Portable Modules

01/10/20207 minutes to read

In this article

Windows PowerShell is written for .NET Frameworkwhile PowerShell Core is written for .NET Core. Portable modules are modules that work in both Windows PowerShell and PowerShell Core. While .NET Framework and .NET Core are highly compatible, there are differences in the available APIs between the two. There are also differences in the APIs available in Windows PowerShell and PowerShell Core. Modules intended to be used in both environments need to be aware of these differences.

Porting an Existing Module

Porting a PSSnapIn

PowerShell SnapIns aren't supported in PowerShell Core. However, it's trivial to convert a PSSnapIn to a PowerShell module. Typically, the PSSnapIn registration code is in a single source file of a class that derives from PSSnapIn. Remove this source file from the build; it's no longer needed.

Use New-ModuleManifest to create a new module manifest that replaces the need for the PSSnapIn registration code. Some of the values from the **PSSnapIn** (such as **Description**) can be reused within the module manifest.

The **RootModule** property in the module manifest should be set to the name of the assembly (dll) implementing the cmdlets.

The .NET Portability Analyzer (aka APIPort)

To port modules written for Windows PowerShell to work with PowerShell Core, start with the .NET Portability Analyzer. Run this tool against your compiled assembly to determine if the .NET APIs used in the module are compatible with .NET Framework, .NET Core, and other .NET runtimes. The tool suggests alternate APIs if they exist. Otherwise, you may need to add runtime checks and restrict capabilities not available in specific runtimes.

Creating a New Module

If creating a new module, the recommendation is to use the .NET CLI.

Installing the PowerShell Standard Module Template

Once the .NET CLI is installed, install a template library to generate a simple PowerShell module. The module will be compatible with Windows PowerShell, PowerShell Core, Windows, Linux, and macOS.

The following example shows how to install the template:

PowerShell Copy

```
dotnet new -i
Microsoft.PowerShell.Standard.Module.Template
```

output

Copy

```
  Restoring packages for C:\Users\Steve\.tem-
plateengine\dotnetcli\v2.1.302\scratch\restor
e.csproj...
  Installing Microsoft.PowerShell.Standard.-
Module.Template 0.1.3.
  Generating MSBuild file
C:\Users\Steve\.templateengine\dotnetcli\v2.1
.302\scratch\obj\restore.cspro-
j.nuget.g.props.
  Generating MSBuild file
C:\Users\Steve\.templateengine\dotnetcli\v2.1
.302\scratch\obj\restore.csproj.nuget.g.tar-
gets.
  Restore completed in 1.66 sec for
C:\Users\Steve\.templateengine\dotnetcli\v2.1
.302\scratch\restore.csproj.

Usage: new [options]

Options:
  -h, --help          Displays help for this
command.
  -l, --list          Lists templates con-
taining the specified name. If no name is
specified, lists all templates.
  -n, --name          The name for the output
being created. If no name is specified, the
name of the current directory is used.
  -o, --output        Location to place the
generated output.
```

```
  -i, --install       Installs a source or a
template pack.
  -u, --uninstall     Uninstalls a source or
a template pack.
  --nuget-source      Specifies a NuGet
source to use during install.
  --type              Filters templates based
on available types. Predefined values are
"project", "item" or "other".
  --force             Forces content to be
generated even if it would change existing
files.
  -lang, --language   Filters templates based
on language and specifies the language of the
template to create.


Templates
Short Name         Language         Tags
---------------------------------------------
---------------------------------------------
----------------------------------
Console Application
console            [C#], F#, VB
Common/Console
Class library
classlib           [C#], F#, VB
Common/Library
PowerShell Standard Module
psmodule           [C#]
Library/PowerShell/Module
...
```

Creating a New Module Project

After the template is installed, you can create a new PowerShell module project using that template. In this example, the sample module is called 'myModule'.

Copy

```
PS> mkdir myModule

    Directory: C:\Users\Steve

Mode LastWriteTime Length Name
---- ------------- ------ ----
d----- 8/3/2018 2:41 PM myModule

PS> cd myModule
PS C:\Users\Steve\myModule> dotnet new psmod-
ule

The template "PowerShell Standard Module" was
created successfully.

Processing post-creation actions...
Running 'dotnet restore' on
C:\Users\Steve\myModule\myModule.csproj...
  Restoring packages for C:\Users\Steve\my-
Module\myModule.csproj...
  Installing PowerShellStandard.Library
5.1.0.
  Generating MSBuild file C:\Users\Steve\my-
Module\obj\myModule.csproj.nuget.g.props.
  Generating MSBuild file C:\Users\Steve\my-
Module\obj\myModule.csproj.nuget.g.targets.
  Restore completed in 1.76 sec for
C:\Users\Steve\myModule\myModule.csproj.

Restore succeeded.
```

Building the Module

Use standard .NET CLI commands to build the project.

PowerShell

Copy

```
dotnet build
```

output

Copy

```
PS C:\Users\Steve\myModule> dotnet build
Microsoft (R) Build Engine version
15.7.179.6572 for .NET Core
Copyright (C) Microsoft Corporation. All
rights reserved.

  Restore completed in 76.85 ms for
C:\Users\Steve\myModule\myModule.csproj.
  myModule -> C:\Users\Steve\myModule\bin\De-
bug\netstandard2.0\myModule.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:05.40
```

Testing the Module

After building the module, you can import it and execute the sample cmdlet.

PowerShell

Copy

```
ipmo .\bin\Debug\netstandard2.0\myModule.dll
Test-SampleCmdlet -?
Test-SampleCmdlet -FavoriteNumber 7 -Fa-
voritePet Cat
```

output

Copy

```
PS C:\Users\Steve\myModule> ipmo
.\bin\Debug\netstandard2.0\myModule.dll
PS C:\Users\Steve\myModule> Test-SampleCmdlet
-?

NAME
    Test-SampleCmdlet

SYNTAX
    Test-SampleCmdlet [-FavoriteNumber] <int>
[[-FavoritePet] {Cat | Dog | Horse}] [<Com-
monParameters>]


ALIASES
    None


REMARKS
    None


PS C:\Users\Steve\myModule> Test-SampleCmdlet
-FavoriteNumber 7 -FavoritePet Cat

FavoriteNumber FavoritePet
-------------- -----------
             7 Cat
```

The following sections describe in detail some of the technologies used by this template.

.NET Standard Library

.NET Standard is a formal specification of .NET APIs that are available in all .NET implementations. Managed code targeting .NET Standard works with the .NET Framework and .NET Core versions that are compatible with that version of the .NET Standard.

Note

Although an API may exist in .NET Standard, the API implementation in .NET Core may throw a `PlatformNotSupportedException` at runtime, so to verify compatibility with Windows PowerShell and PowerShell Core, the best practice is to run tests for your module within both environments. Also run tests on Linux and macOS if your module is intended to be cross-platform.

Targeting .NET Standard helps ensure that, as the module evolves, incompatible APIs don't accidentally get introduced into the module. Incompatibilities are discovered at compile time instead of runtime.

However, it isn't required to target .NET Standard for a module to work with both Windows PowerShell and PowerShell Core, as long as you use compatible APIs. The Intermediate Language (IL) is compatible between the two runtimes. You can target .NET Framework 4.6.1, which is compatible with .NET Standard 2.0. If you don't use APIs outside of .NET Standard 2.0, then your module works with PowerShell Core 6 without recompilation.

PowerShell Standard Library

The PowerShell Standard library is a formal specification of PowerShell APIs available in all PowerShell versions greater than or equal to the version of that standard.

For example, PowerShell Standard 5.1 is compatible with both Windows PowerShell 5.1 and PowerShell Core 6.0 or newer.

We recommend you compile your module using PowerShell Standard Library. The library ensures the APIs are available and implemented in both Windows PowerShell and PowerShell Core 6. PowerShell Standard is intended to always be forwards-compatible. A module built using PowerShell Standard Library 5.1 will always be compatible with future versions of PowerShell.

Module Manifest

Indicating Compatibility With Windows PowerShell and PowerShell Core

After validating that your module works with both Windows PowerShell and PowerShell Core, the module manifest should explicitly indicate compatibility by using the CompatiblePSEditions property. A value of `Desktop` means that the module is compatible with Windows PowerShell, while a value of `Core` means that the module is compatible with PowerShell Core. Including both `Desktop` and `Core` means that the module is compatible with both Windows PowerShell and PowerShell Core.

Note

`Core` does not automatically mean that the module is compatible with Windows, Linux, and macOS.

The **CompatiblePSEditions** property was introduced in PowerShell v5. Module manifests that use the **CompatiblePSEditions** property fail to load in versions prior to PowerShell v5.

Indicating OS Compatibility

First, validate that your module works on Linux and macOS. Next, indicate compatibility with those operating systems in the module manifest. This makes it easier for users to find your module for their operating system when published to the PowerShell Gallery.

Within the module manifest, the `PrivateData` property has a `PSData` sub-property. The optional `Tags` property of `PSData` takes an array of values that show up in PowerShell Gallery. The PowerShell Gallery supports the following compatibility values:

Tag	Description
PSEdition_Core	Compatible with PowerShell Core 6
PSEdition_Desktop	Compatible with Windows PowerShell
Windows	Compatible with Windows
Linux	Compatible with Linux (no specific distro)

Tag	Description
macOS	Compatible with macOS

Example:

PowerShell

Copy

```
@{
    GUID = "4ae9fd46-338a-459c-8186-
07f910774cb8"
    Author = "Microsoft Corporation"
    CompanyName = "Microsoft Corporation"
    Copyright = "(C) Microsoft Corporation.
All rights reserved."
    HelpInfoUri =
"https://go.microsoft.com/fwlink/?
linkid=855962"
    ModuleVersion = "1.2.4"
    PowerShellVersion = "3.0"
    ClrVersion = "4.0"
    RootModule = "PackageManagement.psm1"
    Description = 'PackageManagement (a.k.a.
OneGet) is a new way to discover and install
software packages from around the web.
 It is a manager or multiplexer of existing
package managers (also called package
providers) that unifies Windows package man-
agement with a single Windows PowerShell in-
terface. With PackageManagement, you can do
the following.
  - Manage a list of software repositories in
which packages can be searched, acquired and
installed
  - Discover software packages
  - Seamlessly install, uninstall, and inven-
tory packages from one or more software
repositories'

    CmdletsToExport = @(
        'Find-Package',
        'Get-Package',
        'Get-PackageProvider',
        'Get-PackageSource',
        'Install-Package',
        'Import-PackageProvider'
        'Find-PackageProvider'
        'Install-PackageProvider'
        'Register-PackageSource',
        'Set-PackageSource',
        'Unregister-PackageSource',
        'Uninstall-Package'
        'Save-Package'
    )

    FormatsToProcess  = @('PackageManagemen-
t.format.ps1xml')

    PrivateData = @{
        PSData = @{
            Tags = @('PackageManagement',
'PSEdition_Core', 'PSEdition_Desktop', 'Win-
dows', 'Linux', 'macOS')
            ProjectUri = 'https://oneget.org'
        }
    }
}
```

Dependency on Native Libraries

Modules intended for use across different operating systems or processor architectures may depend on a managed library that itself depends on some native libraries.

Prior to PowerShell 7, one would have to have custom code to load the appropriate native dll so that the managed library can find it correctly.

With PowerShell 7, native binaries to load are searched in sub-folders within the managed library's location following a subset of the .NET RID Catalog notation.

Copy

```
managed.dll folder
                |
                |--- 'win-x64' folder
                |       |--- native.dll
                |
                |--- 'win-x86' folder
                |       |--- native.dll
                |
                |--- 'win-arm' folder
                |       |--- native.dll
                |
                |--- 'win-arm64' folder
                |       |--- native.dll
                |
                |--- 'linux-x64' folder
                |       |--- native.so
                |
                |--- 'linux-x86' folder
                |       |--- native.so
                |
                |--- 'linux-arm' folder
                |       |--- native.so
                |
                |--- 'linux-arm64' folder
```

```
|        |--- native.so
|
|--- 'osx-x64' folder
|        |--- native.dylib
```