Article • 01/31/201919 minutes to read

May 2016

Volume 31 Number 5

[Windows PowerShell]

Writing Windows Services in PowerShell

By Jean-François Larvoire | May 2016 | Get the Code

Windows Services normally are compiled programs written in C, C++, C# or other Microsoft .NET Framework-based languages, and debugging such services can be fairly difficult. A few months ago, inspired by other OSes that allow writing services as simple shell scripts, I began to wonder if there could be an easier way to create them in Windows, as well.

This article presents the end result of that effort: A novel and easy way to create Windows Services, by writing them in the Windows PowerShell scripting language. No more compilation, just a quick edit/test cycle that can be done on any system, not just the developer's own.

I provide a generic service script template called PSService.ps1, which allows you to create and test new Windows Services in minutes, with just a text editor like Notepad. This technique can save a lot of time and development effort for anyone who wants to experiment with Windows Services—or even provide real services for Windows when performance isn't a critical factor. PSService.ps1 can be downloaded from bit.ly/1Y0XRQB .

What Is a Windows Service?

Windows Services are programs that run in the background, with no user interaction. For example, a Web server, which silently responds to HTTP requests for Web pages from the network, is a service, as is a monitoring application that silently logs performance measurements or records hardware sensor events.

Services can start automatically when the system boots. Or they can start on demand, as requested by applications that rely on them. Services run in their own Windows session, distinct from the UI session. They run in a number of system processes, with carefully selected rights to limit security risks.

The Windows Service Control Manager

The services are managed by the Windows Service Control Manager (SCM). The SCM is responsible for configuring services, starting them, stopping them and so forth.

The SCM control panel is accessible via Control Panel | System and Security **|** Administrative Tools **|** Services. As **Figure 1** shows, it displays a list of all configured services, with their name, description, status, startup type and user name.

The Windows Service Control Manager GUI in Windows 10

Figure 1 The Windows Service Control Manager GUI in Windows 10

There are also command-line interfaces to the SCM:

- The old net.exe tool, with its well-known “net start” and “net stop” commands, dates from as far back as MS-DOS! Despite its name, it can be used to start and stop any service, not just network services. Type “net help” for details.
- A more powerful tool called sc.exe, introduced in Windows NT, gives fine control over all aspects of service management. Type “sc /?” for details.

These command-line tools, although still present in Windows 10, are now deprecated in favor of Windows PowerShell service management functions, described later.

Gotcha: Both net.exe and sc.exe use the “short” one-word service name, which, unfortunately, isn’t the same as the more descriptive name displayed by the SCM control panel. To get the correspondence between the two names, use the Windows

PowerShell get-service command.

Service States

Services can be in a variety of states. Some states are required, others are optional. The two basic states that all services must support are stopped and started. These show up respectively as (blank) or Running in under the Status column in **Figure 1**.

A third optional state is Paused. And another implicit state every service supports even if it’s not mentioned is Uninstalled.

A service can make transitions between those states, as shown in **Figure 2**.

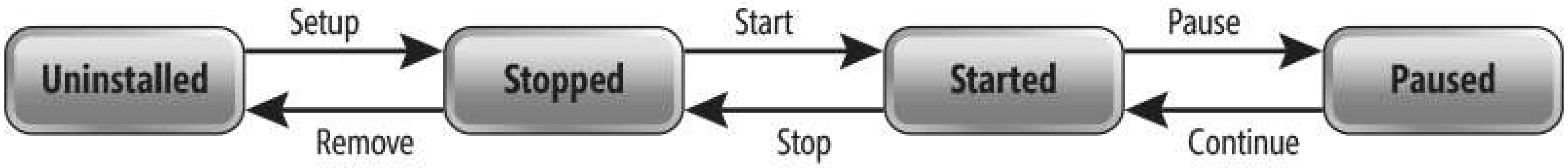

Figure 2 Service States

Finally, there are also several transitory states that services may optionally support: StartPending, StopPending, PausePending, ContinuePending. These are useful only if state transitions take a significant amount of time.

Windows PowerShell Service Management Functions

Windows PowerShell has been the recommended system management shell since Windows Vista. It includes a powerful scripting language and a large library of functions for managing all aspects of the OS. Some of Windows PowerShell strengths are:

- Consistent function names
- Fully object-oriented
- Easy management of any .NET object

Windows PowerShell provides many service management functions, which are known as cmdlets. **Figure 3** shows some examples.

Figure 3 Windows PowerShell Service Management Functions

Function Name	Description
Start-Service	Starts one or more stopped services
Stop-Service	Stops one or more running services
New-Service	Installs a new service
Get-Service	Gets the services on a local or remote computer, with their properties
Set-Service	Starts, stops and suspends a service, and changes its properties

For a complete list of all commands with the string “service” in their names, run:

```
XML
```

```
Get-Command *service*
```

For a list of just the service management functions, run:

```
XML
```

```
Get-Command -module Microsoft.PowerShell.Management *service*
```

Surprisingly, there’s no Windows PowerShell function for removing (that is, uninstalling) a service. This is one of the rare cases when it’s still necessary to use the old sc.exe tool:

XML

```
sc.exe delete $serviceName
```

The .NET ServiceBase Class

All services must create a .NET object deriving from the ServiceBase class. Microsoft documentation describes all properties and methods of that class. **Figure 4** lists a few of these, of particular interest for this project.

Figure 4 Some Properties and Methods of the ServiceBase Class

Member	Description
ServiceName	Short name used to identify the service to the system
CanStop	Whether the service can be stopped once it has started
OnStart()	Actions to take when the service starts
OnStop()	Actions to take when the service stops
Run()	Registers the service executable with the SCM

By implementing these methods, a service application will be manageable by the SCM to start automatically at boot time or on demand; and it'll be manageable by the SCM control panel, by the old net.exe and sc.exe commands, or by the new Windows PowerShell service management functions, to start or stop manually.

Creating an Executable from C# Source Embedded in a Windows PowerShell Script

PowerShell makes it easy to use .NET objects in a script. By default it has built-in support for many .NET object types, sufficient for most purposes. Better still, it’s extensible and allows embedding short C# code snippets in a Windows PowerShell script to add support for any other .NET feature. This capability is provided by the Add-Type command, which, despite its name, can do much more than just adding support for new .NET object types to Windows PowerShell. It can even compile and link a complete C# application into a new executable. For example, this hello.ps1 Windows PowerShell script:

XML

```
$source = @"
  using System;
  class Hello {
    static void Main() {
      Console.WriteLine("Hello World!");
    }
  }
"@
Add-Type -TypeDefinition $source -Language CSharp -OutputAssembly "hello.exe"
  -OutputType ConsoleApplication
```

will create a hello.exe application, that prints “Hello world!”:

XML

```
PS C:\Temp> .\hello.ps1
PS C:\Temp> .\hello.exe
Hello World!
PS C:\Temp>
```

Putting It All Together

PSService.ps1 Features Based on all I’ve discussed so far, I can now create that Windows PowerShell service I’ve been dreaming about, a PSService.ps1 script that can:

- Install and uninstall itself (using Windows PowerShell service management functions).
- Start and stop itself (using the same set of functions).
- Contain a short C# snippet, which creates the PSService.exe that the SCM expects (using the Add-Type command).
- Make the PSService.exe stub call back into the PSService.ps1 script for the actual service operation (in reponse to OnStart, OnStop and other events).
- Be manageable by the SCM control panel and all command-line tools (thanks to the PSService.exe stub).
- Be resilient, and process successfully any command when in any state. (For example, it can automatically stop the service before uninstalling it, or do nothing when asked to start an already started service.)
- Support Windows 7 and all later versions of Windows (using only Windows PowerShell v2 features).

Note that I’ll cover only the critical parts of PSService.ps1 design and implementation in this article. The sample script also contains debugging code, and some support for optional service features, but their description would needlessly complicate the explanations here.

PSService.ps1 Architecture The script is organized in a series of sections:

- A header comment describing the file.
- A comment-based help block.
- The Param block defining command-line switches.
- Global variables.
- Helper routines: Now and Log.
- A C# source block of the PSService.exe stub.
- The main routine, processing every command-line switch.

Global Settings

Immediately beneath the Param block, PSService.ps1 contains global variables defining global settings, which can be changed as needed. The defaults are shown in **Figure 5**.

Figure 5 Global Variable Defaults

Variable	Description	Default
$serviceName	A one-word name used for net start commands, and others	The base name of the script
$serviceDisplayName	A more descriptive name for the service	A Sample PowerShell Service
$installDir	Where to install the service files	${ENV:windir}\System32
$logFile	Name of the file in which to log the service messages	${ENV:windir}\Logs\$serviceName.log
$logName	Name of the Event Log in which to record service events	Application

Using the base name of the file as the service name (for example, PSService for PSService.ps1) lets you create multiple services from the same script, just by copying the script, renaming the copy, then installing the copy.

Command-Line Arguments

To make it easy to use, the script supports command-line arguments that match all state transitions, as shown in **Figure 6**.

Figure 6 Command-Line Arguments for State Transitions

Switch	Description

-Start	Start the service
-Stop	Stop the service
-Setup	Install itself as a service
-Remove	Uninstall the service

(Support for the paused state isn't implemented but would be easy to add, with the corresponding state transition options.)

Figure 7 shows a few more management arguments that the script supports.

Figure 7 Supported Management Arguments

Switch	Description
-Restart	Stop the service, then start it again
-Status	Display the current state of the service
-Service	Run the service instance (for use only by the service.exe stub)
-Version	Display the service version
Common Parameters	-? , -Verbose , -Debug and so forth

Each state transition switch has two modes of operation:

- When invoked by the end user: Use the Windows PowerShell service management functions to trigger a state transition.
- When invoked by the SCM (indirectly via the service.exe stub): Manage the service.ps1 service instance accordingly.

The two cases can be distinguished at run time by checking the user name: In the first case it's a normal user (the system administrator); in the second case it's the actual Windows system user. The system user can be identified like this:

XML

```
$identity = [Security.Principal.WindowsIdentity]::GetCurrent()
$userName = $identity.Name   # Ex: "NT AUTHORITY\SYSTEM" or "Domain\Administrator"
$isSystem = ($userName -eq "NT AUTHORITY\SYSTEM")
```

Installation

The goal of a service installation is to store a copy of the service files in a local directory, then to declare this to the SCM, so that it knows which program to run to start the service.

Here's the sequence of operations performed by the -Setup switch processing:

1. Uninstall any previous instance, if any.
2. Create the installation directory if needed. (This isn't needed for the default: C:\Windows\System32.)
3. Copy the service script into the installation directory.
4. Create a service.exe stub in that same installation directory, from the C# snippet in the script.
5. Register the service.

Note that starting with a single Windows PowerShell source script (PSService.ps1), I end up with three files installed in C:\Windows\System32: PSService.ps1, PSService.pdb and PSService.exe. These three files will need to be removed during uninstallation. The installation is implemented by including two pieces of code in the script:

- The definition of the -Setup switch in the Param block at the beginning of the script:

XML

```
[Parameter(ParameterSetName='Setup', Mandatory=$true)]
[Switch]$Setup,    # Install the service
```

- An if block, as shown in **Figure 8**, for processing the -Setup switch in the main routine at the end of the script.

Figure 8 Setup Code Handler

XML

```
if ($Setup) {
  # Install the service
  # Check if it's necessary (if not installed,
  # or if this script is newer than the installed copy).
  [...] # If necessary and already installed, uninstall the old copy.
  # Copy the service script into the installation directory.
  if ($ScriptFullName -ne $scriptCopy) {
    Copy-Item $ScriptFullName $scriptCopy
  }
  # Generate the service .EXE from the C# source embedded in this script.
  try {
    Add-Type -TypeDefinition $source -Language CSharp -OutputAssembly $exeFullName
      -OutputType ConsoleApplication -ReferencedAssemblies "System.ServiceProcess"
  } catch {
    $msg = $_.Exception.Message
    Write-error "Failed to create the $exeFullName service stub. $msg"
    exit 1
  }
  # Register the service
  $pss = New-Service $serviceName $exeFullName -DisplayName $serviceDisplayName
    -StartupType Automatic
  return
}
```

Startup

The authority responsible for managing services is the SCM. Every startup operation must go through the SCM so it can keep track of service states. So even if the user wants to manually initiate a startup using the service script, that startup must be done through a request to the SCM. In this case, the sequence of operations is:

1. The user (an administrator) runs a first instance: PSService.ps1 -Start.
2. This first instance tells the SCM to start the service: Start-Service $serviceName.
3. The SCM runs PSService.exe. Its Main routine creates a service object, then invokes its Run method.
4. The SCM invokes the service object OnStart method.
5. The C# OnStart method runs a second script instance: PSService.ps1 -Start.
6. This second instance, now running in the background as the system user, starts a third instance, which will remain in memory as the actual service: PSService.ps1 -Service. It’s this last -Service instance that does the actual service task, which you customize for whatever task is desired.

In the end, there will be two tasks running: PSService.exe, and a PowerShell.exe instance running PSService.ps1 -Service.

All this is implemented by having three pieces of code in the script:

- The definition of the -Start switch in the Param block at the beginning of the script:

XML

```
[Parameter(ParameterSetName='Start', Mandatory=$true)]
[Switch]$Start, # Start the service
```

- In the main routine, at the end of the script, an if block processing the -Start switch:

XML

```
if ($Start) {# Start the service
  if ($isSystem) { # If running as SYSTEM, ie. invoked as a service
    Start-Process PowerShell.exe -ArgumentList (
      "-c & '$scriptFullName' -Service")
  } else { # Invoked manually by the administrator
  Start-Service $serviceName # Ask Service Control Manager to start it
  }
  return
}
```

- In the C# source snippet, a Main routine and a handler for the OnStart method that runs PSService.ps1 -Start, as shown in **Figure 9**.

Figure 9 Start Code Handler

C#

```
public static void Main() {
  System.ServiceProcess.ServiceBase.Run(new $serviceName());
}
protected override void OnStart(string [] args) {
  // Start a child process with another copy of this script.
  try {
    Process p = new Process();
    // Redirect the output stream of the child process.
    p.StartInfo.UseShellExecute = false;
    p.StartInfo.RedirectStandardOutput = true;
    p.StartInfo.FileName = "PowerShell.exe";
    p.StartInfo.Arguments = "-c & '$scriptCopyCname' -Start";
    p.Start();
    // Read the output stream first and then wait. (Supposed to avoid deadlocks.)
    string output = p.StandardOutput.ReadToEnd();
    // Wait for the completion of the script startup code,     // which launches the -Service instance.
    p.WaitForExit();
  } catch (Exception e) {
    // Log the failure.
  }
}
```

Getting the Service State

The -Status handler simply asks the SCM for the service status, and sends it to the output pipe:

XML

```
try {
  $pss = Get-Service $serviceName -ea stop # Will error-out if not installed.
} catch {
  "Not Installed"
  return
}
$pss.Status
```

But during the debugging phase, you might encounter script failures, due, for example, to syntax errors in the script and the like. In such cases, the SCM status might end up being incorrect. I've actually run into this several times while preparing this article. To help diagnose that kind of thing, it's prudent to double-check and search for -Service instances:

XML

```
$spid = $null
$processes = @(gwmi Win32_Process -filter "Name = 'powershell.exe'" | where {
  $_.CommandLine -match ".*$scriptCopyCname.*-Service"
})
foreach ($process in $processes) { # Normally there is only one.
  $spid = $process.ProcessId
  Write-Verbose "$serviceName Process ID = $spid"
}
if (($pss.Status -eq "Running") -and (!$spid)) {
# This happened during the debugging phase.
```

```
    Write-Error "The Service Control Manager thinks $serviceName is started,
      but $serviceName.ps1 -Service is not running."
    exit 1
  }
```

Stop and Uninstallation

The Stop and Remove operations basically undo what Setup and Start did:

- -Stop (if invoked by the user) tells the SCM to stop the service.
- If invoked by the system, it simply kills the PSService.ps1 -Service instance.
- -Remove stops the service, unregisters it using sc.exe delete $serviceName, then deletes the files in the installation directory.

The implementation is also very similar to that of the Setup and Start:

1. The definition of each switch in the Param block at the beginning of the script.
2. An if block processing the switch in the main routine, at the end of the script.
3. For the stop operation, in the C# source snippet, a handler for the OnStop method that runs PSService.ps1 -Stop. The stop operation does things differently depending on whether the user is a real user or the system.

Event Logging

Services run in the background, without a UI. This makes them difficult to debug: How can you diagnose what went wrong, when by design nothing is visible? The usual method is to keep a record of all error messages with time stamps, and also to log important events that went well, such as state transitions.

The sample PSService.ps1 script implements two distinct logging methods, and uses both at strategic points (including in parts of the previous code extracts, removed here to clarify the basic operation):

- It writes event objects into the Application log, with the service name as the source name, as shown in **Figure 10**. These event objects are visible in the Event Viewer, and can be filtered and searched using all the capabilities of that tool. You can also get these entries with the Get-Eventlog cmdlet:

Event Viewer with PSService Events

Figure 10 Event Viewer with PSService Events

XML

```
Get-Eventlog -LogName Application -Source PSService | select -First 10
```

- It writes message lines to a text file in the Windows Logs directory, ${ENV:windir}\Logs\$serviceName.log, as shown in **Figure 11**. This log file is readable with Notepad, and can be searched using findstr.exe, or Win32 ports of grep, tail and so forth.

Figure 11 Sample Log File

XML

```
PS C:\Temp> type C:\Windows\Logs\PSService.log
2016-01-02 15:29:47 JFLZB\Larvoire C:\SRC\PowerShell\SRC\PSService.ps1 -Status
2016-01-02 15:30:38 JFLZB\Larvoire C:\SRC\PowerShell\SRC\PSService.ps1 -Setup
2016-01-02 15:30:42 JFLZB\Larvoire PSService.ps1 -Status
2016-01-02 15:31:13 JFLZB\Larvoire PSService.ps1 -Start
2016-01-02 15:31:15 NT AUTHORITY\SYSTEM & 'C:\WINDOWS\System32\PSService.ps1' -Start
2016-01-02 15:31:15 NT AUTHORITY\SYSTEM PSService.ps1 -Start: Starting script
'C:\WINDOWS\System32\PSService.ps1' -Service
2016-01-02 15:31:15 NT AUTHORITY\SYSTEM & 'C:\WINDOWS\System32\PSService.ps1' -Service
2016-01-02 15:31:15 NT AUTHORITY\SYSTEM PSService.ps1 -Service # Beginning background job
2016-01-02 15:31:25 NT AUTHORITY\SYSTEM PSService -Service # Awaken after 10s
2016-01-02 15:31:36 NT AUTHORITY\SYSTEM PSService -Service # Awaken after 10s
2016-01-02 15:31:46 NT AUTHORITY\SYSTEM PSService -Service # Awaken after 10s
2016-01-02 15:31:54 JFLZB\Larvoire PSService.ps1 -Stop
2016-01-02 15:31:55 NT AUTHORITY\SYSTEM & 'C:\WINDOWS\System32\PSService.ps1' -Stop
```

```
2016-01-02 15:31:55 NT AUTHORITY\SYSTEM PSService.ps1 -Stop: Stopping script PSService.ps1 -Service
2016-01-02 15:31:55 NT AUTHORITY\SYSTEM Stopping PID 34164
2016-01-02 15:32:01 JFLZB\Larvoire PSService.ps1 -Remove
PS C:\Temp>
```

A Log function makes it easy to write such messages, automatically prefixing the ISO 8601 time stamp and current user name:

XML

```
Function Log ([String]$string) {
  if (!(Test-Path $logDir)) {
    mkdir $logDir
  }
  "$(Now) $userName $string" |
    out-file -Encoding ASCII -append "$logDir\$serviceName.log"
}
```

Sample Test Session

Here's how the preceding logs were generated:

XML

```
PS C:\Temp> C:\SRC\PowerShell\SRC\PSService.ps1 -Status
Not Installed
PS C:\Temp> PSService.ps1 -Status
PSService.ps1 : The term 'PSService.ps1' is not recognized as the name of a cmdlet, function, script file, or
operable program.
[...]
PS C:\Temp> C:\SRC\PowerShell\SRC\PSService.ps1 -Setup
PS C:\Temp> PSService.ps1 -Status
Stopped
```

```
PS C:\Temp> PSService.ps1 -Start
PS C:\Temp>
```

This shows how to use the service in general. Keep in mind that it must be run by a user with local admin rights, in a Windows PowerShell session running as Administrator. Notice how the PSService.ps1 script wasn't on the path at first, then after the -Setup operation it is. (The first -Status call with no path specified fails; the second -Status call succeeds.)

XML

```
Calling PSService.ps1 -Status at this stage would produce this output: Running. And this, after waiting 30
seconds:
PS C:\Temp> PSService.ps1 -Stop
PS C:\Temp> PSService.ps1 -Remove
PS C:\Temp>
```

Customizing the Service

To create your own service, just do the following:

- Copy the sample service into a new file with a new base name, such as C:\Temp\MyService.ps1.
- Change the long service name in the global variables section.
- Change the TO DO block in the -Service handler at the end of the script. Currently, the while ($true) block just contains dummy code that wakes up every 10 seconds and logs one message in the log file:

XML

```
######### TO DO: Implement your own service code here. ##########
###### Example that wakes up and logs a line every 10 sec: ######
Start-Sleep 10
Log "$script -Service # Awaken after 10s"
```

- Install and start testing:

XML

```
C:\Temp\MyService.ps1 -Setup
MyService.ps1 -Start
type C:\Windows\Logs\MyService.log
```

You shouldn't have to change anything in the rest of the script, except to add support for new SCM features like the Paused state.

Limitations and Issues

The service script must be run in a shell running with administrator rights or you'll get various access denied errors.

The sample script works in Windows versions XP to 10, and the corresponding server versions. In Windows XP, you have to install Windows PowerShell v2, which isn't available by default. Download and install Windows Management Framework v2 for XP (bit.ly/1MpOdpV), which includes Windows PowerShell v2. Note that I've done very little testing in that OS, as it's not supported anymore.

On many systems, the Windows PowerShell script execution is disabled by default. If you get an error like, "the execution of scripts is disabled on this system," when trying to run PSService.ps1, then use:

XML

```
Set-ExecutionPolicy RemoteSigned
```

For more information, see "References."

Obviously, a service script like this can't be as performant as a compiled program. A service script written in Windows PowerShell will be good for prototyping a concept, and for tasks with low performance costs like system monitoring, service clustering and so forth. But for any high performance task, a rewrite in C++ or C# is recommended.

The memory footprint is also not as good as that of a compiled program, because it requires loading a full-fledged Windows PowerShell interpreter in the System session. In today's world, with systems having many gigabytes of RAM, this is not a big deal.

This script is completely unrelated to Mark Russinovich's PsService.exe. I chose the PSService.ps1 name before I knew about the homonymy. I'll keep it for this sample script as I think the name makes its purpose clear. Of course, if you plan to experiment with your own Windows PowerShell service, you must rename it, to get a unique service name from a unique script base name!

References

- Introduction to Windows Service Applications (bit.ly/1UOBJJY)
- How to: Create Windows Services (bit.ly/1VJCnJo)
- ServiceBase Class (bit.ly/1UOC13y .aspx))
- Managing Services (bit.ly/1VJCZyG)
- How to: Debug Windows Service Applications (bit.ly/1RjEhPg)

Jean-François Larvoire *works for Hewlett-Packard Enterprise in Grenoble, France. He has been developing software for 30 years for PC BIOS, Windows drivers, Windows and Linux system management. He can be reached at jf.larvoire@hpe.com.*

Thanks to the following technical expert for reviewing this article: Jeffery Hicks (JDH IT Solutions)

Discuss this article in the MSDN Magazine forum