Powershell: Building a Module, one microstep at a time

Posted on May 27, 2017

I am really quick to build a module out of my scripts and functions. I like how it allows me to organize my functions and use them in other scripts. I also see that many PowerShell scripters are slow to take that step of building a module. We often learn how to build a module without really understanding why they are built that way.

In this post, we will turn a common script scenario into a full module one step at a time. We will take several microsteps to show all the subtle details of why common modules are built the way they are. Lets take the mystery out of building a module to see how simple they can be.

Index

Starting with functions

There is a natural progression when working with PowerShell. You start with small script and work into making much larger ones. When your scripts get large, you start using more functions. These functions could be from someone else or functions that you write yourself. These functions start collecting at the top of your script.

```
    function Get-Info
    {
        param($ComputerName)
        Get-WmiObject -ComputerName $ComputerName -Class Win32_BIOS
    }

    Get-Info -ComputerName localhost
```

You scripts are still long because all these functions are still taking up space in your script. This is the common scenario that we are going to build on. I have one function here but having several of them is common.

Dot sourcing

From here, we would move the functions out into their own `.ps1` file. So if you saved that function into a file called `GetInfo.ps1`, then you could dot source it and call it like this.

```
. .\GetInfo.ps1
GetInfo -ComputerName localhost
```

Dot sourcing is a way to load a script into your current runspace. This function now becomes available to you or the calling script. Leaving out that period would run the script without leaving that defined function available in your script.

You can place multiple functions in that file and treat it like a library of functions. This is the basic idea of a module.

Import-Module

All it takes to turn your function script into a module is the use of `Import-Module` to import it. Here is how that works.

```
Import-Module .\GetInfo.ps1
GetInfo -ComputerName localhost
```

I like this so much more than dot sourcing. I really wish that this was the standard approach over dot sourcing for two reasons. First is that it would be easier to understand and explain to people new to PowerShell. Second, it moves the scripter down the path of making modules much sooner.

.psm1

We can call `Import-Module` on a `.ps1`, but the convention is to use `.psm1` as a module designation. Rename that script to `GetInfo.psm1`.

```
Import-Module .\GetInfo.psm1
GetInfo -ComputerName localhost
```

Now we can say we have a module.

Export-ModuleMember

Sometimes you may have utility functions in your module that should stay internal to the module and not be made available to other scripts. If you want to have public and internal functions, you will need to use `Export-ModuleMember` in the `*.psm1` file to define the exported public functions.

```
function GetInfo{
    param($ComputerName)
    Get-WmiObject -ComputerName $ComputerName -Class Win32_BIOS
}
Export-ModuleMember -Function 'GetInfo'
```

If you don't call `Export-ModuleMember` to specify exactly what you want exported, then everything is exported from this `.psm1` file.

I will show an alternate way to do this when we talk about the module manifest.

Folder names

Then next thing I want to do is place our module into its own folder. The convention here is that the name of the folder matches the name of the `.psm1` file. Our file structure should look something like this now.

```
Scripts
│   myscript.ps1
│
└───GetInfo
        GetInfo.psm1
```

Then we update our `myscript.ps1` file to import that folder.

```
Import-Module .\GetInfo
GetInfo -ComputerName localhost
```

Import-Module will automatically find our `.psm1` file inside that folder. From here on out, that whole folder and all of it's contents will be our module.

$ENV:PSModulePath

There is an environment variable called `$ENV:PSModulePath`. If we look at it, we will see all the locations where we can import a module (by module name instead of by path).

```
PS:> $env:PSModulePath -split ';'

C:\Users\username\Documents\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules
```

This is normally a single string but I split it up so we could read it. Sometimes you will install something that will add a path to that list. If we place our folder inside one of those two locations, we can import our module by name.

So we need to move our module to `C:\Users\$env:username\Documents\WindowsPowerShell\Modules`

```
Move-Item .\GetInfo "C:\Users\$env:username\Documents\WindowsPowerShell\Modu
```

Then our module should show up when we list available modules.

```
Get-Module -ListAvailable
```

And we can import it by name in our script.

```
Import-Module GetInfo
GetInfo -ComputerName localhost
```

Module manifest

We really shouldn't call our module done until we add a module manifest. This adds metadata about your module. It includes author information and versioning. It also will enable PowerShell to auto-load our module if we create it correctly.

The module manifest is just a hashtable saved as a `*.psd1` file. The name of the file should match the name of the folder. By creating this file, it will get loaded when you call `Import-Module`.

New-ModuleManifest

The good news is that we have a `New-ModuleManifest` cmdlet that will create the manifest for us.

```
$manifest = @{
    Path              = '.\GetInfo\GetInfo.psd1'
    RootModule        = 'GetInfo.psm1'
    Author            = 'Kevin Marquette'
}
New-ModuleManifest @manifest
```

Because our manifest is loaded instead of the `*.psm1`, we need to use the `RootModule` parameter to indicate what PowerShell module file should be ran. This is the structure that we have now.

```
Modules
|
└───GetInfo
        GetInfo.psd1
        GetInfo.psm1
```

Here is a clip of the manifest that we just generated.

```
# Module manifest for module 'GetInfo'
# Generated by: Kevin Marquette
# Generated on: 5/21/2017

@{

# Script module or binary module file associated with this manifest.
RootModule = 'GetInfo.psm1'

# Version number of this module.
ModuleVersion = '1.0'

# ID used to uniquely identify this module
GUID = 'dadea276-ae04-4c01-b901-06838167ec7c'

# Author of this module
Author = 'Kevin Marquette'

# Company or vendor of this module
CompanyName = 'Unknown'

# Copyright statement for this module
Copyright = '(c) 2017 Kevin Marquette. All rights reserved.'

# Description of the functionality provided by this module
# Description = ''

# Functions to export from this module, for best performance, do not use wild
FunctionsToExport = '*'

...

}
```

The `New-ModuleManifest` created all those keys and comments for us.

FunctionsToExport

One of the properties in the module manifest is the `FunctionsToExport` property with a default value of `*`. By default, it will export all functions. We want to update that value to have all our public functions in it.

```
FunctionsToExport = "GetInfo"
```

Use an array if you need to list multiple functions.

Using this property in the manifest is just like using `Export-ModuleMember`. I do need to mention that you don't need to use both `Export-ModuleMember` and `FunctionsToExport`. You only need to use one of those to export your functions. If you have a manifest, then you should be using `FunctionsToExport`.

> *In Powershell 6.0, the `FunctionsToExport` default is changing to an empty array `@()`. This is more in-line with best practices. You should either specify the functions to export or set the value to an empty array.*

Module autoloading

One nice feature of having a module manifest with `FunctionsToExport` defined, is that Powershell can auto import your module if you call one of the exported functions. Your module still has to be in the `$ENV:PSModulePath` variable for this to work.

This is why it is important to populate the `FunctionsToExport`. The default value for this is `*` to designate that it is exporting all functions defined in the module. This does work, but the auto import functionality depends on this value. This is often overlooked by a lot of module builders.

$PSModuleAutoloadingPreference

Module auto-loading was introduced in PowerShell 3.0. We were also given `$PSModuleAutoloadingPreference` to control that behavior. If you want to disable module auto-loading for all modules, then set this value to `none`.

```
$PSModuleAutoloadingPreference = 'none'
```

Unless your doing module development, you would generally leave this variable alone.

#Requires -Modules

When you have a script that requires a module, you can add a requires statement (https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_requires) to the top of your script. This will require that the specified module is loaded before your script runs. If the module is installed and if auto-loading is allowed, the requires statement will go ahead and import the module.

```
#Requires -Modules GetInfo
GetInfo -ComputerName localhost
```

This is what our script should end up like if we have a properly crafted module in the correct location.

Putting it all together

Now we know how to build a module layer by layer, I would generally build a module this way.

- Create a folder named `MyModule`
- Create a file called `MyModule.psm1` in that folder to hold your functions

- Use `New-ModuleManifest` to create a `MyModule.psd1` in that folder for the metadata
- Update the `ModuleRoot` and `FunctionsToExport` properties in the `MyModule.psd1`

Start with a library or utility module for your common functions. As your collection of functions grows, then you can break them out into their own modules later.

What's next?

We all start with simple modules like this and I wanted to lay the groundwork for my next post. I will cover a specific design pattern for modules that you see in a lot of community projects in that post. I quickly cover this in my post on building a CI/CD pipeline (/2017-01-21-powershell-module-continious-delivery-pipeline/?utm_source=blog&utm_medium=blog&utm_content=body&utm_content=module).

If you are looking for a way to distribute your module to others on your team, consider creating an internal script repository (/2017-05-30-Powershell-your-first-PSScript-repository/?utm_source=blog&utm_medium=blog&utm_content=module). It is easier than you would expect.

Tags: PowerShell (/tags/#PowerShell) Basics (/tags/#Basics) Modules (/tags/#Modules)

(htt (ht

← PREVIOUS POST (/2017-05-18-POWERSHELL-TYPEEXTENSION-DSL-PART-5/)

NEXT POST → (/2017-05-30-POWERSHELL-YOUR-FIRST-PSSCRIPT-REPOSITORY/)

Kevin Marquette

● (https://github.com/kevinmarquette) ● (https://twitter.com/kevinmarquette)

● (https://linkedin.com/in/kevinmarquette) ● (/feed.xml)

Microsoft® Most Valuable Professional (https://mvp.microsoft.com/en-us/PublicProfile/5003187)

PLANET POWERSHELL Featured Community Blog (http://www.planetpowershell.com/)

Top 5 PowerShell 2018 (top-50-powershell-bloggers-of-2018)

AWARDED TOP 50 POWERSHELL BLOG (http://blog.feedspot.com/powershell_blogs/)