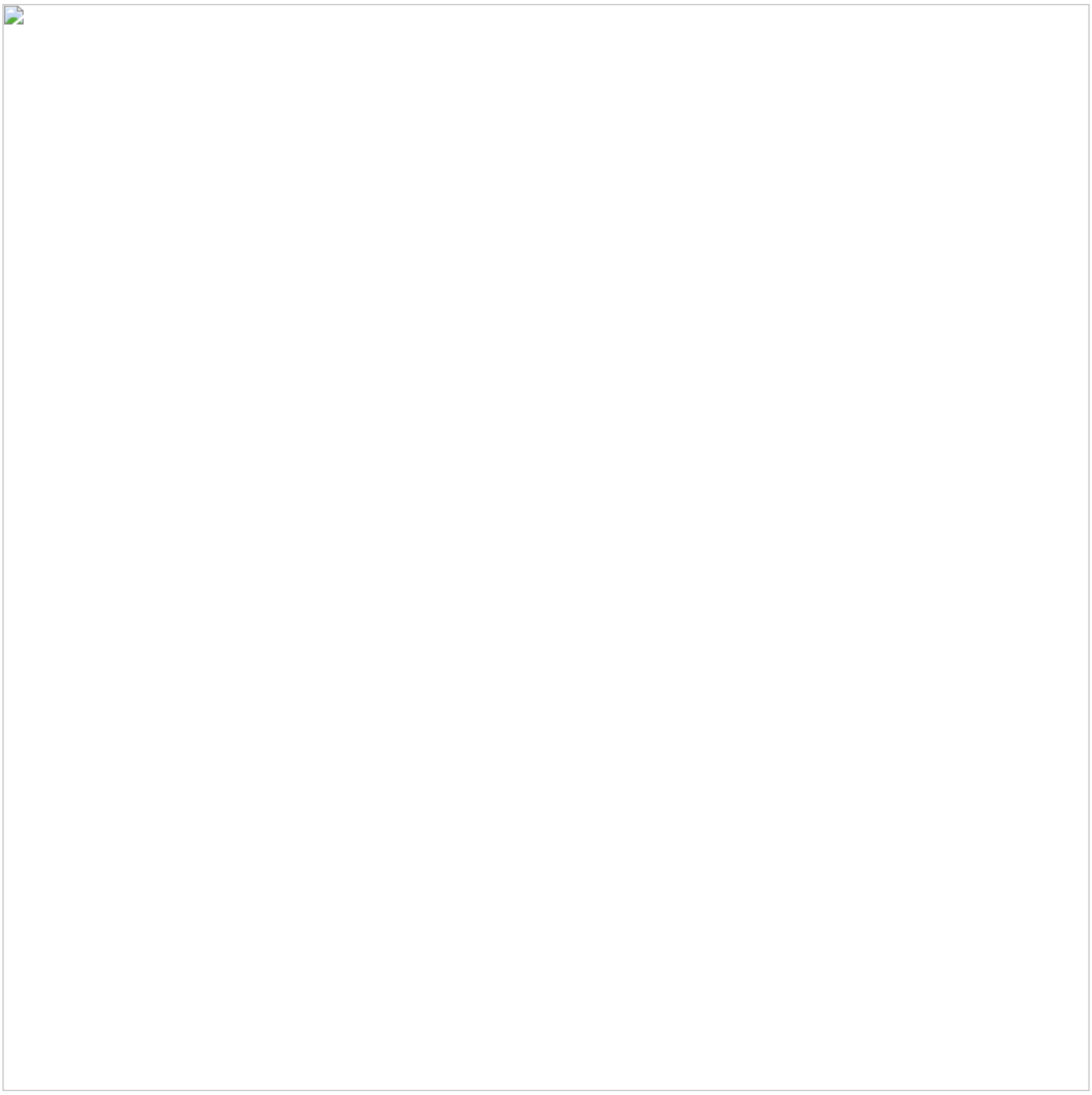
Powershell: Publishing community modules to an internal PSRepository

powershellexplained.com/2018-03-06-Powershell-Managing-community-modules



I touched base on the idea of Using a NuGet server for a PSRepository a few days ago. This is the ideal way to distribute modules to other systems. This can include community modules. Today, I am going to cover the process that I use to republish modules from the PSGallery to an internal PSRepository.

I'm already using the internal repository for my modules and I like the idea of having all my modules published to one place.

There are several reasons to use an internal repository. The primary one is when your servers or systems don't have access to the external internet. Another reason is to control the version of the modules in use across your teams and servers. My primary reason is so I can test them before they get introduced into my environment.

Index

Getting Started

Here is the plan. We write a script to download the modules that we care about. Run some tests. Then publish to our NuGet server. You could do this by hand fairly quickly, but modules get updated all the time. If you don't automate this, you are not going to update them often.

The list

We need to have a way to define the list of modules that we will manage. I use a JSON document called `config.json` that looks like this:

```
{
    "Modules":[
        {
            "Name":"Pester",
            "RequiredVersion":"4.3.1"
        },
        {
            "Name":"PSScriptAnalyzer"
        },
        {
            "Name":"Plaster"
        }
    ],
    "TestOrder":[
        "MyProject1",
        "MyProject2"
    ]
}
```

The first section is the list of modules to republish. The plan is that we would update every module on that list when our process runs. We can lock a module to a specific version with the `RequiredVersion` property if that is what we need.

The last section is how I define what projects to test and in what order to test them. I'll explain why this is important in a bit.

Downloading modules

My script will walk the module list and execute `Save-Module` for each one. In this example, I am saving them to a `downloads` folder.

```
$config = Get-Content -Path .\config.json | ConvertFrom-Json
foreach ( $module in $config.Modules )
{
    $saveParam = @{
        Name = $module.Name
        Path = '.\downloads'
        Repository = 'PSGallery'
    }

    if ( $null -ne $module.RequiredVersion )
    {
        $saveParam.RequiredVersion = $module.RequiredVersion
    }

    Save-Module @saveParam
}
```

At the moment, I download the entire list of modules even if I don't need to update them. The reason I do this is that I need them available for testing. After this runs, we have a local copy of each module.

```
PS> Get-ChildItem .\downloads\

Directory: .\downloads

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         3/5/2018   4:54 PM                Pester
d-----         3/5/2018   4:54 PM                Plaster
d-----         3/5/2018   4:54 PM                PSScriptAnalyzer
```

Import the modules

After I download the module, I import them in the same order.

```
foreach( $module in $config.Modules )
{
    $path = Join-Path .\downloads $module.Name
    Import-Module $path -Force
}
```

This is the first sanity check on these modules. You should be able to call `Import-Module` on every module you download. You may find that a newer version of one module breaks another. Or a module added an important dependency that you missed. Windows Defender was deleting files out of some important modules a few weeks ago. You never know what could go wrong. If a module will not import, then you don't want to be publishing it to your repository.

Test all the things

This is the point where I test all my projects that depend on these modules. All of my modules have a `build.ps1` script that processes my project and runs the Pester tests. This step looks something like this:

```
foreach($project in $config.TestOrder)
{
    $repo = 'https://github.com/KevinMarquette/{0}.git' -f $project
    $buildScript = '{0}\build.ps1' -f $project
    git clone $repo
    & $buildScript -Task 'DependencyTest'
}
```

I get a local copy of the project and execute the `build.ps1` script. I pass `DependencyTest` to my build script so the build script knows that it is running as part of this process. I use that parameter to suppress any module management that my build script may be doing (we already loaded the modules we want it to use). I run all my tests, but this parameter could define what tests need to run if you cannot run them all.

You don't have to have a build script for this to work. Calling `Invoke-Pester` on each project could be all that you need.

Why this is important

The reason I rebuild everything and run all my tests is to know if any module will cause me issues before I introduce it into my environment. I want to catch this failure here before someone else discovers it on their machine, or after they update modules on a server.

I run the full build because my build process depends on a lot of community modules. My built and published module may be fine, but I still want the build to work without issue on every system. It's easy to control versioning on the CI/CD pipeline, but this allows me to detect these problems early.

When the tests fail, I get to intervene and root cause the issue before anything actually breaks. I can pin the version of the module in the `config.json` while I remediate the tests or the project that does not work with the updated module.

Having the opportunity to run these tests is the whole reason I built this pipeline. I already have the tests and this will maximize their value.

Publish the modules

If everything checks out, then we can publish the modules to our internal repository.

```
foreach($module in $config.Modules)
{
    $path = '.\downloads\{0}\*\{0}.psd1' -f $module.Name

    $publishParam = @{
        Path        = $path
        Repository  = 'MyRepository'
        NuGetApiKey = $apiKey
        Force       = $true
    }
    Publish-Module @publishParam
}
```

The `$apiKey` is defined when you set up your NuGet server feed for secured publishing.

NuGet/PowerShellGet issues

I wish everything worked as easily as my post implies that it should. There are 2 big issues that are unresolved in the published versions of these tools. I learned them the hard way.

Publish-Module -Force

`Publish-Module` uses string comparisons instead of version comparisons when publishing modules. So if you publish version `1.3.9` and then try to publish `1.3.12`, the publish will fail. Thankfully you can ignore the version check with `-Force`. This is already resolved in the PowerShellGet source.

PowerShellGet #217

NuGet version normalization

At some point, NuGet started normalizing version numbers. When you go to publish a module, NuGet may decide that it does not like the version number that the module uses and will publish with whatever version it decides is best for that module.

If you decide to republish `PackageManagement` version `1.1.7.0`, NuGet decides that the trailing zero should not be there and removes it. It gets published with version `1.1.7` instead. The published version in your gallery will not match the version of the module in the PSGallery. This makes it hard to compare the two galleries for drift.

This issue is unresolved for `PowerShellGet 1.6.0` and `NuGet 4.4.1.4656`.

But all is not lost, there is a workaround. You can downgrade your NuGet version to `NuGet 2.8.60717.93` (https://www.NuGet.org/NuGet.exe) and it will work. This is one of the last versions of NuGet before they started changing version numbers.

Closing thoughts

It would have been a lot easier to pull all modules directly from the PSGallery all the time. But once I started publishing my own modules after testing them, I liked the idea of doing the same for all the community modules that we now depend on.

By testing them in this way, my servers can pull latest from my internal repository with DSC. My team is already doing frequent module updates because of our own internal modules. I'm now delivering community modules through that same update pipeline. It made it easier for us to consume community modules. It also keeps us all updated, all on the same versions, and more importantly all on verified compatible versions of community modules.

Tags: PowerShell Modules Build

- ← Previous Post
- Next Post →

Kevin Marquette