Full credit is given to the above companies including the OS that this PDF file was generated!

PowerShell Get-Help on command 'Wait-Job'

PS C:\Users\wahid> Get-Help Wait-Job

NAME

Wait-Job

SYNOPSIS

Waits until one or all of the PowerShell jobs running in the session are in a terminating state.

SYNTAX

Wait-Job [-Filter] <System.Collections.Hashtable> [-Any] [-Force] [-Timeout <System.Int32>] [<CommonParameters>]

Wait-Job [-Id] <System.Int32[]> [-Any] [-Force] [-Timeout <System.Int32>] [<CommonParameters>]

Wait-Job [-InstanceId] <System.Guid[]> [-Any] [-Force] [-Timeout <System.Int32>] [<CommonParameters>]

Wait-Job [-Job] <System.Management.Automation.Job[]> [-Any] [-Force] [-Timeout <System.Int32>] [<CommonParameters>]

Wait-Job [-Name] <System.String[]> [-Any] [-Force] [-Timeout <System.Int32>]
[<CommonParameters>]

Wait-Job [-State] {NotStarted | Running | Completed | Failed | Stopped |
Blocked | Suspended | Disconnected | Suspending | Stopping | AtBreakpoint}
[-Any] [-Force] [-Timeout <System.Int32>] [<CommonParameters>]

DESCRIPTION

The `Wait-Job` cmdlet waits for a job to be in a terminating state before continuing execution. The terminating states are:

- Completed

- Failed

- Stopped

- Suspended

- Disconnected

You can wait until a specified job, or all jobs are in a terminating state.
You can also set a maximum wait time for the job using the Timeout parameter, or use the Force parameter to wait for a job in the `Suspended` or `Disconnected` states.

When the commands in the job are complete, `Wait-Job` returns a job object and continues execution.

You can use the `Wait-Job` cmdlet to wait for jobs started by using the `Start-Job` cmdlet or the AsJob parameter of the `Invoke-Command` cmdlet. For

more information about jobs, see about_Jobs (./about/about_Jobs.md).
Starting in Windows PowerShell 3.0, the `Wait-Job` cmdlet also waits for
custom job types, such as workflow jobs and instances of scheduled jobs. To
enable `Wait-Job` to wait for jobs of a particular type, import the module
that supports the custom job type into the session before you run the
`Get-Job` cmdlet, either by using the `Import-Module` cmdlet or by using or
getting a cmdlet in the module. For information about a particular custom job
type, see the documentation of the custom job type feature.

PARAMETERS

-Any <System.Management.Automation.SwitchParameter>

Indicates that this cmdlet returns the job object and continues execution when any job finishes. By default, `Wait-Job` waits until all of the specified jobs are complete before it displays the prompt.

-Filter <System.Collections.Hashtable>

Specifies a hash table of conditions. This cmdlet waits for jobs that satisfy all of the conditions in the hash table. Enter a hash table where the keys are job properties and the values are job property values.

This parameter works only on custom job types, such as workflow jobs and scheduled jobs. It does not work on standard jobs, such as those created by using the `Start-Job` cmdlet. For information about support for this parameter, see the help topic for the job type.

This parameter was introduced in Windows PowerShell 3.0.

-Force <System.Management.Automation.SwitchParameter>

Indicates that this cmdlet continues to wait for jobs in the Suspended or Disconnected state. By default, `Wait-Job` returns, or ends the wait, when jobs are in one of the following states:

- Completed

- Failed

- Stopped

- Suspended

- Disconnected

This parameter was introduced in Windows PowerShell 3.0.

-Id <System.Int32[]>

Specifies an array of IDs of jobs for which this cmdlet waits.

The ID is an integer that uniquely identifies the job in the current session. It is easier to remember and type than the instance ID, but it is unique only in the current session. You can type one or more IDs, separated by commas. To find the ID of a job, type `Get-Job`.

-InstanceId <System.Guid[]>

Specifies an array of instance IDs of jobs for which this cmdlet waits. The default is all jobs.

An instance ID is a GUID that uniquely identifies the job on the computer. To find the instance ID of a job, use `Get-Job`.

-Job <System.Management.Automation.Job[]>

Specifies the jobs for which this cmdlet waits. Enter a variable that contains the job objects or a command that gets the job objects. You can

also use a pipeline operator to send job objects to the `Wait-Job` cmdlet.

By default, `Wait-Job` waits for all jobs created in the current session.

-Name <System.String[]>

Specifies friendly names of jobs for which this cmdlet waits.

-State <System.Management.Automation.JobState>

Specifies a job state. This cmdlet waits only for jobs in the specified state. The acceptable values for this parameter are:

- NotStarted
- Running
- Completed
- Failed
- Stopped
- Blocked
- Suspended
- Disconnected
- Suspending
- Stopping

For more information about job states, see JobState Enumeration (/dotnet/api/system.management.automation.jobstate).

-Timeout <System.Int32>

Specifies the maximum wait time for each job, in seconds. The default value, -1, indicates that the cmdlet waits until the job finishes. The timing starts when you submit the `Wait-Job` command, not the `Start-Job` command.

If this time is exceeded, the wait ends and execution continues, even if the job is still running. The command does not display any error message.

<CommonParameters>

This cmdlet supports the common parameters: Verbose, Debug, ErrorAction, ErrorVariable, WarningAction, WarningVariable, OutBuffer, PipelineVariable, and OutVariable. For more information, see about_CommonParameters (https:/go.microsoft.com/fwlink/?LinkID=113216).

---------- Example 1: Wait for all jobs ----------

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

This command waits for all of the jobs running in the session to finish.

Example 2: Wait for jobs started on remote computers by using Start-Job

```
$s = New-PSSession Server01, Server02, Server03
Invoke-Command -Session $s -ScriptBlock {Start-Job -Name Date1 -ScriptBlock
{Get-Date}}
$done = Invoke-Command -Session $s -Command {Wait-Job -Name Date1}
$done.Count
```

3

This example shows how to use the `Wait-Job` cmdlet with jobs started on

remote computers by using the `Start-Job` cmdlet. Both `Start-Job` and `Wait-Job` commands are submitted to the remote computer by using the `Invoke-Command` cmdlet.

This example uses `Wait-Job` to determine whether a `Get-Date` command running as a job on three different computers is finished.

The first command creates a Windows PowerShell session (PSSession) on each of the three remote computers and stores them in the `$s` variable.

The second command uses `Invoke-Command` to run `Start-Job` in each of the three sessions in `$s`. All of the jobs are named Date1.

The third command uses `Invoke-Command` to run `Wait-Job`. This command waits for the `Date1` jobs on each computer to finish. It stores the resulting collection (array) of job objects in the `$done` variable.

The fourth command uses the Count property of the array of job objects in the `$done` variable to determine how many of the jobs are finished.
------- Example 3: Determine when the first job finishes -------

```
$s = New-PSSession -ComputerName (Get-Content -Path .\Machines.txt)
$c = 'Get-EventLog -LogName System | Where-Object {$PSItem.EntryType -eq
"error" --and $PSItem.Source -eq "LSASRV"} | Out-File -FilePath Errors.txt'
Invoke-Command -Session $s -ScriptBlock {Start-Job -ScriptBlock {$Using:c}
Invoke-Command -Session $s -ScriptBlock {Wait-Job -Any}
```

This example uses the Any parameter of `Wait-Job` to determine when the first of many jobs running in the current session are in a terminating state. It also shows how to use the `Wait-Job` cmdlet to wait for remote jobs to finish.

The first command creates a PSSession on each of the computers listed in the Machines.txt file and stores the PSSession objects in the `$s` variable. The

command uses the `Get-Content` cmdlet to get the contents of the file. The `Get-Content` command is enclosed in parentheses to make sure that it runs before the `New-PSSession` command.

The second command stores a `Get-EventLog` command string, in quotation marks, in the `$c` variable.

The third command uses `Invoke-Command` cmdlet to run `Start-Job` in each of the sessions in `$s`. The `Start-Job` command starts a job that runs the `Get-EventLog` command in the `$c` variable.

The command uses the Using scope modifier to indicate that the `$c` variable was defined on the local computer. The Using scope modifier is introduced in Windows PowerShell 3.0. For more information about the Using scope modifier, see about_Remote_Variables (./about/about_Remote_Variables.md).

The fourth command uses `Invoke-Command` to run a `Wait-Job` command in the sessions. It uses the Any parameter to wait until the first job on the remote computers is terminating state.
--- Example 4: Set a wait time for jobs on remote computers ---

```
PS> $s = New-PSSession -ComputerName Server01, Server02, Server03
PS> $jobs = Invoke-Command -Session $s -ScriptBlock {Start-Job -ScriptBlock
{Get-Date}}
PS> $done = Invoke-Command -Session $s -ScriptBlock {Wait-Job -Timeout 30}
PS>
```

This example shows how to use the Timeout parameter of `Wait-Job` to set a maximum wait time for the jobs running on remote computers.

The first command creates a PSSession on each of three remote computers (Server01, Server02, and Server03), and then stores the PSSession objects in the `$s` variable.

The second command uses `Invoke-Command` to run `Start-Job` in each of the PSSession objects in `$s`. It stores the resulting job objects in the `$jobs` variable.

The third command uses `Invoke-Command` to run `Wait-Job` in each of the sessions in `$s`. The `Wait-Job` command determines whether all of the commands have completed within 30 seconds. It uses the Timeout parameter with a value of 30 to establish the maximum wait time, and then stores the results of the command in the `$done` variable.

In this case, after 30 seconds, only the command on the Server02 computer has completed. `Wait-Job` ends the wait, returns the object that represents the job that was completed, and displays the command prompt.

The `$done` variable contains a job object that represents the job that ran on Server02.

------ Example 5: Wait until one of several jobs finishes ------

```
Wait-Job -id 1,2,5 -Any
```

This command identifies three jobs by their IDs and waits until any one of them are in a terminating state. Execution continues when the first job finishes.

Example 6: Wait for a period, then allow job to continue in background

```
Wait-Job -Name "DailyLog" -Timeout 120
```

This command waits 120 seconds (two minutes) for the DailyLog job to finish. If the job does not finish in the next two minutes, execution continues, and the job continues to run in the background.

---------- Example 7: Wait for a job by name ----------

```
Wait-Job -Name "Job3"
```

This command uses the job name to identify the job for which to wait.

Example 8: Wait for jobs on local computer started with Start-Job

```
$j = Start-Job -ScriptBlock {Get-ChildItem -Filter *.ps1| Where-Object
{$PSItem.LastWriteTime -gt ((Get-Date) - (New-TimeSpan -Days 7))}}
$j | Wait-Job
```

This example shows how to use the `Wait-Job` cmdlet with jobs started on the local computer by using `Start-Job`.

These commands start a job that gets the Windows PowerShell script files that were added or updated in the last week.

The first command uses `Start-Job` to start a job on the local computer. The job runs a `Get-ChildItem` command that gets all of the files that have a .ps1 file name extension that were added or updated in the last week.

The third command uses `Wait-Job` to wait until the job is in a terminating state. When the job finishes, the command displays the job object, which contains information about the job.

Example 9: Wait for jobs started on remote computers by using Invoke-Command

```
$s = New-PSSession -ComputerName Server01, Server02, Server03
$j = Invoke-Command -Session $s -ScriptBlock {Get-Process} -AsJob
$j | Wait-Job
```

This example shows how to use `Wait-Job` with jobs started on remote computers by using the AsJob parameter of `Invoke-Command`. When using AsJob , the job is created on the local computer and the results are automatically returned to the local computer, even though the job runs on the remote computers.

This example uses `Wait-Job` to determine whether a `Get-Process` command running in the sessions on three remote computers is in a terminating state.

The first command creates PSSession objects on three computers and stores them in the `$s` variable.

The second command uses `Invoke-Command` to run `Get-Process` in each of the three sessions in `$s`. The command uses the AsJob parameter to run the command asynchronously as a job. The command returns a job object, just like the jobs started by using `Start-Job`, and the job object is stored in the `$j` variable.

The third command uses a pipeline operator (`|`) to send the job object in `$j` to the `Wait-Job` cmdlet. An `Invoke-Command` command is not required in this case, because the job resides on the local computer.

---------- Example 10: Wait for a job that has an ID ----------

```
Get-Job

Id   Name     State      HasMoreData     Location            Command
--   ----     -----      -----------     --------            -------
1    Job1     Completed  True            localhost,Server01.. get-service
4    Job4     Completed  True            localhost           dir | where

Wait-Job -Id 1
```

This command waits for the job with an ID value of 1.

REMARKS

To see the examples, type: "get-help Wait-Job -examples".

For more information, type: "get-help Wait-Job -detailed".

For technical information, type: "get-help Wait-Job -full".

For online help, type: "get-help Wait-Job -online"