Learn   Discover ⌄   Product documentation ⌄   Development languages ⌄   Topics ⌄   🔍   Sign in

⊕   ✎   ⋮

Understanding a Windows PowerShell Module

Article • 09/18/2021 • **2 contributors**   👍 Feedback

In this article

A *module* is a set of related Windows PowerShell functionalities, grouped together as a convenient unit (usually saved in a single directory). By defining a set of related script files, assemblies, and

related resources as a module, you can reference, load, persist, and share your code much easier than you would otherwise.

The main purpose of a module is to allow the modularization (ie, reuse and abstraction) of Windows PowerShell code. For example, the most basic way of creating a module is to simply save a Windows PowerShell script as a .psm1 file. Doing so allows you to control (ie, make public or private) the functions and variables contained in the script. Saving the script as a .psm1 file also allows you to control the scope of certain variables. Finally, you can also use cmdlets such as Install-Module to organize, install, and use your script as building blocks for larger solutions.

Module Components and Types

A module is made up of four basic components:

1. Some sort of code file - usually either a PowerShell script or a managed cmdlet assembly.

2. Anything else that the above code file may need, such as additional assemblies, help files, or scripts.

3. A manifest file that describes the above files, as well as stores metadata such as author and versioning information.

4. A directory that contains all of the above content, and is located where PowerShell can reasonably find it.

> ⊙ **Note**
>
> none of these components, by themselves, are actually necessary. For example, a module can technically be only a script stored in a .psm1 file. You can also have a module that is nothing but a manifest file, which is used mainly for organizational purposes. You can also write a script that dynamically creates a module, and as such doesn't actually

need a directory to store anything in. The following sections describe the types of modules you can get by mixing and matching the different possible parts of a module together.

Script Modules

As the name implies, a *script module* is a file (`.psm1`) that contains any valid Windows PowerShell code. Script developers and administrators can use this type of module to create modules whose members include functions, variables, and more. At heart, a script module is simply a Windows PowerShell script with a different extension, which allows administrators to use import, export, and management functions on it.

In addition, you can use a manifest file to include other resources in your module, such as data files, other dependent modules, or runtime scripts. Manifest files are also useful for tracking metadata such as authoring and versioning information.

Finally, a script module, like any other module that isn't dynamically created, needs to be saved in a folder that PowerShell can reasonably discover. Usually, this is on the PowerShell module path; but if necessary you can explicitly describe where your module is installed. For more information, see How to Write a PowerShell Script Module.

Binary Modules

A **binary module** is a .NET Framework assembly (`.dll`) that contains compiled code, such as C#. Cmdlet developers can use this type of module to share cmdlets, providers, and more. (Existing snap-ins can also be used as binary modules.) Compared to a script module, a binary module allows you to create cmdlets that are faster or use features (such as multithreading) that are not as easy to code in Windows PowerShell scripts.

As with script modules, you can include a manifest file to describe additional resources that your module uses, and to track metadata about your module. Similarly, you probably should install your binary module in a folder somewhere along the PowerShell module path. For more information, see How to [How to Write a PowerShell Binary Module](#).

Manifest Modules

A **manifest module** is a module that uses a manifest file to describe all of its components, but doesn't have any sort of core assembly or script. (Formally, a manifest module leaves the `ModuleToProcess` or `RootModule` element of the manifest empty.) However, you can still use the other features of a module, such as the ability to load up dependent assemblies or automatically run certain pre-processing scripts. You can also use a manifest module as a convenient way to package up resources that other modules will use, such as nested modules, assemblies, types, or formats. For more information, see [How to Write a PowerShell Module Manifest](#).

Dynamic Modules

A **dynamic module** is a module that is not loaded from, or saved to, a file. Instead, they are created dynamically by a script, using the [New-Module](#) cmdlet. This type of module enables a script to create a module on demand that does not need to be loaded or saved to persistent storage. By its nature, a dynamic module is intended to be short-lived, and therefore cannot be accessed by the `Get-Module` cmdlet. Similarly, they usually do not need module manifests, nor do they likely need permanent folders to store their related assemblies.

Module Manifests

A **module manifest** is a `.psd1` file that contains a hash table. The keys and values in the hash table do the following things:

- Describe the contents and attributes of the module.

- Define the prerequisites.

- Determine how the components are processed.

  Manifests are not required for a module. Modules can reference script files (`.ps1`), script module files (`.psm1`), manifest files (`.psd1`), formatting and type files (`.ps1xml`), cmdlet and provider assemblies (`.dll`), resource files, Help files, localization files, or any other type of file or resource that is bundled as part of the module. For an internationalized script, the module folder also contains a set of message catalog files. If you add a manifest file to the module folder, you can reference the multiple files as a single unit by referencing the manifest.

  The manifest itself describes the following categories of information:

- Metadata about the module, such as the module version number, the author, and the description.

- Prerequisites needed to import the module, such as the Windows PowerShell version, the common language runtime (CLR) version, and the required modules.

- Processing directives, such as the scripts, formats, and types to process.

- Restrictions on the members of the module to export, such as the aliases, functions, variables, and cmdlets to export.

  For more information, see How to Write a PowerShell Module Manifest.

Storing and Installing a Module

Once you have created a script, binary, or manifest module, you can save your work in a location that others may access it. For example, your module can be stored in the system folder where Windows PowerShell is installed, or it can be stored in a user folder.

Generally speaking, you can determine where you should install your module by using one of the paths stored in the `$ENV:PSModulePath` variable. Using one of these paths means that PowerShell can automatically find and load your module when a user makes a call to it in their code. If you store your module somewhere else, you can explicitly let PowerShell know by passing in the location of your module as a parameter when you call `Install-Module`.

Regardless, the path of the folder is referred to as the **base** of the module (ModuleBase), and the name of the script, binary, or manifest module file should be the same as the module folder name, with the following exceptions:

- Dynamic modules that are created by the `New-Module` cmdlet can be named using the `Name` parameter of the cmdlet.

- Modules imported from assembly objects by the `Import-Module -Assembly` command are named according to the following syntax: `"dynamic_code_module_" + assembly.GetName()`.

  For more information, see Installing a PowerShell Module and about_PSModulePath.

Module Cmdlets and Variables

The following cmdlets and variables are provided by Windows PowerShell for the creation and management of modules.

New-Module cmdlet This cmdlet creates a new dynamic module that exists only in memory. The module is created from a script block, and its exported members, such as its functions and variables, are immediately available in the session and remain available until the session is closed.

New-ModuleManifest cmdlet This cmdlet creates a new module manifest (.psd1) file, populates its values, and saves the manifest file to the specified path. This cmdlet can also be used to create a module manifest template that can be filled in manually.

Import-Module cmdlet This cmdlet adds one or more modules to the current session.

Get-Module cmdlet This cmdlet retrieves information about the modules that have been or that can be imported into the current session.

Export-ModuleMember cmdlet This cmdlet specifies the module members (such as cmdlets, functions, variables, and aliases) that are exported from a script module (.psm1) file or from a dynamic module created by using the `New-Module` cmdlet.

Remove-Module cmdlet This cmdlet removes modules from the current session.

Test-ModuleManifest cmdlet This cmdlet verifies that a module manifest accurately describes the components of a module by verifying that the files that are listed in the module manifest file (.psd1) actually exist in the specified paths.

$PSScriptRoot This variable contains the directory from which the script module is being executed. It enables scripts to use the module path to access other resources.

$env:PSModulePath This environment variable contains a list of the directories in which Windows PowerShell modules are stored. Windows

PowerShell uses the value of this variable when importing modules automatically and updating Help topics for modules.

See Also

Writing a Windows PowerShell Module

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

🐛 Open a documentation issue

🗨 Provide product feedback

🌐 English (United States)   ☑✗ Your Privacy Choices   ☀ Theme ⌄

Manage cookies   Previous Versions   Blog⧉   Contribute   Privacy⧉   Terms of Use   Trademarks⧉

© Microsoft 2024