<http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package>

[NuGet Docs](http://docs.nuget.org/)

* [Home](http://docs.nuget.org/)
* [Quick View](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package)
* [Contribute](http://docs.nuget.org/docs/Contribute/Contributing-to-NuGet-Documentation)
* [NuGet Gallery](http://nuget.org/)
* [Install NuGet](http://visualstudiogallery.msdn.microsoft.com/27077b70-9dad-4c64-adcf-c7cf6bc9970c)

Top of Form

Bottom of Form

[Doc Outline](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package)

1. [Creating and Publishing a Package](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Creating_and_Publishing_a_Package)
   1. [Getting Started](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Getting_Started)
   2. [Share your Package](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Share_your_Package)
   3. [Installing NuGet.exe](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Installing_NuGet.exe)
   4. [Creating a Package](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Creating_a_Package)
      1. [From an assembly](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#From_an_assembly)
      2. [From a project](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#From_a_project)
      3. [From a convention based working directory](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#From_a_convention_based_working_directory)
         1. [Create the manifest](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Create_the_manifest)
      4. [Creating a solution-level package](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Creating_a_solution-level_package)
   5. [Publishing](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Publishing)
      1. [Create an account at NuGet.org](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Create_an_account_at_NuGet.org)
   6. [Package Conventions](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Package_Conventions)
   7. [Supporting Multiple .NET Framework Versions and Profiles](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Supporting_Multiple_.NET_Framework_Versions_and_Profiles)
      1. [Framework Version Folder Structure](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Framework_Version_Folder_Structure)
      2. [New in NuGet version 2.0 and above](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#New_in_NuGet_version_2.0_and_above)
      3. [Framework Names](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Framework_Names)
      4. [Assemblies that are not Specific to a Framework Version](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Assemblies_that_are_not_Specific_to_a_Framework_Version)
      5. [Matching Assembly Version to the Target Framework of a Project](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Matching_Assembly_Version_to_the_Target_Framework_of_a_Project)
      6. [Grouping Assemblies by Framework Version](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Grouping_Assemblies_by_Framework_Version)
      7. [Grouping Assemblies by Framework Profile](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Grouping_Assemblies_by_Framework_Profile)
      8. [Common Framework and Profile Targeting Examples](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Common_Framework_and_Profile_Targeting_Examples)
   8. [Deleting packages](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Deleting_packages)
   9. [Automatically Running PowerShell Scripts During Package Installation and Removal](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Automatically_Running_PowerShell_Scripts_During_Package_Installation_and_Removal)
   10. [Import MSBuild targets and props files into project (Requires NuGet 2.5 or above)](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Import_MSBuild_targets_and_props_files_into_project_(Requires_NuGet_2.5_or_above))
   11. [Automatically Displaying a Readme.txt File During Package Installation](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#Automatically_Displaying_a_Readme.txt_File_During_Package_Installation)

Creating and Publishing a Package

Getting Started

1. [Download NuGet.exe](http://nuget.codeplex.com/releases/view/58939)
2. Make sure NuGet.exe is in your path

If you’re more **GUI inclined**, use the [Package Explorer GUI to create packages](http://docs.nuget.org/docs/creating-packages/using-a-gui-to-build-packages).

Share your Package

Upgrade to the latest version of NuGet.exe ([*or download it here*](http://nuget.codeplex.com/releases/view/58939))

NuGet Update -self

Store your [API Key](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#api-key)

NuGet SetApiKey Your-API-Key

Build your NuGet Package

NuGet Pack YourPackage.nuspec

Publish your package

NuGet Push YourPackage.nupkg

Installing NuGet.exe

1. [Download NuGet.exe](http://nuget.codeplex.com/releases/view/58939)
2. Place NuGet in a well known location such as c:\utils on your machine
3. Make sure that NuGet.exe is in your path

To learn about the nuget commands, run nuget help or refer to the [NuGet.exe Command Line Reference](http://docs.nuget.org/docs/reference/command-line-reference).

Creating a Package

There are a few approaches to creating a package. Most packages are very simple and contain a single assembly. In those cases, there are some very easy ways to create packages. We’ll cover more interesting cases later.

From an assembly

If you have an assembly, you can easily generate a nuspec file from metadata within the assembly and create a package.

**nuget spec MyAssembly.dll**

This creates a Nuspec file. Edit the NuSpec file as needed and then

**nuget pack MyAssembly.nuspec**

From a project

For simple packages, creating a package from a csproj or vbproj file is a convenient way to create packages. For example, other packages that have been installed into your project will be referenced as dependencies when you create a package this way.

In the folder where the csproj file is, run:

nuget spec

This creates a special nuspec file with tokens meant to be replaced at pack time based on the project metadata. For example, $version$ gets replaced by the version specified in the AssemblyVersionAttribute applied to your assembly (typically within the AssemblyInfo.cs file).

The following is a list of the supported replacement tokens.

|  |  |
| --- | --- |
| **Token** | **Source** |
| $id$ | The Assembly name |
| $version$ | The assembly version as specified in the assembly’s AssemblyVersionAttribute. |
| $author$ | The company as specified in the AssemblyCompanyAttribute. |
| $description$ | The description as specified in the AssemblyDescriptionAttribute. |

You can then edit this nuspec file if you need to customize it. e.g. if you don't want token replacement for some fields, you hard code then in the nuspec instead.

Also, if you want to pull additional files into the package, you can use a node in the nuspec. e.g. suppose you want to add all the files from some arbitrary other folder into the package, you'd have:

<files>

<file src="..\..\SomeRoot\\*\*\\*.\*" target="" />

</files>

Once your nuspec is ready, you can run:

nuget pack MyProject.csproj

Note that you need to run 'nuget pack' on the project file, not the nuspec itself. But the nuspec will in fact get picked up.

If the project references other projects, you can add the referenced projects as part of the package, or as dependencies with -IncludeReferencedProjects option. This is done recursively. For example, suppose you have project A.csproj, which references B.csproj and C.csproj, while B.csproj references D.csproj & E.csproj, C.csproj references F.csproj & G.csproj. Then, when you run

nuget pack A.csproj -IncludeReferencedProjects

the generated package will contain files from projects B, C, D, E, F & G, in addition to files from project A.

If a referenced project has a corresponding nuspec file with the same name, then that referenced project is added as a dependency instead. Using the same example, suppose now there is file C.nuspec in the same directory as project file C.csproj. When you run

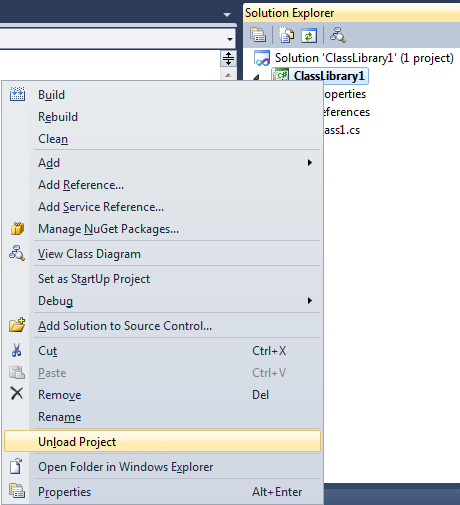
nuget pack A.csproj -IncludeReferencedProjects

the generated package will contain files from projects B, D, E, in addition to files from project A, and the package has dependency on C.

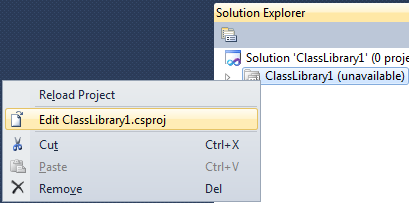
By default, NuGet will use the default build configuration set in the project file (typically Debug). To pack files from a different build configuration (e.g., Release), you can run:

nuget pack MyProject.csproj -Prop Configuration=Release

To change the project default for future packaging calls, modify the project file directly; Visual Studio does not offer a convenient method of altering this setting in the project properties GUI. To edit this file in Visual Studio, first unload it by right-clicking the project and choosing "Unload Project".



To edit the project file, right-click the unloaded project and choose "Edit {your-project-name}.{cs|vb|etc.}proj".



In typical project files, the first <Project><PropertyGroup> will contain a <Configuration> element that can be modified to choose your preferred build configuration (typically Release).

<Configuration Condition=" '$(Configuration)' == '' ">Release</Configuration>

You can also use the -Symbols flag to include a symbols package as well. A symbols package allows others to step into your package’s code in the debugger.

For a detailed walkthrough showing how to create and publish NuGet packages from a project, see [The easy way to publish NuGet packages with sources](http://blog.davidebbo.com/2011/04/easy-way-to-publish-nuget-packages-with.html) on David Ebbo’s blog.

From a convention based working directory

Some packages contain more than just assemblies. They may contain

1. Content and source code that should be injected into the target project.
2. [PowerShell scripts](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package#powershell) and executables.
3. [Configuration file and source code transformations](http://docs.nuget.org/docs/creating-packages/Configuration-File-and-Source-Code-Transformations).

To create a package in this way, you can layout a directory structure that follows the NuGet conventions.

* **tools** - The *tools* folder of a package is for [powershell scripts](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package" \l "powershell) and programs accessible from the Package Manager Console. After the folder is copied to the target project, it is added to the `$env:Path (PATH) environment variable.
* **lib** - Assemblies (*.dll* files) in the *lib* folder are added as assembly references when the package is installed.
* **content** - Files in the *content* folder are copied to the root of your application when the package is installed.

**Think of the *Content* folder as the root of your target application**. For example, if I want a package to add an image in the */images* directory of the target application, make sure to place the image in the *Content/images* folder of the package.

Create the manifest

To create a spec file from scratch, run the following command.

nuget spec

This will create an XML file with the .nuspec extension.

<?xml version="1.0"?>

<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">

<metadata>

<id>MyPackageId</id>

<version>1.0</version>

<authors>philha</authors>

<owners>philha</owners>

<licenseUrl>http://LICENSE\_URL\_HERE\_OR\_DELETE\_THIS\_LINE</licenseUrl>

<projectUrl>http://PROJECT\_URL\_HERE\_OR\_DELETE\_THIS\_LINE</projectUrl>

<iconUrl>http://ICON\_URL\_HERE\_OR\_DELETE\_THIS\_LINE</iconUrl>

<requireLicenseAcceptance>false</requireLicenseAcceptance>

<description>Package description</description>

<tags>Tag1 Tag2</tags>

<dependencies>

<dependency id="SampleDependency" version="1.0" />

</dependencies>

</metadata>

</package>

Edit this file to fill in the values appropriate for your package. Then create the folders needed by your package and copy the correct content in each folder.

mkdir lib

mkdir tools

mkdir content

mkdir content\controllers

copy ..\src\SomeController.cs content

copy ..\src\MyLibrary lib

copy ..\src\SomePowershellScript.ps1 tools

With the working directory in place with content, run the following command:

nuget pack YourPackage.nuspec

To create your package.

Creating a solution-level package

A solution-level package is one that installs a tool or additional commands for the Package Manager console, but does not add references or content to any projects in your solution. For example, the [psake](http://nuget.org/packages/psake) package installs Powershell scripts you can use to automate your build process.

A package is considered a solution-level package if it does not contain any files in its **lib** or **content** directories. If the package has dependencies, they also must not have files in their **lib** or **content** directories.

When a solution-level package is installed, it is tracked in a packages.config file in the .nuget directory, rather than in a packages.config file in a specific project.

Publishing

Create an account at NuGet.org

Head over to http://nuget.org/ and register for an account. Once you do that, click on "My Account" to see an API Key that was generated for you.

In a command console, run the command:

nuget setApiKey Your-API-Key

This will store your API key so that you never need to do this step again on this machine.

Package Conventions

There are two types of conventions that apply when creating packages. The conventions listed in this page are *enforced conventions* which have to follow when building packages. There are also *community* (or *optional*) conventions, which have been formed by the community in order to make it easier for others to understand what your package is all about and make use of it immediately. For information about community conventions, see the [Package Conventions](http://docs.nuget.org/docs/creating-packages/Package-Conventions) page. This page will continue to be updated as new conventions are defined.

Supporting Multiple .NET Framework Versions and Profiles

Many libraries target a specific version of the .NET Framework. For example, you might have one version of your library that's specific to Silverlight, and another version of the same library that takes advantage of .NET Framework 4 features. You do not need to create separate packages for each of these versions. NuGet supports putting multiple versions of the same library in a single package keeping them in separate folders within the package.

Framework Version Folder Structure

When NuGet installs an assembly from a package, it checks the target .NET Framework version of the project you are adding the package to. NuGet then selects the correct version of the assembly in the package by selecting the correct subfolder within the *lib* folder.

To enable NuGet to do this, you use the following naming convention to indicate which assemblies go with which framework versions:

lib\{framework name}{version}

The following example shows a folder structure that supports four versions of a library:

\lib

\net11

\MyAssembly.dll

\net20

\MyAssembly.dll

\net40

\MyAssembly.dll

\sl40

\MyAssembly.dll

New in NuGet version 2.0 and above

Starting in NuGet 2.0, in addition to assembly references, content files as well as PowerShell scripts can be grouped by target frameworks too. The framework folder structure inside lib folder as described above applies exactly the same to content and tools folders.

\content

\net11

\MyContent.txt

\net20

\MyContent20.txt

\net40

\sl40

\MySilverlightContent.html

\tools

init.ps1

\net40

install.ps1

uninstall.ps1

\sl40

install.ps1

uninstall.ps1

A new feature in NuGet 2.0 is that a framework folder can be *empty*, in which case, NuGet will not add assembly references or content files or run the PowerShell scripts for the particular framework version.

**Note**: Because init.ps1 is executed at the solution level and not dependent on project, it must be placed directly under the tools folder. If placed under a framework folder, it will be ignored.

Framework Names

NuGet attempts to parse the folder name into a [FrameworkName](http://msdn.microsoft.com/en-us/library/dd414023.aspx) object. Names are case insensitive, and you can use abbreviations for both framework name and version number.

If you omit the framework name, the .NET Framework is assumed. For example, the following folder structure is equivalent to the previous one:

\lib

\11

\MyAssembly.dll

\20

\MyAssembly.dll

\40

\MyAssembly.dll

\sl4

\MyAssembly.dll

The following is a list of valid framework names and abbreviation:

|  |  |
| --- | --- |
| **Framework Name** | **Abbreviations** |
| .NET Framework | net |
| Silverlight | sl |
| .NETMicroFramework | netmf |

Assemblies that are not Specific to a Framework Version

Assemblies that have no associated framework name or version are stored directly in the *lib* folder.

Matching Assembly Version to the Target Framework of a Project

When NuGet installs a package that has multiple assembly versions, it tries to match the framework name of the assembly with the target framework of the project. If a match is not found, NuGet copies the assembly that's for the highest version that is less than or equal to the project's target framework. For example, if you install a package that has the *lib* folder structure shown in the previous example in a project that targets the .NET Framework 3.5, the assembly in the *2* folder (for .NET Framework 2.0) is selected.

Grouping Assemblies by Framework Version

NuGet copies assemblies from only a single library folder. For example, suppose a package has the following folder structure:

\lib

\Net20

\MyAssembly.dll (v1.0)

\MyAssembly.Core.dll (v1.0)

\Net40

\MyAssembly.dll (v2.0)

When the package is installed in a project that targets the .NET Framework 4, *MyAssembly.dll (v2.0)* is the only assembly installed. *MyAssembly.Core.dll (v1.0)* is not installed. (One reason why NuGet behaves this way is that *MyAssembly.Core* might have been merged into version 2.0 of *MyAssembly*.)

In this example, if you want *MyAssembly.Core.dll* to be installed in a project that targets the .NET Framework 4, you must include it in the *Net40* folder as well as in the *Net20* folder.

The rule about copying assemblies from only one folder also applies to the root *lib* folder. Suppose a package has the following folder structure:

\lib

\MyAssembly.dll (v1.0)

\MyAssembly.Core.dll (v1.0)

\Net40

\MyAssembly.dll (v2.0)

In projects that target the .NET Framework 2.0 and the .NET Framework 3.5, NuGet copies both *MyAssembly.dll* and *MyAssembly.Core.dll*. But as was true of the previous example, in projects that target the .NET Framework 4, only *MyAssembly.dll \*from the \*Net40* folder will be copied.

As in the previous example, if you want *MyAssembly.Core.dll* to be installed in a project that targets the .NET Framework 4, you must include it in the *Net40* folder.

Grouping Assemblies by Framework Profile

NuGet also supports targeting a specific framework profile by appending a dash and the profile name to the end of the folder.

lib\{framework name}-{profile}

For example, to target the Windows Phone profile, place your assembly in a folder named *sl3-wp*.

Profiles supported by NuGet include:

* Client - Client Profile
* Full - Full Profile
* WP - Windows Phone

At the time of this writing, to target the Windows Phone profile, the Silverlight 3 framework must be specified. It is anticipated that in the future, later versions of Silverlight will be supported on the phone.

|  |  |
| --- | --- |
| **Profile Name** | **Abbreviations** |
| Client | client |
| WindowsPhone | wp |
| CompactFramework | cf |
| Full | full |

Common Framework and Profile Targeting Examples

The following provides examples of common targets.

|  |  |  |
| --- | --- | --- |
| **Tool** | **Target** | **Notes** |
| .NET 3.5 | net35 | Just using '35' also works. |
| .NET 4.0 | net40 | Just using '40' also works. |
| .NET 4.0 Client Profile | net40-client |  |
| .NET 4.0 Full Profile | net40-full | Requires full .NET profile |
| .NET 4.0 Compact Framework | net40-cf | net40-compactframework also works. |
| .NET Micro Framework | netmf |  |
| Silverlight 3.0 | sl3 |  |
| Silverlight 4.0 | sl4 |  |
| Silverlight 5.0 | sl5 |  |
| Windows Phone 7.0 | sl3-wp |  |
|  | wp7 | Only in NuGet 2.1+ |
| Windows Phone 7.1 (Mango) | sl4-windowsphone71 |  |
|  | wp71 | Only in NuGet 2.1+ |
| Windows Phone 8 | windowsphone8 | Only in NuGet 2.1+ |
| Windows Store apps (Javascript, C#, VB.NET) | netcore45 |  |
|  | windows8 | Only in NuGet 2.1+ |
| Portable class library for Windows Store apps and .NET 4.5 | portable-windows8+net45 | Only in NuGet 2.1+ |
| Portable class library for Windows Store apps, Silverlight 4.0 and Windows Phone 7.1 | portable-sl4+wp71+windows8 | Only in NuGet 2.1+ |

Deleting packages

NuGet.org does not support permanent deletion of packages, because that would break anyone who is depending on it remaining available. This is particularily true when using the workflow that restores packages on build.

Instead, NuGet.org supports a way to 'unlist' a package, which can be done in the package management page on the web site. When a package is unlisted, it no longer shows up in search and in any package listing, both on NuGet.org and from the NuGet Visual Studio extension (or nuget.exe). However, it remains downloadable by specifying its exact version, which is what allows the Restore workflow to continue working.

If you run into an exceptional situation where you think one of your packages must be deleted, this can be handled manually by the NuGet team. e.g. if there is a copyright infrigement issue, or potentially harmful content, that could be a valid reason to delete it.

Automatically Running PowerShell Scripts During Package Installation and Removal

A package can include PowerShell scripts that automatically run when the package is installed or removed. NuGet automatically runs scripts based on their file names using the following conventions:

* ***Init.ps1*** runs the first time a package is installed in a solution.
  + If the same package is installed into additional projects in the solution, the script is not run during those installations.
  + The script also runs every time the solution is opened. For example, if you install a package, close Visual Studio, and then start Visual Studio and open the solution, the *Init.ps1* script runs again.
* ***Install.ps1*** runs when a package is installed in a project.
  + If the same package is installed in multiple projects in a solution, the script runs each time the package is installed.
  + The package must have files in the *content* or *lib* folder for *Install.ps1* to run. Just having something in the tools folder will not kick this off.
  + If your package also has an *init.ps1*, *install.ps1* runs **after** *init.ps1*.
* ***Uninstall.ps1*** runs every time a package is uninstalled.
* These files should be located in the tools directory of your package.
* At the top of your file, add this line: param($installPath, $toolsPath, $package, $project)
  + $installPath is the path to the folder where the package is installed
  + $toolsPath is the path to the tools directory in the folder where the package is installed
  + $package is a reference to the package object.
  + $project is a reference to the EnvDTE project object and represents the project the package is installed into. *Note:* This will be null in *Init.ps1*. In that case doesn't have a reference to a particular project because it runs at the solution level. The properties of this object are defined in [the MSDN documentation](http://msdn.microsoft.com/en-us/library/51h9a6ew(v=VS.80).aspx).
* When you are testing $project in the console while creating your scripts, you can set it to *$project = Get-Project*

If you want to learn about your actual output, [NuGetPSVariables](http://nuget.org/List/Packages/NuGetPSVariables) is a helper package that writes information out to a series of log files. You can make use of the package with the following call:

Install-Package NuGetPSVariables

NuGetPSVariables displays the log files and uninstalls itself.

Import MSBuild targets and props files into project (Requires NuGet 2.5 or above)

A new convention has been added to the structure of NuGet packages. As a peer to \lib, \content, and \tools, you can now include a '\build' folder in your package. Under this folder, you can place two files with fixed names, **{packageid}.targets** or **{packageid}.props**. These two files can be either directly under \build or under framework-specific folders just like the other folders. The rule for picking the best-matched framework folder is exactly the same as in those.

When NuGet installs a package with \build files, it will add an MSBuild element in the project file pointing to the .targets and .props files. The .props file is added at the *top*, whereas the .targets file is added to the *bottom*.

\build

\Net40

\MyPackage.props

\MyPackage.targets

\Silverlight40

\MyPackage.props

If this package is installed into a .NET 4.0 project, for example, both the .props and .targets files are imported into the target project.

<Project ToolsVersion="4.0" DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<Import Project="..\packages\MyPackage.1.0.0\build\net40\MyPackage.props" Condition="Exists('..\packages\MyPackage.1.0.0\build\net40\MyPackage.props')" />

...

...

<Import Project="..\packages\MyPackage.1.0.0\build\net40\MyPackage.targets" Condition="Exists('..\packages\MyPackage.1.0.0\build\net40\MyPackage.targets')" />

</Project>

Automatically Displaying a Readme.txt File During Package Installation

A package can include a *readme.txt* file in the root of the package. This file will be displayed in Visual Studio immediately after the package is installed.

If the package is installed because it is a dependency of another package, the *readme.txt* file will not be opened. Only the *readme.txt* file of the package that the user is explicitly installing will be shown.

This feature was added in NuGet 1.7. If the client is older than NuGet 1.7 the *readme.txt* file will be ignored.

* [Overview](http://docs.nuget.org/docs/start-here/overview)

NuGet is a Visual Studio 2010 extension that makes it easy to add, remove, and update libraries and...

* [Install](http://docs.nuget.org/docs/start-here/installing-nuget)

NuGet can be installed and updated using the Visual Studio Extension Manager. To check if your copy...

* [Videos](http://docs.nuget.org/docs/start-here/videos)

Watch screencasts and presentations about anything and everything NuGet.

* [FAQ](http://docs.nuget.org/docs/start-here/nuget-faq)

Read the Frequently Asked Questions about NuGet and see if your question made the list.

[Outercurve Foundation](http://outercurve.org/)

© 2011-2013 Outercurve Foundation

[Creative Commons License](http://creativecommons.org/licenses/by/3.0/)

Licensed under a [Creative Commons Attribution 3.0 Unported License](http://creativecommons.org/licenses/by/3.0/) except where otherwise noted.

View or edit this page's source on GitHub: <https://github.com/NuGet/NuGetDocs/tree/master/site/Docs/Creating-Packages/Creating-and-Publishing-a-Package.markdown>