdleDuration           : 00:10:00
IdleTimeout            : 01:00:00
ShowInTaskScheduler    : True
RunElevated            : False
RunWithoutNetwork      : True
DoNotAllowDemandStart  : False
MultipleInstancePolicy : IgnoreNew
JobDefinition          : Microsoft.PowerShell.ScheduledJob.ScheduledJobDefinition
```

Notice that JobDefinition is listed. However, this is not something we can change via job option cmdlets.

We have seen that a scheduled job consists of three parts, which are the job definition, one or more triggers, and job options. The job definition defines what PowerShell code will be executed. Triggers define the schedule of when the job should be executed. Job options define special job execution features such as requiring network services be available or that it be run with administrator privileges. There are defaults, so it is not a requirement to set these options.

Scripting with the Job Cmdlets

The PowerShell job cmdlets are useful, but it can get pretty tedious having to type so many commands with all the various parameters. Well, this is PowerShell, right? So, why don't we make this easier for ourselves by using PowerShell scripts? The function in Listing 12-1 uses Out-GridView to display the job queue. The list includes both scheduled and unscheduled jobs. The user can select one or more jobs from the list and then take an action on the selected jobs, such as removing them.

Listing 12-1. A simple PowerShell job console

```
function Show-UdfJobConsole
{
 [CmdletBinding()]
        param (
              [string]$jobnamefilter
          )
```

```
function ufn_ShowMessageBox
{
    [CmdletBinding()]
    param (
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [string] $message,
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [string] $title,
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [ValidateRange(0,5)]
    [int] $type
    )

 RETURN [System.Windows.Forms.MessageBox]::Show($message , $title, $type)

}

  $actions = "Show Output", "Stop", "Remove", "Resume", "Clear History"

  while ($true)
  {
  $joblist = Get-Job -Name $jobnamefilter |
             Out-GridView -Title "PowerShell Job Console" -OutputMode Multiple

  if ($joblist.count -gt 0)
  {
     $selection = $actions | Out-GridView -Title "Select Action" -OutputMode Single
     foreach ($job in $joblist)
     {
        switch ($selection)
        {
          "Show Output"
          {
            $joboutput = Receive-Job -ID $job.ID -Keep -Verbose 4>&1
            If ($joboutput -eq $null)
            {
               ufn_ShowMessageBox -message "No output to display" -title 'Error' -type 0
            }
            $joboutput |
            Out-GridView -Title 'Job Output - Close window to return to the job list' -Wait
          }
          "Cancel"
          {
            $job | Stop-Job
          }
          "Remove"
          {
            $job | Remove-Job
          }
          "Resume"
          {
            $job | Resume-Job
          }
        }
     }
  }
  else
  {
      Break;
  }
 }
}
```

The function in Listing 12-1 supports one parameter, which is a job name filter used to list jobs that match a given name pattern. A job filter of * will cause all jobs to display. The function contains a nested function, `ufn_ShowMessageBox`, which is just a wrapper to call the Windows message box method, i.e., `[System.Windows.Forms.MessageBox]::Show()`. It's a simple call, but is not easy to remember, so we just include it as a subfunction to be used as needed to display messages to the user. Then, the function stores a list of actions that can be taken on the jobs listed in the gridview. It stores this list as an array in the variable `$actions`. Then, an infinite loop starts, i.e., `While ($true)`. The loop is exited via a `break` statement. Using the `Get-Job` cmdlet, the job list is piped into `Out-GridView` in order to display the list of jobs. Note the use of the `OutputMode` parameter with the value `Multiple`. This allows the user to select one or more rows from the grid. When the user clicks the OK button, the selected rows will be stored in the variable `$joblist`. Before we try to take any actions on the selected jobs, we check to make sure something was selected by testing if `$joblist.count` is greater than 0.

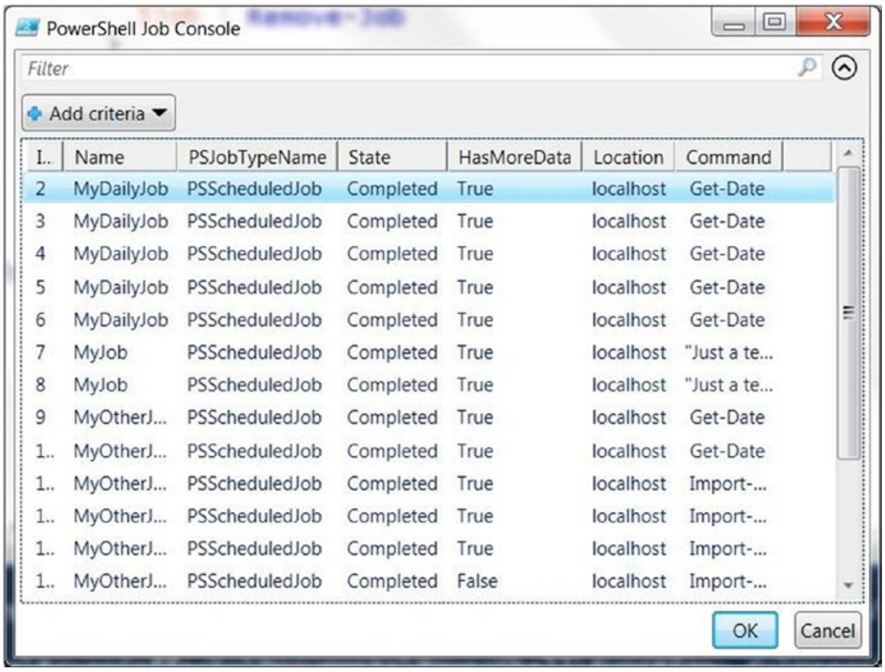
Now the function pipes the `$action` array into a gridview. The idea is to use the gridview as an action menu. Notice that the `Out-GridView` statement has the parameter `OutputMode` set to `Single`, which means the user can only select one item. A `foreach` loop iterates over the selected jobs to execute the selected action.

The rest of the function is pretty simple. A `Switch` statement is used to check the action selected and execute the related code. `$job` is the current job in the `foreach` iteration. One line deserves special attention: `$joboutput = Receive-Job -ID $job.ID -Keep -Verbose 4>&1`. Verbose messages cannot be captured by default, so they will not get displayed in the output view. By using redirection, we are capturing the verbose message stream, which is identified by Channel 4. Note: The function will let us try to perform actions that are not valid for the job status or type so as to keep the example simple. For example, we can attempt to cancel a completed job, but that would fail, and only workflows can be suspended.

Note: For more information about capturing different types of message streams, see the following link: `https://technet.microsoft.com/en-us/library/hh847746.aspx`. Now that we understand how the function works, let's try it with the statement that follows:

```
Show-UdfJobConsole *
```

The function `Show-UdfJobConsole` is called with an asterisk as the `jobnamefilter` parameter, which means all jobs will be displayed. At the time of writing this, on my machine, I get the gridview shown in Figure 12-5. You should see something similar if you have any job output. If there are no jobs, the script just ends.



I..	Name	PSJobTypeName	State	HasMoreData	Location	Command
2	MyDailyJob	PSScheduledJob	Completed	True	localhost	Get-Date
3	MyDailyJob	PSScheduledJob	Completed	True	localhost	Get-Date
4	MyDailyJob	PSScheduledJob	Completed	True	localhost	Get-Date
5	MyDailyJob	PSScheduledJob	Completed	True	localhost	Get-Date
6	MyDailyJob	PSScheduledJob	Completed	True	localhost	Get-Date
7	MyJob	PSScheduledJob	Completed	True	localhost	"Just a te...
8	MyJob	PSScheduledJob	Completed	True	localhost	"Just a te...
9	MyOtherJ...	PSScheduledJob	Completed	True	localhost	Get-Date
1..	MyOtherJ...	PSScheduledJob	Completed	True	localhost	Get-Date
1..	MyOtherJ...	PSScheduledJob	Completed	True	localhost	Import-...
1..	MyOtherJ...	PSScheduledJob	Completed	True	localhost	Import-...
1..	MyOtherJ...	PSScheduledJob	Completed	True	localhost	Import-...
1..	MyOtherJ...	PSScheduledJob	Completed	False	localhost	Import-...

Figure 12-5. *Our custom PowerShell job console*

The point of this function is to show how easy it is to customize PowerShell to fit your needs. If something seems a bit awkward to use or difficult to remember, we can create custom functions to simplify it and increase our productivity. For the function in this example, we would probably want to create an alias to make it even easier to use, as shown here:

```
New-Alias spsj Show-UdfJobConsole
```

We are creating the alias `spsj`, which stands for *Show the PowerShell Job Console*. If we add this to our profile, it will automatically be created whenever we start a session.

Using PowerShell with SQL Server Agent

Although we can use the scheduled jobs features of PowerShell, SQL Server Agent offers a better solution. Since SQL Server Agent can run virtually anything—including SSIS packages (even from the SSIS Catalog), T-SQL code, PowerShell scripts, Analysis Services packages, and Command Exec scripts—it provides flexible integration of application components. Unlike the Task Scheduler, SQL Server Agent jobs can have multiple steps and be configured to send out job notifications when events such as a job failure occur. It also has many more security features than the Task Scheduler does. All these features are provided with a rich visual interface via SQL Server Management Studio.

As mentioned earlier, SQL Server Agent has built-in support to run PowerShell scripts. Let's walk through creating a simple SQL Agent job to execute a PowerShell script. First, start SQL Server Management Studio. Then, connect to the server on which you want to create the job, such as `localhost`. Expand the SQL Server Agent folder, right mouse click on the Jobs folder, and select New Job... as shown in Figure 12-6.

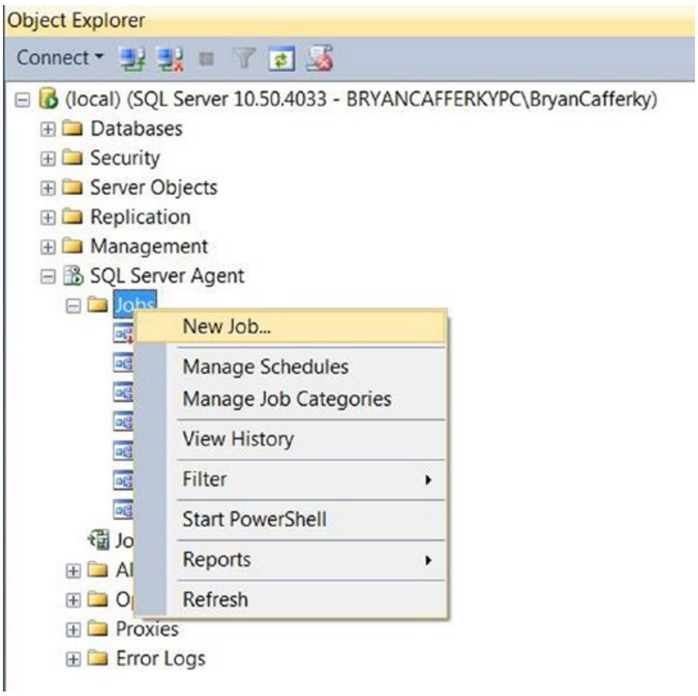
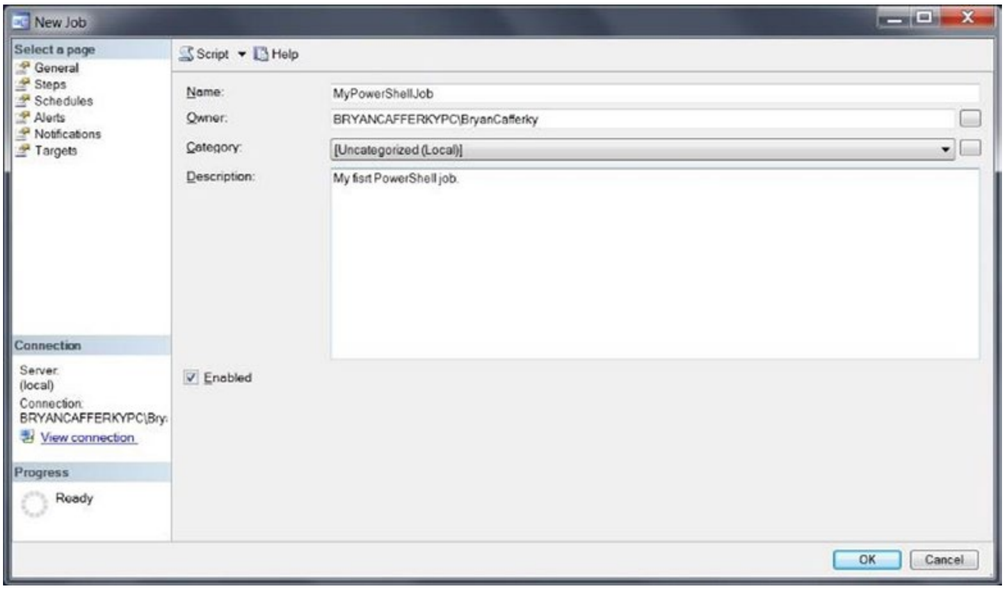


Figure 12-6. *Creating a job in SQL Agent*

After you select New Job..., you will see the New Job dialog box, with which you can create a new SQL Agent Job. Enter a name and a description for the job, as shown in Figure 12-7.

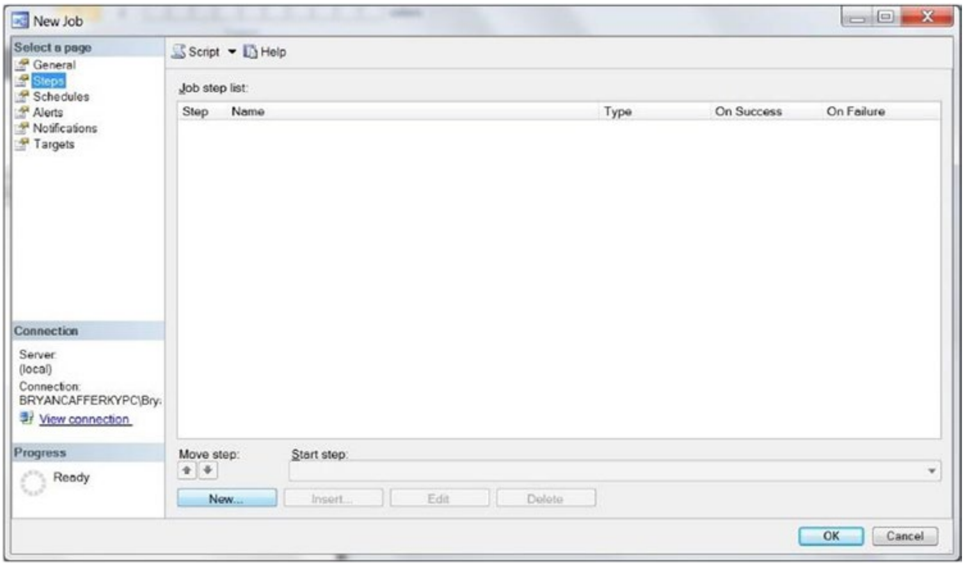


Name: MyPowerShellJob
Owner: BRYANCAFFERKYPC\BryanCafferky
Category: [Uncategorized (Local)]
Description: My fisrt PowerShell job.
☑ Enabled

Server: (local)
Connection: BRYANCAFFERKYPC\Bry

Figure 12-7. *Creating a SQL Agent job—general properties*

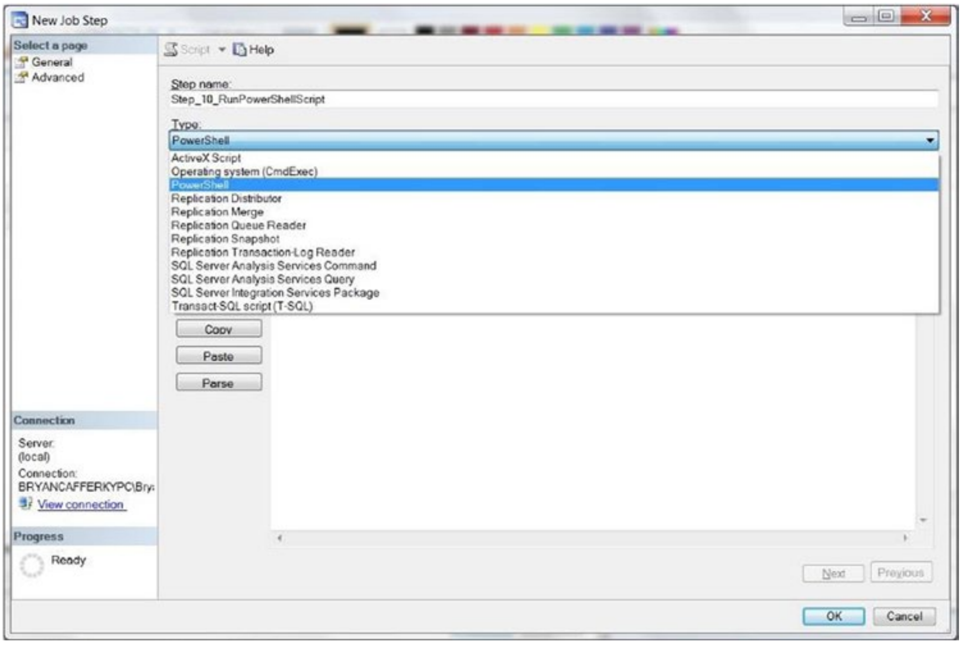
Then click on Steps in the Select a Page navigation panel on the left. This will bring up the Job Step listing, as shown in Figure 12-8.



Server:
(local)
Connection:
BRYANCAFFERKYPC\Bry:

Figure 12-8. *Creating a SQL Agent job step*

On the Job Step screen, click on the New button to create a new job step, which will bring up the Job Step edit screen shown in Figure 12-9.



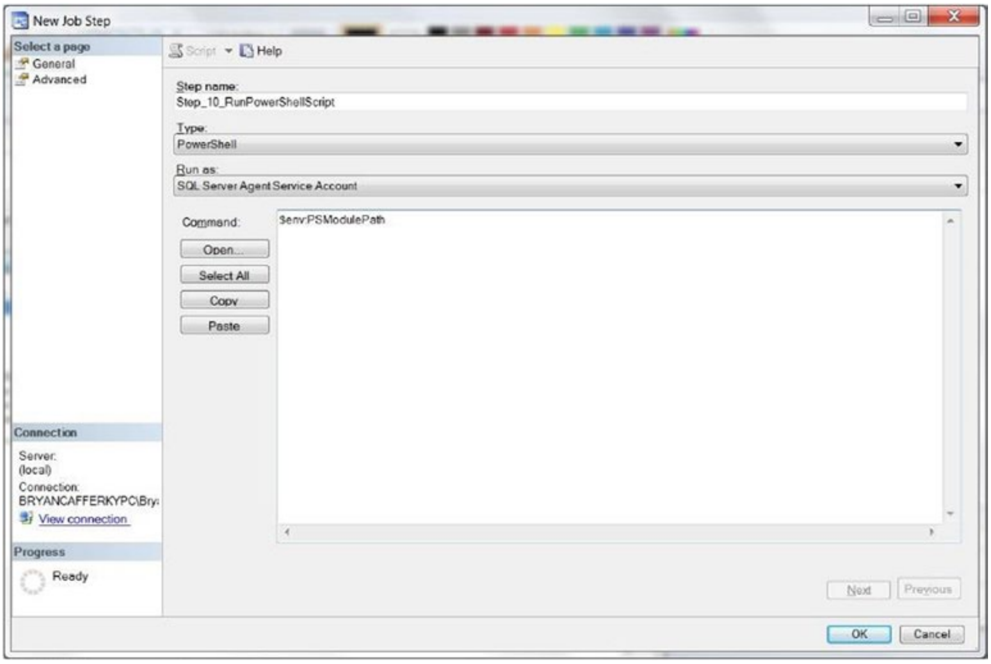
Step name:
Step_10_RunPowerShellScript

Type:
PowerShell

Server:
(local)
Connection:
BRYANCAFFERKYPC\Bry:

Figure 12-9. *Select PowerShell from the Type drop-down menu*

On the Job Step editor screen, enter a step name and select PowerShell from the Type drop-down list. Then, enter the PowerShell code in the text window, as shown in Figure 12-10.



Step name:
Step_10_RunPowerShellScript

Type:
PowerShell

Run as:
SQL Server Agent Service Account

Command:
$env:PSModulePath

Server:
(local)

Connection:
BRYANCAFFERKYPC\Bry:

Figure 12-10. *Entering the PowerShell code*

Enter `$env:PSModulePath` in the text window, which will display the module-path environment variable (as defined) to the logon that is associated with the SQL Agent service. We need to know this in order to know where to place any modules we need to access from our SQL Server Agent jobs. Click on the OK button to save the job and then close the window. Now we can run the job, as shown in Figure 12-11.

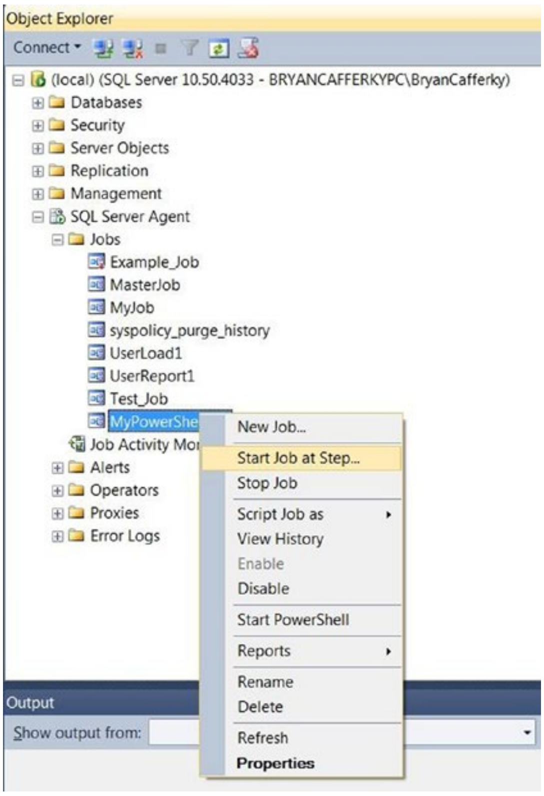
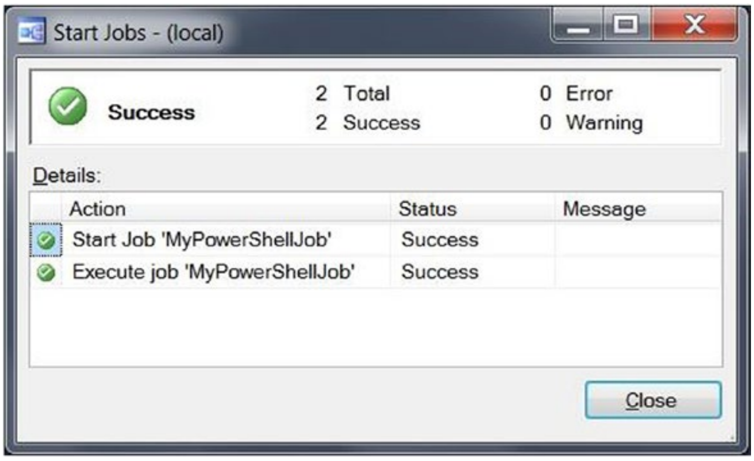


Figure 12-11. *Running the job*

To run the job, right mouse click on the job name and select Start Job. You should see the Start Jobs status window pop up, as shown in Figure 12-12. If there were multiple job steps, we would need to select the one we want to start execution from, but since there is only one step, it just runs the job.


Start Jobs - (local)

Success

2 Total
2 Success
0 Error
0 Warning

Details:

Action	Status	Message
Start Job 'MyPowerShellJob'	Success	
Execute job 'MyPowerShellJob'	Success	

Figure 12-12. *The job-execution dialog box*

If SQL Agent is set up correctly, and the logon associated with the SQL Agent service has permissions to run PowerShell scripts, you should see the status *Success* displayed in the Status column next to the `"Execute job 'MyPowerShellJob'"` action, as shown in Figure 12-12. To see the job output, right mouse click on the job name and select View History, as shown in Figure 12-13.

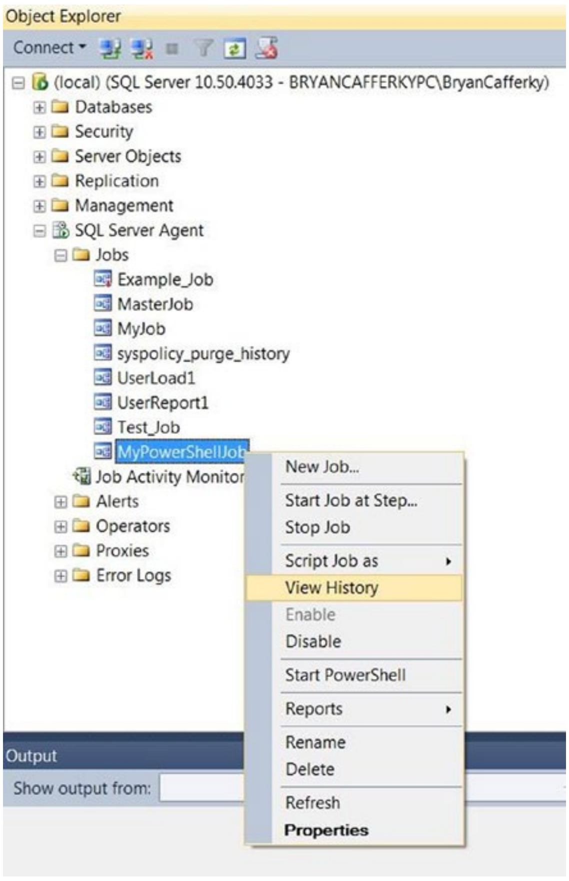
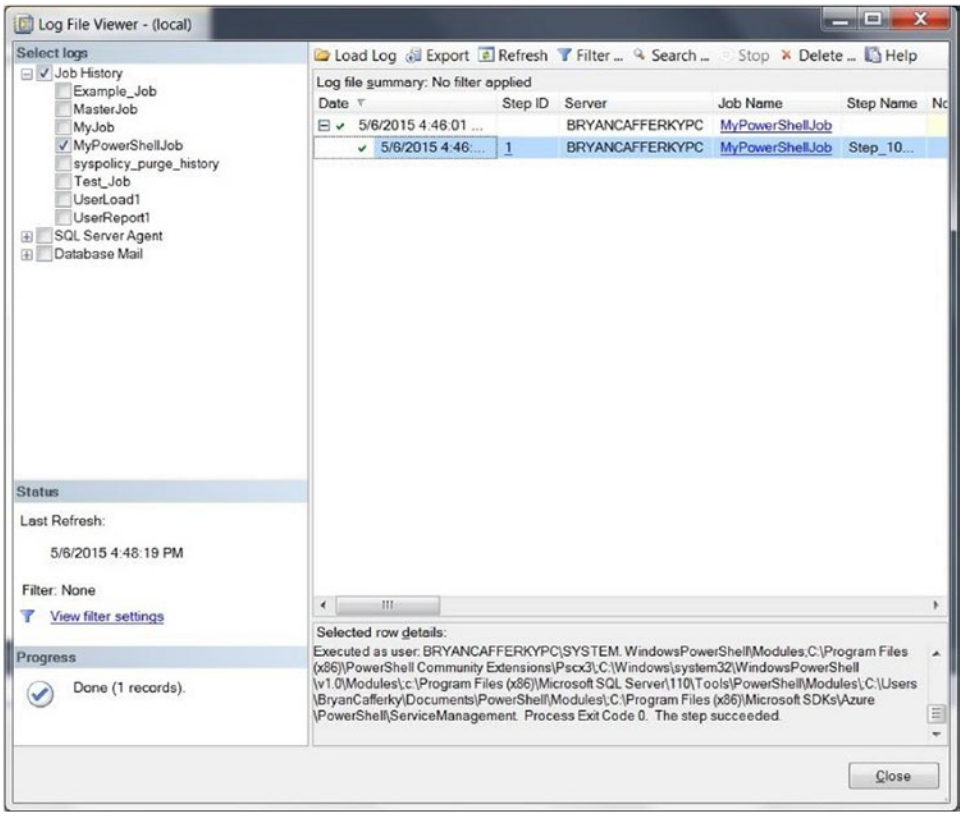


Figure 12-13. *View job history*

This will bring up the Log Viewer, where you can see job's messages, as shown in Figure 12-14.



Log file summary: No filter applied

Date	Step ID	Server	Job Name	Step Name	Nc
⊟ ✓ 5/6/2015 4:46:01 ...		BRYANCAFFERKYPC	MyPowerShellJob		
✓ 5/6/2015 4:46:...	1	BRYANCAFFERKYPC	MyPowerShellJob	Step_10...	

Selected row details:

```
Executed as user: BRYANCAFFERKYPC\SYSTEM. WindowsPowerShell\Modules;C:\Program Files
(x86)\PowerShell Community Extensions\Pscx3\;C:\Windows\system32\WindowsPowerShell
\v1.0\Modules\;c:\Program Files (x86)\Microsoft SQL Server\110\Tools\PowerShell\Modules\;C:\Users
\BryanCafferky\Documents\PowerShell\Modules\;C:\Program Files (x86)\Microsoft SDKs\Azure
\PowerShell\ServiceManagement.  Process Exit Code 0.  The step succeeded.
```

Figure 12-14. *View the job-step results*

In the Log File Summary window of the Log File Viewer, click on the plus sign next to the date in the Date column to expand the folder so we can see the job steps. Then, click on the job-step line and use the scroll bar in the text window at the bottom of the window to scroll down to the end, where we can see the output message—i.e., the value of the PowerShell module-path environment variable. Running PowerShell code from modules allows us to avoid hard coding paths to scripts or pasting code into the job-step text window. However, you need to place the modules in one of the folders displayed in the job output we just viewed.

If your SQL Server Agent is configured to use a proxy account, you will need to give permission to the account to run PowerShell scripts. To do that, expand the SQL Server Agent folder, expand Proxies folder, and right click on the PowerShell folder, as shown in Figure 12-15.

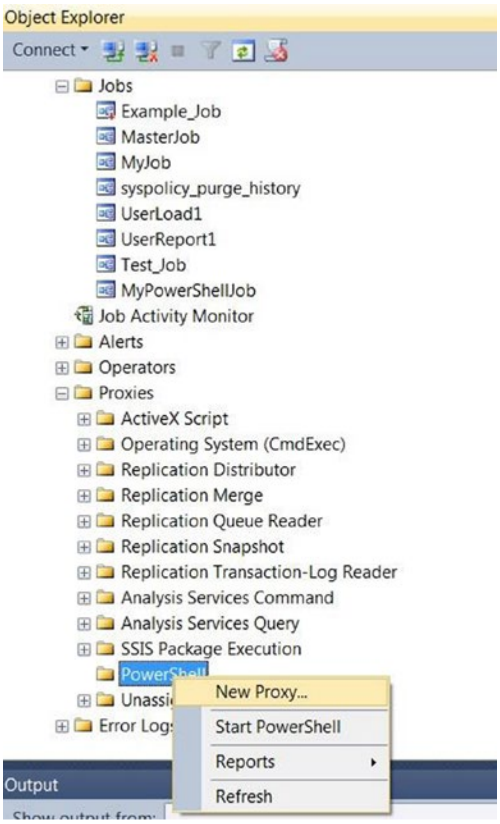
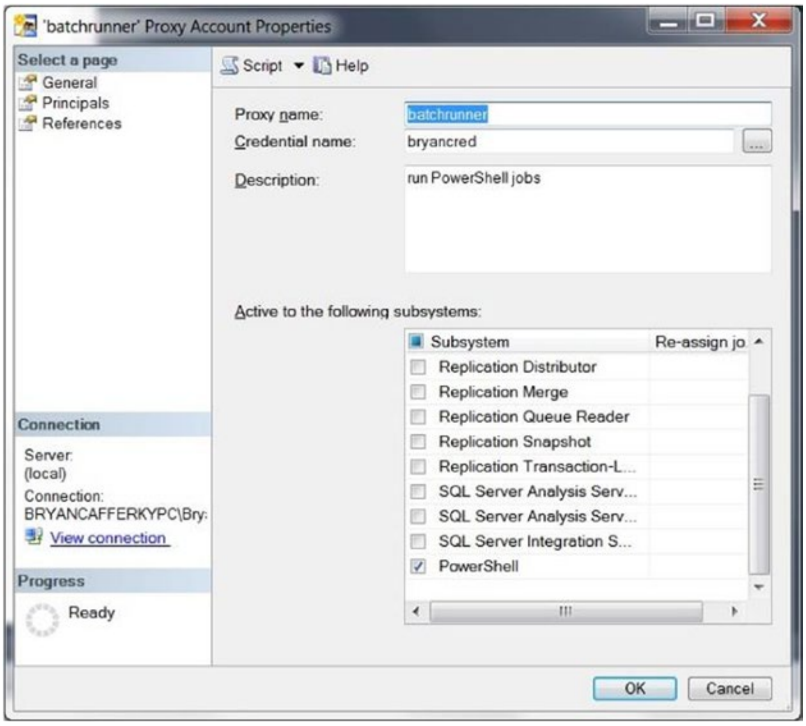


Figure 12-15. *Granting a proxy permission to run PowerShell scripts*

If you see any names listed under the PowerShell folder, these are proxies that have permission to run PowerShell, so you may be able to use one of these. If you do not see any existing proxies, select New Proxy, as shown in Figure 12-15, which will bring up the window to add a new proxy account, as shown in Figure 12-16.


'batchrunner' Proxy Account Properties

Select a page
- General
- Principals
- References

Proxy name: batchrunner
Credential name: bryancred
Description: run PowerShell jobs

Active to the following subsystems:

Subsystem	Re-assign jo
☐ Replication Distributor	
☐ Replication Merge	
☐ Replication Queue Reader	
☐ Replication Snapshot	
☐ Replication Transaction-L...	
☐ SQL Server Analysis Serv...	
☐ SQL Server Analysis Serv...	
☐ SQL Server Integration S...	
✓ PowerShell	

Connection
Server:
(local)
Connection:
BRYANCAFFERKYPC\Bry

Figure 12-16. Adding a proxy account to PowerShell

Enter the proxy name, as shown in Figure 12-16. If you know the credential name that you want to grant PowerShell permissions to, enter that as well. The credential must already exist in order to assign it to the proxy. Use the scroll bars to scroll down to PowerShell in the list and check it so it will be added to the proxy permissions. Note: Other permissions the proxy has will be checked. You can grant additional permissions by checking the related checkboxes. If you don't know the credential name, you can click on the button next to the credential name text box to get some help, as shown in Figure 12-17.

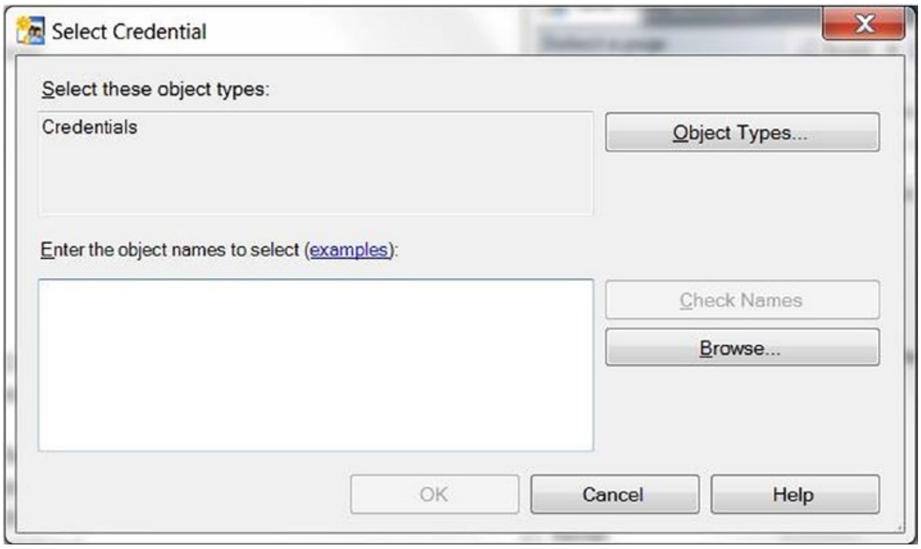
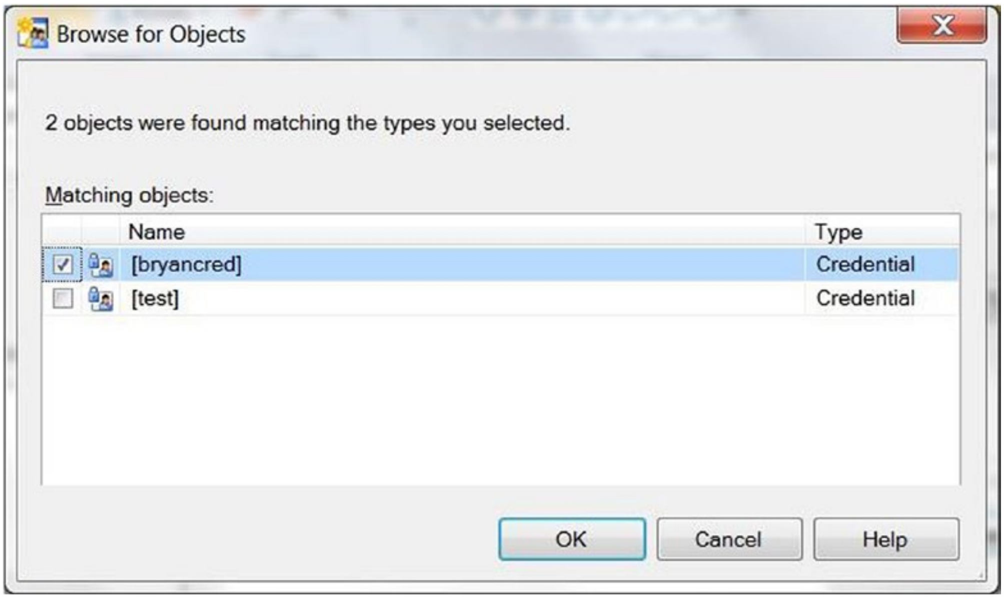


Figure 12-17. *Creating SQL Server Agent proxy—selecting a credential*

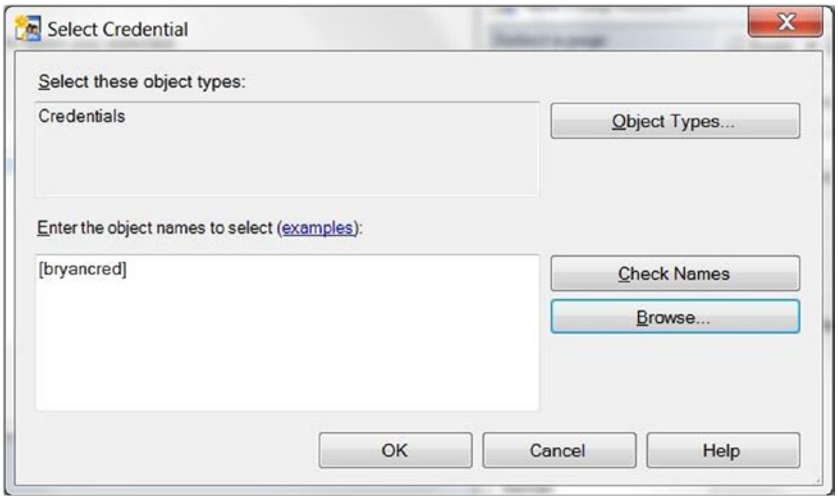
To get a list of credentials available, click the Browse button, which will bring up a list as shown in Figure 12-18.



	Name	Type
✓	[bryancred]	Credential
	[test]	Credential

Figure 12-18. *Creating SQL Server Agent proxy—browsing credentials*

Select the credential from the list using the checkbox. Then, click OK; you will return to the Select Credential window with the selected credential in the text box, as shown in Figure 12-19.

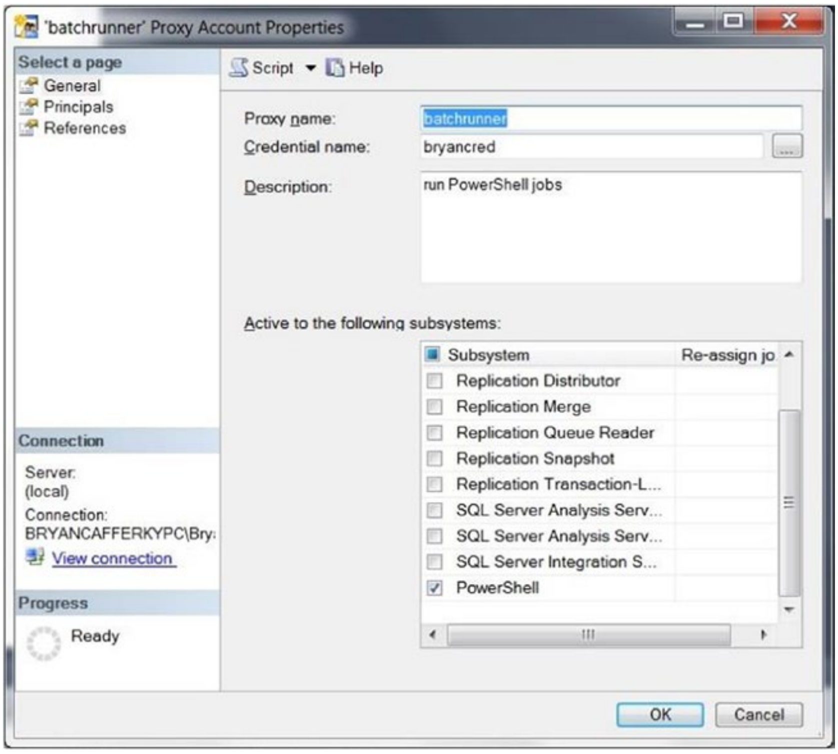

Select Credential
Select these object types:
Object Types...
Enter the object names to select (examples):
Check Names
Browse...
OK
Cancel
Help

Credentials

[bryancred]

Figure 12-19. *Creating SQL Server Agent proxy—credential selected*

Now, just click the OK button to get back to the Proxy Properties window, with the credential field filled in with your selection, as shown in Figure 12-20.



Proxy name: batchrunner
Credential name: bryancred
Description: run PowerShell jobs

Server: (local)
Connection: BRYANCAFFERKYPC\Bry

Active to the following subsystems:

	Subsystem	Re-assign jo
☐	Replication Distributor	
☐	Replication Merge	
☐	Replication Queue Reader	
☐	Replication Snapshot	
☐	Replication Transaction-L...	
☐	SQL Server Analysis Serv...	
☐	SQL Server Analysis Serv...	
☐	SQL Server Integration S...	
✓	PowerShell	

Figure 12-20. *Creating SQL Server Agent proxy—selecting the PowerShell subsystem*

Once SQL Server Agent is configured to run PowerShell code, you have an excellent job-scheduling platform with a rich visual interface—one that can work with SSIS packages, T-SQL code, Analysis Services packages, and Command Exec scripts, thereby providing the flexible integration of application components. The benefits of being able to create multi-step jobs cannot be overemphasized. Many database jobs become complex and require a long sequence of steps to complete them. The visual interface provided in SQL Server Management Studio not only provides easy job maintenance, but also the ability to view job history. As the topic of this book is focused on database development, I should emphasize that SQL Server Agent provides a high level of SQL Server integration.

Accessing SQL Server Agent Jobs from PowerShell

Throughout this book, there is an emphasis on the reach of PowerShell and how easily it can programmatically interface with Windows resources. So, it should come as no surprise that it is also easy to connect to SQL Server Agent and manipulate jobs. We could do this by directly accessing SQL Server Management Objects (SMO), but why not take advantage of the provider interface in the SQLPS module? Let's look at the function in Listing 12-2, which displays a list of SQL Server Agent jobs and will execute the jobs we select.

Listing 12-2. Invoking a SQL Agent job

```
function Invoke-UdfSQLAgentJob
{

 [CmdletBinding()]
        param (
                 [string]$sqlserverpath,
                 [string]$jobfilter     = '*'
               )
  #  Import the SQL Server module
  if(-not(Get-Module -name "sqlps"))
  {
     Import-Module "sqlps" -DisableNameChecking
  }

  # Set where you are in SQL Server...
  set-location $sqlserverpath

  while ($true)
  {

   $jobname = $null

   $jobname = Get-ChildItem |
              Select-Object -Property Name, Description, LastRunDate, LastRunOutcome |
              Where-Object {$_.name -like "$jobfilter*"} |
              Out-GridView -Title "Select a Job to Run" -OutputMode Single

   If (!$jobname) { Break }
```

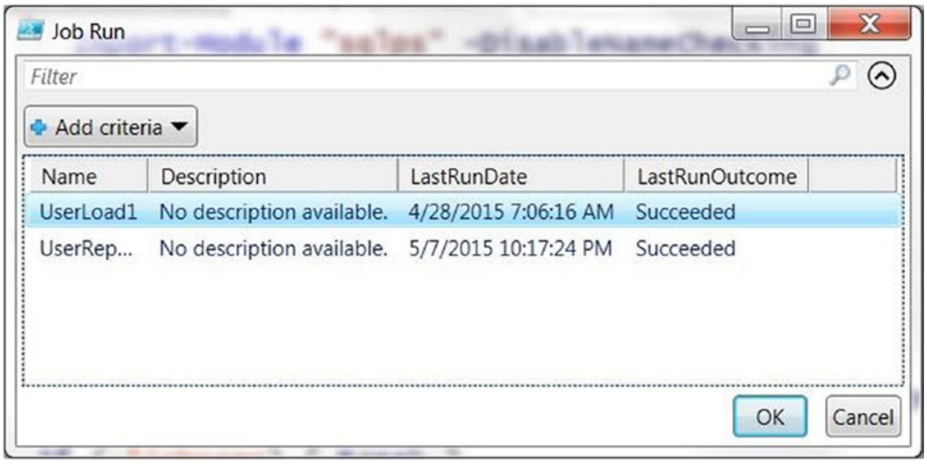
```
    $jobobject = Get-ChildItem | where-object {$_.name -eq $jobname.Name}

    $jobobject.start()

   }

}
```

Notice that the function in Listing 12-2 accepts two parameters: the location of the SQL Server Agent jobs in file-system format, i.e., provider format; and a job-name filter that is used to display only the jobs that match the `$jobfilter` name pattern. Since this function depends on the SQL Server provider, the first thing it does is load the SQLPS module if it is not already loaded. Then, it uses the `Set-Location` cmdlet to point to the SQL Agent Jobs folder on the specified SQL Server. From here, it starts an infinite loop that iterates over the job list and runs the selected job. The job name will be stored in the variable `$jobname`, so we initialize the variable to `$null` at the start of each iteration.

Now, we have a stack of commands. `Get-ChildItem` pipes a list of jobs to the `Select-Object` cmdlet, which extracts specific properties and passes them through the pipeline into the `Where-Object` cmdlet, which filters out job names that do not match the name pattern specified in `$jobfilter`. This is passed into `Out-GridView`, which displays the job properties and replaces the default window title with "Select a Job to Run." The parameter `OutputMode Single` means that only one selection will be allowed. The function will pause until the user either clicks Cancel or OK or hits Escape. At that point, if the OK button was pressed, the selected job is stored in `$jobname`. Then, `Get-ChildItem` is used to load the job object into `$jobobject`. Finally, the job is submitted using the object's `Start` method, and we go back to the top of the loop; i.e., we see the list of jobs again. If `Cancel` was clicked or the Escape key was pressed, the `$jobname` gets no value and the function exits via the `Break` statement. Let's try the function out:

```
Invoke-UdfSQLAgentJob -sqlserverpath 'SQLSERVER:\SQL\MyPC\DEFAULT\JobServer\Jobs\' `
                      -jobfilter 'User*'
```

We run the function, passing the SQL Server instance and a job filter of `User*`, meaning only jobs that start with the string `User` will be displayed. You will need to change the SQL Server instance to a server available in your environment. On my machine, I have two such jobs, so the `Out-GridView` displayed appears as shown in Figure 12-21.



Name	Description	LastRunDate	LastRunOutcome
UserLoad1	No description available.	4/28/2015 7:06:16 AM	Succeeded
UserRep...	No description available.	5/7/2015 10:17:24 PM	Succeeded

Figure 12-21. *The Out-GridView list of jobs displayed by function Invoke-UdfSQLAgentJob*

Using the gridview display shown in Figure 12-21, click on a job and click OK; the job will be executed. This type of function may come in handy when we are working with PowerShell interactively. However, the function name is not easy to remember or type. Let's make it easier by creating an alias for it, as shown here:

```
New-Alias ssaj Invoke-UdfSQLAgentJob  # ssaj stands for SQL Server Agent Jobs
```

Now, we can run the function just by entering `ssaj`, as shown:

```
ssaj -jobfilter 'User*'
```

Not only is this a lot shorter, but we can also still pass parameters, such as the job filter. Of course, we don't want to enter the `New-Alias` command to define this alias every time we start a session. Instead, we can add the new alias to our profile so it will always be available to us.

We can see how easy it is to interface with SQL Server Agent from PowerShell; not only can SQL Server Agent jobs execute PowerShell scripts, but PowerShell scripts can also execute SQL Server Agent jobs. Using the same techniques, we could even create, modify, or delete SQL Server Agent jobs.

Summary

This chapter explained PowerShell's unattended job-execution features. At its simplest, this is the ability to run a script as a background task. However, this requires that the PowerShell session that submitted the job remains active. PowerShell provides a number of cmdlets to support scheduling jobs. As we saw, these cmdlets are really an interface to the Windows Task Scheduler, and, once created, the jobs can be maintained there. We reviewed the cmdlets that support the three components of a scheduled job: the job definition, the trigger(s), and options. The job definition defines *what* we want executed. The triggers define *when* the job should be executed. The options define *how* the job should be executed. We wrapped up the section on job cmdlets by demonstrating how we can write PowerShell code to make it easier to work with PowerShell jobs. Then, we considered why SQL Server Agent is a better avenue for a job-scheduling solution because of its advanced features, SQL Server Integration, and visual interface. We discussed how to create a job that executes a PowerShell script using SQL Server Agent's built-in PowerShell integration. Then, we reviewed how easily we can manipulate SQL Server Agent jobs with PowerShell. If SQL Server Agent is so much better than PowerShell's job-scheduling features, you may be wondering why we even covered it in such detail. The answer is twofold. First, PowerShell's job-execution and scheduling features can be very useful for development. Second, PowerShell's remote execution and workflows leverage the batch-execution features, so we need to understand this before we can delve into remote execution and workflows in the next chapter.