Getting Started With Windows PowerShell Workflow

Windows PowerShell Workflow is a new set of functionality that ships as part of Windows PowerShell 3.0. Windows PowerShell Workflow lets IT pros and developers apply the benefits of workflows to the automation capabilities of Windows PowerShell.

IT professionals and developers often execute management tasks across multiple machines in their IT environment. In general, those multi-machine tasks are long running and need to be robust in the face of errors and reboots. PowerShell Workflows brings new features to PowerShell world, allowing users to automate sequences of tasks that run across multiple computers or devices, while keeping robustness, scalability and performance in mind.

Overview of Windows PowerShell Workflow

A workflow is a sequence of automated steps or activities that execute tasks on or retrieve data from one or more managed nodes (computers or devices). These activities can include individual commands or scripts. Windows PowerShell Workflow enables, IT pros and developers alike, to author sequences of multi-computer management activities — that are either long-running, repeatable, frequent, parallelizable, interruptible, stoppable, or restartable — as workflows. By design, workflows can be resumed from an intentional or accidental suspension or interruption, such as a network outage, a reboot or power loss.

Creating a PowerShell Script-Based Workflow

Workflow is integrated into Windows PowerShell thorough a set of extensions to the PowerShell scripting language. The most important of these extensions is the workflow keyword. To create a workflow, use the **workflow** keyword followed by a name for your workflow followed by the body of the workflow.

```
workflow Invoke-HelloWorld {“Hello World from workflow"}
```

Getting Information about Your Workflow

To find all the data available about the workflow command, call get-command <workflow name>, just like you would do for any other PowerShell command.

```
Get-Command Invoke-HelloWorld
Get-Command Invoke-HelloWorld -syntax
```

To find more details on the workflow command, run the following command:

```
Get-Command Invoke-HelloWorld | Format-List *
```

Running and Managing PowerShell Workflow

To invoke your workflow, type the workflow name on the PowerShell prompt as you would do for any other PowerShell command.

```
Invoke-HelloWorld
```

Additionally, workflows can be executed as a background job by specifying the **–AsJob** parameter.

```
Invoke-HelloWorld –AsJob
$workflowJob = Invoke-HelloWorld -AsJob
```

PowerShell workflows are built on top of the existing PowerShell job infrastructure. To check the status of your workflow jobs, use the regular ***-Job** cmdlets, such as

- **Get-Job** to check the status of the job
- **Stop-Job** to stop the workflow job execution
- **Remove-Job** to remove workflow job
- **Receive-Job** to receive the data produced by workflow execution
- **Wait-Job** to wait for workflow execution to complete

Additionally, one can use the new **Suspend-Job** and **Resume-Job** cmdlets to suspend or continue the workflow execution.

Workflow Common Parameters

There are many workflow common parameters available by default for each workflow command. These workflow common parameters provide the most common functionality that you need for multi-machine management, such as PSComputername. The following is the list of workflow common parameters:

Parameter Name	Description
PSParameterCollection	List of hash tables to specify different parameter values for different target machines
PSComputerName	List of machines that the workflow will be executed on
PSCredential	Credential to use for workflow session to target machines
PSConnectionRetryCount	Number of times workflow should try reconnecting to target machines in case there are connection issues

PSConnectionRetryIntervalSec	Number of seconds to wait between each successive connection retry
PSRunningTimeoutSec	Number of seconds in which workflow execution should finish, otherwise terminate the execution. This timeout does not include the time period when the workflow is in suspended state.
PSElapsedTimeoutSec	Number of seconds in which workflow execution should finish, otherwise terminate the execution, and remove all the workflow data. This timeout does include the time period when the workflow is in suspended state.
PSPersist	Forces the workflow execution to checkpoint the workflow data and state after each executing each step (activity) within the workflow
PSAuthentication	Specifies the mechanism that is used to authenticate the user's credentials. Valid values are Default, Basic, Credssp, Digest, Kerberos, Negotiate, and NegotiateWithImplicitCredential. The default value is Default.
PSAuthenticationLevel	Specifies the authentication level to use for the WMI connection. Valid values are: Unchanged, Default, None, Connect, Call, Packet, PacketIntegrity, PacketPrivacy
PSApplicationName	Specifies the application name segment of the connection URI for connection to target machine. The default value is WSMAN. This value is appropriate for most uses
PSPort	Specifies the network port on the remote computer that is used for this connection. To connect to a remote computer, the remote computer must be listening on the port that the connection uses. The default ports are 5985 (the WinRM port for HTTP) and 5986 (the WinRM port for HTTPS).
PSUseSSL	Uses the Secure Sockets Layer (SSL) protocol to establish a connection to the remote computer. By default, SSL is not used.
PSConfigurationName	Specifies the session configuration that is used for the connection to target machine. If the specified session configuration does not exist on the target machine, the command fails. The default value is Microsoft.PowerShell
PSConnectionURI	Specifies a Uniform Resource Identifier (URI) that defines the connection endpoint for the session. The URI must be fully qualified. The format of this string is as follows: <Transport>://<ComputerName>:<Port>/<ApplicationName> The default value is: http://localhost:5985/WSMAN
PSAllowRedirection	Allows redirection of this connection to an alternate URI. The target machine can return an instruction to redirect to a different URI. By default, Windows PowerShell Workflow does not redirect connections, but you can use this parameter to allow it to redirect the connection.
PSSessionOption	Sets advanced options for the connection to target machine. Enter a SessionOption object that you create by using the New-PSSessionOption cmdlet.
PSCertificateThumbprint	Specifies the digital public key certificate (X509) of a user account that

	has permission to perform this action. Enter the certificate thumbprint of the certificate.
PSPrivateMetadata	Hashtable object that represent user/application level information that can later be used to identify/filter the workflow execution.
AsJob	Runs the workflow as a background job on the local computer. Use this parameter to run workflows that take an extensive time to complete. When you use AsJob, the command returns an object that represents the job, and then displays the command prompt. To manage the job, use the Job cmdlets. To get the job results, use Receive-Job.
JobName	Specifies a friendly name for the background job. By default, jobs are named "Job<n>", where <n> is an ordinal number.
InputObject	Specifies input to the workflow. Enter a variable that contains the objects or type a command or expression that gets the objects.

Workflow Extensions to the PowerShell Language

Parallel Execution of Workflow Activities

The Windows Workflow Foundation runtime supports the execution of parallel activities. This capability is exposed in PowerShell Workflow through the **parallel** keyword and an enhancement to the foreach statement allowing the user to write **foreach –parallel { … }**. Additionally, to execute a collection of activities in-order (not in parallel) within **parallel** block, use **sequence** block.

```
workflow Invoke-ParallelWorkflow {

    # All commands within the following block can execute
    # in any order
    parallel {
        Get-Process -Name PowerShell
        # All the commands within the following block will
        # execute in the specified sequential order

        sequence {
            "In sequence 1 of 2"
            "In sequence 2 of 2"
        }

        "In parallel 1 of 2"
        "In parallel 2 of 2"
        Get-Service -Name WinRM
    }
}

workflow Invoke-ForEachParallel
{
    param([string[]]$computerName)
```

```
        # The contents of the foreach block will be executed in
        parallel
        foreach -parallel($computer in $computerName)
        {
            "Executing on $computer"
        }
    }
```

Workflow Calling Workflow

With this CTP release, users can now call a workflow from inside workflows. This feature enables the reuse of workflows to create higher-level workflows. To use this feature, the workflow (**Invoke-Foo**) that you plan to call from inside the other workflow (**Invoke-Bar**), must be defined in the PowerShell session that is defining the calling workflow.

```
workflow Invoke-Foo {
    "    Hello from Foo Workflow"
}

workflow Invoke-Bar
{
    "    Hello from Bar Workflow"
    "Calling Foo workflow ...."
    Invoke-Foo
}
```

NOTE: *In the example above, changing the* ***Invoke-Foo*** *will have no impact on the result produced by the* ***Invoke-Bar*** *until* ***Invoke-Bar*** *is redefined (activity/function references in workflow are static at compile time.)*

Additionally, you can define nested functions or nested workflows inside the workflows as well.

```
workflow Invoke-NestedCommand
{
    "    Hello from Workflow"

    # Nested worflow defined inside the workflow
    workflow Invoke-NestedWorkflow{
        "     Hello from Nested Workflow"
    }

    # Nested function defined inside the workflow
    function Invoke-NestedFunction
    {
        "     Hello from Nested Function"
    }

    "Calling nested function ...."
    Invoke-NestedFunction

    "Calling nested workflow ...."
```

```
    Invoke-NestedWorkflow
}
```

Running Isolated Blocks of PowerShell Script

Even though each command in workflow is executed with no PowerShell state sharing (e.g: variables created/set by one command are not visible to the next command), it is possible to execute a collection of PowerShell commands or language elements as a single execution unit by using the **inlineScript** keyword.

```
workflow Invoke-InlineScript
{
    # Following commands will share the PowerShell state as
they execute
    inlineScript {
        $a = 2
        $b = $a+2
        $b
    }
}
```

Accessing Workflow Variables from Different Execution Scopes

Like PowerShell remoting, PowerShell script-based workflows support **$using:<variable name>** syntax. This new syntax can be used to import the workflow variable into the context of an inlineScript activity.

Unlike PowerShell, Windows Workflow Foundation does not support dynamic scoping of variables. This means that a variable defined in the parent scope cannot be redefined in a child scope. In PowerShell script-based workflow, to access the workflow's scope variable from any inner scope, use the **$workflow:<variable name>** syntax.

```
workflow Invoke-WithUsingandWorkflowScope
{
    # This is a workflow top-level variable
    $a = 22
    "Initial value of A: is $a"

    # Access $a from Inlinescript (bringing a workflow
    variable to the PowerShell session) using $using
    inlinescript {"PowerShell variable A is: $a"}
    inlinescript {"Workflow variable A is: $using:a"}

    parallel
    {
        sequence
        {
          # Updating a top-level variable with
          $workflow:<variable name>
            $workflow:a = 3
```

```
            # Reading a top-level variable (no $workflow:
            needed)
              "Value of A inside parallel is: $a"
        }
    }
    "Updated value of A is: $a"
}
```

PowerShell Language Restrictions in PowerShell Workflow

PowerShell Workflow does not support the full language semantics of PowerShell. Below is the list of restrictions:

- **Begin**, **Process**, and **End** statements are not supported in a PowerShell workflow.
  Example: `workflow foo {param(); begin { "Hello World" } }`
- **Break** and **Continue** statements along with loop labels are not supported in a PowerShell workflow. Instead, use an 'if' statement to control loop execution.
- Sub expressions are not supported.
  Example: `$myVariable = $(<#Some complicated sub-expression#>)`
- Multiple assignment is not supported.
  Example: `$foo = $bar = "Hello World"`
- In a PowerShell workflow, loop conditions that modify variables are not supported. To change a variable, place the modification statement in the loop body itself.
  Example:
  ```
  workflow foo { $i=0; while ($i++){if ($i -gt 10){break;} $i} }
  ```
- Dynamic parameters are not supported in a PowerShell workflow.
- Assignment to drive-qualified variables is not supported.
  Example: `$env:FOO = "Bar"`
- In a PowerShell workflow, variable names must contain only letters, digits, '-', and '_'.
- Method invocation is in statements is not supported in a PowerShell workflow. To do .NET scripting, use the inlineScript keyword{ <commands> }.
  Example: `workflow foo { inlineScript { "Test".Substring(1,10) } }`

  Reason: This implies that you've got a live object to work on, which is not possible if the workflow is persisted in-between the call that generates the object and the call that uses its method.
- You can only provide one #requires statement per workflow definition.
  Example: `#requires -Assembly Foo; #requires -Assembly Bar`
- Assignment to object properties is not supported.
  Example: `$Something.Property = 1`

- Dot-sourcing (. <command>) and the invocation operator (& <command>) are not supported in a PowerShell workflow.
  Example: `. .\foo.ps1`

  Reason: These techniques are commonly used to pre-populate the session with dynamic commands. Commands must be pre-defined in a workflow.

- Advanced parameter validation is not supported on nested workflows.
  Example:
  ```
  workflow Bar { param([Mandatory] $myParam2) "Hello" }
  workflow Foo { param([Mandatory] $myParam) Bar }
  ```

  Reason: Parameter validation is not supported at all by workflow, but we can simulate it by propagating the validation attributes to the outermost wrapper function ("Foo" in this example). We can't do the same thing for the nested workflow, as it is pre-compiled by Windows Workflow Foundation.

- Positional parameters are not supported in a PowerShell workflow. To invoke a command, use explicit parameter names with all values.
  Example: `Start-Sleep 10`

  Reason: Positional parameters require a lot of run-time validation, such as whether your input can be converted to the parameter in question. They happen after parameter set resolution, which is also a run-time thing. Workflow validation is done at compile-time.

- In a PowerShell workflow, the **switch** statement supports only the '**caseSensitive**' flag with and only constant expressions are supported as switch clauses in a PowerShell workflow.
  Reason: The workflow runtime only supports basic, case-sensitive string comparisons in its switch statement. It does not support regex, file, script block, or case insensitivity.

- The **trap** statements are not supported in a PowerShell workflow. Instead, use **try/catch/finally**.
- Inline help is not supported for PowerShell workflows. Instead, use maml help in the module that packages the workflow.

Persisting Workflow Data

Windows Workflow Foundation allows a workflow execution to persist or checkpoint all of its state by an explicit call to the Persist activity. The same is exposed through PowerShell script-based workflow via the **Checkpoint-Workflow** activity, along with a call to **Persist** activity as a command. Alternatively, you can add **–PSPersist $true** at the end of an activity to achieve similar results.

PowerShell Workflow will not only checkpoint the workflow state but also its output to the persistence store.

```
workflow Set-WorkflowState
{
    "Hello World"

    # Calls the Windows workflow foundation persist
    activity
    Checkpoint-Workflow

    #Simulate long running command
    start-Sleep -seconds 30

    # Calls the Windows Workflow Foundation Persist
    activity
    Persist

    "Hello Mars"
}
```

Suspending a Workflow from within Itself

With PowerShell script-based workflows, there is an option to suspend the workflow from within during execution. This can be achieved by calling the **Suspend-Workflow** command

```
workflow Invoke-SuspendWorkflow
{
    $day = (get-date).dayofweek

    if($day -eq "Friday")
    {
        Write-Warning -Message "Cannot start this long
workflow on Friday ..... Suspending.
         Use Resume-Job $jobInstanceId to resume"
        Suspend-Workflow
    }
    "Running a long workflow that should finish before the
week ends"
}
```

Access to Workflow Common Parameters from within the Workflow

By default, all the workflow common parameters can be accessed (read-only) inside script-based workflows by using the workflow common variable names such as **$pscomputername, $psconfigurationname,** etc. To find the complete list of such variables, use the **Get-PSWorkflowData** activity.

Using **Set-PSWorkflowData**, workflow authors can set the value of workflow common parameters, as well as some other useful variables.

```
workflow Get-WorkflowDataExample
{
    $before = Get-PSWorkflowData[Hashtable] -
VariableToRetrieve All
    "Before Setting: " + $before.PSComputerName
    Set-PSWorkflowData -PSComputerName "foobar"
    $after = Get-PSWorkflowData[string[]] -
VariableToRetrieve PSComputerName
    "After Setting: " + $after
}
```

Importing and Creating Workflows using the Visual Studio Workflow Designer

Apart from authoring workflows using PowerShell script, you can author workflows using the Visual Studio Workflow Designer as well. The workflows produced by workflow designer have XAML extension. PowerShell has support for consuming workflows authored in workflow designer as well as ability to surface PowerShell cmdlets as activities to be used within the workflow designer.

Importing a XAML workflow

Windows PowerShell Workflow allows you to reuse your existing investment in Windows Workflow Foundation by allowing the import of XAML workflows. It also integrates with PowerShell Modules by allowing you to import XAML workflows by using the **Import-Module** cmdlet and passing the *.xaml file name.

```
Import-module -Name <Full path to the XAML file> -Verbose
```

You can also package your XAML workflows via a **module manifest** (*.psd1 file) by specifying them in the **RootModule, NestedModules, RequiredModules** or **RequiredAssemblies** (workflow calling workflow scenario) keys.

Using PowerShell Workflow Activities from the Visual Studio Workflow Designer

Windows PowerShell ships with built-in activities for most of the cmdlets that are present as part of PowerShell installation. These activities are packaged in assemblies with names matching the PowerShell module name containing the cmdlets. Following is the list of PowerShell activity assemblies that are installed in GAC:

Microsoft.PowerShell.Activities
Microsoft.PowerShell.Core.Activities

Microsoft.PowerShell.Diagnostics.Activities
Microsoft.PowerShell.Management.Activities
Microsoft.PowerShell.Security.Activities
Microsoft.PowerShell.Utility.Activities
Microsoft.WSMan.Management.Activities

To use the above activities from workflow designer, follow the standard procedure of importing the assemblies in workflow designer toolbox defined at http://msdn.microsoft.com/en-us/library/dd797579.aspx

List of Non-Supported Windows Workflow Foundation Activities

Windows PowerShell Workflow supports most of the built-in Windows Workflow Foundation activities. For a complete list of built-in Windows Workflow Foundation activities, please see the following link: http://msdn.microsoft.com/en-us/library/dd489459(v=VS.100).aspx

The following is the small subset of Windows Workflow Foundation activities that Windows PowerShell Workflow does not support. These activities are excluded because they typically require host level support such as WCF messaging or transactions that are not supported by the PowerShell Workflow runtime.

Activity Category	Activity Name
Flow control	Pick
	PickBranch
Messaging	Send
	SendReply
	Receive
	ReceiveReply
Transaction	CancellationScope
	CompensableActivity
	Compensate
	Confirm
	TransactionScope
	TransactedReceiveScope
Migration	Interop

List of Cmdlets that Execute Local Only

Even though, PowerShell Workflow inherently supports remote execution of cmdlet actions, there are set of utility and object manipulation cmdlets that are always invoked locally. This is

done to improve performance. These cmdlets can be remotely executed using the InlineScript activity.

Cmdlets that are always execute locally by Workflow	
• Add-Member	• New-PSSessionOption
• Compare-Object	• New-PSTransportOption
• ConvertFrom-Csv	• New-TimeSpan
• ConvertFrom-Json	• Out-Default
• ConvertFrom-StringData	• Out-Host
• Convert-Path	• Out-Null
• ConvertTo-Csv	• Out-String
• ConvertTo-Html	• Select-Object
• ConvertTo-Json	• Sort-Object
• ConvertTo-Xml	• Update-List
• ForEach-Object	• Where-Object
• Get-Host	• Write-Debug
• Get-Member	• Write-Error
• Get-Random	• Write-Host
• Get-Unique	• Write-Output
• Group-Object	• Write-Progress
• Measure-Command	• Write-Verbose
• Measure-Object	• Write-Warning

Cmdlets that Do Not Have Activity Implementation

PowerShell Workflow includes corresponding activities for most of the default cmdlets however there is a small set of cmdlets where this did not make sense. These cmdlets are listed in the following table. To invoke the following cmdlets in a workflow, you have to wrap them in InlineScript

Reason	Cmdlet Names	
Only changes PowerShell session, doesn't work across activities in Workflow (each step in the workflow is isolated)	• Get-Alias • Export-Alias • Import-Alias • New-Alias • Set-Alias • Update-FormatData • Add-History • Clear-History • Get-History • Invoke-History	• New-PSDrive • Remove-PSDrive • Set-StrictMode • Start-Transcript • Stop-Transcript • Remove-TypeData • Update-TypeData • Clear-Variable • Get-Variable • New-Variable

	• Pop-Location • Push-Location • Set-Location • New-Module	• Remove-Variable • Set-Variable • Disconnect-WSMan • Connect-WSMan
No interactive support from Workflow	• Show-Command • Show-ControlPanelItem • Get-Credential • Show-EventLog	• Out-GridView • Read-Host • Debug-Process
No PowerShell script debugging support via Workflow	• Disable-PSBreakpoint • Enable-PSBreakpoint • Get-PSBreakpoint	• Remove-PSBreakpoint • Set-PSBreakpoint • Get-PSCallStack • Set-PSDebug
No PowerShell transaction support via Workflow	• Complete-Transaction • Get-Transaction • Start-Transaction	• Undo-Transaction • Use-Transaction
No Formatting support via Workflow	• Get-FormatData • Format-Custom • Format-List	• Format-Table • Format-Wide
Workflow does its own remoting	• Connect-PSSession • Disconnect-PSSession • Exit-PSSession • Enter-PSSession • Export-PSSession	• Import-PSSession • New-PSSession • New-PSSessionOption • Receive-PSSession
Others	• Export-Console • Get-ControlPanelItem • Out-Default • Out-Null • Write-Host	• Export-ModuleMember • Add-PSSnapin • Get-PSSnapin • Remove-PSSnapin • Trace-Command

Windows PowerShell Workflow Endpoint

Apart from the default remoting endpoint (Microsoft.PowerShell), Windows PowerShell ships with a workflow endpoint: **Microsoft.PowerShell.Workflow.** Unlike the default PowerShell remoting endpoint, the PowerShell Workflow endpoint runs with **-UseSharedProcess** set to true. This setting allows the user to connect to the PowerShell Workflow endpoint from different pssessions and still connect to the same process running on the server machine.

Creating a session to the workflow endpoint ensures that the PowerShell Workflow functionality is loaded.

```
Get-PSSessionConfiguration -Name *workflow
```

Additionally, the PowerShell Workflow endpoint has a very limited set of commands that are necessary for workflow execution and management.

```
Invoke-Command -ComputerName . -ConfigurationName
Microsoft.PowerShell.Workflow -ScriptBlock {get-command}
```

Creating a Session to the Default Workflow Endpoint

Creating a connection to the PowerShell Workflow endpoint is as easy as calling **New-PSWorkflowSession**. This command has all the remoting related parameters that allows you to connect to remote PowerShell Workflow sessions. Alternatively, one can use the full configuration name – **Microsoft.PowerShell.Workflow** with the **New-PSSession** cmdlet as well.

```
$wfsession = New-PSWorkflowSession
$wfsession2 = New-PSSession -ConfigurationName
Microsoft.PowerShell.Workflow
```

Creating a Custom Workflow Endpoint

Creating a custom PowerShell Workflow endpoint is the same as creating a new PowerShell remoting endpoint: you can still use the **Register/Set-PSSessionConfiguration** cmdlet for this purpose. You will notice that among other new parameters, these cmdlets have -**SessionType** and -**SessionTypeOption** parameters.

The **SessionType** parameter lets you specify the type of endpoint you want to create. The **SessionTypeOption** parameter takes an object for specifying additional options for the endpoint. For Workflow-type endpoints, you can specify the various workflow related quotas (such as number of running workflows, allowed workflow activities, list of activities that run in-process, etc.) via the object produced by the **New-PSWorkflowExecutionOption** cmdlet.