

Contents

[Docs at a glance](#)

[What is NuGet?](#)

[Get started](#)

[Install NuGet client tools](#)

[Install and use a package \(dotnet CLI\)](#)

[Install and use a package \(Visual Studio\)](#)

[Install and use a package \(Visual Studio for Mac\)](#)

[Create and publish a .NET Standard package \(dotnet CLI\)](#)

[Create and publish a .NET Standard package \(Visual Studio\)](#)

[Create and publish a .NET Framework package \(Visual Studio\)](#)

[Consume packages](#)

[Overview and workflow](#)

[Find and choose packages](#)

[Install and manage packages](#)

[Visual Studio](#)

[Visual Studio for Mac](#)

[dotnet CLI](#)

[nuget.exe CLI](#)

[Package Manager Console \(PowerShell\)](#)

[Configure NuGet](#)

[Package restore options](#)

[Restore packages](#)

[Troubleshooting](#)

[Reinstall and update packages](#)

[Manage global packages and cache folders](#)

[Manage package trust boundaries](#)

[Work with source control systems](#)

[Common NuGet configurations](#)

[Reference packages in your project](#)

[Package references in project files](#)

[Migrate packages.config to PackageReference](#)
[packages.config](#)

[Create packages](#)

[Overview and workflow](#)

[Create a package \(dotnet CLI\)](#)

[Create a package \(nuget.exe CLI\)](#)

[Create a package \(MSBuild\)](#)

[Support multiple target frameworks in your project file](#)

[Build a prerelease package](#)

[Create a symbol package](#)

[Advanced tasks](#)

[Support multiple target frameworks](#)

[Modify source code and config files](#)

[Select assemblies referenced by projects](#)

[Set package type](#)

[Create a localized package](#)

[Guides for specific content](#)

[Create a UWP package](#)

[Create a native package](#)

[Create UI controls as a NuGet package](#)

[Create an analyzer as a NuGet package](#)

[Create a package for Xamarin with Visual Studio 2015](#)

[Create a package with COM interop assemblies](#)

[Sign packages](#)

[Sign a package](#)

[Signed package signatures and requirements](#)

[Publish packages](#)

[Publish to NuGet.org](#)

[Publish a package](#)

[API keys](#)

[Publish to a private feed](#)

[Overview](#)

[Azure artifacts](#)

[NuGet.Server](#)

[Local feeds](#)

[Concepts](#)

[Package installation process](#)

[Package versioning](#)

[Dependency resolution](#)

[Reference](#)

[.nuspec](#)

[nuget.config file](#)

[Target frameworks](#)

[pack and restore as MSBuild targets](#)

[dotnet CLI](#)

[nuget.exe CLI reference](#)

[add](#)

[config](#)

[delete](#)

[help or ?](#)

[init](#)

[install](#)

[list](#)

[locals](#)

[mirror](#)

[pack](#)

[push](#)

[restore](#)

[setapikey](#)

[sign](#)

[sources](#)

[spec](#)

[update](#)

- [verify](#)
- [trusted-signers](#)
- [Environment variables](#)
- [Long path support](#)
- [PowerShell reference](#)
 - [Add-BindingRedirect](#)
 - [Find-Package](#)
 - [Get-Package](#)
 - [Get-Project](#)
 - [Install-Package](#)
 - [Open-PackagePage](#)
 - [Sync-Package](#)
 - [Uninstall-Package](#)
 - [Update-Package](#)
- [NuGet Server API](#)
 - [Overview](#)
 - [Resources](#)
 - [Autocomplete](#)
 - [Catalog](#)
 - [Package content](#)
 - [Package details URL](#)
 - [Package metadata](#)
 - [Push and delete](#)
 - [Push symbol packages](#)
 - [Report abuse URL](#)
 - [Repository signatures](#)
 - [Search](#)
 - [Service index](#)
 - [How-to: query for all packages using the API](#)
 - [Rate limits](#)
 - [nuget.org protocols](#)
 - [tools.json](#)

NuGet client SDK

Errors and Warnings

[NU1000](#)

[NU1001](#)

[NU1002](#)

[NU1003](#)

[NU1100](#)

[NU1101](#)

[NU1102](#)

[NU1103](#)

[NU1104](#)

[NU1105](#)

[NU1106](#)

[NU1107](#)

[NU1108](#)

[NU1201](#)

[NU1202](#)

[NU1203](#)

[NU1401](#)

[NU1500](#)

[NU1501](#)

[NU1502](#)

[NU1503](#)

[NU1601](#)

[NU1602](#)

[NU1603](#)

[NU1604](#)

[NU1605](#)

[NU1608](#)

[NU1701](#)

[NU1801](#)

[NU3000](#)

NU3001

NU3002

NU3003

NU3004

NU3005

NU3006

NU3007

NU3008

NU3009

NU3010

NU3011

NU3012

NU3013

NU3014

NU3015

NU3016

NU3017

NU3018

NU3019

NU3020

NU3021

NU3022

NU3023

NU3024

NU3025

NU3026

NU3027

NU3028

NU3029

NU3030

NU3031

NU3032

NU3033

NU3034

NU3035

NU3036

NU3037

NU3038

NU3040

NU5000

NU5001

NU5002

NU5003

NU5004

NU5005

NU5007

NU5008

NU5009

NU5010

NU5011

NU5012

NU5013

NU5014

NU5015

NU5016

NU5017

NU5018

NU5019

NU5020

NU5021

NU5022

NU5023

NU5024

NU5025

NU5026

NU5027

NU5028

NU5029

NU5030

NU5031

NU5032

NU5033

NU5034

NU5035

NU5036

NU5037

NU5046

NU5047

NU5048

NU5100

NU5101

NU5102

NU5103

NU5104

NU5105

NU5106

NU5107

NU5108

NU5109

NU5110

NU5111

NU5112

NU5114

NU5115

NU5116

NU5117

[NU5118](#)

[NU5119](#)

[NU5120](#)

[NU5121](#)

[NU5122](#)

[NU5123](#)

[NU5124](#)

[NU5125](#)

[NU5127](#)

[NU5128](#)

[NU5129](#)

[NU5130](#)

[NU5131](#)

[NU5500](#)

Archived content

[project.json management format](#)

[project.json and UWP](#)

[project.json impact](#)

Extensibility

[Extensibility - NuGet plugins](#)

[NuGet Cross Platform Plugins](#)

[NuGet cross platform authentication plugin](#)

[NuGet credential providers for Visual Studio](#)

[nuget.exe credential providers](#)

Visual Studio extensibility

[NuGet API in Visual Studio](#)

[Project system support](#)

[Visual Studio templates](#)

Resources

[Policies](#)

[Governance](#)

[Ecosystem](#)

[NuGet.org policies](#)

[Release notes](#)

[Known Issues](#)

[NuGet 5.x](#)

[NuGet 5.3](#)

[NuGet 5.2](#)

[NuGet 5.1](#)

[NuGet 5.0](#)

[NuGet 4.x](#)

[NuGet 4.9 RTM](#)

[NuGet 4.8 RTM](#)

[NuGet 4.7 RTM](#)

[NuGet 4.6 RTM](#)

[NuGet 4.5 RTM](#)

[NuGet 4.4 RTM](#)

[NuGet 4.3 RTM](#)

[NuGet 4.0 RTM](#)

[NuGet 4.0 RC](#)

[NuGet 3.x](#)

[NuGet 3.5 RTM](#)

[NuGet 3.5 RC](#)

[NuGet 3.5 Beta2](#)

[NuGet 3.5 Beta](#)

[NuGet 3.4.4](#)

[NuGet 3.4.3](#)

[NuGet 3.4.2](#)

[NuGet 3.4.1](#)

[NuGet 3.4](#)

[NuGet 3.4 RC](#)

[NuGet 3.3](#)

[NuGet 3.2.1](#)

[NuGet 3.2](#)

[NuGet 3.2 RC](#)

[NuGet 3.1.1](#)

[NuGet 3.1](#)

[NuGet 3.0.0](#)

[NuGet 3.0 RC2](#)

[NuGet 3.0 RC](#)

[NuGet 3.0 Beta](#)

[NuGet 3.0 Preview](#)

[NuGet 2.x](#)

[NuGet 2.12](#)

[NuGet 2.12 RC](#)

[NuGet 2.9 RC](#)

[NuGet 2.8.7](#)

[NuGet 2.8.6](#)

[NuGet 2.8.5](#)

[NuGet 2.8.3](#)

[NuGet 2.8.2](#)

[NuGet 2.8.1](#)

[NuGet 2.8](#)

[NuGet 2.7.2](#)

[NuGet 2.7.1](#)

[NuGet 2.7](#)

[NuGet 2.6.1-for-WebMatrix](#)

[NuGet 2.6](#)

[NuGet 2.5](#)

[NuGet 2.2.1](#)

[NuGet 2.2](#)

[NuGet 2.1](#)

[NuGet 2.0](#)

[NuGet 1.x](#)

[NuGet 1.8](#)

[NuGet 1.7](#)

[NuGet 1.6](#)

[NuGet 1.5](#)

[NuGet 1.4](#)

[NuGet 1.3](#)

[NuGet 1.2](#)

[NuGet 1.1](#)

[FAQs](#)

[Project format](#)

[NuGet.org](#)

NuGet Documentation

NuGet is the package manager for .NET. It enables developers to create, share, and consume useful .NET libraries. NuGet client tools provide the ability to produce and consume these libraries as "packages".

Introduction to NuGet

[What is NuGet?](#)

[Install NuGet client tools](#)

Get started

[Install and use a package - dotnet CLI](#)

[Install and use a package - Visual Studio](#)

[Create a package - dotnet CLI](#)

[Create a package - Visual Studio](#)

[Create a .NET Framework package - Visual Studio](#)

Consume packages

[Workflow \(overview\)](#)

[Find and choose packages](#)

[Use Visual Studio](#)

[Use dotnet CLI](#)

[Use nuget.exe CLI](#)

[Use Package Manager Console](#)

Create packages

[Workflow \(overview\)](#)

[Use Visual Studio](#)

[Use dotnet CLI](#)

[Use nuget.exe CLI](#)

[Use MSBuild](#)

[Support multiple target frameworks](#)

Publish packages

[Publish to NuGet.org](#)

[Publish to a private feed](#)

NuGet.org

[Overview](#)

[Individual accounts](#)

[Organizations](#)

[API keys](#)

[Publish a package](#)

Reference

[dotnet CLI](#)

[nuget.exe CLI](#)

[Package references](#)

[pack and restore as MSBuild targets](#)

[.nuspec](#)

[nuget.config](#)

[NuGet API](#)

Resources

[Policies - NuGet](#)

[Policies - NuGet.org](#)

[Release notes](#)

[FAQ - NuGet](#)

[FAQ - NuGet.org](#)

An introduction to NuGet

10/15/2019 • 10 minutes to read • [Edit Online](#)

An essential tool for any modern development platform is a mechanism through which developers can create, share, and consume useful code. Often such code is bundled into "packages" that contain compiled code (as DLLs) along with other content needed in the projects that consume these packages.

For .NET (including .NET Core), the Microsoft-supported mechanism for sharing code is **NuGet**, which defines how packages for .NET are created, hosted, and consumed, and [provides the tools](#) for each of those roles.

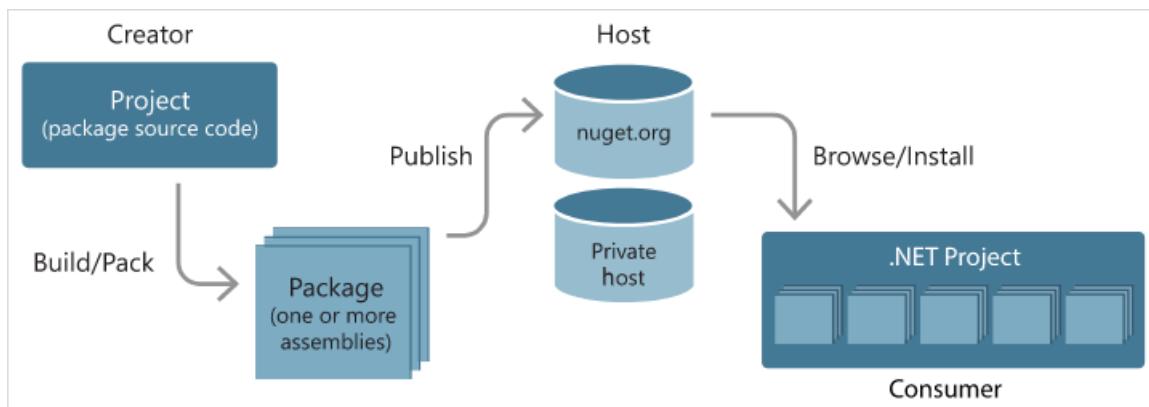
Put simply, a NuGet package is a single ZIP file with the `.nupkg` extension that contains compiled code (DLLs), other files related to that code, and a descriptive manifest that includes information like the package's version number. Developers with code to share create packages and publish them to a public or private host. Package consumers obtain those packages from suitable hosts, add them to their projects, and then call a package's functionality in their project code. NuGet itself then handles all of the intermediate details.

Because NuGet supports private hosts alongside the public [nuget.org](#) host, you can use NuGet packages to share code that's exclusive to an organization or a work group. You can also use NuGet packages as a convenient way to factor your own code for use in nothing but your own projects. In short, a NuGet package is a shareable unit of code, but does not require nor imply any particular means of sharing.

The flow of packages between creators, hosts, and consumers

In its role as a public host, NuGet itself maintains the central repository of over 100,000 unique packages at [nuget.org](#). These packages are employed by millions of .NET/.NET Core developers every day. NuGet also enables you to host packages privately in the cloud (such as on Azure DevOps), on a private network, or even on just your local file system. By doing so, those packages are available to only those developers that have access to the host, giving you the ability to make packages available to a specific group of consumers. The options are explained on [Hosting your own NuGet feeds](#). Through configuration options, you can also control exactly which hosts can be accessed by any given computer, thereby ensuring that packages are obtained from specific sources rather than a public repository like [nuget.org](#).

Whatever its nature, a host serves as the point of connection between package *creators* and package *consumers*. Creators build useful NuGet packages and publish them to a host. Consumers then search for useful and compatible packages on accessible hosts, downloading and including those packages in their projects. Once installed in a project, the packages' APIs are available to the rest of the project code.



Package targeting compatibility

A "compatible" package means that it contains assemblies built for at least one target .NET framework that's

compatible with the consuming project's target framework. Developers can create packages that are specific to one framework, as with UWP controls, or they can support a wider range of targets. To maximize a package's compatibility, developers target [.NET Standard](#), which all .NET and .NET Core projects can consume. This is the most efficient means for both creators and consumers, as a single package (usually containing a single assembly) works for all consuming projects.

Package developers who require APIs outside of .NET Standard, on the other hand, create separate assemblies for the different target frameworks they want to support and include all of those assemblies in the same package (which is called "multi-targeting"). When a consumer installs such a package, NuGet extracts only those assemblies that are needed by the project. This minimizes the package's footprint in the final application and/or assemblies produced by that project. A multi-targeting package is, of course, more difficult for its creator to maintain.

NOTE

Targeting .NET Standard supersedes the previous approach of using various portable class library (PCL) targets. This documentation therefore focuses on creating packages for .NET Standard.

NuGet tools

In addition to hosting support, NuGet also provides a variety of tools used by both creators and consumers. See [Installing NuGet client tools](#) for how to obtain specific tools.

TOOL	PLATFORMS	APPLICABLE SCENARIOS	DESCRIPTION
dotnet CLI	All	Creation, Consumption	CLI tool for .NET Core and .NET Standard libraries, and for SDK-style projects that target .NET Framework (see SDK attribute). Provides certain NuGet CLI capabilities directly within the .NET Core tool chain. As with the <code>nuget.exe</code> CLI, the <code>dotnet</code> CLI does not interact with Visual Studio projects.

TOOL	PLATFORMS	APPLICABLE SCENARIOS	DESCRIPTION
nuget.exe CLI	All	Creation, Consumption	CLI tool for .NET Framework libraries and non-SDK-style projects that target .NET Standard libraries. Provides all NuGet capabilities, with some commands applying specifically to package creators, some applying only to consumers, and others applying to both. For example, package creators use the <code>nuget pack</code> command to create a package from various assemblies and related files, package consumers use <code>nuget install</code> to include packages in a project folder, and everyone uses <code>nuget config</code> to set NuGet configuration variables. As a platform-agnostic tool, the NuGet CLI does not interact with Visual Studio projects.
Package Manager Console	Visual Studio on Windows	Consumption	Provides PowerShell commands for installing and managing packages in Visual Studio projects.
Package Manager UI	Visual Studio on Windows	Consumption	Provides an easy-to-use UI for installing and managing packages in Visual Studio projects.
Manage NuGet UI	Visual Studio for Mac	Consumption	Provides an easy-to-use UI for installing and managing packages in Visual Studio for Mac projects.
MSBuild	Windows	Creation, Consumption	Provides the ability to create packages and restore packages used in a project directly through the MSBuild tool chain.

As you can see, the NuGet tools you work with depend greatly on whether you're creating, consuming, or publishing packages, and the platform on which you're working. Package creators are typically also consumers, as they build on top of functionality that exists in other NuGet packages. And those packages, of course, may in turn depend on still others.

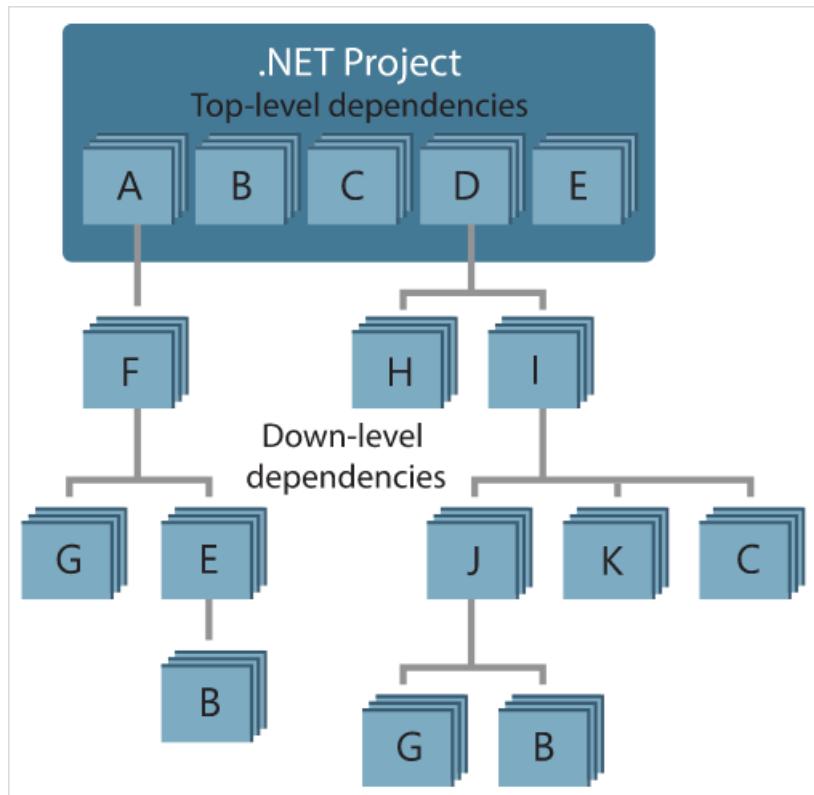
For more information, start with the [Package creation workflow](#) and [Package consumption workflow](#) articles.

Managing dependencies

The ability to easily build on the work of others is one of most powerful features of a package management system. Accordingly, much of what NuGet does is managing that dependency tree or "graph" on behalf of a

project. Simply said, you need only concern yourself with those packages that you're directly using in a project. If any of those packages themselves consume other packages (which can, in turn, consume still others), NuGet takes care of all those down-level dependencies.

The following image shows a project that depends on five packages, which in turn depend on a number of others.



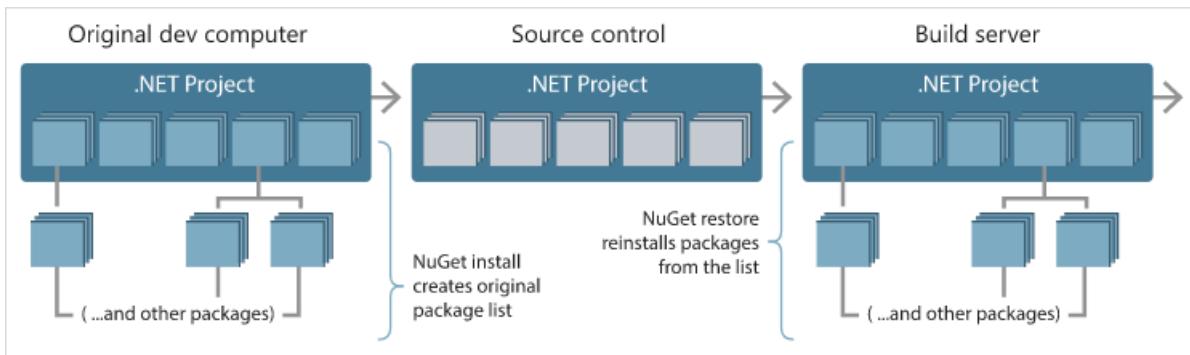
Notice that some packages appear multiple times in the dependency graph. For example, there are three different consumers of package B, and each consumer might also specify a different version for that package (not shown). This is a common occurrence, especially for widely-used packages. NuGet fortunately does all the hard work to determine exactly which version of package B satisfies all consumers. NuGet then does the same for all other packages, no matter how deep the dependency graph.

For more details on how NuGet performs this service, see [Dependency resolution](#).

Tracking references and restoring packages

Because projects can easily move between developer computers, source control repositories, build servers, and so forth, it's highly impractical to keep the binary assemblies of NuGet packages directly bound to a project. Doing so would make each copy of the project unnecessarily bloated (and thereby waste space in source control repositories). It would also make it very difficult to update package binaries to newer versions as updates would have to be applied across all copies of the project.

NuGet instead maintains a simple reference list of the packages upon which a project depends, including both top-level and down-level dependencies. That is, whenever you install a package from some host into a project, NuGet records the package identifier and version number in the reference list. (Uninstalling a package, of course, removes it from the list.) NuGet then provides a means to restore all referenced packages upon request, as described on [Package restore](#).



With only the reference list, NuGet can then reinstall—that is, *restore*—all of those packages from public and/or private hosts at any later time. When committing a project to source control, or sharing it in some other way, you include only the reference list and exclude any package binaries (see [Packages and source control](#).)

The computer that receives a project, such as a build server obtaining a copy of the project as part of an automated deployment system, simply asks NuGet to restore dependencies whenever they're needed. Build systems like Azure DevOps provide "NuGet restore" steps for this exact purpose. Similarly, when developers obtain a copy of a project (as when cloning a repository), they can invoke command like `nuget restore` (NuGet CLI), `dotnet restore` (dotnet CLI), or `Install-Package` (Package Manager Console) to obtain all the necessary packages. Visual Studio, for its part, automatically restores packages when building a project (provided that automatic restore is enabled, as described on [Package restore](#)).

Clearly, then, NuGet's primary role where developers are concerned is maintaining that reference list on behalf of your project and providing the means to efficiently restore (and update) those referenced packages. This list is maintained in one of two *package management formats*, as they're called:

- [PackageReference](#) (or "package references in project files") | (*NuGet 4.0+*) Maintains a list of a project's top-level dependencies directly within the project file, so no separate file is needed. An associated file, `obj/project.assets.json`, is dynamically generated to manage the overall dependency graph of the packages that a project uses along with all down-level dependencies. [PackageReference](#) is always used by .NET Core projects.
- [packages.config](#) : (*NuGet 1.0+*) An XML file that maintains a flat list of all dependencies in the project, including the dependencies of other installed packages. Installed or restored packages are stored in a `packages` folder.

Which package management format is employed in any given project depends on the project type, and the available version of NuGet (and/or Visual Studio). To check what format is being used, simply look for `packages.config` in the project root after installing your first package. If you don't have that file, look in the project file directly for a `<PackageReference>` element.

When you have a choice, we recommend using [PackageReference](#). `packages.config` is maintained for legacy purposes and is no longer under active development.

TIP

Various `nuget.exe` CLI commands, like `nuget install`, do not automatically add the package to the reference list. The list is updated when installing a package with the Visual Studio Package Manager (UI or Console), and with `dotnet.exe` CLI.

What else does NuGet do?

So far you've learned the following characteristics of NuGet:

- NuGet provides the central [nuget.org](#) repository with support for private hosting.

- NuGet provides the tools developers need for creating, publishing, and consuming packages.
- Most importantly, NuGet maintains a reference list of packages used in a project and the ability to restore and update those packages from that list.

To make these processes work efficiently, NuGet does some behind-the-scenes optimizations. Most notably, NuGet manages a package cache and a global packages folder to shortcut installation and reinstallation. The cache avoids downloading a package that's already been installed on the machine. The global packages folder allows multiple projects to share the same installed package, thereby reducing NuGet's overall footprint on the computer. The cache and global packages folder are also very helpful when you're frequently restoring a larger number of packages, as on a build server. For more details on these mechanisms, see [Managing the global packages and cache folders](#).

Within an individual project, NuGet manages the overall dependency graph, which again includes resolving multiple references to different versions of the same package. It's quite common that a project takes a dependency on one or more packages that themselves have the same dependencies. Some of the most useful utility packages on nuget.org are employed by many other packages. In the entire dependency graph, then, you could easily have ten different references to different versions of the same package. To avoid bringing multiple versions of that package into the application itself, NuGet sorts out which single version can be used by all consumers. (For more information, see [Dependency Resolution](#).)

Beyond that, NuGet maintains all the specifications related to how packages are structured (including [localization](#) and [debug symbols](#)) and how they are [referenced](#) (including [version ranges](#) and [pre-release versions](#).) NuGet also provides various APIs to work with its services programmatically, and provides support for developers who write Visual Studio extensions and project templates.

Take a moment to browse the table of contents for this documentation, and you see all of these capabilities represented there, along with release notes dating back to NuGet's beginnings.

Comments, contributions, and issues

Finally, we very much welcome comments and contributions to this documentation—just select the [Feedback](#) and [Edit](#) commands on the top of any page, or visit the [docs repository](#) and [docs issue list](#) on GitHub.

We also welcome contributions to NuGet itself through its [various GitHub repositories](#); NuGet issues can be found on <https://github.com/NuGet/home/issues>.

Enjoy your NuGet experience!

Install NuGet client tools

11/5/2019 • 6 minutes to read • [Edit Online](#)

Looking to install a package? See [Ways to install NuGet packages](#).

To work with NuGet, as a package consumer or creator, you can use command-line interface (CLI) tools as well as NuGet features in Visual Studio. This article briefly outlines the capabilities of the different tools, how to install them, and their comparative [feature availability](#). To get started using NuGet to consume packages, see [Install and use a package \(dotnet CLI\)](#) and [Install and use a package \(Visual Studio\)](#). To get started creating NuGet packages, see [Create and publish a .NET Standard package \(dotnet CLI\)](#) and [Create and publish a .NET Standard package \(Visual Studio\)](#).

TOOL	DESCRIPTION	DOWNLOAD
dotnet.exe	CLI tool for .NET Core and .NET Standard libraries, and for any SDK-style project such as one that targets .NET Framework. Included with the .NET Core SDK and provides core NuGet features on all platforms. (Starting in Visual Studio 2017, the dotnet CLI is automatically installed with any .NET Core related workloads.)	.NET Core SDK
nuget.exe	CLI tool for .NET Framework libraries and for any non-SDK-style project such as one that targets .NET Standard libraries. Provides all NuGet capabilities on Windows, provides most features on Mac and Linux when running under Mono.	nuget.exe
Visual Studio	On Windows, provides NuGet capabilities through the Package Manager UI and Package Manager Console; included with .NET-related workloads. On Mac, provides certain features through the UI. In Visual Studio Code, NuGet features are provided through extensions.	Visual Studio

The [MSBuild CLI](#) also provides the ability to restore and create packages, which is primarily useful on build servers. MSBuild is not a general-purpose tool for working with NuGet.

CLI tools

The two NuGet CLI tools are `dotnet.exe` and `nuget.exe`. See [feature availability](#) for a comparison.

- To target .NET Core or .NET Standard, use the dotnet CLI. The `dotnet` CLI is required for the SDK-style project format, which uses the [SDK attribute](#).
- To target .NET Framework (non-SDK-style project only), use the `nuget.exe` CLI. If the project is migrated from `packages.config` to `PackageReference`, use the dotnet CLI.

dotnet.exe CLI

The .NET Core 2.0 CLI, `dotnet.exe`, works on all platforms (Windows, Mac, and Linux) and provides core NuGet features such as installing, restoring, and publishing packages. `dotnet` provides direct integration with .NET Core project files (such as `.csproj`), which is helpful in most scenarios. `dotnet` is also built directly for each platform and does not require you to install Mono.

Installation:

- On developer computers, install the [.NET Core SDK](#). Starting in Visual Studio 2017, the dotnet CLI is automatically installed with any .NET Core related workloads.
- For build servers, follow the instructions on [Using .NET Core SDK and tools in Continuous Integration](#).

To learn how to use basic commands with the dotnet CLI, see [Install and use packages using the dotnet CLI](#).

nuget.exe CLI

The `nuget.exe` CLI, `nuget.exe`, is the command-line utility for Windows that provides all NuGet capabilities; it can also be run on Mac OSX and Linux using [Mono](#) with some limitations.

To learn how to use basic commands with the `nuget.exe` CLI, see [Install and use packages using the nuget.exe CLI](#).

Installation:

Windows

NOTE

NuGet.exe 5.0 and later require .NET Framework 4.7.2 or later to execute.

1. Visit [nuget.org/downloads](#) and select NuGet 3.3 or higher (2.8.6 is not compatible with Mono). The latest version is always recommended, and 4.1.0+ is required to publish packages to nuget.org.
2. Each download is the `nuget.exe` file directly. Instruct your browser to save the file to a folder of your choice. The file is *not* an installer; you won't see anything if you run it directly from the browser.
3. Add the folder where you placed `nuget.exe` to your PATH environment variable to use the CLI tool from anywhere.

macOS/Linux

Behaviors may vary slightly by OS distribution.

1. Install [Mono 4.4.2 or later](#).
2. Execute the following command at a shell prompt:

```
# Download the latest stable `nuget.exe` to `/usr/local/bin`  
sudo curl -o /usr/local/bin/nuget.exe https://dist.nuget.org/win-x86-commandline/latest/nuget.exe
```

3. Create an alias by adding the following script to the appropriate file for your OS (typically `~/.bash_aliases` or `~/.bash_profile`):

```
# Create an alias for nuget  
alias nuget="mono /usr/local/bin/nuget.exe"
```

4. Reload the shell. Test the installation by entering `nuget` with no parameters. NuGet CLI help should display.

TIP

Use `nuget update -self` on Windows to update an existing nuget.exe to the latest version.

NOTE

The latest recommended NuGet CLI is always available at

<https://dist.nuget.org/win-x86-commandline/latest/nuget.exe>. For compatibility purposes with older continuous integration systems, a previous URL, <https://nuget.org/nuget.exe> currently provides the [deprecated 2.8.6 CLI tool](#).

Visual Studio

- Visual Studio Code: NuGet capabilities are available through marketplace extensions, or use the `dotnet.exe` or `nuget.exe` CLI tools.
- Visual Studio for Mac: certain NuGet capabilities are built in directly. See [Including a NuGet package in your project](#) for a walkthrough. For other capabilities, use the `dotnet.exe` or `nuget.exe` CLI tools.
- Visual Studio on Windows: The **NuGet Package Manager** is included with Visual Studio 2012 and later. Visual Studio provides the [Package Manager UI](#) and the [Package Manager Console](#), through which you can run most NuGet operations.
 - Starting in Visual Studio 2017, the installer includes the NuGet Package Manager with any workload that employs .NET. To install separately, or to verify that the Package Manager is installed, run the Visual Studio installer and check the option under **Individual Components > Code tools > NuGet package manager**.
 - The Package Manager UI and Console are unique to Visual Studio on Windows. They are not presently available on Visual Studio for Mac.
 - A CLI tool is required to support NuGet features in the IDE. You can use either the `dotnet` CLI or the `nuget.exe` CLI. The `dotnet` CLI is installed with some Visual Studio workloads, such as .NET Core. The `nuget.exe` CLI must be installed separately as described earlier.
 - Package Manager Console commands work only within Visual Studio on Windows and do not work within other PowerShell environments.
 - For Visual Studio 2010 and earlier, install the "NuGet Package Manager for Visual Studio" extension.
 - NuGet Extensions for Visual Studio 2013 and 2015 can also be downloaded from <https://dist.nuget.org/index.html>.
 - If you'd like to preview upcoming NuGet features, install a [Visual Studio Preview](#), which works side-by-side with stable releases of Visual Studio. To report problems or share ideas for previews, open an issue on the [NuGet GitHub repository](#).

Feature availability

FEATURE	DOTNET CLI	NUGET CLI (WINDOWS)	NUGET CLI (MONO)	VISUAL STUDIO (WINDOWS)	VISUAL STUDIO FOR MAC
Search packages		✓	✓	✓	✓
Install/uninstall packages	✓	✓(1)	✓	✓	✓
Update packages	✓	✓		✓	✓

FEATURE	DOTNET CLI	NUGET CLI (WINDOWS)	NUGET CLI (MONO)	VISUAL STUDIO (WINDOWS)	VISUAL STUDIO FOR MAC
Restore packages	✓	✓	✓(2)	✓	✓
Manage package feeds (sources)		✓	✓	✓	✓
Manage packages on a feed	✓	✓	✓		
Set API keys for feeds		✓	✓		
Create packages(3)	✓	✓	✓(4)	✓	
Publish packages	✓	✓	✓	✓	
Replicate packages		✓	✓		
Manage <i>global-package</i> and cache folders	✓	✓	✓		
Manage NuGet configuration		✓	✓		

(1) Does not affect project files; use `dotnet.exe` instead.

(2) Works only with `packages.config` file and not with solution (`.sln`) files.

(3) Various advanced package features are available through the CLI only as they aren't represented in the Visual Studio UI tools.

(4) Works with `.nuspec` files but not with project files.

Related topics

- [Install and manage packages using Visual Studio](#)
- [Install and manage packages using PowerShell](#)
- [Install and manage packages using dotnet CLI](#)
- [Install and manage packages using nuget.exe CLI](#)
- [Package Manager Console PowerShell reference](#)
- [Creating a package](#)
- [Publishing a Package](#)

Developers working on Windows can also explore the [NuGet Package Explorer](#), an open-source, stand-alone tool to visually explore, create, and edit NuGet packages. It's very helpful, for example, to make experimental changes to a package structure without rebuilding the package.

Quickstart: Install and use a package using the dotnet CLI

8/6/2019 • 2 minutes to read • [Edit Online](#)

NuGet packages contain reusable code that other developers make available to you for use in your projects. See [What is NuGet?](#) for background. Packages are installed into a .NET Core project using the `dotnet add package` command as described in this article for the popular [Newtonsoft.Json](#) package.

Once installed, refer to the package in code with `using <namespace>` where `<namespace>` is specific to the package you're using. You can then use the package's API.

TIP

Start with nuget.org: Browsing nuget.org is how .NET developers typically find components they can reuse in their own applications. You can search nuget.org directly or find and install packages within Visual Studio as shown in this article.

Prerequisites

- The [.NET Core SDK](#), which provides the `dotnet` command-line tool. Starting in Visual Studio 2017, the dotnet CLI is automatically installed with any .NET Core related workloads.

Create a project

NuGet packages can be installed into a .NET project of some kind. For this walkthrough, create a simple .NET Core console project as follows:

1. Create a folder for the project.
2. Open a command prompt and switch to the new folder.
3. Create the project using the following command:

```
dotnet new console
```

4. Use `dotnet run` to test that the app has been created properly.

Add the Newtonsoft.Json NuGet package

1. Use the following command to install the `Newtonsoft.json` package:

```
dotnet add package Newtonsoft.Json
```

2. After the command completes, open the `.csproj` file to see the added reference:

```
<ItemGroup>
<PackageReference Include="Newtonsoft.Json" Version="12.0.1" />
</ItemGroup>
```

Use the Newtonsoft.Json API in the app

1. Open the `Program.cs` file and add the following line at the top of the file:

```
using Newtonsoft.Json;
```

2. Add the following code before the `class Program` line:

```
public class Account
{
    public string Name { get; set; }
    public string Email { get; set; }
    public DateTime DOB { get; set; }
}
```

3. Replace the `Main` function with the following:

```
static void Main(string[] args)
{
    Account account = new Account
    {
        Name = "John Doe",
        Email = "john@nuget.org",
        DOB = new DateTime(1980, 2, 20, 0, 0, 0, DateTimeKind.Utc),
    };

    string json = JsonConvert.SerializeObject(account, Formatting.Indented);
    Console.WriteLine(json);
}
```

4. Build and run the app by using the `dotnet run` command. The output should be the JSON representation of the `Account` object in the code:

```
{
    "Name": "John Doe",
    "Email": "john@nuget.org",
    "DOB": "1980-02-20T00:00:00Z"
}
```

Next steps

Congratulations on installing and using your first NuGet package!

[Install and use packages using the dotnet CLI](#)

To explore more that NuGet has to offer, select the links below.

- [Overview and workflow of package consumption](#)
- [Finding and choosing packages](#)
- [Package references in project files](#)

Quickstart: Install and use a package in Visual Studio (Windows only)

9/26/2019 • 3 minutes to read • [Edit Online](#)

NuGet packages contain reusable code that other developers make available to you for use in your projects. See [What is NuGet?](#) for background. Packages are installed into a Visual Studio project using the NuGet Package Manager or the Package Manager Console. This article demonstrates the process using the popular [Newtonsoft.Json](#) package and a Windows Presentation Foundation (WPF) project. The same process applies to any other .NET or .NET Core project.

Once installed, refer to the package in code with `using <namespace>` where `<namespace>` is specific to the package you're using. Once the reference is made, you can call the package through its API.

TIP

Start with nuget.org: Browsing [nuget.org](#) is how .NET developers typically find components they can reuse in their own applications. You can search [nuget.org](#) directly or find and install packages within Visual Studio as shown in this article. For general information, see [Find and evaluate NuGet packages](#).

Prerequisites

- Visual Studio 2019 with the .NET Desktop Development workload.

You can install the 2019 Community edition for free from [visualstudio.com](#) or use the Professional or Enterprise editions.

If you're using Visual Studio for Mac, see [Install and use a package in Visual Studio for Mac](#).

Create a project

NuGet packages can be installed into any .NET project, provided that the package supports the same target framework as the project.

For this walkthrough, use a simple WPF app. Create a project in Visual Studio using **File > New Project**, typing **.NET** in the search box, and then selecting the **WPF App (.NET Framework)**. Click **Next**. Accept the default values for **Framework** when prompted.

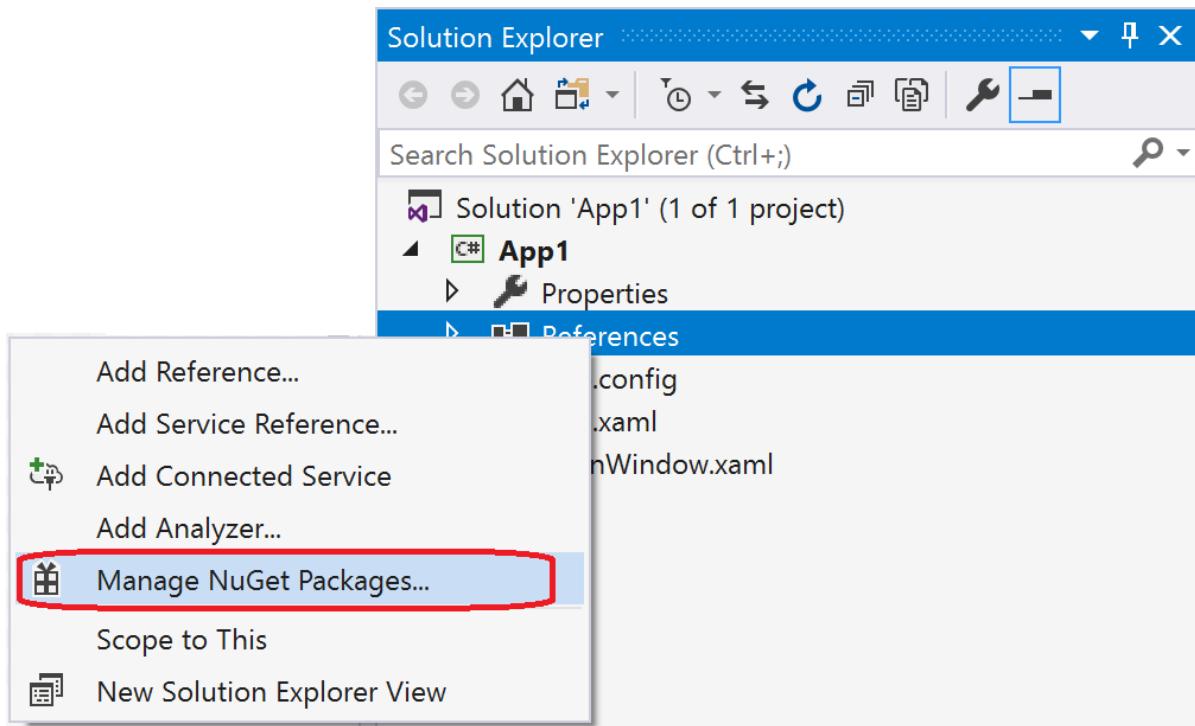
Visual Studio creates the project, which opens in Solution Explorer.

Add the Newtonsoft.Json NuGet package

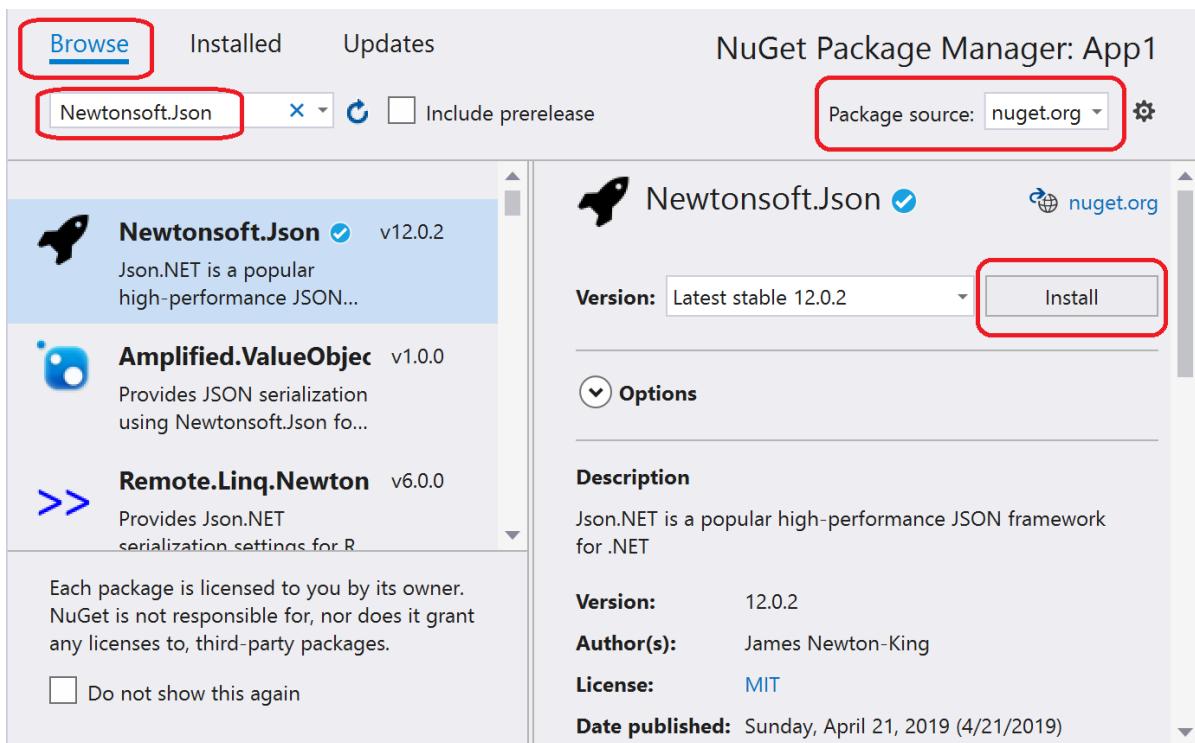
To install the package, you can use either the NuGet Package Manager or the Package Manager Console. When you install a package, NuGet records the dependency in either your project file or a `packages.config` file (depending on the project format). For more information, see [Package consumption overview and workflow](#).

NuGet Package Manager

1. In Solution Explorer, right-click **References** and choose **Manage NuGet Packages**.

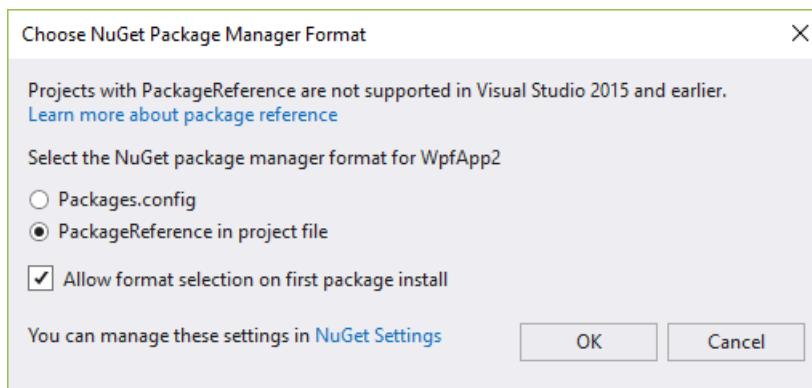


2. Choose "nuget.org" as the **Package source**, select the **Browse** tab, search for **Newtonsoft.Json**, select that package in the list, and select **Install**:



If you want more information on the NuGet Package Manager, see [Install and manage packages using Visual Studio](#).

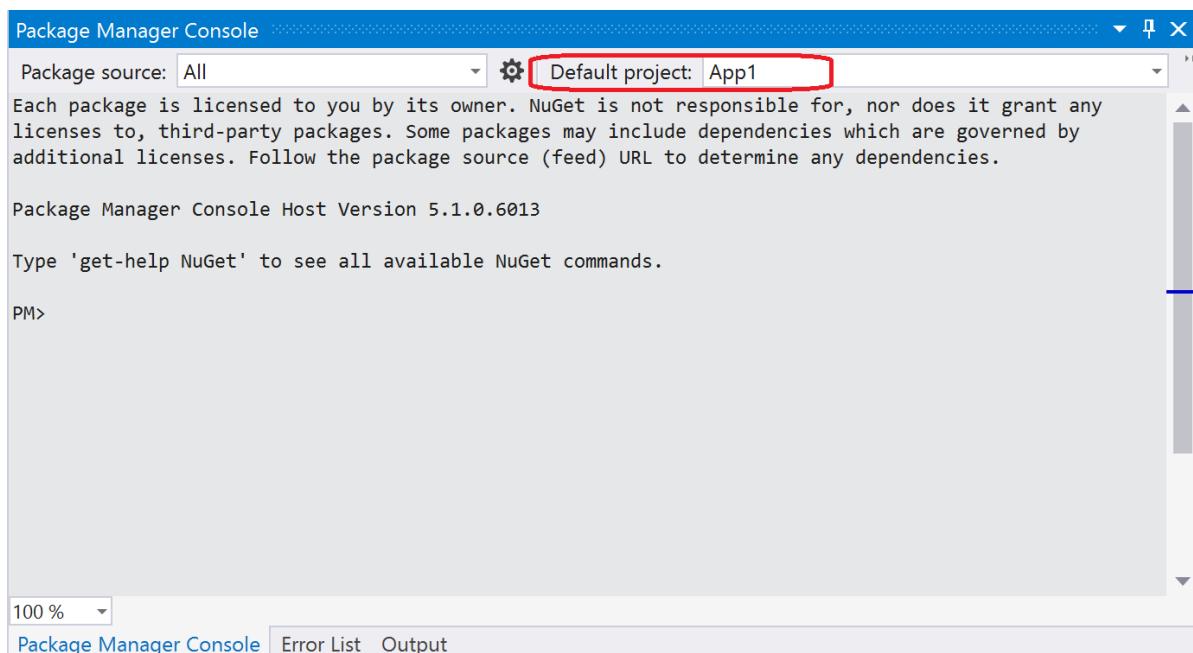
3. Accept any license prompts.
4. (Visual Studio 2017 only) If prompted to select a package management format, select **PackageReference in project file**:



5. If prompted to review changes, select **OK**.

Package Manager Console

1. Select the **Tools > NuGet Package Manager > Package Manager Console** menu command.
2. Once the console opens, check that the **Default project** drop-down list shows the project into which you want to install the package. If you have a single project in the solution, it is already selected.



3. Enter the command `Install-Package Newtonsoft.Json` (see [Install-Package](#)). The console window shows output for the command. Errors typically indicate that the package isn't compatible with the project's target framework.

If you want more information on the Package Manager Console, see [Install and manage packages using Package Manager Console](#).

Use the Newtonsoft.Json API in the app

With the Newtonsoft.Json package in the project, you can call its `JsonConvert.SerializeObject` method to convert an object to a human-readable string.

1. Open `MainWindow.xaml` and replace the existing `Grid` element with the following:

```
<Grid Background="White">
    <StackPanel VerticalAlignment="Center">
        <Button Click="Button_Click" Width="100px" HorizontalAlignment="Center" Content="Click Me" Margin="10"/>
        <TextBlock Name="TextBlock" HorizontalAlignment="Center" Text="TextBlock" Margin="10"/>
    </StackPanel>
</Grid>
```

2. Open the `MainWindow.xaml.cs` file (located in Solution Explorer under the `MainWindow.xaml` node), and insert the following code inside the `MainWindow` class:

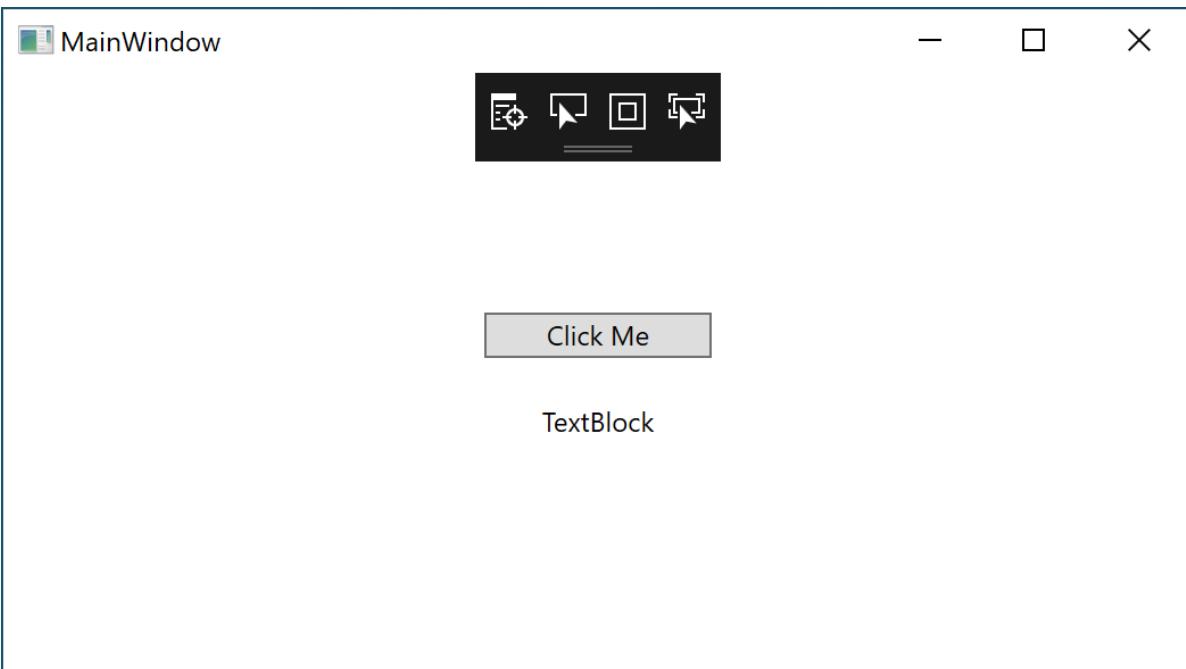
```
public class Account
{
    public string Name { get; set; }
    public string Email { get; set; }
    public DateTime DOB { get; set; }
}

private void Button_Click(object sender, RoutedEventArgs e)
{
    Account account = new Account
    {
        Name = "John Doe",
        Email = "john@microsoft.com",
        DOB = new DateTime(1980, 2, 20, 0, 0, 0, DateTimeKind.Utc),
    };
    string json = JsonConvert.SerializeObject(account, Formatting.Indented);
    TextBlock.Text = json;
}
```

3. Even though you added the `Newtonsoft.Json` package to the project, red squiggles appears under `JsonConvert` because you need a `using` statement at the top of the code file:

```
using Newtonsoft.Json;
```

4. Build and run the app by pressing F5 or selecting **Debug > Start Debugging**:



5. Select on the button to see the contents of the TextBlock replaced with some JSON text:



Next steps

Congratulations on installing and using your first NuGet package!

[Install and manage packages using Visual Studio](#)

[Install and manage packages using Package Manager Console](#)

To explore more that NuGet has to offer, select the links below.

- [● Overview and workflow of package consumption](#)
- [● Finding and choosing packages](#)
- [● Package references in project files](#)

Quickstart: Install and use a package in Visual Studio for Mac

9/3/2019 • 2 minutes to read • [Edit Online](#)

NuGet packages contain reusable code that other developers make available to you for use in your projects. See [What is NuGet?](#) for background. Packages are installed into a Visual Studio for Mac project using the NuGet Package Manager. This article demonstrates the process using the popular [Newtonsoft.Json](#) package and a .NET Core console project. The same process applies to any other Xamarin or .NET Core project.

Once installed, refer to the package in code with `using <namespace>` where `<namespace>` is specific to the package you're using. Once the reference is made, you can call the package through its API.

TIP

Start with nuget.org: Browsing [nuget.org](#) is how .NET developers typically find components they can reuse in their own applications. You can search [nuget.org](#) directly or find and install packages within Visual Studio as shown in this article. For general information, see [Find and evaluate NuGet packages](#).

Prerequisites

- Visual Studio 2019 for Mac.

You can install the 2019 Community edition for free from [visualstudio.com](#) or use the Professional or Enterprise editions.

If you're using Visual Studio on Windows, see [Install and use a package in Visual Studio \(Windows Only\)](#).

Create a project

NuGet packages can be installed into any .NET project, provided that the package supports the same target framework as the project.

For this walkthrough, use a simple .NET Core Console app. Create a project in Visual Studio for Mac using **File > New Solution...**, select the **.NET Core > App > Console Application** template. Click **Next**. Accept the default values for **Target Framework** when prompted.

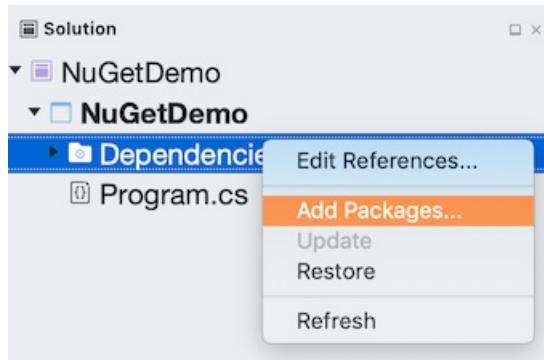
Visual Studio creates the project, which opens in Solution Explorer.

Add the Newtonsoft.Json NuGet package

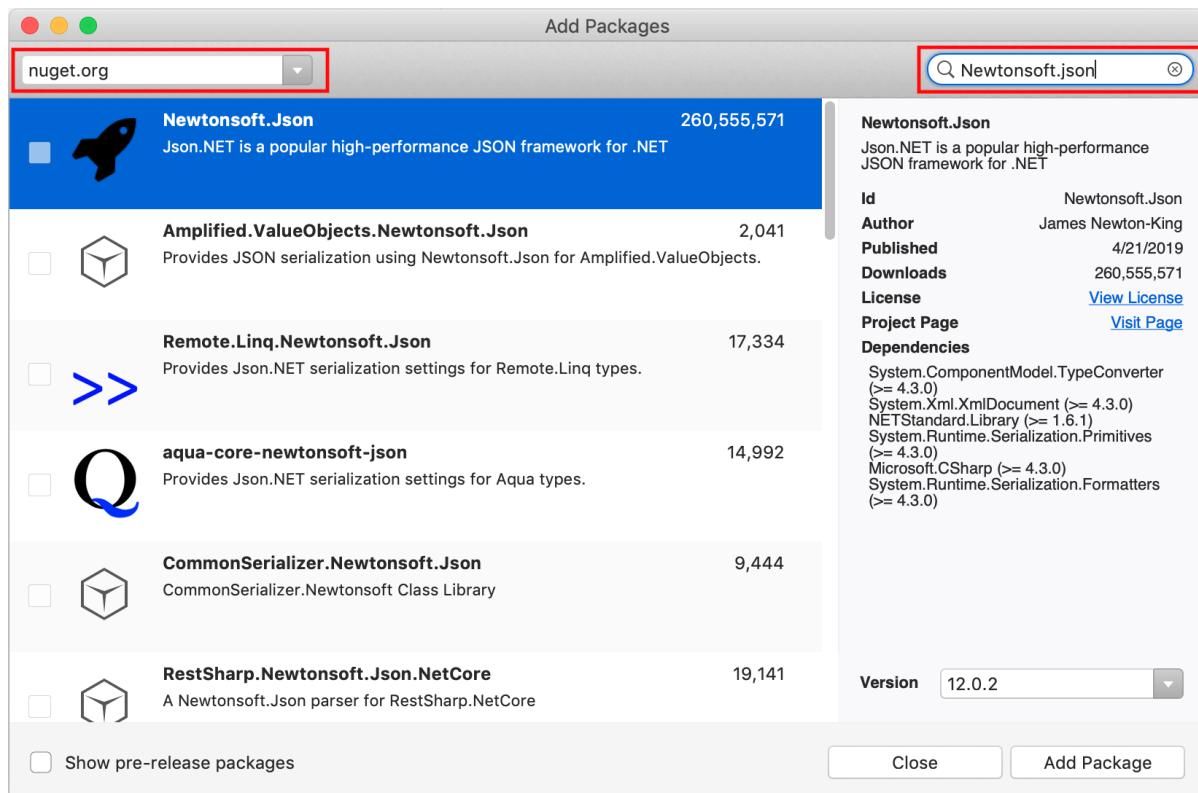
To install the package, you use the NuGet Package Manager. When you install a package, NuGet records the dependency in either your project file or a `packages.config` file (depending on the project format). For more information, see [Package consumption overview and workflow](#).

NuGet Package Manager

1. In Solution Explorer, right-click **Dependencies** and choose **Add Packages....**



2. Choose "nuget.org" as the **Package source** in the top left corner of the dialog, and search for **Newtonsoft.Json**, select that package in the list, and select **Add Packages...**:



If you want more information on the NuGet Package Manager, see [Install and manage packages using Visual Studio for Mac](#).

Use the Newtonsoft.Json API in the app

With the Newtonsoft.Json package in the project, you can call its `JsonConvert.SerializeObject` method to convert an object to a human-readable string.

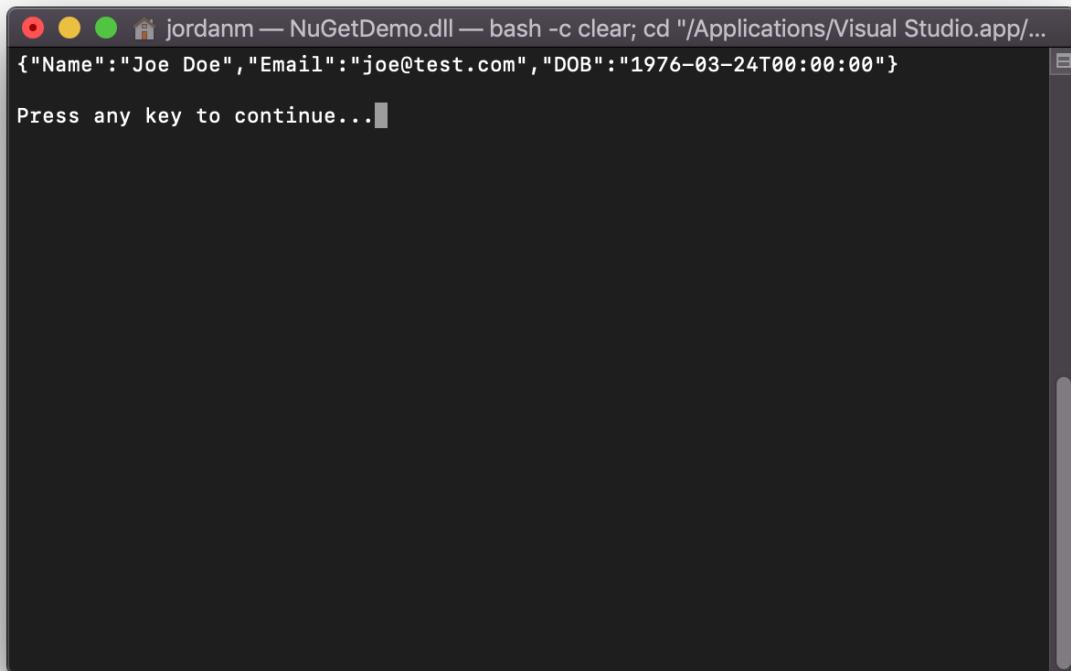
1. Open the `Program.cs` file (located in the Solution Pad) and replace the file contents with the following code:

```
using System;
using Newtonsoft.Json;

namespace NuGetDemo
{
    public class Account
    {
        public string Name { get; set; }
        public string Email { get; set; }
        public DateTime DOB { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Account account = new Account()
            {
                Name = "Joe Doe",
                Email = "joe@test.com",
                DOB = new DateTime(1976, 3, 24)
            };
            string json = JsonConvert.SerializeObject(account);
            Console.WriteLine(json);
        }
    }
}
```

2. Build and run the app by selecting **Run > Start Debugging**:
3. Once the app runs, you'll see the serialized JSON output appear in the console:



```
{"Name": "Joe Doe", "Email": "joe@test.com", "DOB": "1976-03-24T00:00:00"}  
Press any key to continue...
```

Next steps

Congratulations on installing and using your first NuGet package!

Install and manage packages using Visual Studio for Mac

To explore more that NuGet has to offer, select the links below.

- [Overview and workflow of package consumption](#)
- [Package references in project files](#)

Quickstart: Create and publish a package (dotnet CLI)

11/14/2019 • 5 minutes to read • [Edit Online](#)

It's a simple process to create a NuGet package from a .NET Class Library and publish it to nuget.org using the `dotnet` command-line interface (CLI).

Prerequisites

1. Install the [.NET Core SDK](#), which includes the `dotnet` CLI. Starting in Visual Studio 2017, the dotnet CLI is automatically installed with any .NET Core related workloads.
2. [Register for a free account on nuget.org](#) if you don't have one already. Creating a new account sends a confirmation email. You must confirm the account before you can upload a package.

Create a class library project

You can use an existing .NET Class Library project for the code you want to package, or create a simple one as follows:

1. Create a folder called `AppLogger`.
2. Open a command prompt and switch to the `AppLogger` folder.
3. Type `dotnet new classlib`, which uses the name of the current folder for the project.

This creates the new project.

Add package metadata to the project file

Every NuGet package needs a manifest that describes the package's contents and dependencies. In a final package, the manifest is a `.nuspec` file that is generated from the NuGet metadata properties that you include in the project file.

1. Open your project file (`.csproj`) and add the following minimal properties inside the existing `<PropertyGroup>` tag, changing the values as appropriate:

```
<PackageId>AppLogger</PackageId>
<Version>1.0.0</Version>
<Authors>your_name</Authors>
<Company>your_company</Company>
```

IMPORTANT

Give the package an identifier that's unique across nuget.org or whatever host you're using. For this walkthrough we recommend including "Sample" or "Test" in the name as the later publishing step does make the package publicly visible (though it's unlikely anyone will actually use it).

2. Add any optional properties described on [NuGet metadata properties](#).

NOTE

For packages built for public consumption, pay special attention to the **PackageTags** property, as tags help others find your package and understand what it does.

Run the pack command

To build a NuGet package (a `.nupkg` file) from the project, run the `dotnet pack` command, which also builds the project automatically:

```
# Uses the project file in the current folder by default
dotnet pack
```

The output shows the path to the `.nupkg` file:

```
Microsoft (R) Build Engine version 15.5.180.51428 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 29.91 ms for D:\proj\AppLoggerNet\AppLogger\AppLogger.csproj.
AppLogger -> D:\proj\AppLoggerNet\AppLogger\bin\Debug\netstandard2.0\AppLogger.dll
Successfully created package 'D:\proj\AppLoggerNet\AppLogger\bin\Debug\AppLogger.1.0.0.nupkg'.
```

Automatically generate package on build

To automatically run `dotnet pack` when you run `dotnet build`, add the following line to your project file within `<PropertyGroup>`:

```
<GeneratePackageOnBuild>true</GeneratePackageOnBuild>
```

Publish the package

Once you have a `.nupkg` file, you publish it to nuget.org using the `dotnet nuget push` command along with an API key acquired from nuget.org.

NOTE

Virus scanning: All packages uploaded to nuget.org are scanned for viruses and rejected if any viruses are found. All packages listed on nuget.org are also scanned periodically.

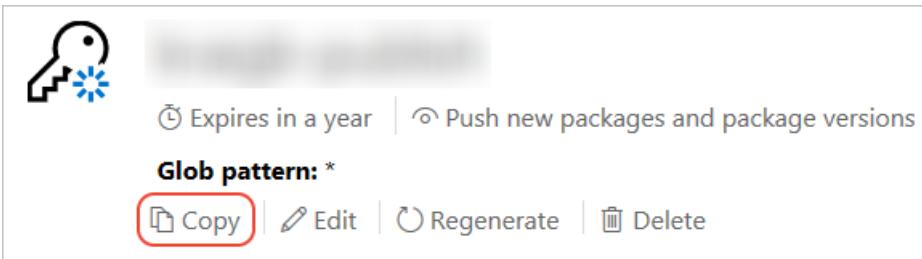
Packages published to nuget.org are also publicly visible to other developers unless you unlist them. To host packages privately, see [Hosting packages](#).

Acquire your API key

1. [Sign into your nuget.org account](#) or create an account if you don't have one already.

For more information on creating your account, see [Individual accounts](#).

2. Select your user name (on the upper right), then select **API Keys**.
3. Select **Create**, provide a name for your key, select **Select Scopes > Push**. Enter * for **Glob pattern**, then select **Create**. (See below for more about scopes.)
4. Once the key is created, select **Copy** to retrieve the access key you need in the CLI:



5. **Important:** Save your key in a secure location because you cannot copy the key again later on. If you return to the API key page, you need to regenerate the key to copy it. You can also remove the API key if you no longer want to push packages via the CLI.

Scoping allows you to create separate API keys for different purposes. Each key has its expiration timeframe and can be scoped to specific packages (or glob patterns). Each key is also scoped to specific operations: push of new packages and updates, push of updates only, or delisting. Through scoping, you can create API keys for different people who manage packages for your organization such that they have only the permissions they need. For more information, see [scoped API keys](#).

Publish with dotnet nuget push

1. Change to the folder containing the `.nupkg` file.
2. Run the following command, specifying your package name (unique package ID) and replacing the key value with your API key:

```
dotnet nuget push AppLogger.1.0.0.nupkg -k qz2jga8pl3dvn2akksyquwcs9ygggg4exypy3bhxy6w6x6 -s https://api.nuget.org/v3/index.json
```

3. dotnet displays the results of the publishing process:

```
info : Pushing AppLogger.1.0.0.nupkg to 'https://www.nuget.org/api/v2/package'...
info :   PUT https://www.nuget.org/api/v2/package/
info :   Created https://www.nuget.org/api/v2/package/ 12620ms
info : Your package was pushed.
```

See [dotnet nuget push](#).

Publish errors

Errors from the `push` command typically indicate the problem. For example, you may have forgotten to update the version number in your project and are therefore trying to publish a package that already exists.

You also see errors when trying to publish a package using an identifier that already exists on the host. The name "AppLogger", for example, already exists. In such a case, the `push` command gives the following error:

```
Response status code does not indicate success: 403 (The specified API key is invalid, has expired, or does not have permission to access the specified package.).
```

If you're using a valid API key that you just created, then this message indicates a naming conflict, which isn't entirely clear from the "permission" part of the error. Change the package identifier, rebuild the project, recreate the `.nupkg` file, and retry the `push` command.

Manage the published package

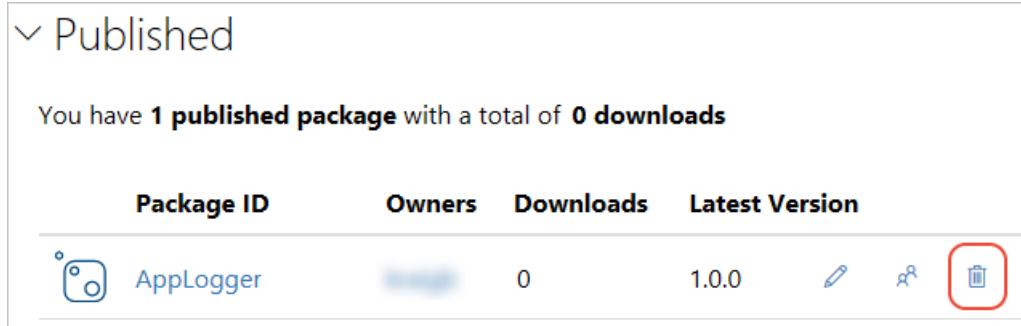
From your profile on nuget.org, select **Manage Packages** to see the one you just published. You also receive a confirmation email. Note that it might take a while for your package to be indexed and appear in search results where others can find it. During that time your package page shows the message below:

⚠ This package has not been published yet. It will appear in search results and will be available for install/restore after both validation and indexing are complete. Package validation and indexing may take up to an hour. [Read more](#).

And that's it! You've just published your first NuGet package to nuget.org that other developers can use in their own projects.

If in this walkthrough you created a package that isn't actually useful (such as a package created with an empty class library), you should *unlist* the package to hide it from search results:

1. On nuget.org, select your user name (upper right of the page), then select **Manage Packages**.
2. Locate the package you want to unlist under **Published** and select the trash can icon on the right:

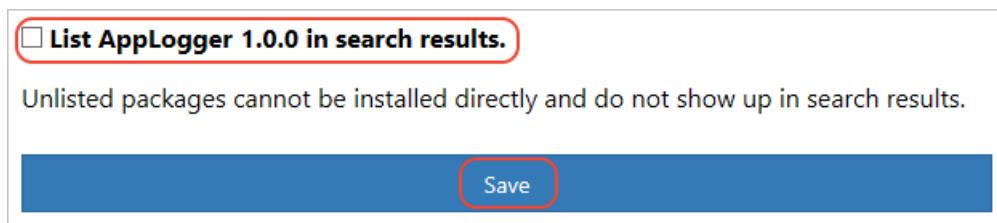


✓ Published

You have **1 published package** with a total of **0 downloads**

Package ID	Owners	Downloads	Latest Version	Actions
AppLogger	[redacted]	0	1.0.0	  

3. On the subsequent page, clear the box labeled **List (package-name) in search results** and select **Save**:



List AppLogger 1.0.0 in search results.

Unlisted packages cannot be installed directly and do not show up in search results.

Save

Next steps

Congratulations on creating your first NuGet package!

[Create a Package](#)

To explore more that NuGet has to offer, select the links below.

- [Publish a Package](#)
- [Pre-release Packages](#)
- [Support multiple target frameworks](#)
- [Package versioning](#)
- [Creating localized packages](#)
- [Creating symbol packages](#)
- [Signing packages](#)

Quickstart: Create and publish a NuGet package using Visual Studio (.NET Standard, Windows only)

11/5/2019 • 9 minutes to read • [Edit Online](#)

It's a simple process to create a NuGet package from a .NET Standard Class Library in Visual Studio on Windows, and then publish it to nuget.org using a CLI tool.

NOTE

If you are using Visual Studio for Mac, refer to [this information](#) on creating a NuGet package, or use the [dotnet CLI tools](#).

Prerequisites

1. Install any edition of Visual Studio 2019 from [visualstudio.com](#) with a .NET Core related workload.
2. If it's not already installed, install the `dotnet` CLI.

For the `dotnet` CLI, starting in Visual Studio 2017, the `dotnet` CLI is automatically installed with any .NET Core related workloads. Otherwise, install the [.NET Core SDK](#) to get the `dotnet` CLI. The `dotnet` CLI is required for .NET Standard projects that use the [SDK-style format](#) (SDK attribute). The default .NET Standard class library template in Visual Studio 2017 and higher, which is used in this article, uses the SDK attribute.

IMPORTANT

If you are working with a non-SDK-style project, follow the procedures in [Create and publish a .NET Framework package \(Visual Studio\)](#) to create and publish the package instead. For this article, the `dotnet` CLI is recommended. Although you can publish any NuGet package using the `nuget.exe` CLI, some of the steps in this article are specific to SDK-style projects and the dotnet CLI. The `nuget.exe` CLI is used for [non-SDK-style projects](#) (typically .NET Framework).

3. [Register for a free account on nuget.org](#) if you don't have one already. Creating a new account sends a confirmation email. You must confirm the account before you can upload a package.

Create a class library project

You can use an existing .NET Standard Class Library project for the code you want to package, or create a simple one as follows:

1. In Visual Studio, choose **File > New > Project**, expand the **Visual C# > .NET Standard** node, select the "Class Library (.NET Standard)" template, name the project AppLogger, and click **OK**.

TIP

Unless you have a reason to choose otherwise, .NET Standard is the preferred target for NuGet packages, as it provides compatibility with the widest range of consuming projects.

2. Right-click on the resulting project file and select **Build** to make sure the project was created properly. The DLL is found within the Debug folder (or Release if you build that configuration instead).

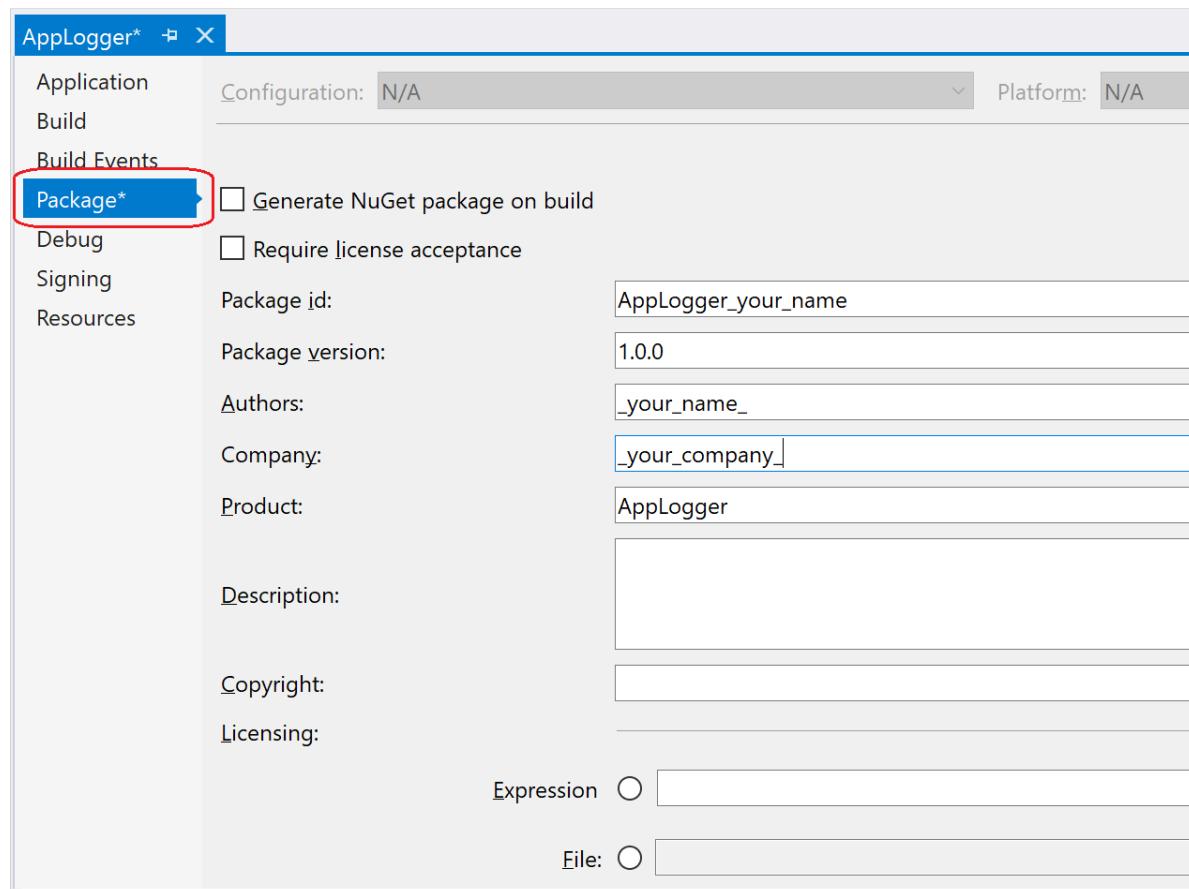
Within a real NuGet package, of course, you implement many useful features with which others can build applications. For this walkthrough, however, you won't write any additional code because a class library from the template is sufficient to create a package. Still, if you'd like some functional code for the package, use the following:

```
namespace AppLogger
{
    public class Logger
    {
        public void Log(string text)
        {
            Console.WriteLine(text);
        }
    }
}
```

Configure package properties

1. Right-click the project in Solution Explorer, and choose **Properties** menu command, then select the **Package** tab.

The **Package** tab appears only for SDK-style projects in Visual Studio, typically .NET Standard or .NET Core class library projects; if you are targeting a non-SDK style project (typically .NET Framework), either [migrate the project](#) or see [Create and publish a .NET Framework package](#) instead for step-by-step instructions.



NOTE

For packages built for public consumption, pay special attention to the **Tags** property, as tags help others find your package and understand what it does.

2. Give your package a unique identifier and fill out any other desired properties. For a mapping of MSBuild properties (SDK-style project) to properties in a `.nuspec`, see [pack targets](#). For descriptions of properties, see the [.nuspec file reference](#). All of the properties here go into the `.nuspec` manifest that Visual Studio creates for the project.

IMPORTANT

You must give the package an identifier that's unique across nuget.org or whatever host you're using. For this walkthrough we recommend including "Sample" or "Test" in the name as the later publishing step does make the package publicly visible (though it's unlikely anyone will actually use it).

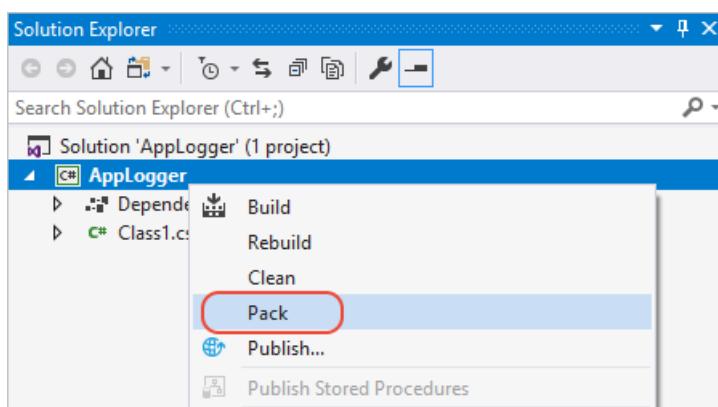
If you attempt to publish a package with a name that already exists, you see an error.

3. (Optional) To see the properties directly in the project file, right-click the project in Solution Explorer and select **Edit AppLogger.csproj**.

This option is only available starting in Visual Studio 2017 for projects that use the SDK-style attribute. Otherwise, right-click the project and choose **Unload Project**. Then right-click the unloaded project and choose **Edit AppLogger.csproj**.

Run the pack command

1. Set the configuration to **Release**.
2. Right click the project in **Solution Explorer** and select the **Pack** command:



If you don't see the **Pack** command, your project is probably not an SDK-style project and you need to use the `nuget.exe` CLI. Either [migrate the project](#) and use `dotnet` CLI, or see [Create and publish a .NET Framework package](#) instead for step-by-step instructions.

3. Visual Studio builds the project and creates the `.nupkg` file. Examine the **Output** window for details (similar to the following), which contains the path to the package file. Note also that the built assembly is in `bin\Release\netstandard2.0` as befits the .NET Standard 2.0 target.

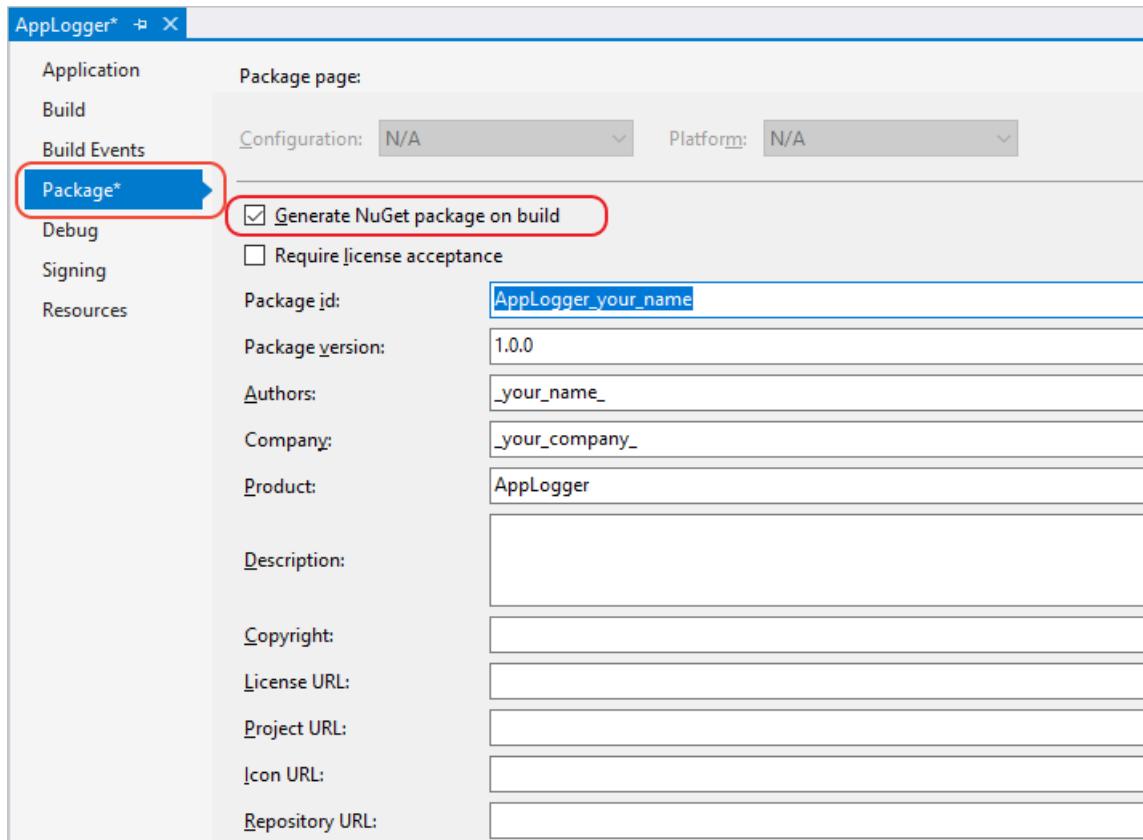
```
1>----- Build started: Project: AppLogger, Configuration: Release Any CPU -----
1>AppLogger -> d:\proj\AppLogger\AppLogger\bin\Release\netstandard2.0\AppLogger.dll
1>Successfully created package 'd:\proj\AppLogger\AppLogger\bin\Release\AppLogger.1.0.0.nupkg'.
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

(Optional) Generate package on build

You can configure Visual Studio to automatically generate the NuGet package when you build the project.

1. In Solution Explorer, right-click the project and choose **Properties**.

2. In the **Package** tab, select **Generate NuGet package on build**.



NOTE

When you automatically generate the package, the time to pack increases the build time for your project.

(Optional) pack with MSBuild

As an alternate to using the **Pack** menu command, NuGet 4.x+ and MSBuild 15.1+ supports a `pack` target when the project contains the necessary package data. Open a command prompt, navigate to your project folder and run the following command. (You typically want to start the "Developer Command Prompt for Visual Studio" from the Start menu, as it will be configured with all the necessary paths for MSBuild.)

For more information, see [Create a package using MSBuild](#).

Publish the package

Once you have a `.nupkg` file, you publish it to nuget.org using either the `nuget.exe` CLI or the `dotnet.exe` CLI along with an API key acquired from nuget.org.

NOTE

Virus scanning: All packages uploaded to nuget.org are scanned for viruses and rejected if any viruses are found. All packages listed on nuget.org are also scanned periodically.

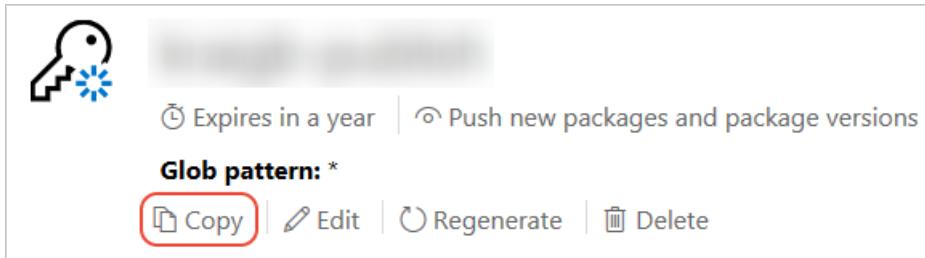
Packages published to nuget.org are also publicly visible to other developers unless you unlist them. To host packages privately, see [Hosting packages](#).

Acquire your API key

1. [Sign into your nuget.org account](#) or create an account if you don't have one already.

For more information on creating your account, see [Individual accounts](#).

2. Select your user name (on the upper right), then select **API Keys**.
3. Select **Create**, provide a name for your key, select **Select Scopes > Push**. Enter * for **Glob pattern**, then select **Create**. (See below for more about scopes.)
4. Once the key is created, select **Copy** to retrieve the access key you need in the CLI:



5. **Important:** Save your key in a secure location because you cannot copy the key again later on. If you return to the API key page, you need to regenerate the key to copy it. You can also remove the API key if you no longer want to push packages via the CLI.

Scoping allows you to create separate API keys for different purposes. Each key has its expiration timeframe and can be scoped to specific packages (or glob patterns). Each key is also scoped to specific operations: push of new packages and updates, push of updates only, or delisting. Through scoping, you can create API keys for different people who manage packages for your organization such that they have only the permissions they need. For more information, see [scoped API keys](#).

Publish with the dotnet CLI or nuget.exe CLI

Select the tab for your CLI tool, either **.NET Core CLI** (dotnet CLI) or **NuGet** (nuget.exe CLI).

- [.NET Core CLI](#)
- [NuGet](#)

This step is the recommended alternative to using `nuget.exe`.

Before you can publish the package, you must first open a command line.

1. Change to the folder containing the `.nupkg` file.
2. Run the following command, specifying your package name (unique package ID) and replacing the key value with your API key:

```
dotnet nuget push AppLogger.1.0.0.nupkg -k qz2jga8pl3dvn2akkysquwcs9ygggg4exypy3bhxy6w6x6 -s https://api.nuget.org/v3/index.json
```

3. dotnet displays the results of the publishing process:

```
info : Pushing AppLogger.1.0.0.nupkg to 'https://www.nuget.org/api/v2/package'...
info :  PUT https://www.nuget.org/api/v2/package/
info :  Created https://www.nuget.org/api/v2/package/ 12620ms
info : Your package was pushed.
```

See [dotnet nuget push](#).

Publish errors

Errors from the `push` command typically indicate the problem. For example, you may have forgotten to update the version number in your project and are therefore trying to publish a package that already exists.

You also see errors when trying to publish a package using an identifier that already exists on the host. The name

"AppLogger", for example, already exists. In such a case, the `push` command gives the following error:

```
Response status code does not indicate success: 403 (The specified API key is invalid, has expired, or does not have permission to access the specified package.).
```

If you're using a valid API key that you just created, then this message indicates a naming conflict, which isn't entirely clear from the "permission" part of the error. Change the package identifier, rebuild the project, recreate the `.nupkg` file, and retry the `push` command.

Manage the published package

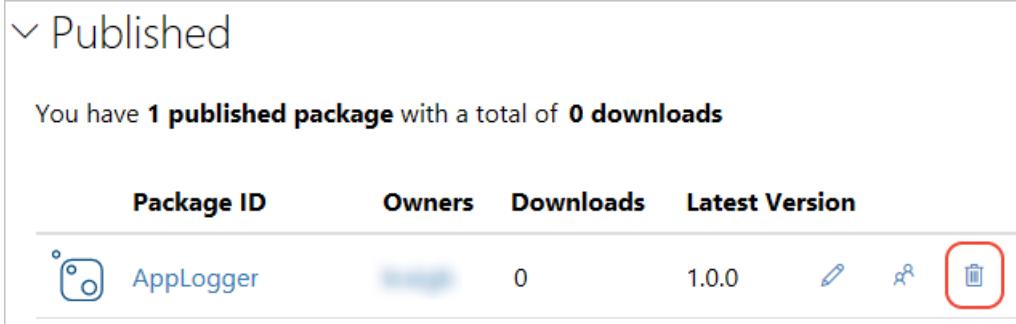
From your profile on nuget.org, select **Manage Packages** to see the one you just published. You also receive a confirmation email. Note that it might take a while for your package to be indexed and appear in search results where others can find it. During that time your package page shows the message below:

⚠ This package has not been published yet. It will appear in search results and will be available for install/restore after both validation and indexing are complete. Package validation and indexing may take up to an hour. [Read more](#).

And that's it! You've just published your first NuGet package to nuget.org that other developers can use in their own projects.

If in this walkthrough you created a package that isn't actually useful (such as a package created with an empty class library), you should *unlist* the package to hide it from search results:

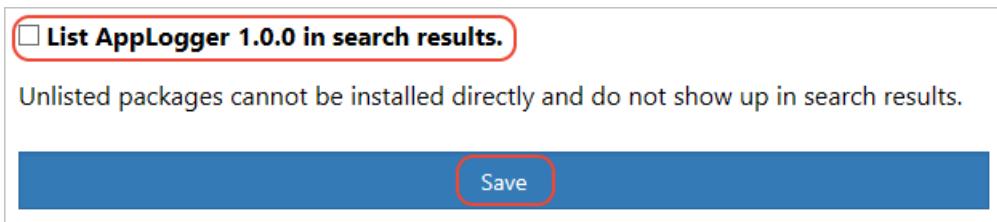
1. On nuget.org, select your user name (upper right of the page), then select **Manage Packages**.
2. Locate the package you want to unlist under **Published** and select the trash can icon on the right:



The screenshot shows the 'Published' section of a user's profile on nuget.org. It displays a single package named 'AppLogger'. The table includes columns for Package ID, Owners, Downloads, and Latest Version. The 'Downloads' column shows 0. The 'Latest Version' column shows 1.0.0. To the right of the package row is a trash can icon, which is circled in red to indicate it is the target for unlisting.

Package ID	Owners	Downloads	Latest Version
AppLogger	[redacted]	0	1.0.0

3. On the subsequent page, clear the box labeled **List (package-name) in search results** and select **Save**:



The screenshot shows a confirmation page for unlisting the package. It features a checkbox labeled 'List AppLogger 1.0.0 in search results.' which is unchecked. Below the checkbox, a message states: 'Unlisted packages cannot be installed directly and do not show up in search results.' At the bottom is a blue 'Save' button.

List AppLogger 1.0.0 in search results.

Unlisted packages cannot be installed directly and do not show up in search results.

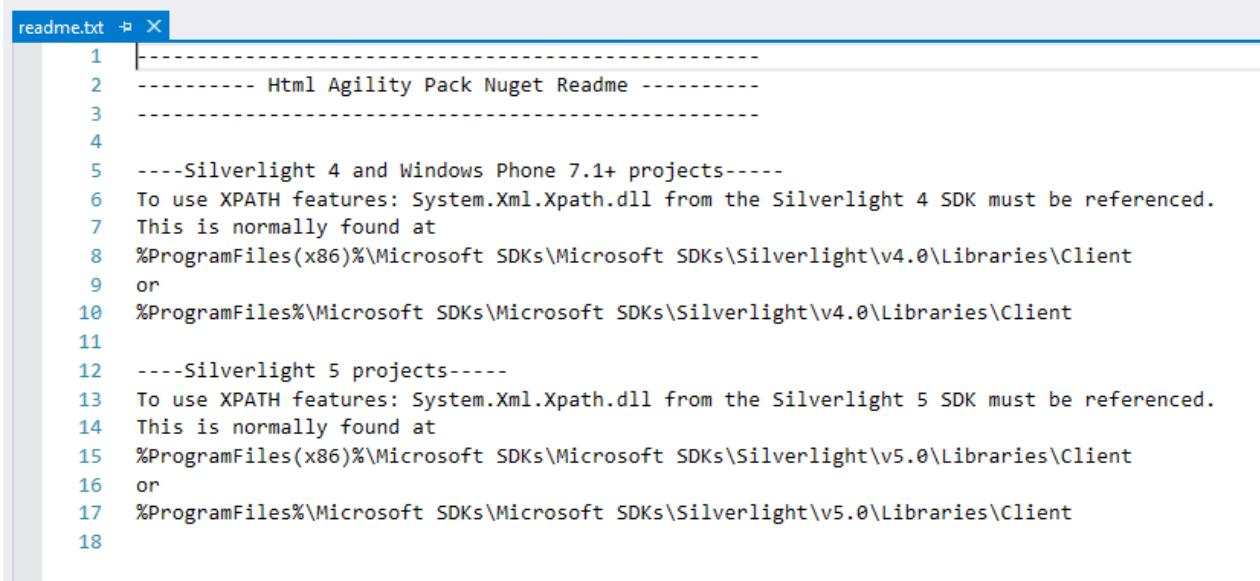
Save

Adding a readme and other files

To directly specify files to include in the package, edit the project file and use the `content` property:

```
<ItemGroup>
  <Content Include="readme.txt">
    <Pack>true</Pack>
    <PackagePath>\</PackagePath>
  </Content>
</ItemGroup>
```

This will include a file named `readme.txt` in the package root. Visual Studio displays the contents of that file as plain text immediately after installing the package directly. (Readme files are not displayed for packages installed as dependencies). For example, here's how the readme for the `HtmlAgilityPack` package appears:



```
1  -----
2  ----- Html Agility Pack Nuget Readme -----
3  -----
4
5  ----Silverlight 4 and Windows Phone 7.1+ projects-----
6  To use XPATH features: System.Xml.XPath.dll from the Silverlight 4 SDK must be referenced.
7  This is normally found at
8  %ProgramFiles(x86)%\Microsoft SDKs\Microsoft SDKs\Silverlight\v4.0\Libraries\Client
9  or
10 %ProgramFiles%\Microsoft SDKs\Microsoft SDKs\Silverlight\v4.0\Libraries\Client
11
12 ----Silverlight 5 projects-----
13 To use XPATH features: System.Xml.XPath.dll from the Silverlight 5 SDK must be referenced.
14 This is normally found at
15 %ProgramFiles(x86)%\Microsoft SDKs\Microsoft SDKs\Silverlight\v5.0\Libraries\Client
16 or
17 %ProgramFiles%\Microsoft SDKs\Microsoft SDKs\Silverlight\v5.0\Libraries\Client
18
```

NOTE

Merely adding the `readme.txt` at the project root will not result in it being included in the resulting package.

Related topics

- [Create a Package](#)
- [Publish a Package](#)
- [Pre-release Packages](#)
- [Support multiple target frameworks](#)
- [Package versioning](#)
- [Creating localized packages](#)
- [.NET Standard Library documentation](#)
- [Porting to .NET Core from .NET Framework](#)

Quickstart: Create and publish a package using Visual Studio (.NET Framework, Windows)

10/15/2019 • 8 minutes to read • [Edit Online](#)

Creating a NuGet package from a .NET Framework Class Library involves creating the DLL in Visual Studio on Windows, then using the `nuget.exe` command line tool to create and publish the package.

NOTE

This Quickstart applies to Visual Studio 2017 and higher versions for Windows only. Visual Studio for Mac does not include the capabilities described here. Use the [dotnet CLI tools](#) instead.

Prerequisites

1. Install any edition of Visual Studio 2017 or higher from [visualstudio.com](#) with any .NET-related workload. Visual Studio 2017 automatically includes NuGet capabilities when a .NET workload is installed.
2. Install the `nuget.exe` CLI by downloading it from [nuget.org](#), saving that `.exe` file to a suitable folder, and adding that folder to your PATH environment variable.
3. [Register for a free account on nuget.org](#) if you don't have one already. Creating a new account sends a confirmation email. You must confirm the account before you can upload a package.

Create a class library project

You can use an existing .NET Framework Class Library project for the code you want to package, or create a simple one as follows:

1. In Visual Studio, choose **File > New > Project**, select the **Visual C#** node, select the "Class Library (.NET Framework)" template, name the project AppLogger, and click **OK**.
2. Right-click on the resulting project file and select **Build** to make sure the project was created properly. The DLL is found within the Debug folder (or Release if you build that configuration instead).

Within a real NuGet package, of course, you implement many useful features with which others can build applications. You can also set the target frameworks however you like. For example, see the guides for [UWP](#) and [Xamarin](#).

For this walkthrough, however, you won't write any additional code because a class library from the template is sufficient to create a package. Still, if you'd like some functional code for the package, use the following:

```
using System;

namespace AppLogger
{
    public class Logger
    {
        public void Log(string text)
        {
            Console.WriteLine(text);
        }
    }
}
```

TIP

Unless you have a reason to choose otherwise, .NET Standard is the preferred target for NuGet packages, as it provides compatibility with the widest range of consuming projects. See [Create and publish a package using Visual Studio \(.NET Standard\)](#).

Configure project properties for the package

A NuGet package contains a manifest (a `.nuspec` file), that contains relevant metadata such as the package identifier, version number, description, and more. Some of these can be drawn from the project properties directly, which avoids having to separately update them in both the project and the manifest. This section describes where to set the applicable properties.

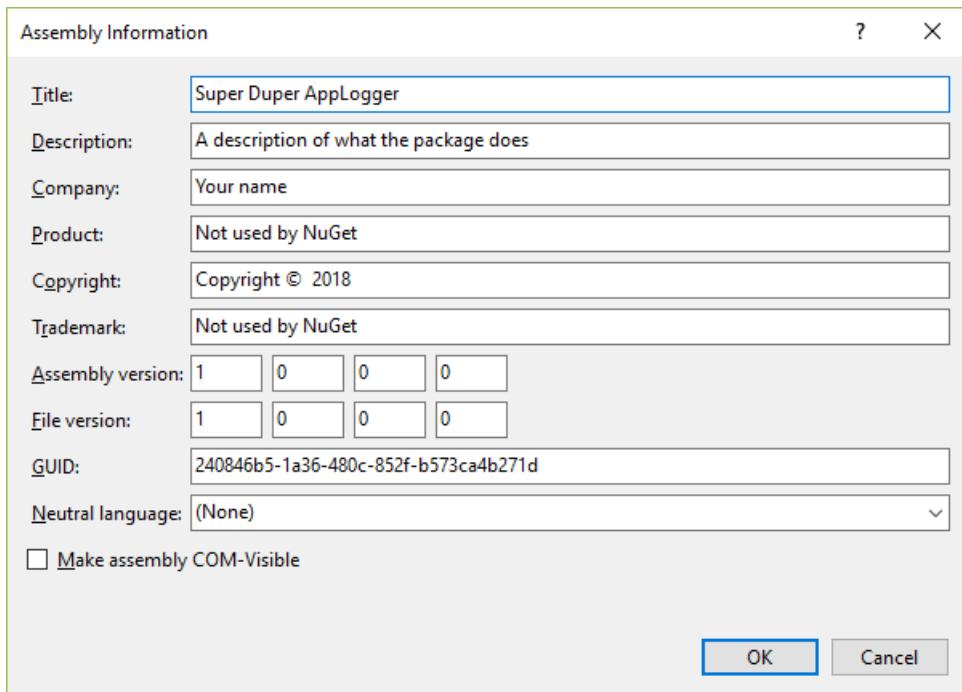
1. Select the **Project > Properties** menu command, then select the **Application** tab.
2. In the **Assembly name** field, give your package a unique identifier.

IMPORTANT

You must give the package an identifier that's unique across nuget.org or whatever host you're using. For this walkthrough we recommend including "Sample" or "Test" in the name as the later publishing step does make the package publicly visible (though it's unlikely anyone will actually use it).

If you attempt to publish a package with a name that already exists, you see an error.

3. Select the **Assembly Information...** button, which brings up a dialog box in which you can enter other properties that carry into the manifest (see [.nuspec file reference - replacement tokens](#)). The most commonly used fields are **Title**, **Description**, **Company**, **Copyright**, and **Assembly version**. These properties ultimately appear with your package on a host like nuget.org, so make sure they're fully descriptive.



4. Optional: to see and edit the properties directly, open the `Properties/AssemblyInfo.cs` file in the project.
5. When the properties are set, set the project configuration to **Release** and rebuild the project to generate the updated DLL.

Generate the initial manifest

With a DLL in hand and project properties set, you now use the `nuget spec` command to generate an initial `.nuspec` file from the project. This step includes the relevant replacement tokens to draw information from the project file.

You run `nuget spec` only once to generate the initial manifest. When updating the package, you either change values in your project or edit the manifest directly.

1. Open a command prompt and navigate to the project folder containing `AppLogger.csproj` file.
2. Run the following command: `nuget spec AppLogger.csproj`. By specifying a project, NuGet creates a manifest that matches the name of the project, in this case `AppLogger.nuspec`. It also include replacement tokens in the manifest.
3. Open `AppLogger.nuspec` in a text editor to examine its contents, which should appear as follows:

```
<?xml version="1.0"?>
<package>
  <metadata>
    <id>Package</id>
    <version>1.0.0</version>
    <authors>YourUsername</authors>
    <owners>YourUsername</owners>
    <license type="expression">MIT</license>
    <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>
    <releaseNotes>Summary of changes made in this release of the package.</releaseNotes>
    <copyright>Copyright 2019</copyright>
    <tags>Tag1 Tag2</tags>
  </metadata>
</package>
```

Edit the manifest

1. NuGet produces an error if you try to create a package with default values in your `.nuspec` file, so you must edit the following fields before proceeding. See [.nuspec file reference - optional metadata elements](#) for a description of how these are used.
 - `licenseUrl`
 - `projectUrl`
 - `iconUrl`
 - `releaseNotes`
 - `tags`
2. For packages built for public consumption, pay special attention to the **Tags** property, as tags help others find your package on sources like nuget.org and understand what it does.
3. You can also add any other elements to the manifest at this time, as described on [.nuspec file reference](#).
4. Save the file before proceeding.

Run the pack command

1. From a command prompt in the folder containing your `.nuspec` file, run the command `nuget pack`.
2. NuGet generates a `.nupkg` file in the form of *identifier-version.nupkg*, which you'll find in the current folder.

Publish the package

Once you have a `.nupkg` file, you publish it to nuget.org using `nuget.exe` with an API key acquired from nuget.org. For nuget.org you must use `nuget.exe` 4.1.0 or higher.

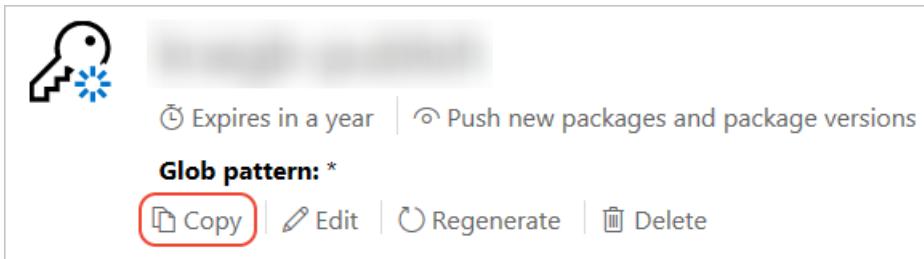
NOTE

Virus scanning: All packages uploaded to nuget.org are scanned for viruses and rejected if any viruses are found. All packages listed on nuget.org are also scanned periodically.

Packages published to nuget.org are also publicly visible to other developers unless you unlist them. To host packages privately, see [Hosting packages](#).

Acquire your API key

1. [Sign into your nuget.org account](#) or create an account if you don't have one already.
For more information on creating your account, see [Individual accounts](#).
2. Select your user name (on the upper right), then select **API Keys**.
3. Select **Create**, provide a name for your key, select **Select Scopes > Push**. Enter * for **Glob pattern**, then select **Create**. (See below for more about scopes.)
4. Once the key is created, select **Copy** to retrieve the access key you need in the CLI:



5. **Important:** Save your key in a secure location because you cannot copy the key again later on. If you return to the API key page, you need to regenerate the key to copy it. You can also remove the API key if you no longer want to push packages via the CLI.

Scoping allows you to create separate API keys for different purposes. Each key has its expiration timeframe and can be scoped to specific packages (or glob patterns). Each key is also scoped to specific operations: push of new packages and updates, push of updates only, or delisting. Through scoping, you can create API keys for different people who manage packages for your organization such that they have only the permissions they need. For more information, see [scoped API keys](#).

Publish with `nuget push`

1. Open a command line and change to the folder containing the `.nupkg` file.
2. Run the following command, specifying your package name and replacing the key value with your API key:

```
nuget push AppLogger.1.0.0.nupkg qz2jga8pl3dvn2akksyquwcs9ygggg4exypy3bhxy6w6x6 -Source  
https://api.nuget.org/v3/index.json
```

3. `nuget.exe` displays the results of the publishing process:

```
Pushing AppLogger.1.0.0.nupkg to 'https://www.nuget.org/api/v2/package'...  
PUT https://www.nuget.org/api/v2/package/  
Created https://www.nuget.org/api/v2/package/ 6829ms  
Your package was pushed.
```

See [nuget push](#).

Publish errors

Errors from the `push` command typically indicate the problem. For example, you may have forgotten to update the version number in your project and are therefore trying to publish a package that already exists.

You also see errors when trying to publish a package using an identifier that already exists on the host. The name "AppLogger", for example, already exists. In such a case, the `push` command gives the following error:

```
Response status code does not indicate success: 403 (The specified API key is invalid,  
has expired, or does not have permission to access the specified package.).
```

If you're using a valid API key that you just created, then this message indicates a naming conflict, which isn't entirely clear from the "permission" part of the error. Change the package identifier, rebuild the project, recreate the `.nupkg` file, and retry the `push` command.

Manage the published package

From your profile on nuget.org, select **Manage Packages** to see the one you just published. You also receive a confirmation email. Note that it might take a while for your package to be indexed and appear in search results where others can find it. During that time your package page shows the message below:

⚠ This package has not been published yet. It will appear in search results and will be available for install/restore after both validation and indexing are complete. Package validation and indexing may take up to an hour. [Read more](#).

And that's it! You've just published your first NuGet package to nuget.org that other developers can use in their own projects.

If in this walkthrough you created a package that isn't actually useful (such as a package created with an empty class library), you should *unlist* the package to hide it from search results:

1. On nuget.org, select your user name (upper right of the page), then select **Manage Packages**.
2. Locate the package you want to unlist under **Published** and select the trash can icon on the right:

✓ Published

You have **1 published package** with a total of **0 downloads**

Package ID	Owners	Downloads	Latest Version	
 AppLogger		0	1.0.0	 

3. On the subsequent page, clear the box labeled **List (package-name) in search results** and select **Save**:

List AppLogger 1.0.0 in search results.

Unlisted packages cannot be installed directly and do not show up in search results.

Save

Next steps

Congratulations on creating your first NuGet package!

[Create a Package](#)

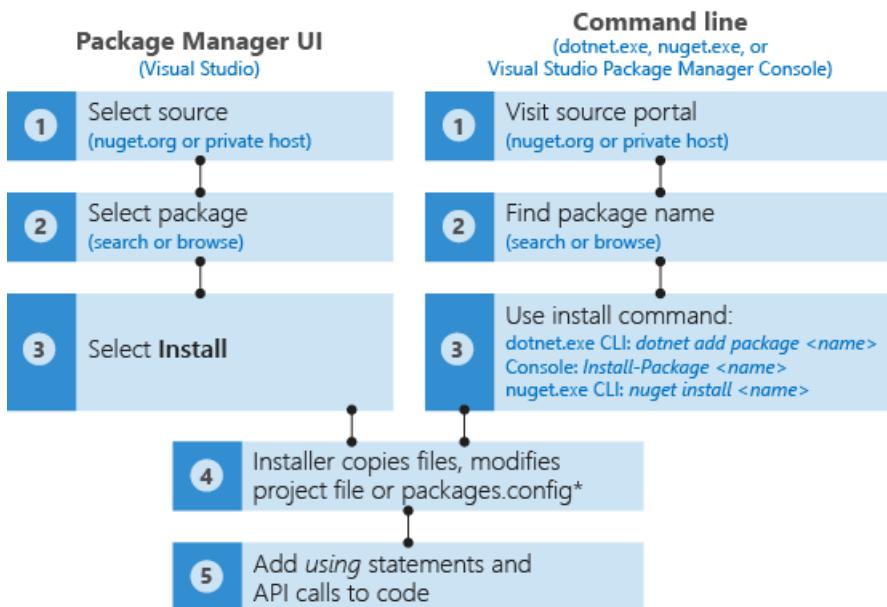
To explore more that NuGet has to offer, select the links below.

- [Publish a Package](#)
- [Pre-release Packages](#)
- [Support multiple target frameworks](#)
- [Package versioning](#)
- [Creating localized packages](#)

Package consumption workflow

8/15/2019 • 3 minutes to read • [Edit Online](#)

Between nuget.org and private package galleries that your organization might establish, you can find tens of thousands of highly useful packages to use in your apps and services. But regardless of the source, consuming a package follows the same general workflow.



* Visual Studio and `dotnet.exe` only. The `nuget install` command does not modify project files or the `packages.config` file; entries must be managed manually.

For further details, see [Finding and Choosing Packages](#) and [What happens when a package is installed?](#).

NuGet remembers the identity and version number of each installed package, recording it in either the project file (using [PackageReference](#)) or [packages.config](#), depending on project type and your version of NuGet. With NuGet 4.0+, [PackageReference](#) is preferred, although this is configurable in Visual Studio through the [Package Manager UI](#). In any case, you can look in the appropriate file at any time to see the full list of dependencies for your project.

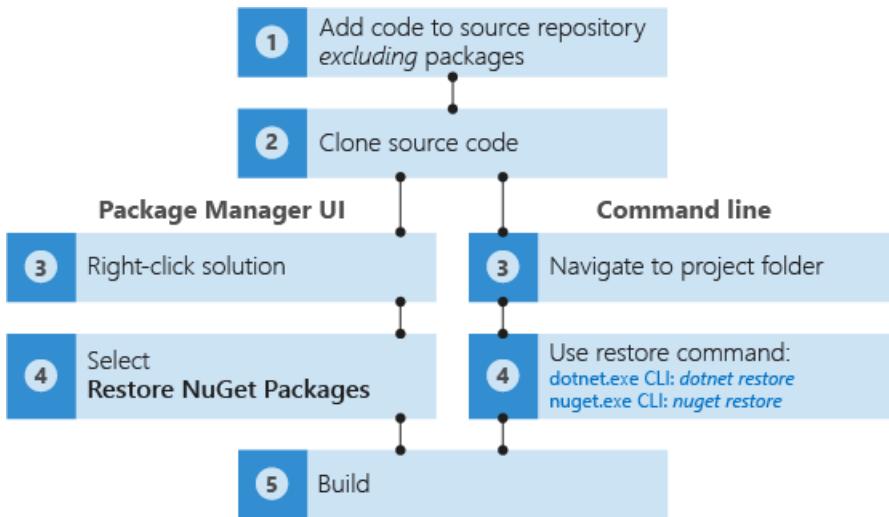
TIP

It's prudent to always check the license for each package you intend to use in your software. On nuget.org, you find a [License Info](#) link on the right side of each package's description page. If a package does not specify license terms, contact the package owner directly using the [Contact owners](#) link on the package page. Microsoft does not license any intellectual property to you from third party package providers and is not responsible for information provided by third parties.

When installing packages, NuGet typically checks if the package is already available from its cache. You can manually clear this cache from the command line, as described on [Managing the global packages and cache folders](#).

NuGet also makes sure that the target frameworks supported by the package are compatible with your project. If the package does not contain compatible assemblies, NuGet displays an error. See [Resolving incompatible package errors](#).

When adding project code to a source repository, you typically don't include NuGet packages. Those who later clone the repository or otherwise acquire the project, including build agents on systems like Visual Studio Team Services, must restore the necessary packages prior to running a build:



Package Restore uses the information in the project file or `packages.config` to reinstall all dependencies. Note that there are differences in the process involved, as described in [Dependency Resolution](#). Also, the diagram above does not show a restore command for the Package Manager Console because if you're with the Console you're already in the context of Visual Studio, which typically restores packages automatically and provides the solution-level command as shown.

Occasionally it's necessary to reinstall packages that are already included in a project, which may also reinstall dependencies. This is easy to do using the `nuget reinstall` command or the NuGet Package Manager Console. For details, see [Reinstalling and Updating Packages](#).

Finally, NuGet's behavior is driven by `Nuget.Config` files. Multiple files can be used to centralize certain settings at different levels, as explained in [Configuring NuGet Behavior](#).

Ways to install a NuGet Package

NuGet packages are downloaded and installed using any of the methods in the following table.

TOOL	DESCRIPTION
dotnet.exe CLI	(All platforms) CLI tool for .NET Core and .NET Standard libraries, and for SDK-style projects that target .NET Framework (see SDK attribute). Retrieves the package identified by <code><package_name></code> and adds a reference to the project file. Also retrieves and installs dependencies.
Visual Studio	(Windows and Mac) Provides a UI through which you can browse, select, and install packages and their dependencies into a project from a specified package source. Adds references to installed packages to the project file. <ul style="list-style-type: none"> • Install and manage packages using Visual Studio • Including a NuGet package in your project (Mac)
Package Manager Console (Visual Studio)	(Windows only) Retrieves and installs the package identified by <code><package_name></code> from a selected source into a specified project in the solution, then adds a reference to the project file. Also retrieves and installs dependencies.

TOOL	DESCRIPTION
nuget.exe CLI	(All platforms) CLI tool for .NET Framework libraries and non-SDK-style projects that target .NET Standard libraries. Retrieves the package identified by <package_name> and expands its contents into a folder in the current directory; can also retrieve all packages listed in a <code>packages.config</code> file. Also retrieves and installs dependencies, but makes no changes to project files or <code>packages.config</code> .

Finding and evaluating NuGet packages for your project

11/20/2019 • 9 minutes to read • [Edit Online](#)

When starting any .NET project, or whenever you identify a functional need for your app or service, you can save yourself lots of time and trouble by using existing NuGet packages that fulfill that need. These packages can come from the public collection on [nuget.org](#), or a private source that's provided by your organization or another third party.

Finding packages

When you visit [nuget.org](#) or open the Package Manager UI in Visual Studio, you see a list of packages sorted by total downloads. This immediately shows you the most widely-used packages across the millions of .NET projects. There's a good chance, then, that at least some of the packages listed on the first few pages will be useful in your projects.

There are 107,603 packages **Include prerelease**

 Newtonsoft.Json  <small>by: jamesnk</small> Json.NET <small>↓ 109,214,262 total downloads ⏲ last updated a month ago ⚡ Latest version: 11.0.1 ⚡ json</small> <small>Json.NET is a popular high-performance JSON framework for .NET</small>
 NUnit  <small>by: rprouse charliepoole</small> <small>↓ 18,733,362 total downloads ⏲ last updated 4 days ago ⚡ Latest version: 3.10.1 ⚡ nunit test testing tdd framework f...</small> <small>NUnit features a fluent assert syntax, parameterized, generic and theory tests and is user-extensible. This package includes the NUnit 3 framework assembly, which is referenced by your tests. You will need to install version 3 of the nunit3-console program or a third-party runner that supports... More information</small>
 EntityFramework  <small>by: microsoft EntityFramework aspnet</small> <small>↓ 40,924,086 total downloads ⏲ last updated 5 months ago ⚡ Latest version: 6.2.0 ⚡ Microsoft EntityFramework EF D...</small> <small>Entity Framework is Microsoft's recommended data access technology for new applications.</small>

Notice the **Include prerelease** option on the upper right of the page. When selected, [nuget.org](#) shows all versions of packages including beta and other early releases. To show only stable releases, clear the option.

For specific needs, searching by tags (within the Visual Studio Package Manager or on a portal like [nuget.org](#)) is the most common means of discovering a suitable package. For example, searching on "json" lists all NuGet packages that are tagged with that keyword and thus have some relationship to the JSON data format.

2,335 packages returned for json

Include prerelease



[JSON](#) by: conatuscreative

↓ 99,788 total downloads | ⏲ last updated 5/31/2011 | ⚡ Latest version: 1.0.1 | ⚡ JSON parser
A JSON parser in C#, supporting dynamic deserialization in .NET 4.0



[Newtonsoft.Json](#) by: jamesnk

Json.NET
↓ 109,214,262 total downloads | ⏲ last updated a month ago | ⚡ Latest version: 11.0.1 | ⚡ json
Json.NET is a popular high-performance JSON framework for .NET



[System.IdentityModel.Tokens.Jwt](#) by: microsoft AzureAD

↓ 13,002,260 total downloads | ⏲ last updated a month ago | ⚡ Latest version: 5.2.1 | ⚡ .NET Windows Authentication Id...
Includes types that provide support for creating, serializing and validating JSON Web Tokens.



[json-serialize](#) by: kmalakoff

↓ 21,898 total downloads | ⏲ last updated 9/17/2012 | ⚡ Latest version: 1.1.2 | ⚡ json-serialize json-serializejs json util se...
JSON-Serialize.js provides conventions and helpers to manage serialization and deserialization of instances to/from JSON.

You can also search using the package ID, if you know it. See [Search Syntax](#) below.

At this time, search results are sorted only by relevance, so you generally want to look through at least the first few pages of results for packages that suit your needs, or refine your search terms to be more specific.

Does the package support my project's target framework?

NuGet installs a package into a project only if that package's supported frameworks include the project's target framework. If the package is not compatible, NuGet issues an error.

Some packages list their supported frameworks directly in the nuget.org gallery, but because such data is not required, many packages do not include that list. At present there is no means to search nuget.org for packages that support a specific target framework (the feature is under consideration, see [NuGet Issue 2936](#)).

Fortunately, you can determine supported frameworks through two other means:

1. Attempt to install a package into a project using the [Install-Package](#) command in the NuGet Package Manager Console. If the package is incompatible, this command shows you the package's supported frameworks.
2. Download the package from its page on nuget.org using the **Manual download** link under **Info**. Change the extension from `.nupkg` to `.zip`, and open the file to examine the content of its `lib` folder. There you see subfolders for each of the supported frameworks, where each subfolder is named with a target framework moniker (TFM; see [Target Frameworks](#)). If you see no subfolders under `lib` and only a single DLL, then you must attempt to install the package in your project to discover its compatibility.

Pre-release packages

Many package authors make preview and beta releases available as they continue to make improvements and seek feedback on their latest revisions.

By default, nuget.org shows pre-release packages in search results. To search only stable releases, clear the **Include prerelease** option on the upper right of the page

2,380 packages returned for logging

Include prerelease



NLog by: 304NotModified jkowalski Xharze

↓ 11,417,261 total downloads | last updated 17 days ago | Latest version: 4.4.13 | NLog logging log tracing logfile...

NLog is a logging platform for .NET with rich log routing and management capabilities. It can help you produce and manage high-quality logs for your application regardless of its size or complexity. This package installs NLog.dll with includes core logging functionality. For your main project also... [More information](#)



logging by: conatuscreative apitize

↓ 3,211 total downloads | last updated 6/14/2013 | Latest version: 0.9.1 | logging

A simple logging abstraction. You don't have the option of using immediate-mode logging, so you never accidentally kill performance.



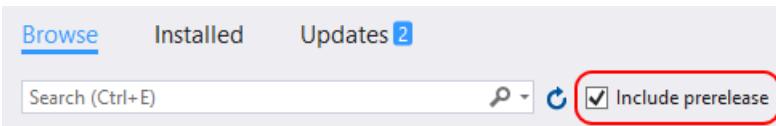
log4net by: cincura.net Apache.Logging

Apache log4net

↓ 14,841,688 total downloads | last updated 3/11/2017 | Latest version: 2.0.8 | logging log tracing logfiles

In Visual Studio, and when using the NuGet and dotnet CLI tools, NuGet does not include pre-release versions by default. To change this behavior, do the following steps:

- **Package Manager UI in Visual Studio:** In the **Manage NuGet Packages** UI, set the **Include prerelease** box. Setting or clearing this box refreshes the Package Manager UI and the list of available versions you can install.



- **Package Manager Console:** Use the `-IncludePrerelease` switch with the `Find-Package`, `Get-Package`, `Install-Package`, `Sync-Package`, and `Update-Package` commands. Refer to the [PowerShell Reference](#).
- **nuget.exe CLI:** Use the `-prerelease` switch with the `install`, `update`, `delete`, and `mirror` commands. Refer to the [NuGet CLI reference](#).
- **dotnet.exe CLI:** Specify the exact pre-release version using the `-v` argument. Refer to the [dotnet add package reference](#).

Native C++ packages

NuGet supports native C++ packages that can be used in C++ projects in Visual Studio. This enables the **Manage NuGet Packages** context-menu command for projects, introduces a `native` target framework, and provides MSBuild integration.

To find native packages on [nuget.org](#), search using `tag:native`. Such packages typically provide `.targets` and `.props` files, which NuGet imports automatically when the package is added to a project.

Evaluating packages

The best way to evaluate the usefulness of a package is to download it and try it out in your code (all packages on nuget.org are routinely scanned for viruses, by the way). After all, every highly popular package got started with only a few developers using it, and you might be one of the early adopters!

At the same time, using a NuGet package means taking a dependency on it, so you want to make sure it's robust and reliable. Because installing and directly testing a package is time-consuming, you can also learn a lot about a package's quality by using the information on a package's listing page:

- **Downloads statistics:** on the package page on nuget.org, the **Statistics** section shows total downloads, downloads of the most recent version, and average downloads per day. Larger numbers indicate that many

other developers have taken a dependency on the package, which means that it has proven itself.

Statistics

↓ 92,919,266 total downloads

↗ 10,344 downloads of latest version

↗ 36,785 downloads per day (avg)

[View full stats](#)

- *GitHub Usage:* on the package page, the **GitHub Usage** section lists public GitHub repositories that depend on this package and that have a high number of stars on GitHub. A GitHub repository's number of stars generally indicates how popular that repository is with GitHub users (more stars usually means more popular). Please visit [GitHub's Getting Started page](#) for more information on GitHub's star and repository ranking system.

▽ GitHub Usage

Showing the top 10 GitHub repositories that depend on Microsoft.AspNetCore.Mvc:

Repository	Stars
aspnetboilerplate/aspnetboilerplate ASP.NET Boilerplate - Web Application Framework	★ 7.0K
aspnet/AspNetCore.Docs Documentation for ASP.NET Core	★ 5.4K

NOTE

A package's GitHub Usage section is generated automatically, periodically, without human review of individual repositories, and solely for informational purposes in order to show you GitHub repositories that depend on the package and that are popular with GitHub users.

- *Version history:* on the package page, look under **Info** for the date of the most recent update and examine the **Version History**. A well-maintained package has recent updates and a rich version history. Neglected packages have few updates and often haven't been updated in some time.

Version History

Version	Downloads	Last updated
11.0.1-beta1 (current version)	11,667	12 days ago
10.0.3	5,781,529	6 months ago
10.0.2	4,745,507	8 months ago
10.0.1	11,320,059	9 months ago
9.0.1	11,284,341	6/22/2016

[+ Show more](#)

- *Recent installs*: on the package page under **Statistics**, select **View full stats**. The full stats page shows the package installs over the last six weeks by version number. A package that other developers are actively using is typically a better choice than one that's not.
- *Support*: on the package page under **Info**, select **Project Site** (if available) to see what support options the author provides. A project with a dedicated site is generally better supported.
- *Developer history*: on the package page under **Owners**, select an owner to see what other packages they've published. Those with multiple packages are more likely to continue supporting their work in the future.
- *Open source contributions*: many packages are maintained in open-source repositories, making it possible for developers depending on them to directly contribute bug fixes and feature improvements. The contribution history of any given package is also a good indicator of how many developers are actively involved.
- *Interview the owners*: new developers can certainly be equally committed to producing great packages for you to use, and it's good to give them a chance to bring something new to the NuGet ecosystem. With this in mind, reach out directly to the package developers through the **Contact Owners** option under **Info** on the listing page. Chances are, they'll be happy to work with you to serve your needs!
- *Reserved Package ID Prefixes*: many package owners have applied for and have been granted a [reserved package ID prefix](#). When you see the visual checkmark next to a package ID on [nuget.org](#), or in Visual Studio, that means that the package owner has met our [criteria](#) for ID prefix reservation. This means the package owner is being clear on identifying themselves and their package.

NOTE

Always be mindful of a package's license terms, which you can see by selecting **License Info** on a package's listing page on [nuget.org](#). If a package does not specify license terms, contact the package owner directly using the **Contact owners** link on the package page. Microsoft does not license any intellectual property to you from third party package providers and is not responsible for information provided by third parties.

License URL deprecation

As we transition from [licenseUrl](#) to [license](#), some NuGet clients and NuGet feeds may not yet have the ability to surface licensing information in some cases. To maintain backward compatibility, the license URL points to this document which talks about how to retrieve the license information in such cases.

If clicking on the license URL for a package brought you to this page, it implies the package contains a license file

and

- You are connected to a feed that does not yet know how to interpret and surface the new license information to the client **OR**
- You are using a client that does not yet know how to interpret and read the new license information that is potentially provided by the feed **OR**
- A combination of both

Here is how you could read the information contained in the license file inside the package:

1. Download the NuGet package, and unzip its contents to a folder.
2. Open the `.nuspec` file which would be at the root of that folder.
3. It should have a tag like `<license type="file">license\license.txt</license>`. This implies the license file is named `license.txt` and it is inside a folder called `license` which would also be at the root of that folder.
4. Navigate to the `license` folder and open the `license.txt` file.

For the MSBuild equivalent to setting the license in the `.nuspec`, take a look at [Packing a license expression or a license file](#).

Search Syntax

NuGet package search works the same on nuget.org, from the NuGet CLI, and within the NuGet Package Manager extension in Visual Studio. In general, search is applied to keywords as well as package descriptions.

- **Keywords:** Search looks for relevant packages that contain any of the provided keywords. Example:
`modern UI`. To search for packages that contain all of the provided keywords, use "+" between the terms, such as `modern+UI`.
- **Phrases:** Entering terms within quotation marks looks for exact case-insensitive matches to those terms. Example: `"modern UI" package`
- **Filtering:** You can apply a search term to a specific property by using the syntax `<property>:<term>` where `<property>` (case-insensitive) can be `id`, `packageid`, `version`, `title`, `tags`, `author`, `description`, `summary`, and `owner`. Terms can be contained in quotes if needed, and you can search for multiple properties at the same time. Also, searches on the `id` property are substring matches, whereas `packageid` uses an exact match. Examples:

```
id:NuGet.Core           # Match any part of the id property
Id:"Nuget.Core"
ID:jQuery
title:jquery            # Searches title as shown on the package listing
PackageId:jquery        # Match the package id exactly
id:jquery id:ui         # Search for multiple terms in the id
id:jquery tags:validation # Search multiple properties
id:"jquery.ui"          # Phrase search
invalid:jquery ui       # Unsupported properties are ignored, so this
                        # is the same as searching on jquery ui
```

Install and manage packages in Visual Studio using the NuGet Package Manager

7/23/2019 • 6 minutes to read • [Edit Online](#)

The NuGet Package Manager UI in Visual Studio on Windows allows you to easily install, uninstall, and update NuGet packages in projects and solutions. For the experience in Visual Studio for Mac, see [Including a NuGet package in your project](#). The Package Manager UI is not included with Visual Studio Code.

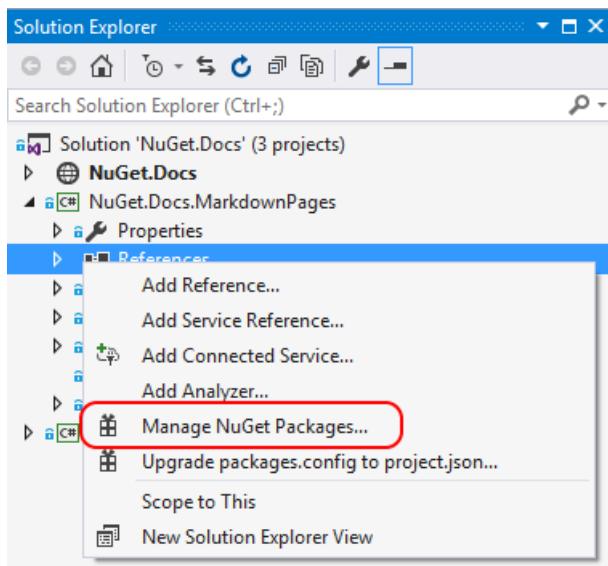
NOTE

If you're missing the NuGet Package Manager in Visual Studio 2015, check **Tools > Extensions and Updates...** and search for the *NuGet Package Manager* extension. If you're unable to use the extensions installer in Visual Studio, download the extension directly from <https://dist.nuget.org/index.html>.

Starting in Visual Studio 2017, NuGet and the NuGet Package Manager are automatically installed with any .NET-related workloads. Install it individually by selecting the **Individual components > Code tools > NuGet package manager** option in the Visual Studio installer.

Find and install a package

1. In **Solution Explorer**, right-click either **References** or a project and select **Manage NuGet Packages....**



2. The **Browse** tab displays packages by popularity from the currently selected source (see [package sources](#)). Search for a specific package using the search box on the upper left. Select a package from the list to display its information, which also enables the **Install** button along with a version-selection dropdown.

NuGet Package Manager: NuGet.Docs.MarkdownPages

Installed Updates 1

Json

Include prerelease

Package source: nuget.org

Newtonsoft.Json by James Newton-King, 32M v9.0.1

Json.NET is a popular high-performance JSON framework for .NET

JSON by Daniel Crenna, 35.2K downloads v1.0.1

A JSON parser in C#, supporting dynamic deserialization in .NET 4.0. A JSON parser in C#, su...

ServiceStack.Text by Service Stack, 1.48M dc v4.0.60

.NET's fastest JSON, JSV and CSV Text Serializers

RestSharp by John Sheehan, RestSharp Comm v105.2.3

Simple REST and HTTP API Client

json-serialize by Kevin Malakoff, 6.77K downlo v1.1.2

JSON-Serialize.js provides conventions and helpers

Newtonsoft.Json

Version: Latest stable 9.0.1

Install

Options

Description

Json.NET is a popular high-performance JSON framework for .NET

Version: 9.0.1

Author(s): James Newton-King

License: <https://raw.github.com/JamesNK/Newtonsoft.Json/master/LICENSE.md>

Date published: Wednesday, June 22, 2016 (6/22/2016)

Project URL: <http://www.newtonsoft.com/json>

3. Select the desired version from the drop-down and select **Install**. Visual Studio installs the package and its dependencies into the project. You may be asked to accept license terms. When installation is complete, the added packages appear on the **Installed** tab. Packages are also listed in the **References** node of Solution Explorer, indicating that you can refer to them in the project with `using` statements.

NuGet.Docs.MarkdownPages

Properties

References

Analyzers

HtmlAgilityPack

MarkdownSharp

Microsoft.CSharp

Microsoft.Web.Infrastructure

Newtonsoft.Json

Octokit

System

System.Core

System.Data

System.Data.DataSetExtensions

System.Net.Http

System.Web

System.Web.Helpers

System.Web.Razor

System.Web.WebPages

System.Web.WebPages.Deployment

System.Web.WebPages.Razor

System.Xml

System.Xml.Linq

Heading.cs

MarkdownPathFactory.cs

MarkdownWebPage.cs

packages.config

Topic.cs

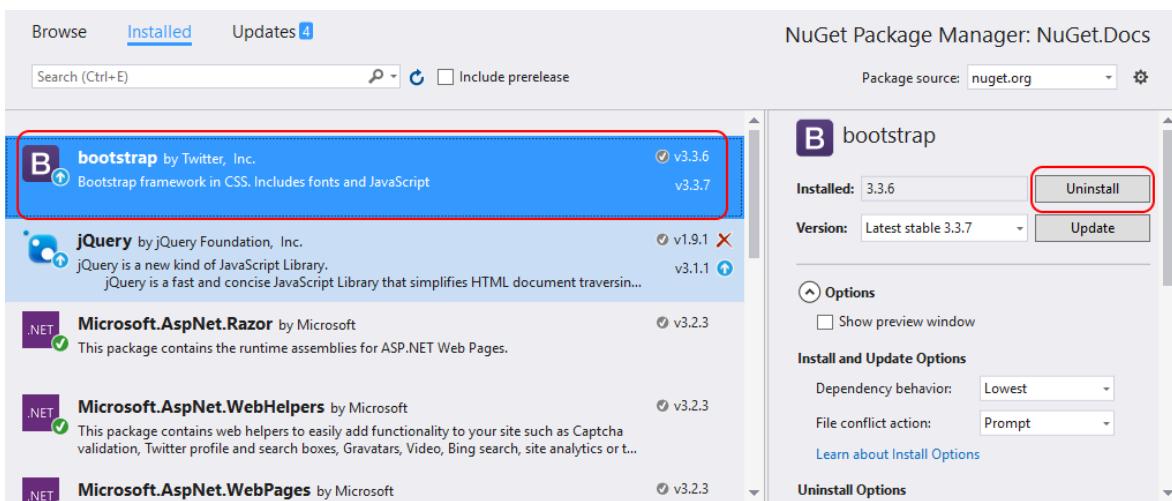
TIP

To include prerelease versions in the search, and to make prerelease versions available in the version drop-down, select the **Include prerelease** option.

Uninstall a package

1. In **Solution Explorer**, right-click either **References** or the desired project, and select **Manage NuGet Packages....**

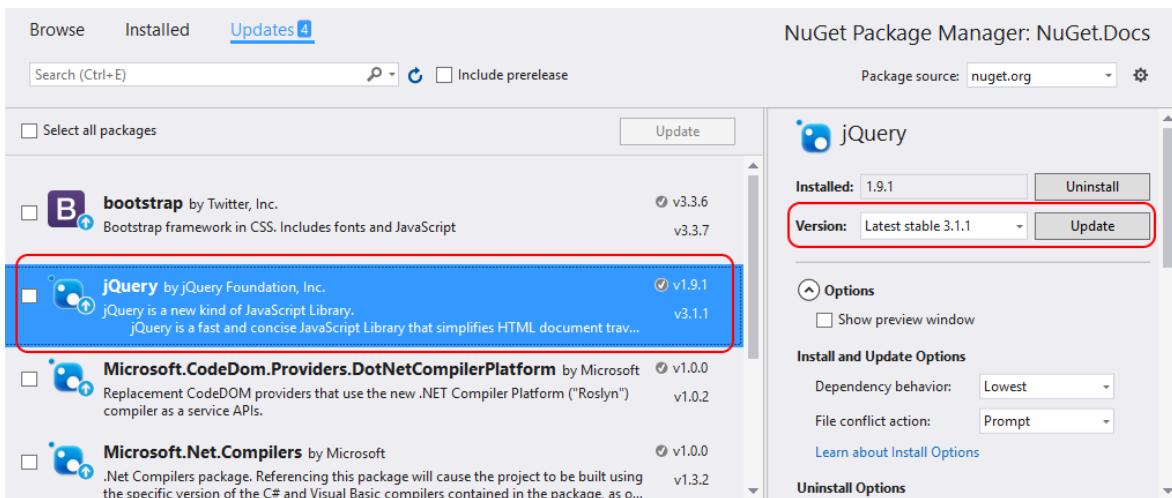
2. Select the **Installed** tab.
3. Select the package to uninstall (using search to filter the list if necessary) and select **Uninstall**.



4. Note that the **Include prerelease** and **Package source** controls have no effect when uninstalling packages.

Update a package

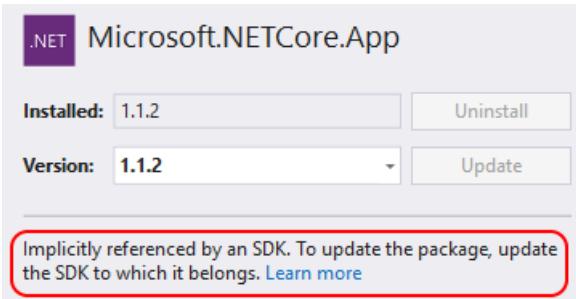
1. In **Solution Explorer**, right-click either **References** or the desired project, and select **Manage NuGet Packages...** (In web site projects, right-click the **Bin** folder.)
2. Select the **Updates** tab to see packages that have available updates from the selected package sources. Select **Include prerelease** to include prerelease packages in the update list.
3. Select the package to update, select the desired version from the drop-down on the right, and select **Update**.



4. For some packages, the **Update** button is disabled and a message appears saying that it's "Implicitly referenced by an SDK" (or "AutoReferenced"). This message indicates that the package is part of a larger framework or SDK and should not be updated independently. (Such packages are internally marked with `<IsImplicitlyDefined>True</IsImplicitlyDefined>`.) For example, `Microsoft.NETCore.App` is part of the .NET Core SDK, and the package version is not the same as the version of the runtime framework used by the application. You need to [update your .NET Core installation](#) to get new versions of the ASP.NET Core and .NET Core runtime. [See this document for more details on .NET Core metapackages and versioning](#). This applies to the following commonly used packages:

- Microsoft.AspNetCore.All

- Microsoft.AspNetCore.App
- Microsoft.NETCore.App
- NETStandard.Library

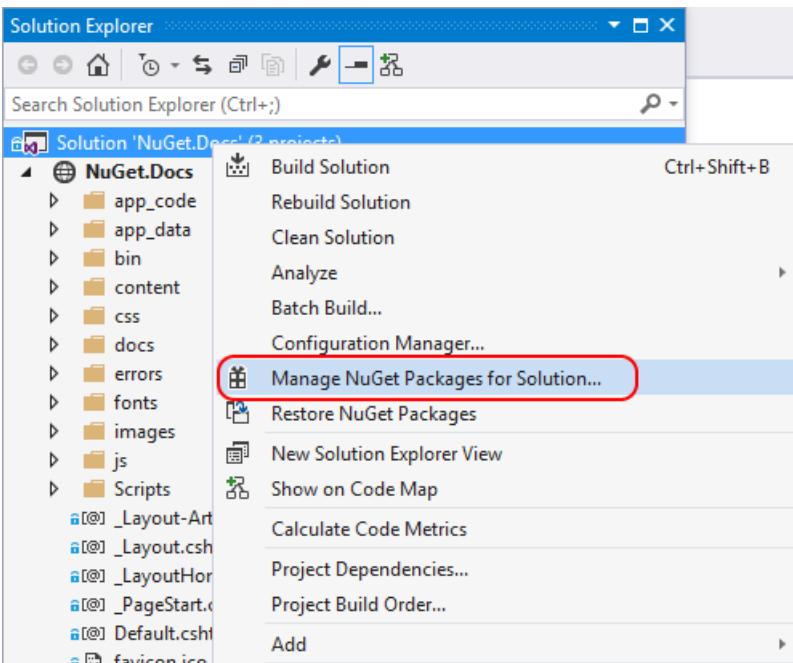


5. To update multiple packages to their newest versions, select them in the list and select the **Update** button above the list.
6. You can also update an individual package from the **Installed** tab. In this case, the details for the package include a version selector (subject to the **Include prerelease** option) and an **Update** button.

Manage packages for the solution

Managing packages for a solution is a convenient means to work with multiple projects simultaneously.

1. Select the **Tools > NuGet Package Manager > Manage NuGet Packages for Solution...** menu command, or right-click the solution and select **Manage NuGet Packages...**:



2. When managing packages for the solution, the UI lets you select the projects that are affected by the operations:

Manage Packages for Solution

EntityFramework by Microsoft, 19M download v6.1.3

Entity Framework is Microsoft's recommended data access technology for new applications.

Newtonsoft.Json by James Newton-King, 32M download v9.0.1

Json.NET is a popular high-performance JSON framework for .NET

bootstrap by Twitter, Inc., 5.78M downloads v3.3.6

Bootstrap framework in CSS. Includes fonts and JavaScript

jQuery by jQuery Foundation, Inc., 22.6M download v3.0.0.1

jQuery is a new kind of JavaScript Library. jQuery is a fast and concise JavaScript Libr...

Microsoft.AspNet.Mvc by Microsoft, 17.6M download v5.2.3

This package contains the runtime assemblies for ASP.NET MVC.

.NET EntityFramework

Version(s) - 0

Project	Version
<input checked="" type="checkbox"/> NuGet.Docs	
<input checked="" type="checkbox"/> NuGet.Docs.MarkdownPages	
<input checked="" type="checkbox"/> NuGet.Docs.References	

Installed: not installed

Version: Latest stable 6.1.3

Uninstall

Install

Consolidate tab

Developers typically consider it bad practice to use different versions of the same NuGet package across different projects in the same solution. When you choose to manage packages for a solution, the Package Manager UI provides a **Consolidate** tab on which you can easily see where packages with distinct version numbers are used by different projects in the solution:

Solution1 - NuGet - Solution*

NuGet - Solution* Consolidate 1

Manage Packages for Solution

EntityFramework by Microsoft v6.2.0

Entity Framework is Microsoft's recommended data access technology for new applications.

.NET EntityFramework

Version(s) - 2

Project	Version
<input checked="" type="checkbox"/> ClassLibrary1	6.2.0
<input checked="" type="checkbox"/> ConsoleApp1\ConsoleApp1.	6.1.0

Installed: multiple versions in

Uninstall

Version: Latest stable 6.2.0

Install

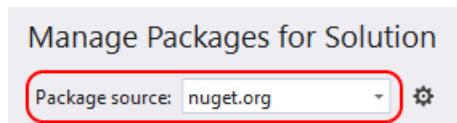
In this example, the ClassLibrary1 project is using EntityFramework 6.2.0, whereas ConsoleApp1 is using EntityFramework 6.1.0. To consolidate package versions, do the following:

- Select the projects to update in the project list.
- Select the version to use in all those projects in the **Version** control, such as EntityFramework 6.2.0.
- Select the **Install** button.

The Package Manager installs the selected package version into all selected projects, after which the package no longer appears on the **Consolidate** tab.

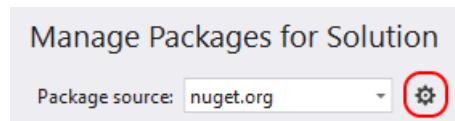
Package sources

To change the source from which Visual Studio obtains packages, select one from the source selector:

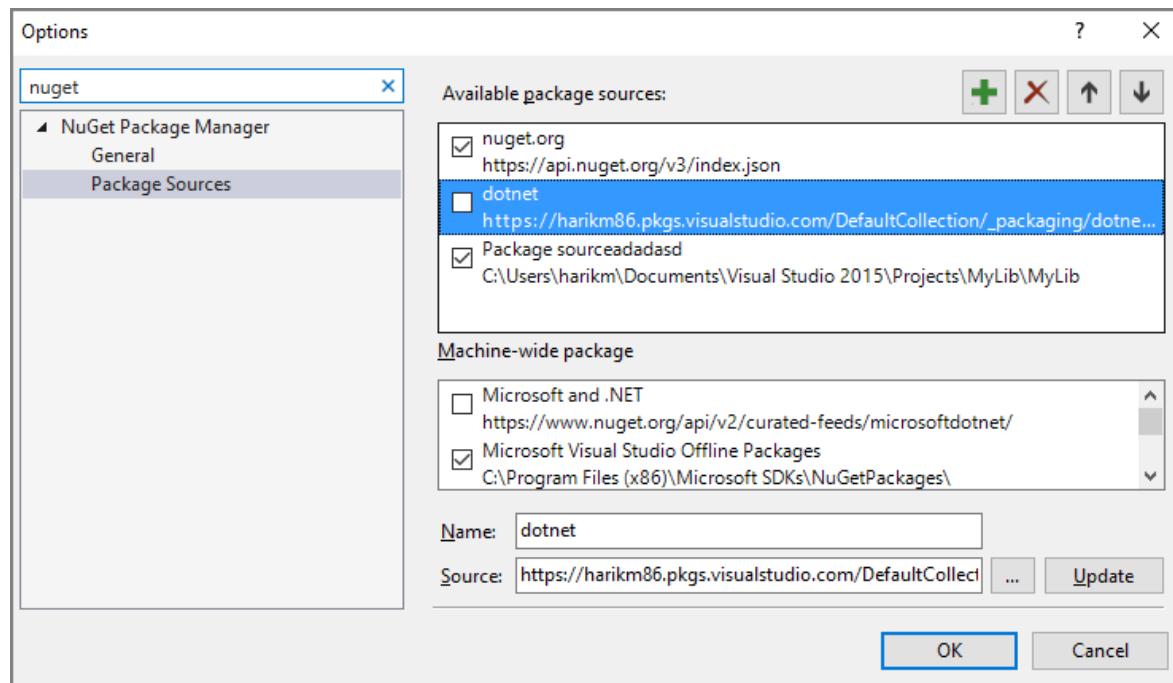


To manage package sources:

1. Select the **Settings** icon in the Package Manager UI outlined below or use the **Tools > Options** command and scroll to **NuGet Package Manager**:



2. Select the **Package Sources** node:



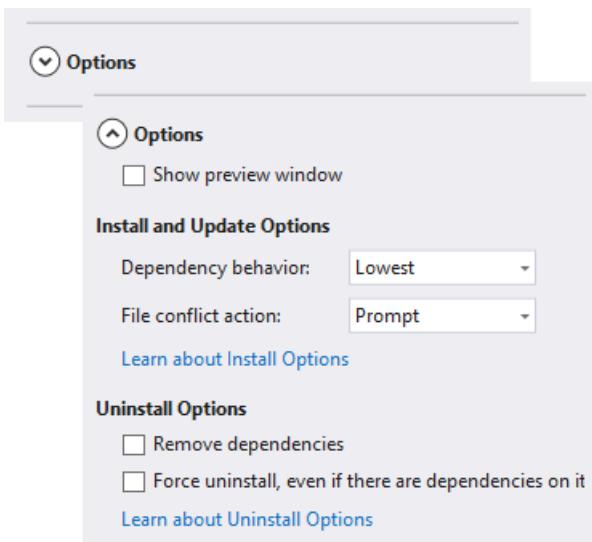
3. To add a source, select **+**, edit the name, enter the URL or path in the **Source** control, and select **Update**. The source now appears in the selector drop-down.
4. To change a package source, select it, make edits in the **Name** and **Source** boxes, and select **Update**.
5. To disable a package source, clear the box to the left of the name in the list.
6. To remove a package source, select it and then select the **X** button.
7. Using the up and down arrow buttons does not change the priority order of the package sources. Visual Studio ignores the order of package sources, using the package from whichever source is first to respond to requests. For more information, see [Package restore](#).

TIP

If a package source reappears after deleting it, it may be listed in a computer-level or user-level `NuGet.Config` files. See [Common NuGet configurations](#) for the location of these files, then remove the source by editing the files manually or using the `nuget sources` command.

Package manager Options control

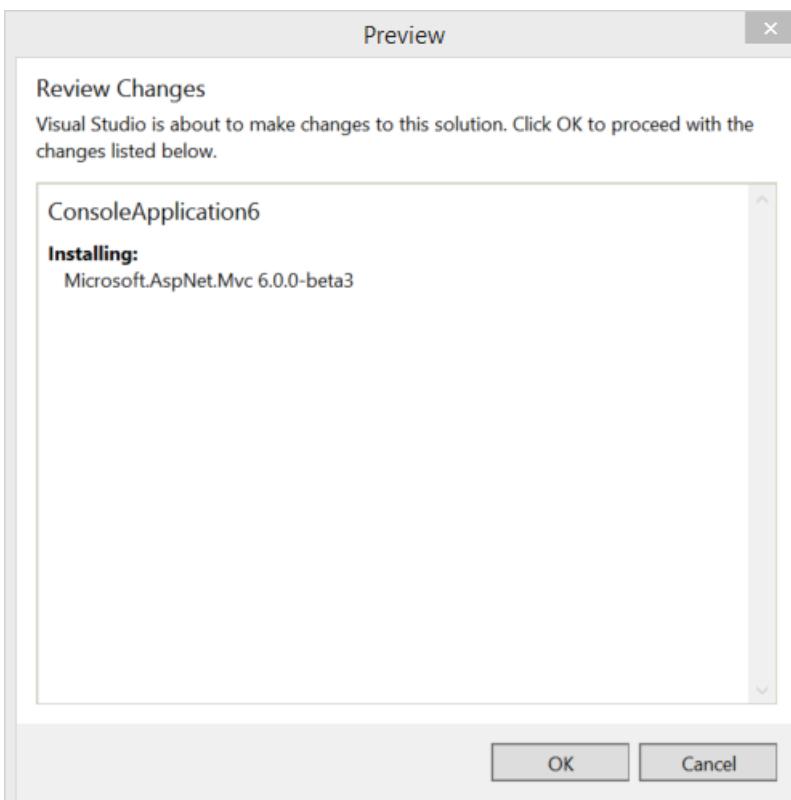
When a package is selected, the Package Manager UI displays a small, expandable **Options** control below the version selector (shown here both collapsed and expanded). Note that for some project types, only the **Show preview window** option is provided.



The following sections explain these options.

Show preview window

When selected, a modal window displays which the dependencies of a chosen package before the package is installed:



Install and Update Options

(Not available for all project types.)

Dependency behavior configures how NuGet decides which versions of dependent packages to install:

- *Ignore dependencies* skips installing any dependencies, which typically breaks the package being installed.
- *Lowest* [Default] installs the dependency with the minimal version number that meets the requirements of the primary chosen package.
- *Highest Patch* installs the version with the same major and minor version numbers, but the highest patch number. For example, if version 1.2.2 is specified then the highest version that starts with 1.2 will be installed
- *Highest Minor* installs the version with the same major version number but the highest minor number and patch number. If version 1.2.2 is specified, then the highest version that starts with 1 will be installed
- *Highest* installs the highest available version of the package.

File conflict action specifies how NuGet should handle packages that already exist in the project or local machine:

- *Prompt* instructs NuGet to ask whether to keep or overwrite existing packages.
- *Ignore All* instructs NuGet to skip overwriting any existing packages.
- *Overwrite All* instructs NuGet to overwrite any existing packages.

Uninstall Options

(Not available for all project types.)

Remove dependencies: when selected, removes any dependent packages if they're not referenced elsewhere in the project.

Force uninstall even if there are dependencies on it: when selected, uninstalls a package even if it's still being referenced in the project. This is typically used in combination with **Remove dependencies** to remove a package and whatever dependencies it installed. Using this option may, however, lead to broken references in the project. In such cases, you may need to [reinstall those other packages](#).

Install and manage packages using the dotnet CLI

8/8/2019 • 2 minutes to read • [Edit Online](#)

The CLI tool allows you to easily install, uninstall, and update NuGet packages in projects and solutions. It runs on Windows, Mac OS X, and Linux.

The dotnet CLI is for use in your .NET Core and .NET Standard project (SDK-style project types), and for any other SDK-style projects (for example, an SDK-style project that targets .NET Framework). For more information, see [SDK attribute](#).

This article shows you basic usage for a few of the most common dotnet CLI commands. For most of these commands, the CLI tool looks for a project file in the current directory, unless a project file is specified in the command (the project file is an optional switch). For a complete list of commands and the arguments you may use, see the [.NET Core command-line interface \(CLI\) tools](#).

Prerequisites

- The [.NET Core SDK](#), which provides the `dotnet` command-line tool. Starting in Visual Studio 2017, the dotnet CLI is automatically installed with any .NET Core related workloads.

Install a package

`dotnet add package` adds a package reference to the project file, then runs `dotnet restore` to install the package.

1. Open a command line and switch to the directory that contains your project file.
2. Use the following command to install a Nuget package:

```
dotnet add package <PACKAGE_NAME>
```

For example, to install the `Newtonsoft.Json` package, use the following command

```
dotnet add package Newtonsoft.Json
```

3. After the command completes, look at the project file to make sure the package was installed.

You can open the `.csproj` file to see the added reference:

```
<ItemGroup>
<PackageReference Include="Newtonsoft.Json" Version="12.0.1" />
</ItemGroup>
```

Install a specific version of a package

If the version is not specified, NuGet installs the latest version of the package. You can also use the `dotnet add package` command to install a specific version of a Nuget package:

```
dotnet add package <PACKAGE_NAME> -v <VERSION>
```

For example, to add version 12.0.1 of the `Newtonsoft.Json` package, use this command:

```
dotnet add package Newtonsoft.Json -v 12.0.1
```

List package references

You can list the package references for your project using the [dotnet list package](#) command.

```
dotnet list package
```

Remove a package

Use the [dotnet remove package](#) command to remove a package reference from the project file.

```
dotnet remove package <PACKAGE_NAME>
```

For example, to remove the `Newtonsoft.Json` package, use the following command

```
dotnet remove package Newtonsoft.Json
```

Update a package

NuGet installs the latest version of the package when you use the `dotnet add package` command unless you specify the package version (`-v` switch).

Restore packages

Use the [dotnet restore](#) command, which restores packages listed in the project file (see [PackageReference](#)). With .NET Core 2.0 and later, restore is done automatically with `dotnet build` and `dotnet run`. As of NuGet 4.0, this runs the same code as `nuget restore`.

As with the other `dotnet` CLI commands, first open a command line and switch to the directory that contains your project file.

To restore a package using `dotnet restore`:

```
dotnet restore
```

Manage packages using the nuget.exe CLI

8/15/2019 • 3 minutes to read • [Edit Online](#)

The CLI tool allows you to easily update and restore NuGet packages in projects and solutions. This tool provides all NuGet capabilities on Windows, and also provides most features on Mac and Linux when running under Mono.

The `nuget.exe` CLI is for your .NET Framework project and non-SDK-style projects (for example, a non-SDK style project that targets .NET Standard libraries). If you are using a non-SDK-style project that has been migrated to `PackageReference`, use the `dotnet` CLI instead. The `nuget.exe` CLI requires a `packages.config` file for package references.

NOTE

In most scenarios, we recommend [migrating non-SDK-style projects](#) that use `packages.config` to `PackageReference`, and then you can use the `dotnet` CLI instead of the `nuget.exe` CLI. Migration is not currently available for C++ and ASP.NET projects.

This article shows you basic usage for a few of the most common `nuget.exe` CLI commands. For most of these commands, the CLI tool looks for a project file in the current directory, unless a project file is specified in the command. For a complete list of commands and the arguments you may use, see the [nuget.exe CLI reference](#).

Prerequisites

- Install the `nuget.exe` CLI by downloading it from [nuget.org](#), saving that `.exe` file to a suitable folder, and adding that folder to your PATH environment variable.

Install a package

The `install` command downloads and installs a package into a project, defaulting to the current folder, using specified package sources. Install new packages into the `packages` folder in your project root directory.

IMPORTANT

The `install` command does not modify a project file or `packages.config`; in this way it's similar to `restore` in that it only adds packages to disk but does not change a project's dependencies. To add a dependency, either add a package through the Package Manager UI or Console in Visual Studio, or modify `packages.config` and then run either `install` or `restore`.

- Open a command line and switch to the directory that contains your project file.
- Use the following command to install a NuGet package to the `packages` folder.

```
nuget install <packageID> -OutputDirectory packages
```

To install the `Newtonsoft.json` package to the `packages` folder, use the following command:

```
nuget install Newtonsoft.Json -OutputDirectory packages
```

Alternatively, you can use the following command to install a NuGet package using an existing `packages.config` file to the `packages` folder. This does not add the package to your project dependencies, but installs it locally.

```
nuget install packages.config -OutputDirectory packages
```

Install a specific version of a package

If the version is not specified when you use the `install` command, NuGet installs the latest version of the package. You can also install a specific version of a NuGet package:

```
nuget install <packageID | configFilePath> -Version <version>
```

For example, to add version 12.0.1 of the `Newtonsoft.json` package, use this command:

```
nuget install Newtonsoft.Json -Version 12.0.1
```

For more information on the limitations and behavior of `install`, see [Install a package](#).

Remove a package

To delete one or more packages, delete the packages you want to remove from the `packages` folder.

If you want to reinstall packages, use the `restore` or `install` command.

List packages

You can display a list of packages from a given source using the `list` command. Use the `-Source` option to restrict the search.

```
nuget list -Source <source>
```

For example, list packages in the `packages` folder.

```
nuget list -Source C:\Users\username\source\repos\MyProject\packages
```

If you use a search term, the search includes names of packages, tags, and package descriptions.

```
nuget list <search term>
```

Update an individual package

NuGet installs the latest version of the package when you use the `install` command unless you specify the package version.

Update all packages

Use the `update` command to update all packages. Updates all packages in a project (using `packages.config`) to their latest available versions. It is recommended to run `restore` before running `update`.

```
nuget update
```

Restore packages

Use the [restore](#) command, which downloads and installs any packages missing from the *packages* folder.

For projects migrated to `PackageReference`, use [msbuild -t:restore](#) to restore packages instead.

`restore` only adds packages to disk but does not change a project's dependencies. To restore project dependencies, modify `packages.config`, then use the `restore` command.

As with the other `nuget.exe` CLI commands, first open a command line and switch to the directory that contains your project file.

To restore a package using `restore`:

```
nuget restore MySolution.sln
```

Get the CLI version

Use this command:

```
nuget help
```

The first line in the help output shows the version. To avoid scrolling up, use `nuget help | more` instead.

Install and manage packages with the Package Manager Console in Visual Studio (PowerShell)

11/5/2019 • 5 minutes to read • [Edit Online](#)

The NuGet Package Manager Console lets you use [NuGet PowerShell commands](#) to find, install, uninstall, and update NuGet packages. Using the console is necessary in cases where the Package Manager UI does not provide a way to perform an operation. To use `nuget.exe` CLI commands in the console, see [Using the nuget.exe CLI in the console](#).

The console is built into Visual Studio on Windows. It is not included with Visual Studio for Mac or Visual Studio Code.

Find and install a package

For example, finding and installing a package is done with three easy steps:

1. Open the project/solution in Visual Studio, and open the console using the **Tools > NuGet Package Manager > Package Manager Console** command.
2. Find the package you want to install. If you already know this, skip to step 3.

```
# Find packages containing the keyword "elmah"
Find-Package elmah
```

3. Run the install command:

```
# Install the Elmah package to the project named MyProject.
Install-Package Elmah -ProjectName MyProject
```

IMPORTANT

All operations that are available in the console can also be done with the [NuGet CLI](#). However, console commands operate within the context of Visual Studio and a saved project/solution and often accomplish more than their equivalent CLI commands. For example, installing a package through the console adds a reference to the project whereas the CLI command does not. For this reason, developers working in Visual Studio typically prefer using the console to the CLI.

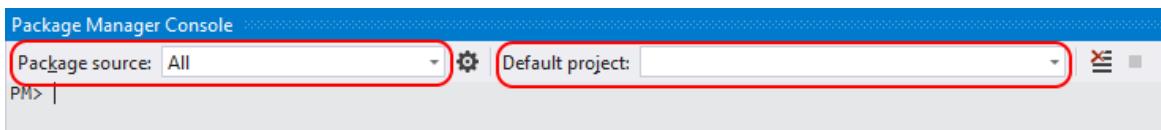
TIP

Many console operations depend on having a solution opened in Visual Studio with a known path name. If you have an unsaved solution, or no solution, you can see the error, "Solution is not opened or not saved. Please ensure you have an open and saved solution." This indicates that the console cannot determine the solution folder. Saving an unsaved solution, or creating and saving a solution if you don't have one open, should correct the error.

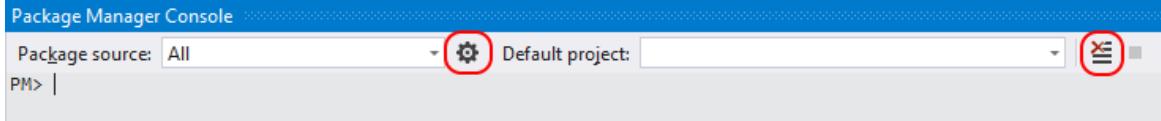
Opening the console and console controls

1. Open the console in Visual Studio using the **Tools > NuGet Package Manager > Package Manager Console** command. The console is a Visual Studio window that can be arranged and positioned however you like (see [Customize window layouts in Visual Studio](#)).

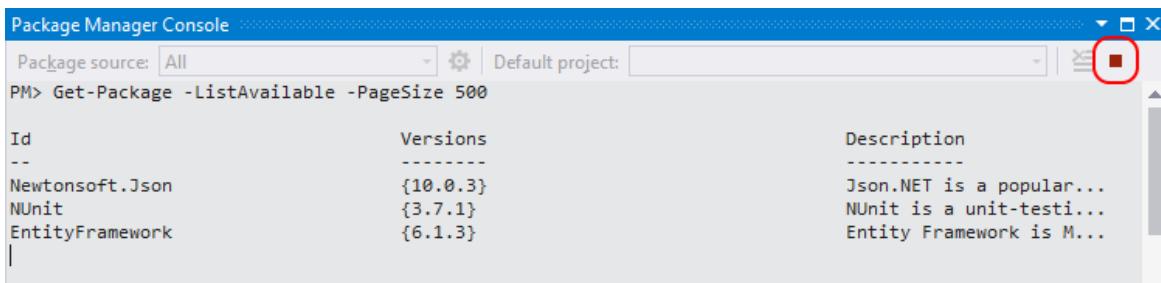
2. By default, console commands operate against a specific package source and project as set in the control at the top of the window:



3. Selecting a different package source and/or project changes those defaults for subsequent commands. To override these settings without changing the defaults, most commands support `-Source` and `-ProjectName` options.
4. To manage package sources, select the gear icon. This is a shortcut to the **Tools > Options > NuGet Package Manager > Package Sources** dialog box as described on the [Package Manager UI](#) page. Also, the control to the right of the project selector clears the console's contents:



5. The rightmost button interrupts a long-running command. For example, running `Get-Package -ListAvailable -PageSize 500` lists the top 500 packages on the default source (such as [nuget.org](#)), which could take several minutes to run.



Install a package

```
# Add the Elmah package to the default project as specified in the console's project selector
Install-Package Elmah

# Add the Elmah package to a project named UtilitiesLib that is not the default
Install-Package Elmah -ProjectName UtilitiesLib
```

See [Install-Package](#).

Installing a package in the console performs the same steps as described on [What happens when a package is installed](#), with the following additions:

- The Console displays applicable license terms in its window with implied agreement. If you do not agree to the terms, you should uninstall the package immediately.
- Also a reference to the package is added to the project file and appears in **Solution Explorer** under the **References** node, you need to save the project to see the changes in the project file directly.

Uninstall a package

```
# Uninstalls the Elmah package from the default project
Uninstall-Package Elmah

# Uninstalls the Elmah package and all its unused dependencies
Uninstall-Package Elmah -RemoveDependencies

# Uninstalls the Elmah package even if another package depends on it
Uninstall-Package Elmah -Force
```

See [Uninstall-Package](#). Use [Get-Package](#) to see all packages currently installed in the default project if you need to find an identifier.

Uninstalling a package performs the following actions:

- Removes references to the package from the project (and whatever management format is in use). References no longer appear in **Solution Explorer**. (You might need to rebuild the project to see it removed from the **Bin** folder.)
- Reverses any changes made to `app.config` or `web.config` when the package was installed.
- Removes previously-installed dependencies if no remaining packages use those dependencies.

Update a package

```
# Checks if there are newer versions available for any installed packages
Get-Package -updates

# Updates a specific package using its identifier, in this case jQuery
Update-Package jQuery

# Update all packages in the project named MyProject (as it appears in Solution Explorer)
Update-Package -ProjectName MyProject

# Update all packages in the solution
Update-Package
```

See [Get-Package](#) and [Update-Package](#)

Find a package

```
# Find packages containing keywords
Find-Package elmah
Find-Package logging

# List packages whose ID begins with Elmah
Find-Package Elmah -StartWith

# By default, Get-Package returns a list of 20 packages; use -First to show more
Find-Package logging -First 100

# List all versions of the package with the ID of "jquery"
Find-Package jquery -AllVersions -ExactMatch
```

See [Find-Package](#). In Visual Studio 2013 and earlier, use [Get-Package](#) instead.

Availability of the console

Starting in Visual Studio 2017, NuGet and the NuGet Package Manager are automatically installed when you select any .NET-related workloads; you can also install it individually by checking the **Individual components**

> **Code tools** > **NuGet package manager** option in the Visual Studio installer.

Also, if you're missing the NuGet Package Manager in Visual Studio 2015 and earlier, check **Tools** > **Extensions and Updates...** and search for the NuGet Package Manager extension. If you're unable to use the extensions installer in Visual Studio, you can download the extension directly from <https://dist.nuget.org/index.html>.

The Package Manager Console is not presently available with Visual Studio for Mac. The equivalent commands, however, are available through the [NuGet CLI](#). Visual Studio for Mac does have a UI for managing NuGet packages. See [Including a NuGet package in your project](#).

The Package Manager Console is not included with Visual Studio Code.

Extend the Package Manager Console

Some packages install new commands for the console. For example, `MvcScaffolding` creates commands like `Scaffold` shown below, which generates ASP.NET MVC controllers and views:

```
PM> Install-Package MvcScaffolding
'T4Scaffolding (≥ 1.0.0)' not installed. Attempting to retrieve dependency from
source...
Done.
Successfully installed 'T4Scaffolding 1.0.0'.
Successfully installed 'MvcScaffolding 1.0.0'.
Successfully added 'T4Scaffolding 1.0.0' to MvcApplication5.
Successfully added 'MvcScaffolding 1.0.0' to MvcApplication5.

PM> Scaffold C
    Controller
    CustomScaffolder
    CustomTemplate
```

Set up a NuGet PowerShell profile

A PowerShell profile lets you make commonly-used commands available wherever you use PowerShell. NuGet supports a NuGet-specific profile typically found at the following location:

```
%UserProfile%\Documents\WindowsPowerShell\NuGet_profile.ps1
```

To find the profile, type `$profile` in the console:

```
$profile
C:\Users<user>\Documents\WindowsPowerShell\NuGet_profile.ps1
```

For more details, refer to [Windows PowerShell Profiles](#).

Use the nuget.exe CLI in the console

To make the `nuget.exe` [CLI](#) available in the Package Manager Console, install the [NuGet.CommandLine](#) package from the console:

```
# Other versions are available, see https://www.nuget.org/packages/NuGet.CommandLine/
Install-Package NuGet.CommandLine -Version 4.4.1
```

Restore packages using Package Restore

8/23/2019 • 10 minutes to read • [Edit Online](#)

To promote a cleaner development environment and to reduce repository size, NuGet **Package Restore** installs all of a project's dependencies listed in either the project file or `packages.config`. The .NET Core 2.0+ `dotnet build` and `dotnet run` commands do an automatic package restore. Visual Studio can restore packages automatically when it builds a project, and you can restore packages at any time through Visual Studio, `nuget restore`, `dotnet restore`, and `xbuild` on Mono.

Package Restore makes sure that all a project's dependencies are available, without having to store them in source control. To configure your source control repository to exclude the package binaries, see [Packages and source control](#).

Package Restore overview

Package Restore first installs the direct dependencies of a project as needed, then installs any dependencies of those packages throughout the entire dependency graph.

If a package isn't already installed, NuGet first attempts to retrieve it from the [cache](#). If the package isn't in the cache, NuGet tries to download the package from all enabled sources in the list at **Tools > Options > NuGet Package Manager > Package Sources** in Visual Studio. During restore, NuGet ignores the order of package sources, and uses the package from whichever source is first to respond to requests. For more information about how NuGet behaves, see [Common NuGet configurations](#).

NOTE

NuGet doesn't indicate a failure to restore a package until all the sources have been checked. At that time, NuGet reports a failure for only the last source in the list. The error implies that the package wasn't present on *any* of the other sources, even though errors aren't shown for each of those sources individually.

Restore packages

Package Restore tries to install all package dependencies to the correct state matching the package references in your project file (`.csproj`) or your `packages.config` file. (In Visual Studio, the references appear in Solution Explorer under the **Dependencies \ NuGet** or the **References** node.)

1. If the package references in your project file are correct, use your preferred tool to restore packages.

- [Visual Studio \(automatic restore or manual restore\)](#)
- [dotnet CLI](#)
- [nuget.exe CLI](#)
- [MSBuild](#)
- [Azure Pipelines](#)
- [Azure DevOps Server](#)

If the package references in your project file (`.csproj`) or your `packages.config` file are incorrect (they do not match your desired state following Package Restore), then you need to either install or update packages instead.

For projects using `PackageReference`, after a successful restore, the package should be present in the *global-packages* folder and the `obj/project.assets.json` file is recreated. For projects using `packages.config`, the

package should appear in the project's `packages` folder. The project should now build successfully.

2. After running Package Restore, if you still experience missing packages or package-related errors (such as error icons in Solution Explorer in Visual Studio), you may need to follow instructions described in [Troubleshooting Package Restore errors](#) or, alternatively, [reinstall and update packages](#).

In Visual Studio, the Package Manager Console provides several flexible options for reinstalling packages. See [Using Package-Update](#).

Restore using Visual Studio

In Visual Studio on Windows, either:

- Restore packages automatically, or
- Restore packages manually

Restore packages automatically using Visual Studio

Package Restore happens automatically when you create a project from a template or build a project, subject to the options in [Enable and disable package restore](#). In NuGet 4.0+, restore also happens automatically when you make changes to a SDK-style project (typically a .NET Core or .NET Standard project).

1. Enable automatic package restore by choosing **Tools > Options > NuGet Package Manager**, and then selecting **Automatically check for missing packages during build in Visual Studio** under **Package Restore**.

For non-SDK-style projects, you first need to select **Allow NuGet to download missing packages** to enable the automatic restore option.

2. Build the project.

If one or more individual packages still aren't installed properly, **Solution Explorer** shows an error icon.

Right-click and select **Manage NuGet Packages**, and use **Package Manager** to uninstall and reinstall the affected packages. For more information, see [Reinstall and update packages](#)

If you see the error "This project references NuGet package(s) that are missing on this computer," or "One or more NuGet packages need to be restored but couldn't be because consent has not been granted," [enable automatic restore](#). For older projects, also see [Migrate to automatic package restore](#). Also see [Package Restore troubleshooting](#).

Restore packages manually using Visual Studio

1. Enable package restore by choosing **Tools > Options > NuGet Package Manager**. Under **Package Restore** options, select **Allow NuGet to download missing packages**.

2. In **Solution Explorer**, right click the solution and select **Restore NuGet Packages**.

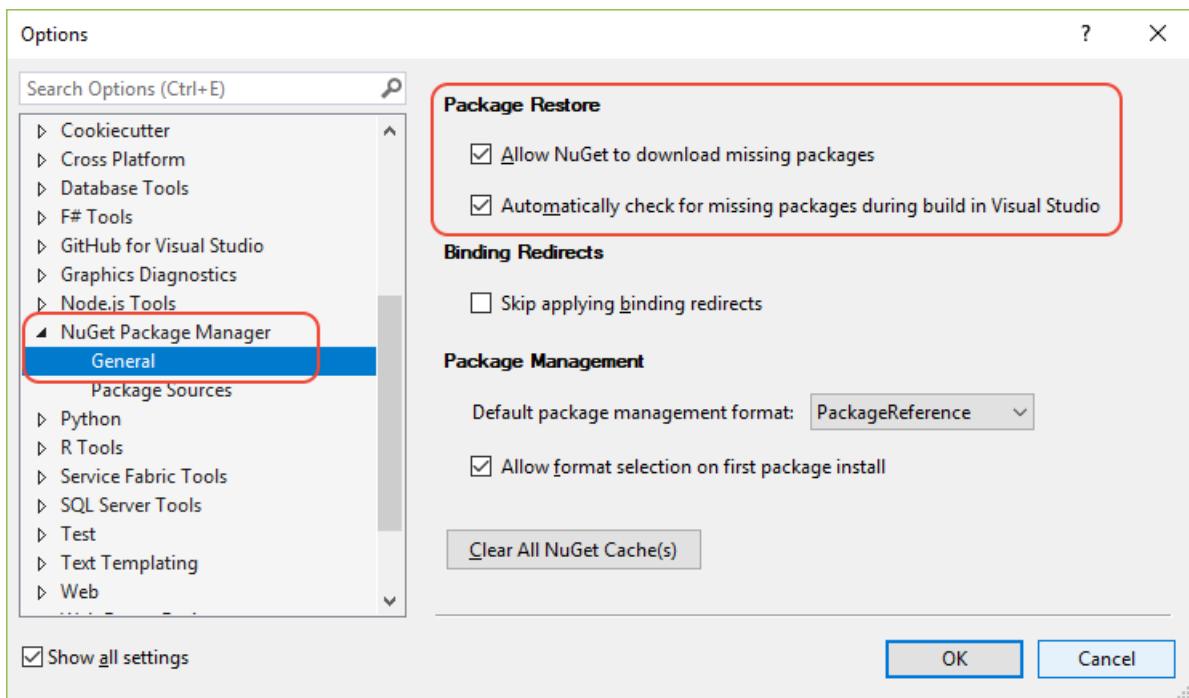
If one or more individual packages still aren't installed properly, **Solution Explorer** shows an error icon.

Right-click and select **Manage NuGet Packages**, and then use **Package Manager** to uninstall and reinstall the affected packages. For more information, see [Reinstall and update packages](#)

If you see the error "This project references NuGet package(s) that are missing on this computer," or "One or more NuGet packages need to be restored but couldn't be because consent has not been granted," [enable automatic restore](#). For older projects, also see [Migrate to automatic package restore](#). Also see [Package Restore troubleshooting](#).

Enable and disable package restore in Visual Studio

In Visual Studio, you control Package Restore primarily through **Tools > Options > NuGet Package Manager**:



- **Allow NuGet to download missing packages** controls all forms of package restore by changing the

`packageRestore/enabled` setting in the [packageRestore section](#) of the `NuGet.Config` file, at `%AppData%\NuGet\` on Windows, or `~/.nuget/NuGet/` on Mac/Linux. This setting also enables the **Restore NuGet Packages** command on the solution's context menu in Visual Studio, .

```
<configuration>
  <packageRestore>
    <!-- The 'enabled' key is True when the "Allow NuGet to download missing packages" checkbox is set.
        Clearing the box sets this to False, disabling command-line, automatic, and MSBuild-integrated restore. -->
    <add key="enabled" value="True" />
  </packageRestore>
</configuration>
```

NOTE

To globally override the `packageRestore/enabled` setting, set the environment variable **EnableNuGetPackageRestore** with a value of True or False before launching Visual Studio or starting a build.

- **Automatically check for missing packages during build in Visual Studio** controls automatic restore by changing the `packageRestore/automatic` setting in the [packageRestore section](#) of the `NuGet.Config` file. When this option is set to True, running a build from Visual Studio automatically restores any missing packages. This setting doesn't affect builds run from the MSBuild command line.

```
...
<configuration>
  <packageRestore>
    <!-- The 'automatic' key is set to True when the "Automatically check for missing packages during build in Visual Studio" checkbox is set. Clearing the box sets this to False and disables automatic restore. -->
    <add key="automatic" value="True" />
  </packageRestore>
</configuration>
```

To enable or disable Package Restore for all users on a computer, a developer or company can add the configuration settings to the global `nuget.config` file. The global `nuget.config` is in Windows at `%ProgramData%\NuGet\Config`, sometimes under a specific `\{IDE\}\{Version\}\{SKU\}\` Visual Studio folder, or in Mac/Linux at `~/.local/share`. Individual users can then selectively enable restore as needed on a project level. For more details on how NuGet prioritizes multiple config files, see [Common NuGet configurations](#).

IMPORTANT

If you edit the `packageRestore` settings directly in `nuget.config`, restart Visual Studio, so that the **Options** dialog box shows the current values.

Restore using the dotnet CLI

Use the `dotnet restore` command, which restores packages listed in the project file (see [PackageReference](#)). With .NET Core 2.0 and later, restore is done automatically with `dotnet build` and `dotnet run`. As of NuGet 4.0, this runs the same code as `nuget restore`.

As with the other `dotnet` CLI commands, first open a command line and switch to the directory that contains your project file.

To restore a package using `dotnet restore`:

```
dotnet restore
```

IMPORTANT

To add a missing package reference to the project file, use `dotnet add package`, which also runs the `restore` command.

Restore using the nuget.exe CLI

Use the `restore` command, which downloads and installs any packages missing from the *packages* folder.

For projects migrated to [PackageReference](#), use `msbuild -t:restore` to restore packages instead.

`restore` only adds packages to disk but does not change a project's dependencies. To restore project dependencies, modify `packages.config`, then use the `restore` command.

As with the other `nuget.exe` CLI commands, first open a command line and switch to the directory that contains your project file.

To restore a package using `restore`:

```
nuget restore MySolution.sln
```

IMPORTANT

The `restore` command does not modify a project file or `packages.config`. To add a dependency, either add a package through the Package Manager UI or Console in Visual Studio, or modify `packages.config` and then run either `install` or `restore`.

Restore using MSBuild

To restore packages listed in the project file with `PackageReference`, use the the `msbuild -t:restore` command. This command is available only in NuGet 4.x+ and MSBuild 15.1+, which are included with Visual Studio 2017 and higher versions. Both `nuget restore` and `dotnet restore` use this command for applicable projects.

1. Open a Developer command prompt (In the **Search** box, type **Developer command prompt**).

You typically want to start the Developer Command Prompt for Visual Studio from the **Start** menu, as it will be configured with all the necessary paths for MSBuild.

2. Switch to the folder containing the project file and type the following command.

```
# Uses the project file in the current folder by default  
msbuild -t:restore
```

3. Type the following command to rebuild the project.

```
msbuild
```

Make sure that the MSBuild output indicates that the build completed successfully.

Restore using Azure Pipelines

When you create a build definition in Azure Pipelines, include the NuGet `restore` or .NET Core `restore` task in the definition before any build tasks. Some build templates include the restore task by default.

Restore using Azure DevOps Server

Azure DevOps Server and TFS 2013 and later automatically restore packages during build, if you're using a TFS 2013 or later Team Build template. For earlier TFS versions, you can include a build step to run a command-line restore option, or optionally migrate the build template to a later version. For more information, see [Set up package restore with Team Foundation Build](#).

Constrain package versions with restore

When NuGet restores packages through any method, it honors any constraints you specified in `packages.config` or the project file:

- In `packages.config`, you can specify a version range in the `allowedVersion` property of the dependency. See [Constrain upgrade versions](#) for more information. For example:

```
<package id="Newtonsoft.json" version="6.0.4" allowedVersions="[6,7)" />
```

- In a project file, you can use `PackageReference` to specify a dependency's range directly. For example:

```
<PackageReference Include="Newtonsoft.json" Version="[6, 7)" />
```

In all cases, use the notation described in [Package versioning](#).

Force restore from package sources

By default, NuGet restore operations use packages from the *global-packages* and *http-cache* folders, which are described in [Manage the global packages and cache folders](#).

To avoid using the *global-packages* folder, do one of the following:

- Clear the folder using `nuget locals global-packages -clear` or `dotnet nuget locals global-packages --clear`.
- Temporarily change the location of the *global-packages* folder before the restore operation, using one of the following methods:
 - Set the `NUGET_PACKAGES` environment variable to a different folder.
 - Create a `NuGet.Config` file that sets `globalPackagesFolder` (if using `PackageReference`) or `repositoryPath` (if using `packages.config`) to a different folder. For more information, see [configuration settings](#).
 - MSBuild only: Specify a different folder with the `RestorePackagesPath` property.

To avoid using the cache for HTTP sources, do one of the following:

- Use the `-NoCache` option with `nuget restore`, or the `--no-cache` option with `dotnet restore`. These options don't affect restore operations through the Visual Studio Package Manager or console.
- Clear the cache using `nuget locals http-cache -clear` or `dotnet nuget locals http-cache --clear`.
- Temporarily set the `NUGET_HTTP_CACHE_PATH` environment variable to a different folder.

Migrate to automatic package restore (Visual Studio)

For NuGet 2.6 and earlier, an MSBuild-integrated package restore was previously supported but that is no longer true. (It was typically enabled by right-clicking a solution in Visual Studio and selecting **Enable NuGet Package Restore**). If your project uses the deprecated MSBuild-integrated package restore, please migrate to automatic package restore.

Projects that use MSBuild-Integrated package restore typically contain a `.nuget` folder with three files: `NuGet.config`, `nuget.exe`, and `NuGet.targets`. The presence of a `NuGet.targets` file determines whether NuGet will continue to use the MSBuild-untegrated approach, so this file must be removed during the migration.

To migrate to automatic package restore:

1. Close Visual Studio.
2. Delete `.nuget/nuget.exe` and `.nuget/NuGet.targets`.
3. For each project file, remove the `<RestorePackages>` element and remove any reference to `NuGet.targets`.

To test the automatic package restore:

1. Remove the `packages` folder from the solution.
2. Open the solution in Visual Studio and start a build.

Automatic package restore should download and install each dependency package, without adding them to source control.

Troubleshooting

See [Troubleshoot package restore](#).

Troubleshooting package restore errors

8/8/2019 • 5 minutes to read • [Edit Online](#)

This article focuses on common errors when restoring packages and steps to resolve them.

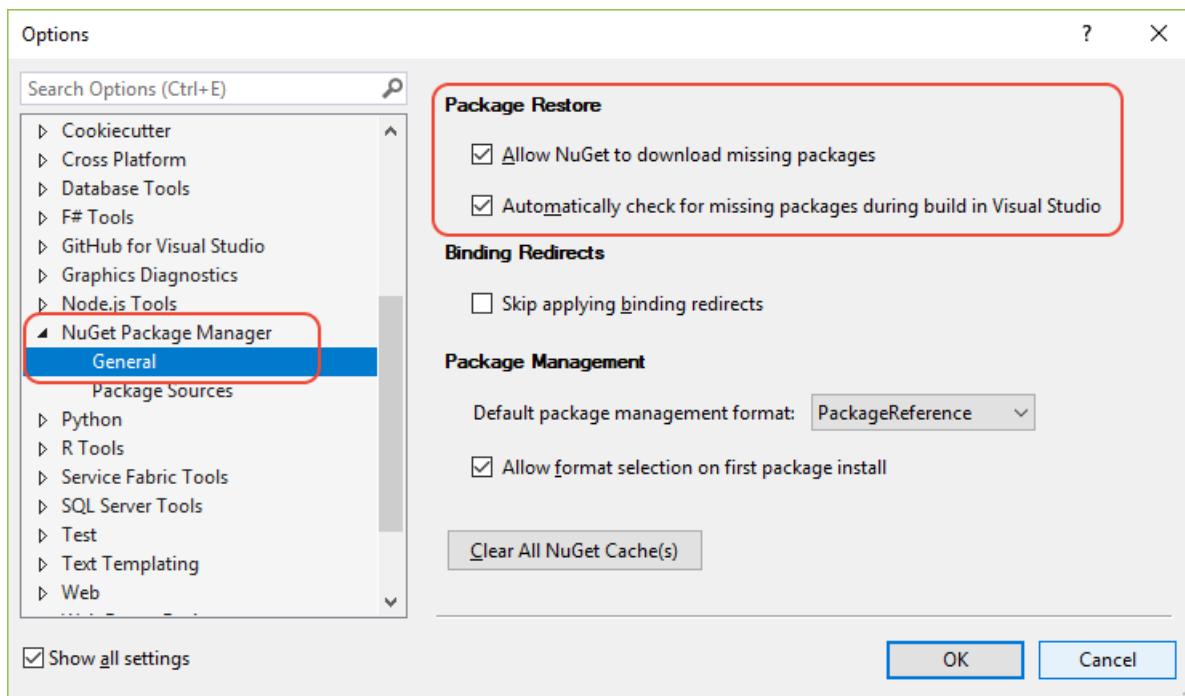
Package Restore tries to install all package dependencies to the correct state matching the package references in your project file (.csproj) or your *packages.config* file. (In Visual Studio, the references appear in Solution Explorer under the **Dependencies \ NuGet** or the **References** node.) To follow the required steps to restore packages, see [Restore packages](#). If the package references in your project file (.csproj) or your *packages.config* file are incorrect (they do not match your desired state following Package Restore), then you need to either install or update packages instead of using Package Restore.

If the instructions here do not work for you, [please file an issue on GitHub](#) so that we can examine your scenario more carefully. Do not use the "Is this page helpful?" control that may appear on this page because it doesn't give us the ability to contact you for more information.

Quick solution for Visual Studio users

If you're using Visual Studio, first enable package restore as follows. Otherwise continue to the sections that follow.

1. Select the **Tools > NuGet Package Manager > Package Manager Settings** menu command.
2. Set both options under **Package Restore**.
3. Select **OK**.
4. Build your project again.



These settings can also be changed in your `NuGet.config` file; see the [consent](#) section. If your project is an older project that uses MSBuild-integrated package restore, you may need to [migrate](#) to automatic package restore.

This project references NuGet package(s) that are missing on this computer

Complete error message:

This project references NuGet package(s) that are missing on this computer.
Use NuGet Package Restore to download them. The missing file is {name}.

This error occurs when you attempt to build a project that contains references to one or more NuGet packages, but those packages are not presently installed on the computer or in the project.

- When using the [PackageReference](#) management format, the error means that the package is not installed in the *global-packages* folder as described on [Managing the global packages and cache folders](#).
- When using [packages.config](#), the error means that the package is not installed in the `packages` folder at the solution root.

This situation commonly occurs when you obtain the project's source code from source control or another download. Packages are typically omitted from source control or downloads because they can be restored from package feeds like nuget.org (see [Packages and source control](#)). Including them would otherwise bloat the repository or create unnecessarily large .zip files.

The error can also happen if your project file contains absolute paths to package locations, and you move the project.

Use one of the following methods to restore the packages:

- If you've moved the project file, edit the file directly to update the package references.
- [Visual Studio \(automatic restore or manual restore\)](#)
- [dotnet CLI](#)
- [nuget.exe CLI](#)
- [MSBuild](#)
- [Azure Pipelines](#)
- [Azure DevOps Server](#)

After a successful restore, the package should be present in the *global-packages* folder. For projects using [PackageReference](#), a restore should recreate the `obj/project.assets.json` file; for projects using `packages.config`, the package should appear in the project's `packages` folder. The project should now build successfully. If not, [file an issue on GitHub](#) so we can follow up with you.

Assets file project.assets.json not found

Complete error message:

Assets file '<path>\project.assets.json' not found. Run a NuGet package restore to generate this file.

The `project.assets.json` file maintains a project's dependency graph when using the [PackageReference](#) management format, which is used to make sure that all necessary packages are installed on the computer. Because this file is generated dynamically through package restore, it's typically not added to source control. As a result, this error occurs when building a project with a tool such as `msbuild` that does not automatically restore packages.

In this case, run `msbuild -t:restore` followed by `msbuild`, or use `dotnet build` (which restores packages automatically). You can also use any of the package restore methods in the [previous section](#).

One or more NuGet packages need to be restored but couldn't be because consent has not been granted

Complete error message:

One or more NuGet packages need to be restored but couldn't be because consent has not been granted. To give consent, open the Visual Studio Options dialog, click on the NuGet Package Manager node and check 'Allow NuGet to download missing packages during build.' You can also give consent by setting the environment variable 'EnableNuGetPackageRestore' to 'true'. Missing packages: {name}

This error indicates that package restore is disabled in your NuGet configuration.

You can change the applicable settings in Visual Studio as described earlier under [Quick solution for Visual Studio users](#).

You can also edit these settings directly in the applicable `nuget.config` file (typically `%AppData%\NuGet\NuGet.Config` on Windows and `~/.nuget/NuGet.Config` on Mac/Linux). Make sure the `enabled` and `automatic` keys under `packageRestore` are set to True:

```
<!-- Package restore is enabled -->
<configuration>
  <packageRestore>
    <add key="enabled" value="True" />
    <add key="automatic" value="True" />
  </packageRestore>
</configuration>
```

IMPORTANT

If you edit the `packageRestore` settings directly in `nuget.config`, restart Visual Studio so that the options dialog box shows the current values.

Other potential conditions

- You may encounter build errors due to missing files, with a message saying to use NuGet restore to download them. However, running a restore might say, "All packages are already installed and there is nothing to restore." In this case, delete the `packages` folder (when using `packages.config`) or the `obj/project.assets.json` file (when using `PackageReference`) and run restore again. If the error still persists, use `nuget locals all -clear` or `dotnet locals all --clear` from the command line to clear the *global-packages* and cache folders as described on [Managing the global packages and cache folders](#).
- When obtaining a project from source control, your project folders may be set to read-only. Change the folder permissions and try restoring packages again.
- You may be using an old version of NuGet. Check [nuget.org/downloads](#) for the latest recommended versions. For Visual Studio 2015, we recommend 3.6.0.

If you encounter other problems, [file an issue on GitHub](#) so we can get more details from you.

How to reinstall and update packages

8/15/2019 • 6 minutes to read • [Edit Online](#)

There are a number of situations, described below under [When to Reinstall a Package](#), where references to a package might get broken within a Visual Studio project. In these cases, uninstalling and then reinstalling the same version of the package will restore those references to working order. Updating a package simply means installing an updated version, which often restores a package to working order.

In Visual Studio, the Package Manager Console provides many flexible options for updating and reinstalling packages.

Updating and reinstalling packages is accomplished as follows:

METHOD	UPDATE	REINSTALL
Package Manager console (described in Using Update-Package)	<code>Update-Package</code> command	<code>Update-Package -reinstall</code> command
Package Manager UI	On the Updates tab, select one or more packages and select Update	On the Installed tab, select a package, record its name, then select Uninstall . Switch to the Browse tab, search for the package name, select it, then select Install).
nuget.exe CLI	<code>nuget update</code> command	For all packages, delete the package folder, then run <code>nuget install</code> . For a single package, delete the package folder and use <code>nuget install <id></code> to reinstall the same one.

NOTE

For the dotnet CLI, the equivalent procedure is not required. In a similar scenario, you can [restore packages with the dotnet CLI](#).

In this article:

- [When to Reinstall a Package](#)
- [Constraining upgrade versions](#)

When to Reinstall a Package

1. **Broken references after package restore:** If you've opened a project and restored NuGet packages, but still see broken references, try reinstalling each of those packages.
2. **Project is broken due to deleted files:** NuGet does not prevent you from removing items added from packages, so it's easy to inadvertently modify contents installed from a package and break your project. To restore the project, reinstall the affected packages.
3. **Package update broke the project:** If an update to a package breaks a project, the failure is generally caused by a dependency package which may have also been updated. To restore the state of the dependency, reinstall that specific package.
4. **Project retargeting or upgrade:** This can be useful when a project has been retargeted or upgraded and if the

package requires reinstallation due to the change in target framework. NuGet shows a build error in such cases immediately after project retargeting, and subsequent build warnings let you know that the package may need to be reinstalled. For project upgrade, NuGet shows an error in the Project Upgrade Log.

5. **Reinstalling a package during its development:** Package authors often need to reinstall the same version of package they're developing to test the behavior. The `Install-Package` command does not provide an option to force a reinstall, so use `Update-Package -reinstall` instead.

Constraining upgrade versions

By default, reinstalling or updating a package *always* installs the latest version available from the package source.

In projects using the `packages.config` management format, however, you can specifically constrain the version range. For example, if you know that your application works only with version 1.x of a package but not 2.0 and above, perhaps due to a major change in the package API, then you'd want to constrain upgrades to 1.x versions. This prevents accidental updates that would break the application.

To set a constraint, open `packages.config` in a text editor, locate the dependency in question, and add the `allowedVersions` attribute with a version range. For example, to constrain updates to version 1.x, set `allowedVersions` to `[1,2)`:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="ExamplePackage" version="1.1.0" allowedVersions="[1,2)" />
  <!-- ... -->
</packages>
```

In all cases, use the notation described in [Package versioning](#).

Using Update-Package

Being mindful of the [Considerations](#) described below, you can easily reinstall any package using the [Update-Package command](#) in the Visual Studio Package Manager Console (**Tools > NuGet Package Manager > Package Manager Console**).

```
Update-Package -Id <package_name> -reinstall
```

Using this command is much easier than removing a package and then trying to locate the same package in the NuGet gallery with the same version. Note that the `-Id` switch is optional.

The same command without `-reinstall` updates a package to a newer version, if applicable. The command gives an error if the package in question is not already installed in a project; that is, `Update-Package` does not install packages directly.

```
Update-Package <package_name>
```

By default, `Update-Package` affects all projects in a solution. To limit the action to a specific project, use the `-ProjectName` switch, using the name of the project as it appears in Solution Explorer:

```
# Reinstall the package in just MyProject
Update-Package <package_name> -ProjectName MyProject -reinstall
```

To *update* all packages in a project (or reinstall using `-reinstall`), use `-ProjectName` without specifying any

particular package:

```
Update-Package -ProjectName MyProject
```

To update all packages in a solution, just use `Update-Package` by itself with no other arguments or switches. Use this form carefully, because it can take considerable time to perform all the updates:

```
# Updates all packages in all projects in the solution
Update-Package
```

Updating packages in a project or solution using [PackageReference](#) always updates to the latest version of the package (excluding pre-release packages). Projects that use `packages.config` can, if desired, limit update versions as described below in [Constraining upgrade versions](#).

For full details on the command, see the [Update-Package](#) reference.

Considerations

The following may be affected when reinstalling a package:

1. Reinstalling packages according to project target framework retargeting

- In a simple case, just reinstalling a package using `Update-Package -reinstall <package_name>` works. A package that is installed against an old target framework gets uninstalled and the same package gets installed against the current target framework of the project.
- In some cases, there may be a package that does not support the new target framework.
 - If a package supports portable class libraries (PCLs) and the project is retargeted to a combination of platforms no longer supported by the package, references to the package will be missing after reinstalling.
 - This can surface for packages you're using directly or for packages installed as dependencies. It's possible for the package you're using directly to support the new target framework while its dependency does not.
 - If reinstalling packages after retargeting your application results in build or runtime errors, you may need to revert your target framework or search for alternative packages that properly support your new target framework.

2. requireReinstallation attribute added in packages.config after project retargeting or upgrade

- If NuGet detects that packages were affected by retargeting or upgrading a project, it adds `requireReinstallation="true"` attribute in `packages.config` to all affected package references. Because of this, each subsequent build in Visual Studio raises build warnings for those packages so you can remember to reinstall them.

3. Reinstalling packages with dependencies

- `Update-Package -reinstall` reinstalls the same version of the original package, but installs the latest version of dependencies unless specific version constraints are provided. This allows you to update only the dependencies as required to fix an issue. However, if this rolls a dependency back to an earlier version, you can use `Update-Package <dependency_name>` to reinstall that one dependency without affecting the dependent package.
- `Update-Package -reinstall <packageName> -ignoreDependencies` reinstalls the same version of the original package but does not reinstall dependencies. Use this when updating package dependencies might result in a broken state

4. Reinstalling packages when dependent versions are involved

- As explained above, reinstalling a package does not change versions of any other installed packages that

depend on it. It's possible, then, that reinstalling a dependency could break the dependent package.

Managing the global packages, cache, and temp folders

7/23/2019 • 4 minutes to read • [Edit Online](#)

Whenever you install, update, or restore a package, NuGet manages packages and package information in several folders outside of your project structure:

NAME	DESCRIPTION AND LOCATION (PER USER)
global-packages	<p>The <i>global-packages</i> folder is where NuGet installs any downloaded package. Each package is fully expanded into a subfolder that matches the package identifier and version number. Projects using the PackageReference format always use packages directly from this folder. When using the packages.config, packages are installed to the <i>global-packages</i> folder, then copied into the project's packages folder.</p> <ul style="list-style-type: none">Windows: <code>%userprofile%\.nuget\packages</code>Mac/Linux: <code>~/.nuget/packages</code>Override using the <code>NUGET_PACKAGES</code> environment variable, the <code>globalPackagesFolder</code> or <code>repositoryPath</code> configuration settings (when using <code>PackageReference</code> and <code>packages.config</code>, respectively), or the <code>RestorePackagesPath</code> MSBuild property (MSBuild only). The environment variable takes precedence over the configuration setting.
http-cache	<p>The Visual Studio Package Manager (NuGet 3.x+) and the <code>dotnet</code> tool store copies of downloaded packages in this cache (saved as <code>.dat</code> files), organized into subfolders for each package source. Packages are not expanded, and the cache has an expiration time of 30 minutes.</p> <ul style="list-style-type: none">Windows: <code>%localappdata%\NuGet\v3-cache</code>Mac/Linux: <code>~/.local/share/NuGet/v3-cache</code>Override using the <code>NUGET_HTTP_CACHE_PATH</code> environment variable.
temp	<p>A folder where NuGet stores temporary files during its various operations.</p> <ul style="list-style-type: none">Windows: <code>%temp%\NuGetScratch</code>Mac/Linux: <code>/tmp/NuGetScratch</code>
plugins-cache 4.8+	<p>A folder where NuGet stores the results from the operation claims request.</p> <ul style="list-style-type: none">Windows: <code>%localappdata%\NuGet\plugins-cache</code>Mac/Linux: <code>~/.local/share/NuGet/plugins-cache</code>Override using the <code>NUGET_PLUGINS_CACHE_PATH</code> environment variable.

NOTE

NuGet 3.5 and earlier uses `packages-cache` instead of the `http-cache`, which is located in `%localappdata%\NuGet\Cache`.

By using the cache and *global-packages* folders, NuGet generally avoids downloading packages that already exist on the computer, improving the performance of install, update, and restore operations. When using `PackageReference`, the *global-packages* folder also avoids keeping downloaded packages inside project folders, where they might be inadvertently added to source control, and reduces NuGet's overall impact on computer storage.

When asked to retrieve a package, NuGet first looks in the *global-packages* folder. If the exact version of package is not there, then NuGet checks all non-HTTP package sources. If the package is still not found, NuGet looks for the package in the `http-cache` unless you specify `--no-cache` with `dotnet.exe` commands or `-NoCache` with `nuget.exe` commands. If the package is not in the cache, or the cache isn't used, NuGet then retrieves the package over HTTP.

For more information, see [What happens when a package is installed?](#).

Viewing folder locations

You can view locations using the [nuget locals command](#):

```
# Display locals for all folders: global-packages, http cache, temp and plugins cache
nuget locals all -list
```

Typical output (Windows; "user1" is the current username):

```
http-cache: C:\Users\user1\AppData\Local\NuGet\v3-cache
global-packages: C:\Users\user1\.nuget\packages\
temp: C:\Users\user1\AppData\Local\Temp\NuGetScratch
plugins-cache: C:\Users\user1\AppData\Local\NuGet\plugins-cache
```

(`package-cache` is used in NuGet 2.x and appears with NuGet 3.5 and earlier.)

You can also view folder locations using the [dotnet nuget locals command](#):

```
dotnet nuget locals all --list
```

Typical output (Mac/Linux; "user1" is the current username):

```
info : http-cache: /home/user1/.local/share/NuGet/v3-cache
info : global-packages: /home/user1/.nuget/packages/
info : temp: /tmp/NuGetScratch
info : plugins-cache: /home/user1/.local/share/NuGet/plugins-cache
```

To display the location of a single folder, use `http-cache`, `global-packages`, `temp`, or `plugins-cache` instead of `all`.

Clearing local folders

If you encounter package installation problems or otherwise want to ensure that you're installing packages from a remote gallery, use the `locals --clear` option (`dotnet.exe`) or `locals -clear` (`nuget.exe`), specifying the folder to clear, or `all` to clear all folders:

```

# Clear the 3.x+ cache (use either command)
dotnet nuget locals http-cache --clear
nuget locals http-cache -clear

# Clear the 2.x cache (NuGet CLI 3.5 and earlier only)
nuget locals packages-cache -clear

# Clear the global packages folder (use either command)
dotnet nuget locals global-packages --clear
nuget locals global-packages -clear

# Clear the temporary cache (use either command)
dotnet nuget locals temp --clear
nuget locals temp -clear

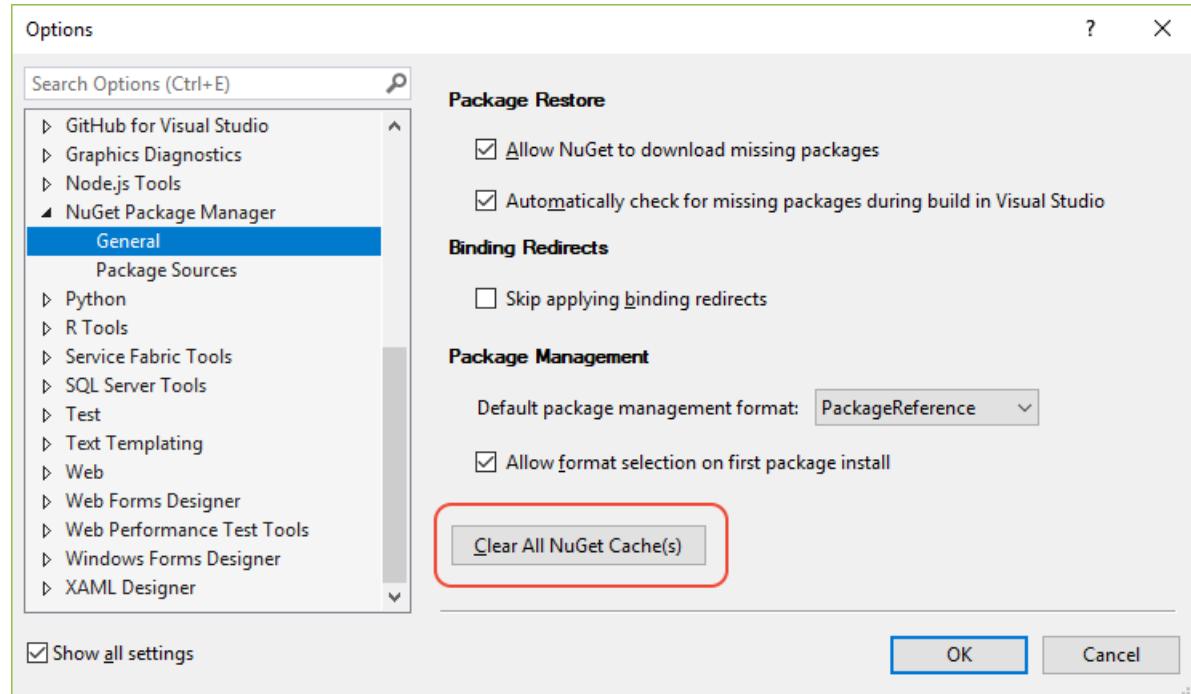
# Clear the plugins cache (use either command)
dotnet nuget locals plugins-cache --clear
nuget locals plugins-cache -clear

# Clear all caches (use either command)
dotnet nuget locals all --clear
nuget locals all -clear

```

Any packages used by projects that are currently open in Visual Studio are not cleared from the *global-packages* folder.

Starting in Visual Studio 2017, use the **Tools > NuGet Package Manager > Package Manager Settings** menu command, then select **Clear All NuGet Cache(s)**. Managing the cache isn't presently available through the Package Manager Console. In Visual Studio 2015, use the CLI commands instead.



Troubleshooting errors

The following errors can occur when using `nuget locals` or `dotnet nuget locals`:

- *Error: The process cannot access the file because it is being used by another process or Clearing local resources failed: Unable to delete one or more files*

One or more files in the folder are in use by another process; for example, a Visual Studio project is open that refers to packages in the *global-packages* folder. Close those processes and try again.

- *Error: Access to the path is denied or The directory is not empty*

You don't have permission to delete files in the cache. Change the folder permissions, if possible, and try again. Otherwise, contact your system administrator.

- *Error: The specified path, file name, or both are too long. The fully qualified file name must be less than 260 characters, and the directory name must be less than 248 characters.*

Shorten the folder names and try again.

Manage package trust boundaries

11/14/2019 • 2 minutes to read • [Edit Online](#)

Signed packages don't require any specific action to be installed; however, if the content has been modified since it was signed, the installation is blocked with error [NU3008](#).

WARNING

Packages signed with untrusted certificates are considered as unsigned and are installed without any warnings or errors like any other unsigned package.

Configure package signature requirements

NOTE

Requires NuGet 4.9.0+ and Visual Studio version 15.9 and later on Windows

You can configure how NuGet clients validate package signatures by setting the `signatureValidationMode` to `require` in the `nuget.config` file using the `nuget config` command.

```
nuget.exe config -set signatureValidationMode=require
```

```
<config>
  <add key="signatureValidationMode" value="require" />
</config>
```

This mode will verify that all packages are signed by any of the certificates trusted in the `nuget.config` file. This file allows you to specify which authors and/or repositories are trusted based on the certificate's fingerprint.

Trust package author

To trust packages based on the author signature use the `trusted-signers` command to set the `author` property in the `nuget.config`.

```
nuget.exe trusted-signers Add -Name MyCompanyCert -CertificateFingerprint
CE40881FF5F0AD3E58965DA20A9F571EF1651A56933748E1BF1C99E537C4E039 -FingerprintAlgorithm SHA256
```

```
<trustedSigners>
  <author name="MyCompanyCert">
    <certificate fingerprint="CE40881FF5F0AD3E58965DA20A9F571EF1651A56933748E1BF1C99E537C4E039"
hashAlgorithm="SHA256" allowUntrustedRoot="false" />
  </author>
</trustedSigners>
```

TIP

Use the `nuget.exe verify` command to get the `SHA256` value of the certificate's fingerprint.

Trust all packages from a repository

To trust packages based on the repository signature use the `repository` element:

```
<trustedSigners>
  <repository name="nuget.org" serviceIndex="https://api.nuget.org/v3/index.json">
    <certificate fingerprint="0E5F38F57DC1BCC806D8494F4F90FBCEDD988B4676070...."
      hashAlgorithm="SHA256"
      allowUntrustedRoot="false" />
  </repository>
</trustedSigners>
```

Trust Package Owners

Repository signatures include additional metadata to determine the owners of the package at the time of submission. You can restrict packages from a repository based on a list of owners:

```
<trustedSigners>
  <repository name="nuget.org" serviceIndex="https://api.nuget.org/v3/index.json">
    <certificate fingerprint="0E5F38F57DC1BCC806D8494F4F90FBCEDD988B4676070...."
      hashAlgorithm="SHA256"
      allowUntrustedRoot="false" />
    <owners>microsoft;nuget</owners>
  </repository>
</trustedSigners>
```

If a package has multiple owners, and any one of those owners is in the trusted list, the package installation will succeed.

Untrusted Root certificates

In some situations you may want to enable verification using certificates that do not chain to a trusted root in the local machine. You can use the `allowUntrustedRoot` attribute to customize this behavior.

Sync repository certificates

Package repositories should announce the certificates they use in their [service index](#). Eventually the repository will update these certificates, e.g. when the certificate expires. When that happens, clients with specific policies will require an update to the configuration to include the newly added certificate. You can easily upgrade the trusted signers associated to a repository by using the [nuget.exe trusted-signers sync command](#).

Schema reference

The complete schema reference for the client policies can be found in the [nuget.config reference](#)

Related articles

- [Signing NuGet Packages](#)
- [Signed Packages Reference](#)

Omitting NuGet packages in source control systems

8/14/2019 • 2 minutes to read • [Edit Online](#)

Developers typically omit NuGet packages from their source control repositories and rely instead on [package restore](#) to reinstall a project's dependencies before a build.

The reasons for relying on package restore include the following:

1. Distributed version control systems, such as Git, include full copies of every version of every file within the repository. Binary files that are frequently updated lead to significant bloat and lengthens the time it takes to clone the repository.
2. When packages are included in the repository, developers are liable to add references directly to package contents on disk rather than referencing packages through NuGet, which can lead to hard-coded path names in the project.
3. It becomes harder to clean your solution of any unused package folders, as you need to ensure you don't delete any package folders still in use.
4. By omitting packages, you maintain clean boundaries of ownership between your code and the packages from others that you depend upon. Many NuGet packages are maintained in their own source control repositories already.

Although package restore is the default behavior with NuGet, some manual work is necessary to omit packages—namely, the `packages` folder in your project—from source control, as described in this article.

Omitting packages with Git

Use the [.gitignore file](#) to ignore NuGet packages (`.nupkg`) the `packages` folder, and `project.assets.json`, among other things. For reference, see the [sample .gitignore for Visual Studio projects](#):

The important parts of the `.gitignore` file are:

```
# Ignore NuGet Packages
*.nupkg

# The packages folder can be ignored because of Package Restore
**/[Pp]ackages/*

# except build/, which is used as an MSBuild target.
!**/[Pp]ackages/build/

# Uncomment if necessary however generally it will be regenerated when needed
#!**/[Pp]ackages/repositories.config

# NuGet v3's project.json files produces more ignorable files
*.nuget.props
*.nuget.targets

# Ignore other intermediate files that NuGet might create. project.lock.json is used in conjunction
# with project.json (NuGet v3); project.assets.json is used in conjunction with the PackageReference
# format (NuGet v4 and .NET Core).
project.lock.json
project.assets.json
```

Omitting packages with Team Foundation Version Control

NOTE

Follow these instructions if possible *before* adding your project to source control. Otherwise, manually delete the `packages` folder from your repository and check in that change before continuing.

To disable source control integration with TFVC for selected files:

1. Create a folder called `.nuget` in your solution folder (where the `.sln` file is).
 - Tip: on Windows, to create this folder in Windows Explorer, use the name `.nuget.` *with* the trailing dot.
2. In that folder, create a file named `NuGet.Config` and open it for editing.
3. Add the following text as a minimum, where the `disableSourceControlIntegration` setting instructs Visual Studio to skip everything in the `packages` folder:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <solution>
    <add key="disableSourceControlIntegration" value="true" />
  </solution>
</configuration>
```

4. If you are using TFS 2010 or earlier, cloak the `packages` folder in your workspace mappings.
5. On TFS 2012 or later, or with Visual Studio Team Services, create a `.tfignore` file as described on [Add Files to the Server](#). In that file, include the content below to explicitly ignore modifications to the `\packages` folder on the repository level and a few other intermediate files. (You can create the file in Windows Explorer using the name a `.tfignore.` with the trailing dot, but you might need to disable the "Hide known file extensions" option first):

```
# Ignore NuGet Packages
*.nupkg

# Ignore the NuGet packages folder in the root of the repository. If needed, prefix 'packages'
# with additional folder names if it's not in the same folder as .tfignore.
packages

# Omit temporary files
project.lock.json
project.assets.json
*.nuget.props
```

6. Add `NuGet.Config` and `.tfignore` to source control and check in your changes.

Common NuGet configurations

9/18/2019 • 8 minutes to read • [Edit Online](#)

NuGet's behavior is driven by the accumulated settings in one or more `NuGet.Config` (XML) files that can exist at project-, user-, and computer-wide levels. A global `NuGetDefaults.Config` file also specifically configures package sources. Settings apply to all commands issued in the CLI, the Package Manager Console, and the Package Manager UI.

Config file locations and uses

SCOPE	NUGET.CONFIG FILE LOCATION	DESCRIPTION
Solution	Current folder (aka Solution folder) or any folder up to the drive root.	In a solution folder, settings apply to all projects in subfolders. Note that if a config file is placed in a project folder, it has no effect on that project.
User	Windows: <code>%appdata%\NuGet\NuGet.Config</code> Mac/Linux: <code>~/.config/NuGet/NuGet.Config</code> or <code>~/.nuget/NuGet/NuGet.Config</code> (varies by OS distribution)	Settings apply to all operations, but are overridden by any project-level settings.
Computer	Windows: <code>%ProgramFiles(x86)%\NuGet\Config</code> Mac/Linux: <code>\$XDG_DATA_HOME</code> . If <code>\$XDG_DATA_HOME</code> is null or empty, <code>~/.local/share</code> or <code>/usr/local/share</code> will be used (varies by OS distribution)	Settings apply to all operations on the computer, but are overridden by any user- or project-level settings.

Notes for earlier versions of NuGet:

- NuGet 3.3 and earlier used a `.nuget` folder for solution-wide settings. This folder is not used in NuGet 3.4+.
- For NuGet 2.6 to 3.x, the computer-level config file on Windows was located in `%ProgramData%\NuGet\Config\{IDE}\{Version}\{SKU}\NuGet.Config`, where `{IDE}` can be `VisualStudio`, `{Version}` was the Visual Studio version such as `14.0`, and `{SKU}` is either `Community`, `Pro`, or `Enterprise`. To migrate settings to NuGet 4.0+, simply copy the config file to `%ProgramFiles(x86)%\NuGet\Config`. On Linux, this previous location was `/etc/opt`, and on Mac, `/Library/Application Support`.

Changing config settings

A `NuGet.Config` file is a simple XML text file containing key/value pairs as described in the [NuGet Configuration Settings](#) topic.

Settings are managed using the NuGet CLI [config command](#):

- By default, changes are made to the user-level config file.
- To change settings in a different file, use the `-configFile` switch. In this case files can use any filename.
- Keys are always case sensitive.

- Elevation is required to change settings in the computer-level settings file.

WARNING

Although you can modify the file in any text editor, NuGet (v3.4.3 and later) silently ignores the entire configuration file if it contains malformed XML (mismatched tags, invalid quotation marks, etc.). This is why it's preferable to manage setting using `nuget config`.

Setting a value

Windows:

```
# Set repositoryPath in the user-level config file
nuget config -set repositoryPath=c:\packages

# Set repositoryPath in project-level files
nuget config -set repositoryPath=c:\packages -configfile c:\my.Config
nuget config -set repositoryPath=c:\packages -configfile .\myApp\NuGet.Config

# Set repositoryPath in the computer-level file (requires elevation)
nuget config -set repositoryPath=c:\packages -configfile %ProgramFiles(x86)%\NuGet\Config\NuGet.Config
```

Mac/Linux:

```
# Set repositoryPath in the user-level config file
nuget config -set repositoryPath=/home/packages

# Set repositoryPath in project-level files
nuget config -set repositoryPath=/home/projects/packages -configfile /home/my.Config
nuget config -set repositoryPath=/home/packages -configfile home/myApp/NuGet.Config

# Set repositoryPath in the computer-level file (requires elevation)
nuget config -set repositoryPath=/home/packages -configfile $XDG_DATA_HOME/NuGet.Config
```

NOTE

In NuGet 3.4 and later you can use environment variables in any value, as in `repositoryPath=%PACKAGEHOME%` (Windows) and `repositoryPath=$PACKAGEHOME` (Mac/Linux).

Removing a value

To remove a value, specify a key with an empty value.

```
# Windows
nuget config -set repositoryPath= -configfile c:\my.Config

# Mac/Linux
nuget config -set repositoryPath= -configfile /home/my.Config
```

Creating a new config file

Copy the template below into the new file and then use `nuget config -configFile <filename>` to set values:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
</configuration>
```

How settings are applied

Multiple `NuGet.Config` files allow you to store settings in different locations so that they apply to a single project, a group of projects, or all projects. These settings collectively apply to any NuGet operation invoked from the command line or from Visual Studio, with settings that exist "closest" to a project or the current folder taking precedence.

Specifically, NuGet loads settings from the different config files in the following order:

1. The [NuGetDefaults.Config file](#), which contains settings related only to package sources.
2. The computer-level file.
3. The user-level file.
4. The file specified with `-configFile`.
5. Files found in every folder in the path from the drive root to the current folder (where nuget.exe is invoked or the folder containing the Visual Studio project). For example, if a command is invoked in `c:\A\B\C`, NuGet looks for and loads config files in `c`; then `c\A`, then `c\A\B`, and finally `c\A\B\C`.

As NuGet finds settings in these files, they are applied as follows:

1. For single-item elements, NuGet replaced any previously-found value for the same key. This means that settings that are "closest" to the current folder or project override any others found earlier. For example, the `defaultPushSource` setting in `NuGetDefaults.Config` is overridden if it exists in any other config file.
2. For collection elements (such as `<packageSources>`), NuGet combines the values from all configuration files into a single collection.
3. When `<clear />` is present for a given node, NuGet ignores previously defined configuration values for that node.

Settings walkthrough

Let's say you have the following folder structure on two separate drives:

```
disk_drive_1
  User
disk_drive_2
  Project1
    Source
  Project2
    Source
  tmp
```

You then have four `NuGet.Config` files in the following locations with the given content. (The computer-level file is not included in this example, but would behave similarly to the user-level file.)

File A. User-level file, (`%appdata%\NuGet\NuGet.Config` on Windows, `~/.config/NuGet/NuGet.Config` on Mac/Linux):

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <activePackageSource>
    <add key="NuGet official package source" value="https://api.nuget.org/v3/index.json" />
  </activePackageSource>
</configuration>
```

File B. `disk_drive_2/NuGet.Config`:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <config>
    <add key="repositoryPath" value="disk_drive_2/tmp" />
  </config>
  <packageRestore>
    <add key="enabled" value="True" />
  </packageRestore>
</configuration>

```

File C. disk_drive_2/Project1/NuGet.Config:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <config>
    <add key="repositoryPath" value="External/Packages" />
    <add key="defaultPushSource" value="https://MyPrivateRepo/ES/api/v2/package" />
  </config>
  <packageSources>
    <clear /> <!-- ensure only the sources defined below are used -->
    <add key="MyPrivateRepo - ES" value="https://MyPrivateRepo/ES/nuget" />
  </packageSources>
</configuration>

```

File D. disk_drive_2/Project2/NuGet.Config:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <!-- Add this repository to the list of available repositories -->
    <add key="MyPrivateRepo - DQ" value="https://MyPrivateRepo/DQ/nuget" />
  </packageSources>
</configuration>

```

NuGet then loads and applies settings as follows, depending on where it's invoked:

- **Invoked from disk_drive_1/users:** Only the default repository listed in the user-level configuration file (A) is used, because that's the only file found on disk_drive_1.
- **Invoked from disk_drive_2 or disk_drive_2/tmp:** The user-level file (A) is loaded first, then NuGet goes to the root of disk_drive_2 and finds file (B). NuGet also looks for a configuration file in /tmp but does not find one. As a result, the default repository on nuget.org is used, package restore is enabled, and packages get expanded in disk_drive_2/tmp.
- **Invoked from disk_drive_2/Project1 or disk_drive_2/Project1/Source:** The user-level file (A) is loaded first, then NuGet loads file (B) from the root of disk_drive_2, followed by file (C). Settings in (C) override those in (B) and (A), so the `repositoryPath` where packages get installed is `disk_drive_2/Project1/External/Packages` instead of `disk_drive_2/tmp`. Also, because (C) clears `<packageSources>`, nuget.org is no longer available as a source leaving only `https://MyPrivateRepo/ES/nuget`.
- **Invoked from disk_drive_2/Project2 or disk_drive_2/Project2/Source:** The user-level file (A) is loaded first followed by file (B) and file (D). Because `packageSources` is not cleared, both `nuget.org` and `https://MyPrivateRepo/DQ/nuget` are available as sources. Packages get expanded in disk_drive_2/tmp as specified in (B).

NuGet defaults file

The `NuGetDefaults.Config` file exists to specify package sources from which packages are installed and updated, and to control the default target for publishing packages with `nuget push`. Because administrators can conveniently (using Group Policy, for example) deploy consistent `NuGetDefaults.Config` files to developer and build machines, they can ensure that everyone in the organization is using the correct package sources rather than nuget.org.

IMPORTANT

The `NuGetDefaults.Config` file never causes a package source to be removed from a developer's NuGet configuration.

That means if the developer has already used NuGet and therefore has the nuget.org package source registered, it won't be removed after the creation of a `NuGetDefaults.Config` file.

Furthermore, neither `NuGetDefaults.Config` nor any other mechanism in NuGet can prevent access to package sources like nuget.org. If an organization wishes to block such access, it must use other means such as firewalls to do so.

NuGetDefaults.Config location

The following table describes where the `NuGetDefaults.Config` file should be stored, depending on the target OS:

OS PLATFORM	NUGETDEFAULTS.CONFIG LOCATION
Windows	Visual Studio 2017 or NuGet 4.x+: <code>%ProgramFiles(x86)%\NuGet\Config</code> Visual Studio 2015 and earlier or NuGet 3.x and earlier: <code>%PROGRAMDATA%\NuGet</code>
Mac/Linux	<code>\$XDG_DATA_HOME</code> (typically <code>~/.local/share</code> or <code>/usr/local/share</code> , depending on OS distribution)

NuGetDefaults.Config settings

- `packageSources` : this collection has the same meaning as `packageSources` in regular config files and specifies the default sources. NuGet uses the sources in order when installing or updating packages in projects using the `packages.config` management format. For projects using the `PackageReference` format, NuGet uses local sources first, then sources on network shares, then HTTP sources, regardless of the order in the configuration files. NuGet always ignores the order of sources with restore operations.
- `disabledPackageSources` : this collection also has the same meaning as in `NuGet.Config` files, where each affected source is listed by its name and a true/false value indicating whether it's disabled. This allows the source name and URL to remain in `packageSources` without having it turned on by default. Individual developers can then re-enable the source by setting the source's value to false in other `NuGet.Config` files without having to find the correct URL again. This is also useful to supply developers with a full list of internal source URLs for an organization while enabling only an individual team's source by default.
- `defaultPushSource` : specifies the default target for `nuget push` operations, overriding the built-in default of nuget.org. Administrators can deploy this setting to avoid publishing internal packages to the public nuget.org by accident, as developers specifically need to use `nuget push -Source` to publish to nuget.org.

Example NuGetDefaults.Config and application

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!-- defaultPushSource key works like the 'defaultPushSource' key of NuGet.Config files. -->
    <!-- This can be used by administrators to prevent accidental publishing of packages to nuget.org. -->
    <config>
        <add key="defaultPushSource" value="https://contoso.com/packages/" />
    </config>

    <!-- Default Package Sources; works like the 'packageSources' section of NuGet.Config files. -->
    <!-- This collection cannot be deleted or modified but can be disabled/enabled by users. -->
    <packageSources>
        <add key="Contoso Package Source" value="https://contoso.com/packages/" />
        <add key="nuget.org" value="https://api.nuget.org/v3/index.json" />
    </packageSources>

    <!-- Default Package Sources that are disabled by default. -->
    <!-- Works like the 'disabledPackageSources' section of NuGet.Config files. -->
    <!-- Sources cannot be modified or deleted either but can be enabled/disabled by users. -->
    <disabledPackageSources>
        <add key="nuget.org" value="true" />
    </disabledPackageSources>
</configuration>
```

Package references (PackageReference) in project files

11/14/2019 • 9 minutes to read • [Edit Online](#)

Package references, using the `PackageReference` node, manage NuGet dependencies directly within project files (as opposed to a separate `packages.config` file). Using `PackageReference`, as it's called, doesn't affect other aspects of NuGet; for example, settings in `NuGet.config` files (including package sources) are still applied as explained in [Common NuGet configurations](#).

With `PackageReference`, you can also use MSBuild conditions to choose package references per target framework, or other groupings. It also allows for fine-grained control over dependencies and content flow. (See [For more details NuGet pack and restore as MSBuild targets](#).)

Project type support

By default, `PackageReference` is used for .NET Core projects, .NET Standard projects, and UWP projects targeting Windows 10 Build 15063 (Creators Update) and later, with the exception of C++ UWP projects. .NET Framework projects support `PackageReference`, but currently default to `packages.config`. To use `PackageReference`, [migrate](#) the dependencies from `packages.config` into your project file, then remove `packages.config`.

ASP.NET apps targeting the full .NET Framework include only [limited support](#) for `PackageReference`. C++ and JavaScript project types are unsupported.

Adding a PackageReference

Add a dependency in your project file using the following syntax:

```
<ItemGroup>
  <!-- ... -->
  <PackageReference Include="Contoso.Utility.UsefulStuff" Version="3.6.0" />
  <!-- ... -->
</ItemGroup>
```

Controlling dependency version

The convention for specifying the version of a package is the same as when using `packages.config`:

```
<ItemGroup>
  <!-- ... -->
  <PackageReference Include="Contoso.Utility.UsefulStuff" Version="3.6.0" />
  <!-- ... -->
</ItemGroup>
```

In the example above, 3.6.0 means any version that is $\geq 3.6.0$ with preference for the lowest version, as described on [Package versioning](#).

Using PackageReference for a project with no PackageReferences

Advanced: If you have no packages installed in a project (no `PackageReferences` in project file and no `packages.config` file), but want the project to be restored as `PackageReference` style, you can set a Project property `RestoreProjectStyle` to `PackageReference` in your project file.

```

<PropertyGroup>
  <!-- ... -->
  <RestoreProjectStyle>PackageReference</RestoreProjectStyle>
  <!-- ... -->
</PropertyGroup>

```

This may be useful, if you reference projects which are PackageReference styled (existing csproj or SDK-style projects). This will enable packages that those projects refer to, to be "transitively" referenced by your project.

PackageReference and sources

In PackageReference projects, the transitive dependency versions are resolved at restore time. As such, in PackageReference projects all sources need to be available for all restores.

Floating Versions

Floating versions are supported with `PackageReference` :

```

<ItemGroup>
  <!-- ... -->
  <PackageReference Include="Contoso.Utility.UsefulStuff" Version="3.6.*" />
  <PackageReference Include="Contoso.Utility.UsefulStuff" Version="3.6.0-beta*" />
  <!-- ... -->
</ItemGroup>

```

Controlling dependency assets

You might be using a dependency purely as a development harness and might not want to expose that to projects that will consume your package. In this scenario, you can use the `PrivateAssets` metadata to control this behavior.

```

<ItemGroup>
  <!-- ... -->

  <PackageReference Include="Contoso.Utility.UsefulStuff" Version="3.6.0">
    <PrivateAssets>all</PrivateAssets>
  </PackageReference>

  <!-- ... -->
</ItemGroup>

```

The following metadata tags control dependency assets:

TAG	DESCRIPTION	DEFAULT VALUE
IncludeAssets	These assets will be consumed	all
ExcludeAssets	These assets will not be consumed	none
PrivateAssets	These assets will be consumed but won't flow to the parent project	contentfiles;analyzers;build

Allowable values for these tags are as follows, with multiple values separated by a semicolon except with `all` and `none` which must appear by themselves:

VALUE	DESCRIPTION
compile	Contents of the <code>lib</code> folder and controls whether your project can compile against the assemblies within the folder
runtime	Contents of the <code>lib</code> and <code>runtimes</code> folder and controls whether these assemblies will be copied out to the build output directory
contentFiles	Contents of the <code>contentfiles</code> folder
build	<code>.props</code> and <code>.targets</code> in the <code>build</code> folder
buildMultitargeting	(4.0) <code>.props</code> and <code>.targets</code> in the <code>buildMultitargeting</code> folder, for cross-framework targeting
buildTransitive	(5.0+) <code>.props</code> and <code>.targets</code> in the <code>buildTransitive</code> folder, for assets that flow transitively to any consuming project. See the feature page.
analyzers	.NET analyzers
native	Contents of the <code>native</code> folder
none	None of the above are used.
all	All of the above (except <code>none</code>)

In the following example, everything except the content files from the package would be consumed by the project and everything except content files and analyzers would flow to the parent project.

```

<ItemGroup>
  <!-- ... -->

  <PackageReference Include="Contoso.Utility.UsefulStuff" Version="3.6.0">
    <IncludeAssets>all</IncludeAssets>
    <ExcludeAssets>contentFiles</ExcludeAssets>
    <PrivateAssets>contentFiles;analyzers</PrivateAssets>
  </PackageReference>

  <!-- ... -->
</ItemGroup>

```

Note that because `build` is not included with `PrivateAssets`, targets and props *will* flow to the parent project. Consider, for example, that the reference above is used in a project that builds a NuGet package called AppLogger. AppLogger can consume the targets and props from `Contoso.Utility.UsefulStuff`, as can projects that consume AppLogger.

NOTE

When `developmentDependency` is set to `true` in a `.nuspec` file, this marks a package as a development-only dependency, which prevents the package from being included as a dependency in other packages. With `PackageReference` (NuGet 4.8+), this flag also means that it will exclude compile-time assets from compilation. For more information, see [DevelopmentDependency support for PackageReference](#).

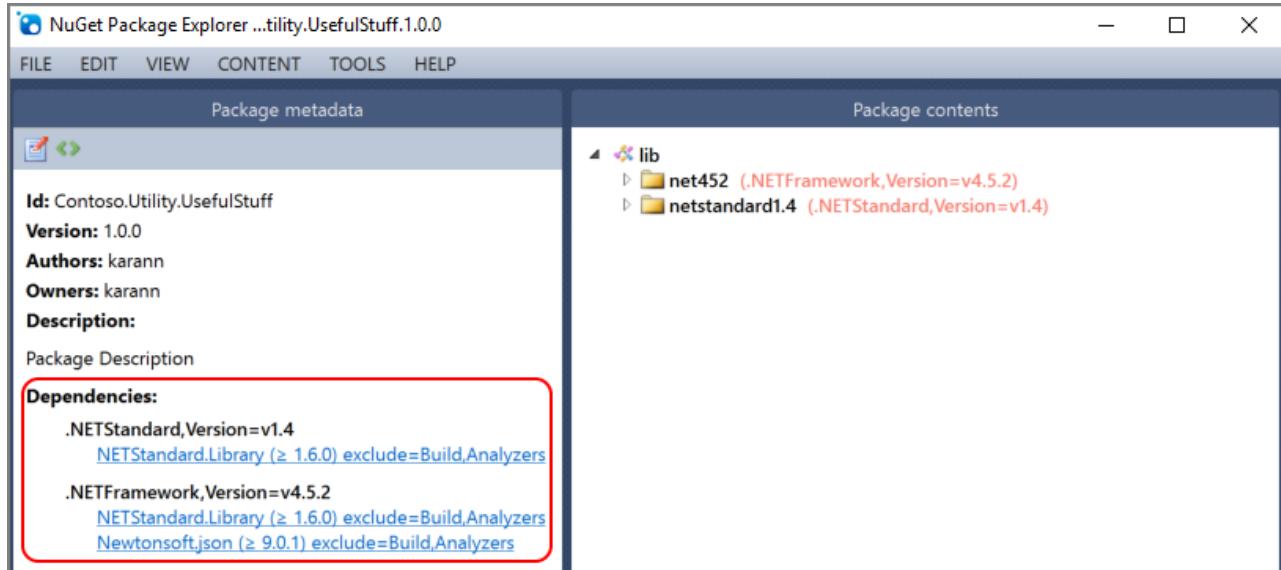
Adding a PackageReference condition

You can use a condition to control whether a package is included, where conditions can use any MSBuild variable or a variable defined in the targets or props file. However, at presently, only the `TargetFramework` variable is supported.

For example, say you're targeting `netstandard1.4` as well as `net452` but have a dependency that is applicable only for `net452`. In this case you don't want a `netstandard1.4` project that's consuming your package to add that unnecessary dependency. To prevent this, you specify a condition on the `PackageReference` as follows:

```
<ItemGroup>
  <!-- ... -->
  <PackageReference Include="Newtonsoft.Json" Version="9.0.1" Condition="\"$(TargetFramework)" == "net452\" />
  <!-- ... -->
</ItemGroup>
```

A package built using this project will show that `Newtonsoft.Json` is included as a dependency only for a `net452` target:



Conditions can also be applied at the `ItemGroup` level and will apply to all children `PackageReference` elements:

```
<ItemGroup Condition = "\"$(TargetFramework)" == "net452\">
  <!-- ... -->
  <PackageReference Include="Newtonsoft.Json" Version="9.0.1" />
  <PackageReference Include="Contoso.Utility.UsefulStuff" Version="3.6.0" />
  <!-- ... -->
</ItemGroup>
```

Locking dependencies

This feature is available with NuGet 4.9 or above and with Visual Studio 2017 15.9 or above.

Input to NuGet restore is a set of Package References from the project file (top-level or direct dependencies) and the output is a full closure of all the package dependencies including transitive dependencies. NuGet tries to always produce the same full closure of package dependencies if the input `PackageReference` list has not changed. However, there are some scenarios where it is unable to do so. For example:

- When you use floating versions like `<PackageReference Include="My.Sample.Lib" Version="4.*"/>`. While the intention here is to float to the latest version on every restore of packages, there are scenarios where users

require the graph to be locked to a certain latest version and float to a later version, if available, upon an explicit gesture.

- A newer version of the package matching `PackageReference` version requirements is published. E.g.
 - Day 1: if you specified `<PackageReference Include="My.Sample.Lib" Version="4.0.0"/>` but the versions available on the NuGet repositories were 4.1.0, 4.2.0 and 4.3.0. In this case, NuGet would have resolved to 4.1.0 (nearest minimum version)
 - Day 2: Version 4.0.0 gets published. NuGet will now find the exact match and start resolving to 4.0.0
- A given package version is removed from the repository. Though nuget.org does not allow package deletions, not all package repositories have this constraints. This results in NuGet finding the best match when it cannot resolve to the deleted version.

Enabling lock file

In order to persist the full closure of package dependencies you can opt-in to the lock file feature by setting the MSBuild property `RestorePackagesWithLockFile` for your project:

```
<PropertyGroup>
  <!-- ... -->
  <RestorePackagesWithLockFile>true</RestorePackagesWithLockFile>
  <!-- ... -->
</PropertyGroup>
```

If this property is set, NuGet restore will generate a lock file - `packages.lock.json` file at the project root directory that lists all the package dependencies.

NOTE

Once a project has `packages.lock.json` file in its root directory, the lock file is always used with restore even if the property `RestorePackagesWithLockFile` is not set. So another way to opt-in to this feature is to create a dummy blank `packages.lock.json` file in the project's root directory.

`restore` behavior with lock file

If a lock file is present for project, NuGet uses this lock file to run `restore`. NuGet does a quick check to see if there were any changes in the package dependencies as mentioned in the project file (or dependent projects' files) and if there were no changes it just restores the packages mentioned in the lock file. There is no re-evaluation of package dependencies.

If NuGet detects a change in the defined dependencies as mentioned in the project file(s), it re-evaluates the package graph and updates the lock file to reflect the new package closure for the project.

For CI/CD and other scenarios, where you would not want to change the package dependencies on the fly, you can do so by setting the `lockedmode` to `true`:

For `dotnet.exe`, run:

```
> dotnet.exe restore --locked-mode
```

For `msbuild.exe`, run:

```
> msbuild.exe -t:restore -p:RestoreLockedMode=true
```

You may also set this conditional MSBuild property in your project file:

```
<PropertyGroup>
  <!-- ... -->
  <RestoreLockedMode>true</RestoreLockedMode>
  <!-- ... -->
</PropertyGroup>
```

If locked mode is `true`, restore will either restore the exact packages as listed in the lock file or fail if you updated the defined package dependencies for the project after lock file was created.

Make lock file part of your source repository

If you are building an application, an executable and the project in question is at the start of the dependency chain then do check in the lock file to the source code repository so that NuGet can make use of it during restore.

However, if your project is a library project that you do not ship or a common code project on which other projects depend upon, you **should not** check in the lock file as part of your source code. There is no harm in keeping the lock file but the locked package dependencies for the common code project may not be used, as listed in the lock file, during the restore/build of a project that depends on this common-code project.

Eg.

```
ProjectA
  |-----> PackageX 2.0.0
  |-----> ProjectB
    |----->PackageX 1.0.0
```

If `ProjectA` has a dependency on a `PackageX` version `2.0.0` and also references `ProjectB` that depends on `PackageX` version `1.0.0`, then the lock file for `ProjectB` will list a dependency on `PackageX` version `1.0.0`.

However, when `ProjectA` is built, its lock file will contain a dependency on `PackageX` version `2.0.0` and **not** `1.0.0` as listed in the lock file for `ProjectB`. Thus, the lock file of a common code project has little say over the packages resolved for projects that depend on it.

Lock file extensibility

You can control various behaviors of restore with lock file as described below:

OPTION	MSBUILD EQUIVALENT OPTION	DESCRIPTION
<code>--use-lock-file</code>	<code>RestorePackagesWithLockFile</code>	Opts into the usage of a lock file.
<code>--locked-mode</code>	<code>RestoreLockedMode</code>	Enables locked mode for restore. This is useful in CI/CD scenarios where you want repeatable builds.
<code>--force-evaluate</code>	<code>RestoreForceEvaluate</code>	This option is useful with packages with floating version defined in the project. By default, NuGet restore will not update the package version automatically upon each restore unless you run restore with this option.

OPTION	MSBUILD EQUIVALENT OPTION	DESCRIPTION
--lock-file-path	NuGetLockFilePath	Defines a custom lock file location for a project. By default, NuGet supports <code>packages.lock.json</code> at the root directory. If you have multiple projects in the same directory, NuGet supports project specific lock file <code>packages.<project_name>.lock.json</code>

Migrate from packages.config to PackageReference

8/15/2019 • 5 minutes to read • [Edit Online](#)

Visual Studio 2017 Version 15.7 and later supports migrating a project from the [packages.config](#) management format to the [PackageReference](#) format.

Benefits of using PackageReference

- **Manage all project dependencies in one place:** Just like project to project references and assembly references, NuGet package references (using the `PackageReference` node) are managed directly within project files rather than using a separate packages.config file.
- **Uncluttered view of top-level dependencies:** Unlike packages.config, PackageReference lists only those NuGet packages you directly installed in the project. As a result, the NuGet Package Manager UI and the project file aren't cluttered with down-level dependencies.
- **Performance improvements:** When using PackageReference, packages are maintained in the *global-packages* folder (as described on [Managing the global packages and cache folders](#) rather than in a `packages` folder within the solution. As a result, PackageReference performs faster and consumes less disk space.
- **Fine control over dependencies and content flow:** Using the existing features of MSBuild allows you to [conditionally reference a NuGet package](#) and choose package references per target framework, configuration, platform, or other pivots.
- **PackageReference is under active development:** See [PackageReference issues on GitHub](#). packages.config is no longer under active development.

Limitations

- NuGet PackageReference is not available in Visual Studio 2015 and earlier. Migrated projects can be opened only in Visual Studio 2017 and later.
- Migration is not currently available for C++ and ASP.NET projects.
- Some packages may not be fully compatible with PackageReference. For more information, see [package compatibility issues](#).

Known Issues

1. The `Migrate packages.config to PackageReference...` option is not available in the right-click context menu

Issue

When a project is first opened, NuGet may not have initialized until a NuGet operation is performed. This causes the migration option to not show up in the right-click context menu on `packages.config` or `References`.

Workaround

Perform any one of the following NuGet actions:

- Open the Package Manager UI - Right-click on `References` and select `Manage NuGet Packages...`
- Open the Package Manager Console - From `Tools > NuGet Package Manager`, select `Package Manager Console`
- Run NuGet restore - Right-click on the solution node in the Solution Explorer and select `Restore NuGet Packages`
- Build the project which also triggers NuGet restore

You should now be able to see the migration option. Note that this option is not supported and will not show up for ASP.NET and C++ project types.

Migration steps

NOTE

Before migration begins, Visual Studio creates a backup of the project to allow you to [roll back to packages.config](#) if necessary.

1. Open a solution containing project using `packages.config`.
2. In **Solution Explorer**, right-click on the **References** node or the `packages.config` file and select **Migrate packages.config to PackageReference....**
3. The migrator analyzes the project's NuGet package references and attempts to categorize them into **Top-level dependencies** (NuGet packages that you installed directly) and **Transitive dependencies** (packages that were installed as dependencies of top-level packages).

NOTE

PackageReference supports transitive package restore and resolves dependencies dynamically, meaning that transitive dependencies need not be installed explicitly.

4. (Optional) You may choose to treat a NuGet package classified as a transitive dependency as a top-level dependency by selecting the **Top-Level** option for the package. This option is automatically set for packages containing assets that do not flow transitively (those in the `build`, `buildCrossTargeting`, `contentFiles`, or `analyzers` folders) and those marked as a development dependency (`developmentDependency = "true"`).
5. Review any [package compatibility issues](#).
6. Select **OK** to begin the migration.
7. At the end of the migration, Visual Studio provides a report with a path to the backup, the list of installed packages (top-level dependencies), a list of packages referenced as transitive dependencies, and a list of compatibility issues identified at the start of migration. The report is saved to the backup folder.
8. Validate that the solution builds and runs. If you encounter problems, [file an issue on GitHub](#).

How to roll back to packages.config

1. Close the migrated project.
2. Copy the project file and `packages.config` from the backup (typically `<solution_root>\MigrationBackup\<unique_guid>\<project_name>\`) to the project folder. Delete the `obj` folder if it exists in the project root directory.
3. Open the project.
4. Open the Package Manager Console using the **Tools > NuGet Package Manager > Package Manager Console** menu command.
5. Run the following command in the Console:

```
update-package -reinstall
```

Create a package after migration

Once the migration is complete, we recommend that you add a reference to the [nuget.build.tasks.pack](#) nuget package, and then use `msbuild -t:pack` to create the package. Although in some scenarios you could use `dotnet.exe pack` instead of `msbuild -t:pack`, it is not recommended.

Package compatibility issues

Some aspects that were supported in packages.config are not supported in PackageReference. The migrator analyzes and detects such issues. Any package that has one or more of the following issues may not behave as expected after the migration.

"install.ps1" scripts are ignored when the package is installed after the migration

Description	With PackageReference, install.ps1 and uninstall.ps1 PowerShell scripts are not executed while installing or uninstalling a package.
Potential impact	Packages that depend on these scripts to configure some behavior in the destination project might not work as expected.

"content" assets are not available when the package is installed after the migration

Description	Assets in a package's <code>content</code> folder are not supported with PackageReference and are ignored. PackageReference adds support for <code>contentFiles</code> to have better transitive support and shared content.
Potential impact	Assets in <code>content</code> are not copied into the project and project code that depends on the presence of those assets requires refactoring.

XDT transforms are not applied when the package is installed after the upgrade

Description	XDT transforms are not supported with PackageReference and <code>.xdt</code> files are ignored when installing or uninstalling a package.
Potential impact	XDT transforms are not applied to any project XML files, most commonly, <code>web.config.install.xdt</code> and <code>web.config.uninstall.xdt</code> , which means the project's <code>web.config</code> file is not updated when the package is installed or uninstalled.

Assemblies in the lib root are ignored when the package is installed after the migration

Description	With PackageReference, assemblies present at the root of <code>lib</code> folder without a target framework specific sub-folder are ignored. NuGet looks for a sub-folder matching the target framework moniker (TFM) corresponding to the project's target framework and installs the matching assemblies into the project.
Potential impact	Packages that do not have a sub-folder matching the target framework moniker (TFM) corresponding to the project's target framework may not behave as expected after the transition or fail installation during the migration

Found an issue? Report it!

If you run into a problem with the migration experience, please [file an issue on the NuGet GitHub repository](#).

packages.config reference

8/15/2019 • 2 minutes to read • [Edit Online](#)

The `packages.config` file is used in some project types to maintain the list of packages referenced by the project. This allows NuGet to easily restore the project's dependencies when the project is transported to a different machine, such as a build server, without all those packages.

If used, `packages.config` is typically located in a project root. It's automatically created when the first NuGet operation is run, but can also be created manually before running any commands such as `nuget restore`.

Projects that use [PackageReference](#) do not use `packages.config`.

Schema

The schema is simple: following the standard XML header is a single `<packages>` node that contains one or more `<package>` elements, one for each reference. Each `<package>` element can have the following attributes:

ATTRIBUTE	REQUIRED	DESCRIPTION
<code>id</code>	Yes	The identifier of the package, such as <code>Newtonsoft.json</code> or <code>Microsoft.AspNet.Mvc</code> .
<code>version</code>	Yes	The exact version of the package to install, such as <code>3.1.1</code> or <code>4.2.5.11-beta</code> . A version string must have at least three numbers; a fourth is optional, as is a pre-release suffix. Ranges are not allowed.
<code>targetFramework</code>	No	The target framework moniker (TFM) to apply when installing the package. This is initially set to the project's target when a package is installed. As a result, different <code><package></code> elements can have different TFMs. For example, if you create a project targeting <code>.NET 4.5.2</code> , packages installed at that point will use the TFM of <code>net452</code> . If you later retarget the project to <code>.NET 4.6</code> and add more packages, those will use TFM of <code>net46</code> . A mismatch between the project's target and <code>targetFramework</code> attributes will generate warnings, in which case you can reinstall the affected packages.

ATTRIBUTE	REQUIRED	DESCRIPTION
allowedVersions	No	A range of allowed versions for this package applied during package update (see Constraining upgrade versions). It does <i>not</i> affect what package is installed during an install or restore operation. See Package versioning for syntax. The PackageManager UI also disables all versions outside the allowed range.
developmentDependency	No	If the consuming project itself creates a NuGet package, setting this to <code>true</code> for a dependency prevents that package from being included when the consuming package is created. The default is <code>false</code> .

Examples

The following `packages.config` refers to two dependencies:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="jQuery" version="3.1.1" targetFramework="net46" />
  <package id="NLog" version="4.3.10" targetFramework="net46" />
</packages>
```

The following `packages.config` refers to nine packages, but `Microsoft.Net.Compilers` will not be included when building the consuming package because of the `developmentDependency` attribute. The reference to `Newtonsoft.Json` also restricts updates to 8.x and 9.x versions only.

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="Microsoft.CodeDom.Providers.DotNetCompilerPlatform" version="1.0.0" targetFramework="net46" />
  <package id="Microsoft.Net.Compilers" version="1.0.0" targetFramework="net46" developmentDependency="true" />
  <package id="Microsoft.Web.Infrastructure" version="1.0.0.0" targetFramework="net46" />
  <package id="Microsoft.Web.Xdt" version="2.1.1" targetFramework="net46" />
  <package id="Newtonsoft.Json" version="8.0.3" allowedVersions="[8,10)" targetFramework="net46" />
  <package id="NuGet.Core" version="2.11.1" targetFramework="net46" />
  <package id="NuGet.Server" version="2.11.2" targetFramework="net46" />
  <package id="RouteMagic" version="1.3" targetFramework="net46" />
  <package id="WebActivatorEx" version="2.1.0" targetFramework="net46" />
</packages>
```

Package creation workflow

8/15/2019 • 2 minutes to read • [Edit Online](#)

Creating a package starts with the compiled code (typically .NET assemblies) that you want to package and share with others, either through the public nuget.org gallery or a private gallery within your organization. The package can also include additional files such as a readme that is displayed when the package is installed, and can include transformations to certain project files.

A package can also serve to only pull in any number of other dependencies, without containing any code of its own. Such a package is a convenient way to deliver an SDK that's composed of multiple independent packages. In other cases, a package may contain only symbol (.pdb) files to aid debugging.

NOTE

When you create a package for use by other developers, it's important to understand that they are taking a dependency on your work. As such, creating and publishing a package also implies a commitment to fixing bugs and making other updates, or at the very least making the package available as open source so others can help to maintain it.

Whatever the case, creating a package begins with deciding its identifier, version number, license, copyright information, and any other necessary content. Once done, you can use the "pack" command to put everything together into a .nupkg file. This file can be published to a NuGet feed, like nuget.org.

TIP

A NuGet package with the .nupkg extension is simply a ZIP file. To easily examine any package's contents, change the extension to .zip and expand its contents as usual. Just be sure to change the extension back to .nupkg before attempting to upload it to a host.

To learn and understand the creation process, start with [Creating a package](#) which guides you through the core processes common to all packages.

From there, you can consider a number of other options for your package:

- [Supporting Multiple Target Frameworks](#) describes how to create a package with multiple variants for different .NET Frameworks.
- [Creating Localized Packages](#) describes how to structure a package with multiple language resources and how to use separate localized satellite packages.
- [Pre-release Packages](#) demonstrates how to release alpha, beta, and rc packages to those customers who are interested.
- [Source and Config File Transformations](#) describes how you can do both one-way token replacements in files that are added to a project, and modify `web.config` and `app.config` with settings that are also backed out when the package is uninstalled.
- [Symbol Packages](#) offers guidance for supplying symbols for your library that allow consumers to step into your code while debugging.
- [Package versioning](#) discusses how to identify the exact versions that you allow for your dependencies (other packages that you consume from your package).
- [Native Packages](#) describes the process for creating a package for C++ consumers.
- [Signing Packages](#) describes the process for adding a digital signature to a package.

When you're then ready to publish a package to nuget.org, follow the simple process in [Publish a package](#).

If you want to use a private feed instead of nuget.org, see the [Hosting Packages Overview](#)

Create a NuGet package using the dotnet CLI

10/15/2019 • 5 minutes to read • [Edit Online](#)

No matter what your package does or what code it contains, you use one of the CLI tools, either `nuget.exe` or `dotnet.exe`, to package that functionality into a component that can be shared with and used by any number of other developers. This article describes how to create a package using the dotnet CLI. To install the `dotnet` CLI, see [Install NuGet client tools](#). Starting in Visual Studio 2017, the dotnet CLI is included with .NET Core workloads.

For .NET Core and .NET Standard projects that use the [SDK-style format](#), and any other SDK-style projects, NuGet uses information in the project file directly to create a package. For step-by-step tutorials, see [Create .NET Standard Packages with dotnet CLI](#) or [Create .NET Standard Packages with Visual Studio](#).

`msbuild -t:pack` is functionality equivalent to `dotnet pack`. To build with MSBuild, see [Create a NuGet package using MSBuild](#).

IMPORTANT

This topic applies to [SDK-style](#) projects, typically .NET Core and .NET Standard projects.

Set properties

The following properties are required to create a package.

- `PackageId`, the package identifier, which must be unique across the gallery that hosts the package. If not specified, the default value is `AssemblyName`.
- `Version`, a specific version number in the form *Major.Minor.Patch[-Suffix]* where *-Suffix* identifies [pre-release versions](#). If not specified, the default value is 1.0.0.
- The package title as it should appear on the host (like [nuget.org](#))
- `Authors`, author and owner information. If not specified, the default value is `AssemblyName`.
- `Company`, your company name. If not specified, the default value is `AssemblyName`.

In Visual Studio, you can set these values in the project properties (right-click the project in Solution Explorer, choose **Properties**, and select the **Package** tab). You can also set these properties directly in the project files (`.csproj`).

```
<PropertyGroup>
  <PackageId>AppLogger</PackageId>
  <Version>1.0.0</Version>
  <Authors>your_name</Authors>
  <Company>your_company</Company>
</PropertyGroup>
```

IMPORTANT

Give the package an identifier that's unique across [nuget.org](#) or whatever package source you're using.

The following example shows a simple, complete project file with these properties included. (You can create a new default project using the `dotnet new classlib` command.)

```

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <PackageId>AppLogger</PackageId>
    <Version>1.0.0</Version>
    <Authors>your_name</Authors>
    <Company>your_company</Company>
  </PropertyGroup>
</Project>

```

You can also set the optional properties, such as `Title`, `PackageDescription`, and `PackageTags`, as described in [MSBuild pack targets](#), [Controlling dependency assets](#), and [NuGet metadata properties](#).

NOTE

For packages built for public consumption, pay special attention to the `PackageTags` property, as tags help others find your package and understand what it does.

For details on declaring dependencies and specifying version numbers, see [Package references in project files](#) and [Package versioning](#). It is also possible to surface assets from dependencies directly in the package by using the `<IncludeAssets>` and `<ExcludeAssets>` attributes. For more information, see [Controlling dependency assets](#).

Choose a unique package identifier and set the version number

The package identifier and the version number are the two most important values in the project because they uniquely identify the exact code that's contained in the package.

Best practices for the package identifier:

- **Uniqueness:** The identifier must be unique across nuget.org or whatever gallery hosts the package. Before deciding on an identifier, search the applicable gallery to check if the name is already in use. To avoid conflicts, a good pattern is to use your company name as the first part of the identifier, such as `Contoso.`.
- **Namespace-like names:** Follow a pattern similar to namespaces in .NET, using dot notation instead of hyphens. For example, use `Contoso.Utility.UsefulStuff` rather than `Contoso-Utility-UsefulStuff` or `Contoso.Utility_UsefulStuff`. Consumers also find it helpful when the package identifier matches the namespaces used in the code.
- **Sample Packages:** If you produce a package of sample code that demonstrates how to use another package, attach `.Sample` as a suffix to the identifier, as in `Contoso.Utility.UsefulStuff.Sample`. (The sample package would of course have a dependency on the other package.) When creating a sample package, use the `contentFiles` value in `<IncludeAssets>`. In the `content` folder, arrange the sample code in a folder called `\Samples\<identifier>` as in `\Samples\Contoso.Utility.UsefulStuff.Sample`.

Best practices for the package version:

- In general, set the version of the package to match the project (or assembly), though this is not strictly required. This is a simple matter when you limit a package to a single assembly. Overall, remember that NuGet itself deals with package versions when resolving dependencies, not assembly versions.
- When using a non-standard version scheme, be sure to consider the NuGet versioning rules as explained in [Package versioning](#). NuGet is mostly [semver 2](#) compliant.

For information on dependency resolution, see [Dependency resolution with PackageReference](#). For older information that may also be helpful to better understand versioning, see this series of blog posts.

- [Part 1: Taking on DLL Hell](#)
- [Part 2: The core algorithm](#)

- Part 3: Unification via Binding Redirects

Run the pack command

To build a NuGet package (a `.nupkg` file) from the project, run the `dotnet pack` command, which also builds the project automatically:

```
# Uses the project file in the current folder by default
dotnet pack
```

The output shows the path to the `.nupkg` file.

```
Microsoft (R) Build Engine version 15.5.180.51428 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 29.91 ms for D:\proj\AppLoggerNet\AppLogger\AppLogger.csproj.
AppLogger -> D:\proj\AppLoggerNet\AppLogger\bin\Debug\netstandard2.0\AppLogger.dll
Successfully created package 'D:\proj\AppLoggerNet\AppLogger\bin\Debug\AppLogger.1.0.0.nupkg'.
```

Automatically generate package on build

To automatically run `dotnet pack` when you run `dotnet build`, add the following line to your project file within `<PropertyGroup>`:

```
<GeneratePackageOnBuild>true</GeneratePackageOnBuild>
```

When you run `dotnet pack` on a solution, this packs all the projects in the solution that are packable (`property` is set to `true`).

NOTE

When you automatically generate the package, the time to pack increases the build time for your project.

Test package installation

Before publishing a package, you typically want to test the process of installing a package into a project. The tests make sure that the necessary files all end up in their correct places in the project.

You can test installations manually in Visual Studio or on the command line using the normal [package installation steps](#).

IMPORTANT

Packages are immutable. If you correct a problem, change the contents of the package and pack again, when you retest you will still be using the old version of the package until you [clear your global packages](#) folder. This is especially relevant when testing packages that don't use a unique prerelease label on every build.

Next Steps

Once you've created a package, which is a `.nupkg` file, you can publish it to the gallery of your choice as described on [Publishing a Package](#).

You might also want to extend the capabilities of your package or otherwise support other scenarios as described in the following topics:

- [Package versioning](#)
- [Support multiple target frameworks](#)
- [Transformations of source and configuration files](#)
- [Localization](#)
- [Pre-release versions](#)
- [Set package type](#)
- [Create packages with COM interop assemblies](#)

Finally, there are additional package types to be aware of:

- [Native Packages](#)
- [Symbol Packages](#)

Create a package using the nuget.exe CLI

11/5/2019 • 18 minutes to read • [Edit Online](#)

No matter what your package does or what code it contains, you use one of the CLI tools, either `nuget.exe` or `dotnet.exe`, to package that functionality into a component that can be shared with and used by any number of other developers. To install NuGet CLI tools, see [Install NuGet client tools](#). Note that Visual Studio does not automatically include a CLI tool.

- For non-SDK-style projects, typically .NET Framework projects, follow the steps described in this article to create a package. For step-by-step instructions using Visual Studio and the `nuget.exe` CLI, see [Create and publish a .NET Framework package](#).
- For .NET Core and .NET Standard projects that use the [SDK-style format](#), and any other SDK-style projects, see [Create a NuGet package using the dotnet CLI](#).
- For projects migrated from `packages.config` to [PackageReference](#), use `msbuild -t:pack`.

Technically speaking, a NuGet package is just a ZIP file that's been renamed with the `.nupkg` extension and whose contents match certain conventions. This topic describes the detailed process of creating a package that meets those conventions.

Packaging begins with the compiled code (assemblies), symbols, and/or other files that you want to deliver as a package (see [Overview and workflow](#)). This process is independent from compiling or otherwise generating the files that go into the package, although you can draw from information in a project file to keep the compiled assemblies and packages in sync.

IMPORTANT

This topic applies to non-SDK-style projects, typically projects other than .NET Core and .NET Standard projects using Visual Studio 2017 and higher versions and NuGet 4.0+.

Decide which assemblies to package

Most general-purpose packages contain one or more assemblies that other developers can use in their own projects.

- In general, it's best to have one assembly per NuGet package, provided that each assembly is independently useful. For example, if you have a `Utilities.dll` that depends on `Parser.dll`, and `Parser.dll` is useful on its own, then create one package for each. Doing so allows developers to use `Parser.dll` independently of `Utilities.dll`.
- If your library is composed of multiple assemblies that aren't independently useful, then it's fine to combine them into one package. Using the previous example, if `Parser.dll` contains code that's used only by `Utilities.dll`, then it's fine to keep `Parser.dll` in the same package.
- Similarly, if `Utilities.dll` depends on `Utilities.resources.dll`, where again the latter is not useful on its own, then put both in the same package.

Resources are, in fact, a special case. When a package is installed into a project, NuGet automatically adds assembly references to the package's DLLs, *excluding* those that are named `.resources.dll` because they are assumed to be localized satellite assemblies (see [Creating localized packages](#)). For this reason, avoid using `.resources.dll` for files

that otherwise contain essential package code.

If your library contains COM interop assemblies, follow additional the guidelines in [Create packages with COM interop assemblies](#).

The role and structure of the .nuspec file

Once you know what files you want to package, the next step is creating a package manifest in a `.nuspec` XML file.

The manifest:

1. Describes the package's contents and is itself included in the package.
2. Drives both the creation of the package and instructs NuGet on how to install the package into a project. For example, the manifest identifies other package dependencies such that NuGet can also install those dependencies when the main package is installed.
3. Contains both required and optional properties as described below. For exact details, including other properties not mentioned here, see the [.nuspec reference](#).

Required properties:

- The package identifier, which must be unique across the gallery that hosts the package.
- A specific version number in the form *Major.Minor.Patch[-Suffix]* where *-Suffix* identifies [pre-release versions](#)
- The package title as it should appear on the host (like nuget.org)
- Author and owner information.
- A long description of the package.

Common optional properties:

- Release notes
- Copyright information
- A short description for the [Package Manager UI in Visual Studio](#)
- A locale ID
- Project URL
- License as an expression or file (`licenseUrl` is being deprecated, use the [license](#) nuspec metadata element)
- An icon URL
- Lists of dependencies and references
- Tags that assist in gallery searches

The following is a typical (but fictitious) `.nuspec` file, with comments describing the properties:

```

<?xml version="1.0"?>
<package xmlns="http://schemas.microsoft.com/packaging/2013/05/nuspec.xsd">
  <metadata>
    <!-- The identifier that must be unique within the hosting gallery -->
    <id>Contoso.Utility.UsefulStuff</id>

    <!-- The package version number that is used when resolving dependencies -->
    <version>1.8.3-beta</version>

    <!-- Authors contain text that appears directly on the gallery -->
    <authors>Dejana Tesic, Rajeev Dey</authors>

    <!--
        Owners are typically nuget.org identities that allow gallery
        users to easily find other packages by the same owners.
    -->
    <owners>dejanatc, rjdey</owners>

    <!-- Project URL provides a link for the gallery -->
    <projectUrl>http://github.com/contoso/UsefulStuff</projectUrl>

    <!-- License information is displayed on the gallery -->
    <license type="expression">Apache-2.0</license>

    <!-- The icon is used in Visual Studio's package manager UI -->
    <iconUrl>http://github.com/contoso/UsefulStuff/nuget_icon.png</iconUrl>

    <!--
        If true, this value prompts the user to accept the license when
        installing the package.
    -->
    <requireLicenseAcceptance>false</requireLicenseAcceptance>

    <!-- Any details about this particular release -->
    <releaseNotes>Bug fixes and performance improvements</releaseNotes>

    <!--
        The description can be used in package manager UI. Note that the
        nuget.org gallery uses information you add in the portal.
    -->
    <description>Core utility functions for web applications</description>

    <!-- Copyright information -->
    <copyright>Copyright ©2016 Contoso Corporation</copyright>

    <!-- Tags appear in the gallery and can be used for tag searches -->
    <tags>web utility http json url parsing</tags>

    <!-- Dependencies are automatically installed when the package is installed -->
    <dependencies>
      <dependency id="Newtonsoft.Json" version="9.0" />
    </dependencies>
  </metadata>

  <!-- A readme.txt to display when the package is installed -->
  <files>
    <file src="readme.txt" target="" />
  </files>
</package>

```

For details on declaring dependencies and specifying version numbers, see [packages.config](#) and [Package versioning](#). It is also possible to surface assets from dependencies directly in the package by using the `include` and `exclude` attributes on the `dependency` element. See [.nuspec Reference - Dependencies](#).

Because the manifest is included in the package created from it, you can find any number of additional examples by

examining existing packages. A good source is the *global-packages* folder on your computer, the location of which is returned by the following command:

```
nuget locals -list global-packages
```

Go into any *package\version* folder, copy the `.nupkg` file to a `.zip` file, then open that `.zip` file and examine the `.nuspec` within it.

NOTE

When creating a `.nuspec` from a Visual Studio project, the manifest contains tokens that are replaced with information from the project when the package is built. See [Creating the .nuspec from a Visual Studio project](#).

Create the `.nuspec` file

Creating a complete manifest typically begins with a basic `.nuspec` file generated through one of the following methods:

- [A convention-based working directory](#)
- [An assembly DLL](#)
- [A Visual Studio project](#)
- [New file with default values](#)

You then edit the file by hand so that it describes the exact content you want in the final package.

IMPORTANT

Generated `.nuspec` files contain placeholders that must be modified before creating the package with the `nuget pack` command. That command fails if the `.nuspec` contains any placeholders.

From a convention-based working directory

Because a NuGet package is just a ZIP file that's been renamed with the `.nupkg` extension, it's often easiest to create the folder structure you want on your local file system, then create the `.nuspec` file directly from that structure. The `nuget pack` command then automatically adds all files in that folder structure (excluding any folders that begin with `.`, allowing you to keep private files in the same structure).

The advantage to this approach is that you don't need to specify in the manifest which files you want to include in the package (as explained later in this topic). You can simply have your build process produce the exact folder structure that goes into the package, and you can easily include other files that might not be part of a project otherwise:

- Content and source code that should be injected into the target project.
- PowerShell scripts
- Transformations to existing configuration and source code files in a project.

The folder conventions are as follows:

FOLDER	DESCRIPTION	ACTION UPON PACKAGE INSTALL
(root)	Location for <code>readme.txt</code>	Visual Studio displays a <code>readme.txt</code> file in the package root when the package is installed.

FOLDER	DESCRIPTION	ACTION UPON PACKAGE INSTALL
lib/{tfm}	Assembly (<code>.dll</code>), documentation (<code>.xml</code>), and symbol (<code>.pdb</code>) files for the given Target Framework Moniker (TFM)	Assemblies are added as references for compile as well as runtime; <code>.xml</code> and <code>.pdb</code> copied into project folders. See Supporting multiple target frameworks for creating framework target-specific sub-folders.
ref/{tfm}	Assembly (<code>.dll</code>), and symbol (<code>.pdb</code>) files for the given Target Framework Moniker (TFM)	Assemblies are added as references only for compile time; So nothing will be copied into project bin folder.
runtimes	Architecture-specific assembly (<code>.dll</code>), symbol (<code>.pdb</code>), and native resource (<code>.pri</code>) files	Assemblies are added as references only for runtime; other files are copied into project folders. There should always be a corresponding (TFM) <code>AnyCPU</code> specific assembly under <code>/ref/{tfm}</code> folder to provide corresponding compile time assembly. See Supporting multiple target frameworks .
content	Arbitrary files	Contents are copied to the project root. Think of the content folder as the root of the target application that ultimately consumes the package. To have the package add an image in the application's <code>/images</code> folder, place it in the package's <code>content/images</code> folder.
build	(3.x+) MSBuild <code>.targets</code> and <code>.props</code> files	Automatically inserted into the project.
buildMultiTargeting	(4.0+) MSBuild <code>.targets</code> and <code>.props</code> files for cross-framework targeting	Automatically inserted into the project.
buildTransitive	(5.0+) MSBuild <code>.targets</code> and <code>.props</code> files that flow transitively to any consuming project. See the feature page.	Automatically inserted into the project.
tools	Powershell scripts and programs accessible from the Package Manager Console	The <code>tools</code> folder is added to the <code>PATH</code> environment variable for the Package Manager Console only (Specifically, <i>not</i> to the <code>PATH</code> as set for MSBuild when building the project).

Because your folder structure can contain any number of assemblies for any number of target frameworks, this method is necessary when creating packages that support multiple frameworks.

In any case, once you have the desired folder structure in place, run the following command in that folder to create the `.nuspec` file:

```
nuget spec
```

Again, the generated `.nuspec` contains no explicit references to files in the folder structure. NuGet automatically

includes all files when the package is created. You still need to edit placeholder values in other parts of the manifest, however.

From an assembly DLL

In the simple case of creating a package from an assembly, you can generate a `.nuspec` file from the metadata in the assembly using the following command:

```
nuget spec <assembly-name>.dll
```

Using this form replaces a few placeholders in the manifest with specific values from the assembly. For example, the `<id>` property is set to the assembly name, and `<version>` is set to the assembly version. Other properties in the manifest, however, don't have matching values in the assembly and thus still contain placeholders.

From a Visual Studio project

Creating a `.nuspec` from a `.csproj` or `.vbproj` file is convenient because other packages that have been installed into those project are automatically referenced as dependencies. Simply use the following command in the same folder as the project file:

```
# Use in a folder containing a project file <project-name>.csproj or <project-name>.vbproj
nuget spec
```

The resulting `<project-name>.nuspec` file contains *tokens* that are replaced at packaging time with values from the project, including references to any other packages that have already been installed.

If you have package dependencies to include in the `.nuspec`, instead use `nuget pack`, and get the `.nuspec` file from within the generated `.nupkg` file. For example, use the following command.

```
# Use in a folder containing a project file <project-name>.csproj or <project-name>.vbproj
nuget pack myproject.csproj
```

A token is delimited by `$` symbols on both sides of the project property. For example, the `<id>` value in a manifest generated in this way typically appears as follows:

```
<id>$id$</id>
```

This token is replaced with the `AssemblyName` value from the project file at packing time. For the exact mapping of project values to `.nuspec` tokens, see the [Replacement Tokens reference](#).

Tokens relieve you from needing to update crucial values like the version number in the `.nuspec` as you update the project. (You can always replace the tokens with literal values, if desired).

Note that there are several additional packaging options available when working from a Visual Studio project, as described in [Running nuget pack to generate the .nupkg file](#) later on.

Solution-level packages

NuGet 2.x only. Not available in NuGet 3.0+.

NuGet 2.x supported the notion of a solution-level package that installs tools or additional commands for the Package Manager Console (the contents of the `tools` folder), but does not add references, content, or build customizations to any projects in the solution. Such packages contain no files in its direct `lib`, `content`, or `build` folders, and none of its dependencies have files in their respective `lib`, `content`, or `build` folders.

NuGet tracks installed solution-level packages in a `packages.config` file in the `.nuget` folder, rather than the

project's `packages.config` file.

New file with default values

The following command creates a default manifest with placeholders, which ensures you start with the proper file structure:

```
nuget spec [<package-name>]
```

If you omit `<package-name>`, the resulting file is `Package.nuspec`. If you provide a name such as `Contoso.Utility.UsefulStuff`, the file is `Contoso.Utility.UsefulStuff.nuspec`.

The resulting `.nuspec` contains placeholders for values like the `projectUrl`. Be sure to edit the file before using it to create the final `.nupkg` file.

Choose a unique package identifier and setting the version number

The package identifier (`<id>` element) and the version number (`<version>` element) are the two most important values in the manifest because they uniquely identify the exact code that's contained in the package.

Best practices for the package identifier:

- **Uniqueness:** The identifier must be unique across nuget.org or whatever gallery hosts the package. Before deciding on an identifier, search the applicable gallery to check if the name is already in use. To avoid conflicts, a good pattern is to use your company name as the first part of the identifier, such as `contoso.`.
- **Namespace-like names:** Follow a pattern similar to namespaces in .NET, using dot notation instead of hyphens. For example, use `Contoso.Utility.UsefulStuff` rather than `Contoso-Utility-UsefulStuff` or `Contoso.Utility.UsefulStuff`. Consumers also find it helpful when the package identifier matches the namespaces used in the code.
- **Sample Packages:** If you produce a package of sample code that demonstrates how to use another package, attach `.Sample` as a suffix to the identifier, as in `Contoso.Utility.UsefulStuff.Sample`. (The sample package would of course have a dependency on the other package.) When creating a sample package, use the convention-based working directory method described earlier. In the `content` folder, arrange the sample code in a folder called `\Samples\<identifier>` as in `\Samples\Contoso.Utility.UsefulStuff.Sample`.

Best practices for the package version:

- In general, set the version of the package to match the library, though this is not strictly required. This is a simple matter when you limit a package to a single assembly, as described earlier in [Deciding which assemblies to package](#). Overall, remember that NuGet itself deals with package versions when resolving dependencies, not assembly versions.
- When using a non-standard version scheme, be sure to consider the NuGet versioning rules as explained in [Package versioning](#).

The following series of brief blog posts are also helpful to understand versioning:

- [Part 1: Taking on DLL Hell](#)
- [Part 2: The core algorithm](#)
- [Part 3: Unification via Binding Redirects](#)

Add a readme and other files

To directly specify files to include in the package, use the `<files>` node in the `.nuspec` file, which *follows* the `<metadata>` tag:

```

<?xml version="1.0"?>
<package xmlns="http://schemas.microsoft.com/packaging/2013/05/nuspec.xsd">
  <metadata>
    <!-- ... -->
  </metadata>
  <files>
    <!-- Add a readme -->
    <file src="readme.txt" target="" />

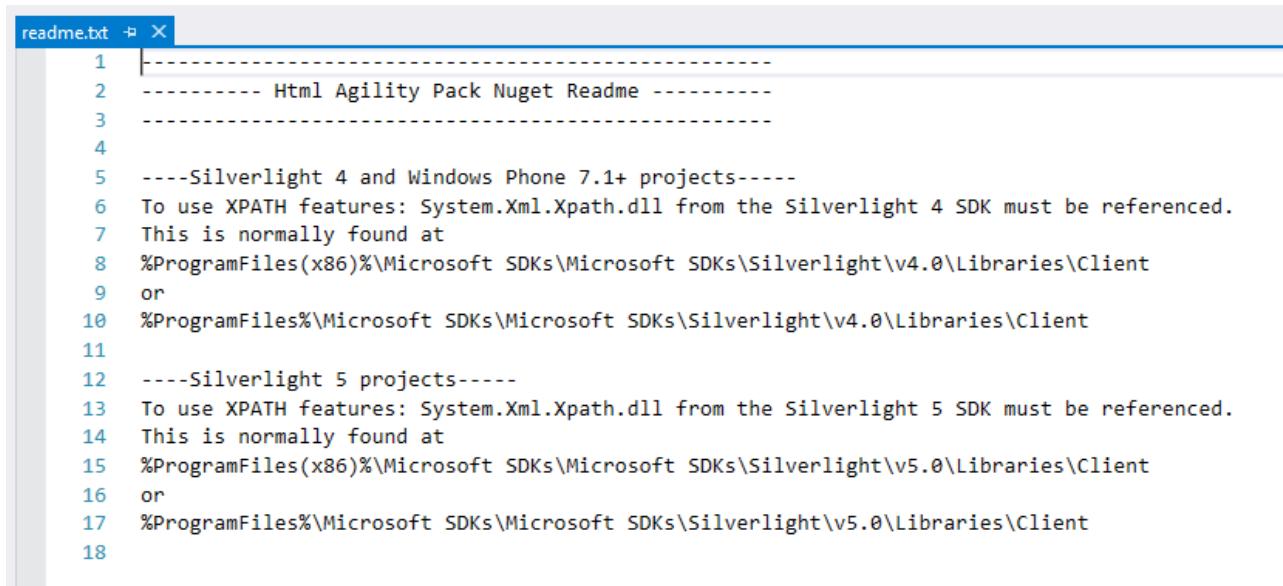
    <!-- Add files from an arbitrary folder that's not necessarily in the project -->
    <file src="..\..\SomeRoot\**\*.*" target="" />
  </files>
</package>

```

TIP

When using the convention-based working directory approach, you can place the `readme.txt` in the package root and other content in the `content` folder. No `<file>` elements are necessary in the manifest.

When you include a file named `readme.txt` in the package root, Visual Studio displays the contents of that file as plain text immediately after installing the package directly. (Readme files are not displayed for packages installed as dependencies). For example, here's how the readme for the `HtmlAgilityPack` package appears:



```

readme.txt + X
1  -----
2  ----- Html Agility Pack Nuget Readme -----
3  -----
4
5  ----Silverlight 4 and Windows Phone 7.1+ projects-----
6  To use XPATH features: System.Xml.XPath.dll from the Silverlight 4 SDK must be referenced.
7  This is normally found at
8  %ProgramFiles(x86)%\Microsoft SDKs\Microsoft SDKs\Silverlight\v4.0\Libraries\Client
9  or
10 %ProgramFiles%\Microsoft SDKs\Microsoft SDKs\Silverlight\v4.0\Libraries\Client
11
12 ----Silverlight 5 projects-----
13 To use XPATH features: System.Xml.XPath.dll from the Silverlight 5 SDK must be referenced.
14 This is normally found at
15 %ProgramFiles(x86)%\Microsoft SDKs\Microsoft SDKs\Silverlight\v5.0\Libraries\Client
16 or
17 %ProgramFiles%\Microsoft SDKs\Microsoft SDKs\Silverlight\v5.0\Libraries\Client
18

```

NOTE

If you include an empty `<files>` node in the `.nuspec` file, NuGet doesn't include any other content in the package other than what's in the `lib` folder.

Include MSBuild props and targets in a package

In some cases, you might want to add custom build targets or properties in projects that consume your package, such as running a custom tool or process during build. You do this by placing files in the form

`<package_id>.targets` or `<package_id>.props` (such as `Contoso.Utility.UsefulStuff.targets`) within the `\build` folder of the project.

Files in the root `\build` folder are considered suitable for all target frameworks. To provide framework-specific files, first place them within appropriate subfolders, such as the following:

```
\build
  \netstandard1.4
    \Contoso.Utility.UsefulStuff.props
    \Contoso.Utility.UsefulStuff.targets
  \net462
    \Contoso.Utility.UsefulStuff.props
    \Contoso.Utility.UsefulStuff.targets
```

Then in the `.nuspec` file, be sure to refer to these files in the `<files>` node:

```
<?xml version="1.0"?>
<package>
  <metadata minClientVersion="2.5">
    <!-- ... -->
  </metadata>
  <files>
    <!-- Include everything in \build -->
    <file src="build\***" target="build" />

    <!-- Other files -->
    <!-- ... -->
  </files>
</package>
```

Including MSBuild props and targets in a package was [introduced with NuGet 2.5](#), therefore it is recommended to add the `minClientVersion="2.5"` attribute to the `metadata` element, to indicate the minimum NuGet client version required to consume the package.

When NuGet installs a package with `\build` files, it adds MSBuild `<Import>` elements in the project file pointing to the `.targets` and `.props` files. (`.props` is added at the top of the project file; `.targets` is added at the bottom.) A separate conditional MSBuild `<Import>` element is added for each target framework.

MSBuild `.props` and `.targets` files for cross-framework targeting can be placed in the `\buildMultiTargeting` folder. During package installation, NuGet adds the corresponding `<Import>` elements to the project file with the condition, that the target framework is not set (the MSBuild property `$(TargetFramework)` must be empty).

With NuGet 3.x, targets are not added to the project but are instead made available through `{projectName}.nuget.g.targets` and `{projectName}.nuget.g.props`.

Run `nuget pack` to generate the `.nupkg` file

When using an assembly or the convention-based working directory, create a package by running `nuget pack` with your `.nuspec` file, replacing `<project-name>` with your specific filename:

```
nuget pack <project-name>.nuspec
```

When using a Visual Studio project, run `nuget pack` with your project file, which automatically loads the project's `.nuspec` file and replaces any tokens within it using values in the project file:

```
nuget pack <project-name>.csproj
```

NOTE

Using the project file directly is necessary for token replacement because the project is the source of the token values. Token replacement does not happen if you use `nuget pack` with a `.nuspec` file.

In all cases, `nuget pack` excludes folders that start with a period, such as `.git` or `.hg`.

NuGet indicates if there are any errors in the `.nuspec` file that need correcting, such as forgetting to change placeholder values in the manifest.

Once `nuget pack` succeeds, you have a `.nupkg` file that you can publish to a suitable gallery as described in [Publishing a Package](#).

TIP

A helpful way to examine a package after creating it is to open it in the [Package Explorer](#) tool. This gives you a graphical view of the package contents and its manifest. You can also rename the resulting `.nupkg` file to a `.zip` file and explore its contents directly.

Additional options

You can use various command-line switches with `nuget pack` to exclude files, override the version number in the manifest, and change the output folder, among other features. For a complete list, refer to the [pack command reference](#).

The following options are a few that are common with Visual Studio projects:

- **Referenced projects:** If the project references other projects, you can add the referenced projects as part of the package, or as dependencies, by using the `-IncludeReferencedProjects` option:

```
nuget pack MyProject.csproj -IncludeReferencedProjects
```

This inclusion process is recursive, so if `MyProject.csproj` references projects B and C, and those projects reference D, E, and F, then files from B, C, D, E, and F are included in the package.

If a referenced project includes a `.nuspec` file of its own, then NuGet adds that referenced project as a dependency instead. You need to package and publish that project separately.

- **Build configuration:** By default, NuGet uses the default build configuration set in the project file, typically *Debug*. To pack files from a different build configuration, such as *Release*, use the `-properties` option with the configuration:

```
nuget pack MyProject.csproj -properties Configuration=Release
```

- **Symbols:** to include symbols that allow consumers to step through your package code in the debugger, use the `-Symbols` option:

```
nuget pack MyProject.csproj -symbols
```

Test package installation

Before publishing a package, you typically want to test the process of installing a package into a project. The tests make sure that the necessary files all end up in their correct places in the project.

You can test installations manually in Visual Studio or on the command line using the normal [package installation steps](#).

For automated testing, the basic process is as follows:

1. Copy the `.nupkg` file to a local folder.
2. Add the folder to your package sources using the `nuget sources add -name <name> -source <path>` command (see [nuget sources](#)). Note that you need only set this local source once on any given computer.
3. Install the package from that source using `nuget install <packageID> -source <name>` where `<name>` matches the name of your source as given to `nuget sources`. Specifying the source ensures that the package is installed from that source alone.
4. Examine your file system to check that files are installed correctly.

Next Steps

Once you've created a package, which is a `.nupkg` file, you can publish it to the gallery of your choice as described on [Publishing a Package](#).

You might also want to extend the capabilities of your package or otherwise support other scenarios as described in the following topics:

- [Package versioning](#)
- [Supporting multiple target frameworks](#)
- [Transformations of source and configuration files](#)
- [Localization](#)
- [Pre-release versions](#)
- [Set package type](#)
- [Create packages with COM interop assemblies](#)

Finally, there are additional package types to be aware of:

- [Native Packages](#)
- [Symbol Packages](#)

Create a NuGet package using MSBuild

11/5/2019 • 6 minutes to read • [Edit Online](#)

When you create a NuGet package from your code, you package that functionality into a component that can be shared with and used by any number of other developers. This article describes how to create a package using MSBuild. MSBuild comes preinstalled with every Visual Studio workload that contains NuGet. Additionally you can also use MSBuild through the dotnet CLI with `dotnet msbuild`.

For .NET Core and .NET Standard projects that use the [SDK-style format](#), and any other SDK-style projects, NuGet uses information in the project file directly to create a package. For a non-SDK-style project that uses `<PackageReference>`, NuGet also uses the project file to create a package.

SDK-style projects have the pack functionality available by default. For non SDK-style PackageReference projects, you need to add the NuGet.Build.Tasks.Pack package to the project dependencies. For detailed information about MSBuild pack targets, see [NuGet pack and restore as MSBuild targets](#).

The command that creates a package, `msbuild -t:pack`, is functionality equivalent to `dotnet pack`.

IMPORTANT

This topic applies to [SDK-style](#) projects, typically .NET Core and .NET Standard projects, and to non-SDK-style projects that use `PackageReference`.

Set properties

The following properties are required to create a package.

- `PackageId`, the package identifier, which must be unique across the gallery that hosts the package. If not specified, the default value is `AssemblyName`.
- `Version`, a specific version number in the form *Major.Minor.Patch[-Suffix]* where *-Suffix* identifies [pre-release versions](#). If not specified, the default value is 1.0.0.
- The package title as it should appear on the host (like nuget.org)
- `Authors`, author and owner information. If not specified, the default value is `AssemblyName`.
- `Company`, your company name. If not specified, the default value is `AssemblyName`.

In Visual Studio, you can set these values in the project properties (right-click the project in Solution Explorer, choose **Properties**, and select the **Package** tab). You can also set these properties directly in the project files (`.csproj`).

```
<PropertyGroup>
  <PackageId>ClassLibDotNetStandard</PackageId>
  <Version>1.0.0</Version>
  <Authors>your_name</Authors>
  <Company>your_company</Company>
</PropertyGroup>
```

IMPORTANT

Give the package an identifier that's unique across nuget.org or whatever package source you're using.

The following example shows a simple, complete project file with these properties included.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <PackageId>ClassLibDotNetStandard</PackageId>
    <Version>1.0.0</Version>
    <Authors>your_name</Authors>
    <Company>your_company</Company>
  </PropertyGroup>
</Project>
```

You can also set the optional properties, such as `Title`, `PackageDescription`, and `PackageTags`, as described in [MSBuild pack targets](#), [Controlling dependency assets](#), and [NuGet metadata properties](#).

NOTE

For packages built for public consumption, pay special attention to the `PackageTags` property, as tags help others find your package and understand what it does.

For details on declaring dependencies and specifying version numbers, see [Package references in project files](#) and [Package versioning](#). It is also possible to surface assets from dependencies directly in the package by using the `<IncludeAssets>` and `<ExcludeAssets>` attributes. For more information, see [Controlling dependency assets](#).

Choose a unique package identifier and set the version number

The package identifier and the version number are the two most important values in the project because they uniquely identify the exact code that's contained in the package.

Best practices for the package identifier:

- **Uniqueness:** The identifier must be unique across nuget.org or whatever gallery hosts the package. Before deciding on an identifier, search the applicable gallery to check if the name is already in use. To avoid conflicts, a good pattern is to use your company name as the first part of the identifier, such as `Contoso.`.
- **Namespace-like names:** Follow a pattern similar to namespaces in .NET, using dot notation instead of hyphens. For example, use `Contoso.Utility.UsefulStuff` rather than `Contoso-Utility-UsefulStuff` or `Contoso.Utility.UsefulStuff`. Consumers also find it helpful when the package identifier matches the namespaces used in the code.
- **Sample Packages:** If you produce a package of sample code that demonstrates how to use another package, attach `.Sample` as a suffix to the identifier, as in `Contoso.Utility.UsefulStuff.Sample`. (The sample package would of course have a dependency on the other package.) When creating a sample package, use the `contentFiles` value in `<IncludeAssets>`. In the `content` folder, arrange the sample code in a folder called `\Samples\<identifier>` as in `\Samples\Contoso.Utility.UsefulStuff.Sample`.

Best practices for the package version:

- In general, set the version of the package to match the project (or assembly), though this is not strictly required. This is a simple matter when you limit a package to a single assembly. Overall, remember that NuGet itself deals with package versions when resolving dependencies, not assembly versions.
- When using a non-standard version scheme, be sure to consider the NuGet versioning rules as explained in [Package versioning](#). NuGet is mostly [semver 2 compliant](#).

For information on dependency resolution, see [Dependency resolution with PackageReference](#). For older information that may also be helpful to better understand versioning, see this series of blog posts.

- [Part 1: Taking on DLL Hell](#)
- [Part 2: The core algorithm](#)
- [Part 3: Unification via Binding Redirects](#)

Add the NuGet.Build.Tasks.Pack package

If you are using MSBuild with a non-SDK-style project and `PackageReference`, add the `NuGet.Build.Tasks.Pack` package to your project.

1. Open the project file and add the following after the `<PropertyGroup>` element:

```
<ItemGroup>
  <!-- ... -->
  <PackageReference Include="NuGet.Build.Tasks.Pack" Version="5.2.0"/>
  <!-- ... -->
</ItemGroup>
```

2. Open a Developer command prompt (In the **Search** box, type **Developer command prompt**).

You typically want to start the Developer Command Prompt for Visual Studio from the **Start** menu, as it will be configured with all the necessary paths for MSBuild.

3. Switch to the folder containing the project file and type the following command to install the `NuGet.Build.Tasks.Pack` package.

```
# Uses the project file in the current folder by default
msbuild -t:restore
```

Make sure that the MSBuild output indicates that the build completed successfully.

Run the `msbuild -t:pack` command

To build a NuGet package (a `.nupkg` file) from the project, run the `msbuild -t:pack` command, which also builds the project automatically:

In the Developer command prompt for Visual Studio, type the following command:

```
# Uses the project file in the current folder by default
msbuild -t:pack
```

The output shows the path to the `.nupkg` file.

```
Microsoft (R) Build Engine version 16.1.76+g14b0a930a7 for .NET Framework
Copyright (C) Microsoft Corporation. All rights reserved.

Build started 8/5/2019 3:09:15 PM.
Project "C:\Users\username\source\repos\ClassLib_DotNetStandard\ClassLib_DotNetStandard.csproj" on node 1
(pack target(s)).
GenerateTargetFrameworkMonikerAttribute:
Skipping target "GenerateTargetFrameworkMonikerAttribute" because all output files are up-to-date with respect
to the input files.
CoreCompile:
...
CopyFilesToOutputDirectory:
  Copying file from
  "C:\Users\username\source\repos\ClassLib_DotNetStandard\obj\Debug\netstandard2.0\ClassLib_DotNetStandard.dll"
  to "C:\Use
  rs\username\source\repos\ClassLib_DotNetStandard\bin\Debug\netstandard2.0\ClassLib_DotNetStandard.dll".
  ClassLib_DotNetStandard ->
C:\Users\username\source\repos\ClassLib_DotNetStandard\bin\Debug\netstandard2.0\ClassLib_DotNetStandard.dll
  Copying file from
  "C:\Users\username\source\repos\ClassLib_DotNetStandard\obj\Debug\netstandard2.0\ClassLib_DotNetStandard.pdb"
  to "C:\Use
  rs\username\source\repos\ClassLib_DotNetStandard\bin\Debug\netstandard2.0\ClassLib_DotNetStandard.pdb".
GenerateNuspec:
  Successfully created package
  'C:\Users\username\source\repos\ClassLib_DotNetStandard\bin\Debug\AppLogger.1.0.0.nupkg'.
Done Building Project "C:\Users\username\source\repos\ClassLib_DotNetStandard\ClassLib_DotNetStandard.csproj"
(pack target(s)).

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

Time Elapsed 00:00:01.21
```

Automatically generate package on build

To automatically run `msbuild -t:pack` when you build or restore the project, add the following line to your project file within `<PropertyGroup>`:

```
<GeneratePackageOnBuild>true</GeneratePackageOnBuild>
```

When you run `msbuild -t:pack` on a solution, this packs all the projects in the solution that are packable (property is set to `true`).

NOTE

When you automatically generate the package, the time to pack increases the build time for your project.

Test package installation

Before publishing a package, you typically want to test the process of installing a package into a project. The tests make sure that the necessary files all end up in their correct places in the project.

You can test installations manually in Visual Studio or on the command line using the normal [package installation steps](#).

IMPORTANT

Packages are immutable. If you correct a problem, change the contents of the package and pack again, when you retest you will still be using the old version of the package until you [clear your global packages](#) folder. This is especially relevant when testing packages that don't use a unique prerelease label on every build.

Next Steps

Once you've created a package, which is a `.nupkg` file, you can publish it to the gallery of your choice as described on [Publishing a Package](#).

You might also want to extend the capabilities of your package or otherwise support other scenarios as described in the following topics:

- [NuGet pack and restore as MSBuild targets](#)
- [Package versioning](#)
- [Support multiple target frameworks](#)
- [Transformations of source and configuration files](#)
- [Localization](#)
- [Pre-release versions](#)
- [Set package type](#)
- [Create packages with COM interop assemblies](#)

Finally, there are additional package types to be aware of:

- [Native Packages](#)
- [Symbol Packages](#)

Support multiple .NET Framework versions in your project file

10/15/2019 • 2 minutes to read • [Edit Online](#)

When you first create a project, we recommend you create a .NET Standard class library, as it provides compatibility with the widest range of consuming projects. By using .NET Standard, you add [cross-platform support](#) to a .NET library by default. However, in some scenarios, you may also need to include code that targets a particular framework. This article shows you how to do that for [SDK-style](#) projects.

For SDK-style projects, you can configure support for multiple targets frameworks ([TFM](#)) in your project file, then use `dotnet pack` or `msbuild /t:pack` to create the package.

NOTE

nuget.exe CLI does not support packing SDK-style projects, so you should only use `dotnet pack` or `msbuild /t:pack`. We recommend that you [include all the properties](#) that are usually in the `.nuspec` file in the project file instead. To target multiple .NET Framework versions in a non-SDK-style project, see [Supporting multiple .NET Framework versions](#).

Create a project that supports multiple .NET Framework versions

1. Create a new .NET Standard class library either in Visual Studio or use `dotnet new classlib`.

We recommend that you create a .NET Standard class library for best compatibility.

2. Edit the `.csproj` file to support the target frameworks. For example, change

```
<TargetFramework>netstandard2.0</TargetFramework>
```

to:

```
<TargetFrameworks>netstandard2.0;net45</TargetFrameworks>
```

Make sure that you change the XML element changed from singular to plural (add the "s" to both the open and close tags).

3. If you have any code that only works in one TFM, you can use `#if NET45` or `#if NETSTANDARD2_0` to separate TFM-dependent code. (For more information, see [How to multitarget](#).) For example, you can use the following code:

```
public string Platform {  
    get {  
        #if NET45  
            return ".NET Framework"  
        #elif NETSTANDARD2_0  
            return ".NET Standard"  
        #else  
            #error This code block does not match csproj TargetFrameworks list  
        #endif  
    }  
}
```

4. Add any NuGet metadata you want to the `.csproj` as MSBuild properties.

For the list of available package metadata and the MSBuild property names, see [pack target](#). Also see [Controlling dependency assets](#).

If you want to separate build-related properties from NuGet metadata, you can use a different `PropertyGroup`, or put the NuGet properties in another file and use MSBuild's `Import` directive to include it. `Directory.Build.Props` and `Directory.Build.Targets` are also supported starting with MSBuild 15.0.

5. Now, use `dotnet pack` and the resulting `.nupkg` targets both .NET Standard 2.0 and .NET Framework 4.5.

Here is the `.csproj` file that is generated using the preceding steps and .NET Core SDK 2.2.

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <TargetFrameworks>netstandard2.0;net45</TargetFrameworks>
  <Description>Sample project that targets multiple TFM</Description>
</PropertyGroup>

</Project>
```

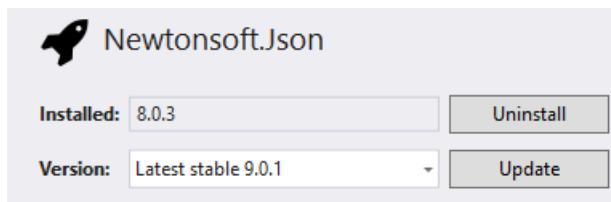
See also

- [How to specify target frameworks](#)
- [Cross-platform targeting](#)

Building pre-release packages

11/5/2019 • 2 minutes to read • [Edit Online](#)

Whenever you release an updated package with a new version number, NuGet considers that one as the "latest stable release" as shown, for example in the Package Manager UI within Visual Studio:



A stable release is one that's considered reliable enough to be used in production. The latest stable release is also the one that will be installed as a package update or during package restore (subject to constraints as described in [Reinstalling and updating packages](#)).

To support the software release lifecycle, NuGet 1.6 and later allows for the distribution of pre-release packages, where the version number includes a semantic versioning suffix such as `-alpha`, `-beta`, or `-rc`. For more information, see [Package versioning](#).

You can specify such versions using one of the following ways:

- **If your project uses `PackageReference`**: include the semantic version suffix in the `.csproj` file's `PackageVersion` element:

```
<PropertyGroup>
  <PackageVersion>1.0.1-alpha</PackageVersion>
</PropertyGroup>
```

- **If your project has a `packages.config` file**: include the semantic version suffix in the `.nuspec` file's `version` element:

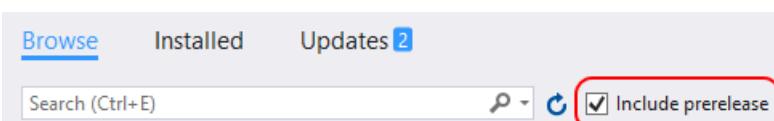
```
<version>1.0.1-alpha</version>
```

When you're ready to release a stable version, just remove the suffix and the package takes precedence over any pre-release versions. Again, see [Package versioning](#).

Installing and updating pre-release packages

By default, NuGet does not include pre-release versions when working with packages, but you can change this behavior as follows:

- **Package Manager UI in Visual Studio**: In the **Manage NuGet Packages** UI, check the **Include prerelease** box:



Setting or clearing this box will refresh the Package Manager UI and the list of available versions you can install.

- **Package Manager Console:** Use the `-IncludePrerelease` switch with the `Find-Package`, `Get-Package`, `Install-Package`, `Sync-Package`, and `Update-Package` commands. Refer to the [PowerShell Reference](#).
- **NuGet CLI:** Use the `-prerelease` switch with the `install`, `update`, `delete`, and `mirror` commands. Refer to the [NuGet CLI reference](#)

Semantic versioning

The [Semantic Versioning or SemVer convention](#) describes how to utilize strings in version numbers to convey the meaning of the underlying code.

In this convention, each version has three parts, `Major.Minor.Patch`, with the following meaning:

- `Major` : Breaking changes
- `Minor` : New features, but backwards compatible
- `Patch` : Backwards compatible bug fixes only

Pre-release versions are then denoted by appending a hyphen and a string after the patch number. Technically speaking, you can use *any* string after the hyphen and NuGet will treat the package as pre-release. NuGet then displays the full version number in the applicable UI, leaving consumers to interpret the meaning for themselves.

With this in mind, it's generally good to follow recognized naming conventions such as the following:

- `-alpha` : Alpha release, typically used for work-in-progress and experimentation
- `-beta` : Beta release, typically one that is feature complete for the next planned release, but may contain known bugs.
- `-rc` : Release candidate, typically a release that's potentially final (stable) unless significant bugs emerge.

NOTE

NuGet 4.3.0+ supports [Semantic Versioning v2.0.0](#), which supports pre-release numbers with dot notation, as in `1.0.1-build.23`. Dot notation is not supported with NuGet versions before 4.3.0. In earlier versions of NuGet, you could use a form like `1.0.1-build23` but this was always considered a pre-release version.

Whatever suffixes you use, however, NuGet will give them precedence in reverse alphabetical order:

```
1.0.1
1.0.1-zzz
1.0.1-rc
1.0.1-open
1.0.1-beta.12
1.0.1-beta.5
1.0.1-beta
1.0.1-alpha.2
1.0.1-alpha
```

As shown, the version without any suffix will always take precedence over pre-release versions.

Leading 0s are not needed with semver2, but they are with the old version schema. If you use numerical suffixes with pre-release tags that might use double-digit numbers (or more), use leading zeroes as in `beta.01` and `beta.05` to ensure that they sort correctly when the numbers get larger. This recommendation only applies to the old version schema.

Creating symbol packages (.snupkg)

11/20/2019 • 3 minutes to read • [Edit Online](#)

Symbol packages allow you to improve the debugging experience of your NuGet packages.

Prerequisites

[nuget.exe v4.9.0 or above](#) or [dotnet.exe v2.2.0 or above](#), which implement the required [NuGet protocols](#).

Creating a symbol package

If you're using dotnet.exe or MSBuild, you need to set the `IncludeSymbols` and `SymbolPackageFormat` properties to create a .snupkg file in addition to the .nupkg file.

- Either add the following properties to your .csproj file:

```
<PropertyGroup>
  <IncludeSymbols>true</IncludeSymbols>
  <SymbolPackageFormat>snupkg</SymbolPackageFormat>
</PropertyGroup>
```

- Or specify these properties on the command-line:

```
dotnet pack MyPackage.csproj -p:IncludeSymbols=true -p:SymbolPackageFormat=snupkg
```

or

```
msbuild MyPackage.csproj /t:pack /p:IncludeSymbols=true /p:SymbolPackageFormat=snupkg
```

If you're using NuGet.exe, you can use the following commands to create a .snupkg file in addition to the .nupkg file:

```
nuget pack MyPackage.nuspec -Symbols -SymbolPackageFormat snupkg
nuget pack MyPackage.csproj -Symbols -SymbolPackageFormat snupkg
```

The `SymbolPackageFormat` property can have one of two values: `symbols.nupkg` (the default) or `snupkg`. If this property is not specified, a legacy symbol package will be created.

NOTE

The legacy format `.symbols.nupkg` is still supported but only for compatibility reasons (see [Legacy Symbol Packages](#)). NuGet.org's symbol server only accepts the new symbol package format - `.snupkg`.

Publishing a symbol package

- For convenience, first save your API key with NuGet (see [publish a package](#)).

```
nuget SetApiKey Your-API-Key
```

2. After publishing your primary package to nuget.org, push the symbol package as follows.

```
nuget push MyPackage.snupkg
```

3. You can also push both primary and symbol packages at the same time using the below command. Both .nupkg and .snupkg files need to be present in the current folder.

```
nuget push MyPackage.nupkg
```

NuGet will publish both packages to nuget.org. `MyPackage.nupkg` will be published first, followed by `MyPackage.snupkg`.

NOTE

If the symbol package isn't published, check that you've configured the NuGet.org source as `https://api.nuget.org/v3/index.json`. Symbol package publishing is only supported by [the NuGet V3 API](#).

NuGet.org symbol server

NuGet.org supports its own symbols server repository and only accepts the new symbol package format - `.snupkg`. Package consumers can use the symbols published to nuget.org symbol server by adding `https://symbols.nuget.org/download/symbols` to their symbol sources in Visual Studio, which allows stepping into package code in the Visual Studio debugger. See [Specify symbol \(.pdb\) and source files in the Visual Studio debugger](#) for details on that process.

NuGet.org symbol package constraints

NuGet.org has the following constraints for symbol packages:

- Only the following file extensions are allowed in symbol packages: `.pdb`, `.nuspec`, `.xml`, `.psmdcp`, `.rels`, `.p7s`
- Only managed [Portable PDBs](#) are supported on NuGet.org's symbol server.
- The PDBs and their associated .nupkg DLLs need to be built with the compiler in Visual Studio version 15.9 or above (see [PDB crypto hash](#))

Symbol packages published to NuGet.org will fail validation if these constraints aren't met.

Symbol package validation and indexing

Symbol packages published to [NuGet.org](#) undergo several validations, including malware scanning. If a package fails a validation check, its package details page will display an error message. In addition, the package's owners will receive an email with instructions on how to fix the identified issues.

When the symbol package has passed all validations, the symbols will be indexed by NuGet.org's symbol servers. Once indexed, the symbol will be available for consumption from the NuGet.org symbol servers.

Package validation and indexing usually takes under 15 minutes. If the package publishing is taking longer than expected, visit [status.nuget.org](#) to check if NuGet.org is experiencing any interruptions. If all systems are operational and the package hasn't been successfully published within an hour, please login to nuget.org and contact us using the Contact Support link on the package details page.

Symbol package structure

The .nupkg file would be exactly the same as it is today, but the .snupkg file would have the following characteristics:

1. The .snupkg will have the same id and version as the corresponding .nupkg.
2. The .snupkg will have the exact folder structure as the nupkg for any DLL or EXE files with the distinction that instead of DLLs/EXEs, their corresponding PDBs will be included in the same folder hierarchy. Files and folders with extensions other than PDB will be left out of the snupkg.
3. The .nuspec file in the .snupkg will also specify a new PackageType as below. This should be the only one PackageType specified.

```
<packageTypes>
  <packageType name="SymbolsPackage"/>
</packageTypes>
```

4. If an author decides to use a custom nuspec to build their nupkg and snupkg, the snupkg should have the same folder hierarchy and files detailed in 2).
5. `authors` and `owners` field will be excluded from the snupkg's nuspec.
6. Do not use the `<license>` element. A .snupkg is covered under the same license as the corresponding .nupkg.

See also

Consider using Source Link to enable source code debugging of .NET assemblies. For more information, please refer to the [Source Link guidance](#).

For more information on symbol packages, please refer to the [NuGet Package Debugging & Symbols Improvements](#) design spec.

Support multiple .NET versions

11/5/2019 • 6 minutes to read • [Edit Online](#)

Many libraries target a specific version of the .NET Framework. For example, you might have one version of your library that's specific to UWP, and another version that takes advantage of features in .NET Framework 4.6. To accommodate this, NuGet supports putting multiple versions of the same library in a single package.

This article describes the layout of a NuGet package, regardless of how the package or assemblies are built (that is, the layout is the same whether using multiple non-SDK-style `.csproj` files and a custom `.nuspec` file, or a single multi-targeted SDK-style `.csproj`). For an SDK-style project, NuGet [pack targets](#) knows how the package must be laid out and automates putting the assemblies in the correct lib folders and creating dependency groups for each target framework (TFM). For detailed instructions, see [Support multiple .NET Framework versions in your project file](#).

You must manually lay out the package as described in this article when using the convention-based working directory method described in [Creating a package](#). For an SDK-style project, the automated method is recommended, but you may also choose to manually lay out the package as described in this article.

Framework version folder structure

When building a package that contains only one version of a library or target multiple frameworks, you always make subfolders under `lib` using different case-sensitive framework names with the following convention:

```
lib\{framework name}\{version}
```

For a complete list of supported names, see the [Target Frameworks reference](#).

You should never have a version of the library that is not specific to a framework and placed directly in the root `lib` folder. (This capability was supported only with `packages.config`). Doing so would make the library compatible with any target framework and allow it to be installed anywhere, likely resulting in unexpected runtime errors. Adding assemblies in the root folder (such as `lib\abc.dll`) or subfolders (such as `lib\abc\abc.dll`) has been deprecated and is ignored when using the `PackagesReference` format.

For example, the following folder structure supports four versions of an assembly that are framework-specific:

```
\lib
  \net46
    \MyAssembly.dll
  \net461
    \MyAssembly.dll
  \uap
    \MyAssembly.dll
  \netcore
    \MyAssembly.dll
```

To easily include all these files when building the package, use a recursive `**` wildcard in the `<files>` section of your `.nuspec`:

```
<files>
  <file src="lib\**" target="lib/{framework name}\{version}" />
</files>
```

Architecture-specific folders

If you have architecture-specific assemblies, that is, separate assemblies that target ARM, x86, and x64, you must place them in a folder named `runtimes` within sub-folders named `{platform}-{architecture}\lib\{framework}` or `{platform}-{architecture}\native`. For example, the following folder structure would accommodate both native and managed DLLs targeting Windows 10 and the `uap10.0` framework:

```
\runtimes
  \win10-arm
    \native
      \lib\uap10.0
  \win10-x86
    \native
      \lib\uap10.0
  \win10-x64
    \native
      \lib\uap10.0
```

These assemblies will only be available at runtime, so if you want to provide the corresponding compile time assembly as well then have `AnyCPU` assembly in `/ref/{tfm}` folder.

Please note, NuGet always picks these compile or runtime assets from one folder so if there are some compatible assets from `/ref` then `/lib` will be ignored to add compile-time assemblies. Similarly, if there are some compatible assets from `/runtime` then also `/lib` will be ignored for runtime.

See [Create UWP Packages](#) for an example of referencing these files in the `.nuspec` manifest.

Also, see [Packing a Windows store app component with NuGet](#)

Matching assembly versions and the target framework in 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 for the highest version that is less than or equal to the project's target framework, if available. If no compatible assembly is found, NuGet returns an appropriate error message.

For example, consider the following folder structure in a package:

```
\lib
  \net45
    \MyAssembly.dll
  \net461
    \MyAssembly.dll
```

When installing this package in a project that targets .NET Framework 4.6, NuGet installs the assembly in the `net45` folder, because that's the highest available version that's less than or equal to 4.6.

If the project targets .NET Framework 4.6.1, on the other hand, NuGet installs the assembly in the `net461` folder.

If the project targets .NET framework 4.0 and earlier, NuGet throws an appropriate error message for not finding the compatible assembly.

Grouping assemblies by framework version

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

```
\lib
  \net40
    \MyAssembly.dll (v1.0)
    \MyAssembly.Core.dll (v1.0)
  \net45
    \MyAssembly.dll (v2.0)
```

When the package is installed in a project that targets .NET Framework 4.5, `MyAssembly.dll` (v2.0) is the only assembly installed. `MyAssembly.Core.dll` (v1.0) is not installed because it's not listed in the `net45` folder. NuGet behaves this way because `MyAssembly.Core.dll` might have merged into version 2.0 of `MyAssembly.dll`.

If you want `MyAssembly.Core.dll` to be installed for .NET Framework 4.5, place a copy in the `net45` 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}
```

The supported profiles are as follows:

- `client` : Client Profile
- `full` : Full Profile
- `wp` : Windows Phone
- `cf` : Compact Framework

Declaring dependencies (Advanced)

When packing a project file, NuGet tries to automatically generate the dependencies from the project. The information in this section about using a `.nuspec` file to declare dependencies is typically necessary for advanced scenarios only.

(Version 2.0+) You can declare package dependencies in the `.nuspec` corresponding to the target framework of the target project using `<group>` elements within the `<dependencies>` element. For more information, see [dependencies element](#).

Each group has an attribute named `targetFramework` and contains zero or more `<dependency>` elements. Those dependencies are installed together when the target framework is compatible with the project's framework profile. See [Target frameworks](#) for the exact framework identifiers.

We recommend using one group per Target Framework Moniker (TFM) for files in the `lib/` and `ref/` folders.

The following example shows different variations of the `<group>` element:

```
<dependencies>

  <group targetFramework="net472">
    <dependency id="jQuery" version="1.10.2" />
    <dependency id="WebActivatorEx" version="2.2.0" />
  </group>

  <group targetFramework="net20">
  </group>

</dependencies>
```

Determining which NuGet target to use

When packaging libraries targeting the Portable Class Library it can be tricky to determine which NuGet target you should use in your folder names and `.nuspec` file, especially if targeting only a subset of the PCL. The following external resources will help you with this:

- [Framework profiles in .NET](#) (stephencleary.com)
- [Portable Class Library profiles](#) (plnkr.co): Table enumerating PCL profiles and their equivalent NuGet targets
- [Portable Class Library profiles tool](#) (github.com): command line tool for determining PCL profiles available on your system

Content files and PowerShell scripts

WARNING

Mutable content files and script execution are available with the `packages.config` format only; they are deprecated with all other formats and should not be used for any new packages.

With `packages.config`, content files and PowerShell scripts can be grouped by target framework using the same folder convention inside the `content` and `tools` folders. For example:

```
\content
  \net46
    \MyContent.txt
  \net461
    \MyContent461.txt
  \uap
    \MyUWPContent.html
  \netcore
\tools
  init.ps1
  \net46
    install.ps1
    uninstall.ps1
  \uap
    install.ps1
    uninstall.ps1
```

If a framework folder is left empty, NuGet doesn't add assembly references or content files or run the PowerShell scripts for that framework.

NOTE

Because `init.ps1` is executed at the solution level and not dependent on project, it must be placed directly under the `tools` folder. It's ignored if placed under a framework folder.

Transforming source code and configuration files

11/13/2018 • 5 minutes to read • [Edit Online](#)

For projects using `packages.config`, NuGet supports the ability to make transformations to source code and configuration files at package install and uninstall times. Only Source code transformations are applied when a package is installed in a project using [PackageReference](#).

A **source code transformation** applies one-way token replacement to files in the package's `content` or `contentFiles` folder (`content` for customers using `packages.config` and `contentFiles` for [PackageReference](#)) when the package is installed, where tokens refer to Visual Studio [project properties](#). This allows you to insert a file into the project's namespace, or to customize code that would typically go into `global.asax` in an ASP.NET project.

A **config file transformation** allows you to modify files that already exist in a target project, such as `web.config` and `app.config`. For example, your package might need to add an item to the `modules` section in the config file. This transformation is done by including special files in the package that describe the sections to add to the configuration files. When a package is uninstalled, those same changes are then reversed, making this a two-way transformation.

Specifying source code transformations

1. Files that you want to insert from the package into the project must be located within the package's `content` and `contentFiles` folders. For example, if you want a file called `ContosoData.cs` to be installed in a `Models` folder of the target project, it must be inside the `content\Models` and `contentFiles\{lang\}\{tfm\}\Models` folders in the package.
2. To instruct NuGet to apply token replacement at install time, append `.pp` to the source code file name. After installation, the file will not have the `.pp` extension.

For example, to make transformations in `ContosoData.cs`, name the file in the package `ContosoData.cs.pp`. After installation it will appear as `ContosoData.cs`.

3. In the source code file, use case-insensitive tokens of the form `$token$` to indicate values that NuGet should replace with project properties:

```
namespace $rootnamespace$.Models
{
    public struct CategoryInfo
    {
        public string categoryid;
        public string description;
        public string htmlUrl;
        public string rssUrl;
        public string title;
    }
}
```

Upon installation, NuGet replaces `$rootnamespace$` with `Fabrikam` assuming the target project's whose root namespace is `Fabrikam`.

The `$rootnamespace$` token is the most commonly used project property; all others are listed in [project properties](#). Be mindful, of course, that some properties might be specific to the project type.

Specifying config file transformations

As described in the sections that follow, config file transformations can be done in two ways:

- Include `app.config.transform` and `web.config.transform` files in your package's `content` folder, where the `.transform` extension tells NuGet that these files contain the XML to merge with existing config files when the package is installed. When a package is uninstalled, that same XML is removed.
- Include `app.config.install.xdt` and `web.config.install.xdt` files in your package's `content` folder, using [XDT syntax](#) to describe the desired changes. With this option you can also include a `.uninstall.xdt` file to reverse changes when the package is removed from a project.

NOTE

Transformations are not applied to `.config` files referenced as a link in Visual Studio.

The advantage of using XDT is that instead of simply merging two static files, it provides a syntax for manipulating the structure of an XML DOM using element and attribute matching using full XPath support. XDT can then add, update, or remove elements, place new elements at a specific location, or replace/remove elements (including child nodes). This makes it straightforward to create uninstall transforms that back out all transformations done during package installation.

XML transforms

The `app.config.transform` and `web.config.transform` in a package's `content` folder contain only those elements to merge into the project's existing `app.config` and `web.config` files.

As an example, suppose the project initially contains the following content in `web.config`:

```
<configuration>
  <system.webServer>
    <modules>
      <add name="ContosoUtilities" type="Contoso.Utilities" />
    </modules>
  </system.webServer>
</configuration>
```

To add a `MyNuModule` element to the `modules` section during package install, create a `web.config.transform` file in the package's `content` folder that looks like this:

```
<configuration>
  <system.webServer>
    <modules>
      <add name="MyNuModule" type="Sample.MyNuModule" />
    </modules>
  </system.webServer>
</configuration>
```

After NuGet installs the package, `web.config` will appear as follows:

```

<configuration>
  <system.webServer>
    <modules>
      <add name="ContosoUtilities" type="Contoso.Utilities" />
      <add name="MyNuModule" type="Sample.MyNuModule" />
    </modules>
  </system.webServer>
</configuration>

```

Notice that NuGet didn't replace the `modules` section, it just merged the new entry into it by adding only new elements and attributes. NuGet will not change any existing elements or attributes.

When the package is uninstalled, NuGet will examine the `.transform` files again and remove the elements it contains from the appropriate `.config` files. Note that this process will not affect any lines in the `.config` file that you modify after package installation.

As a more extensive example, the [Error Logging Modules and Handlers for ASP.NET \(ELMAH\)](#) package adds many entries into `web.config`, which are again removed when a package is uninstalled.

To examine its `web.config.transform` file, download the ELMAH package from the link above, change the package extension from `.nupkg` to `.zip`, and then open `content\web.config.transform` in that ZIP file.

To see the effect of installing and uninstalling the package, create a new ASP.NET project in Visual Studio (the template is under **Visual C# > Web** in the New Project dialog), and select an empty ASP.NET application. Open `web.config` to see its initial state. Then right-click the project, select **Manage NuGet Packages**, browse for ELMAH on nuget.org, and install the latest version. Notice all the changes to `web.config`. Now uninstall the package and you see `web.config` revert to its prior state.

XDT transforms

You can modify config files using [XDT syntax](#). You can also have NuGet replace tokens with [project properties](#) by including the property name within `$` delimiters (case-insensitive).

For example, the following `app.config.install.xdt` file will insert an `appSettings` element into `app.config` containing the `FullPath`, `FileName`, and `ActiveConfigurationSettings` values from the project:

```

<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <appSettings xdt:Transform="Insert">
    <add key="FullPath" value="$FullPath$" />
    <add key="FileName" value="$filename$" />
    <add key="ActiveConfigurationSettings" value="$ActiveConfigurationSettings$" />
  </appSettings>
</configuration>

```

For another example, suppose the project initially contains the following content in `web.config`:

```

<configuration>
  <system.webServer>
    <modules>
      <add name="ContosoUtilities" type="Contoso.Utilities" />
    </modules>
  </system.webServer>
</configuration>

```

To add a `MyNuModule` element to the `modules` section during package install, the package's `web.config.install.xdt` would contain the following:

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <system.webServer>
    <modules>
      <add name="MyNuModule" type="Sample.MyNuModule" xdt:Transform="Insert" />
    </modules>
  </system.webServer>
</configuration>
```

After installing the package, `web.config` will look like this:

```
<configuration>
  <system.webServer>
    <modules>
      <add name="ContosoUtilities" type="Contoso.Utilities" />
      <add name="MyNuModule" type="Sample.MyNuModule" />
    </modules>
  </system.webServer>
</configuration>
```

To remove only the `MyNuModule` element during package uninstall, the `web.config.uninstall.xdt` file should contain the following:

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <system.webServer>
    <modules>
      <add name="MyNuModule" xdt:Transform="Remove" xdt:Locator="Match(name)" />
    </modules>
  </system.webServer>
</configuration>
```

Select Assemblies Referenced By Projects

6/28/2019 • 2 minutes to read • [Edit Online](#)

Explicit assembly references allows a subset of assemblies to be used for IntelliSense and compiling, while all assemblies are available at run-time. `PackageReference` and `packages.config` work differently, and as a result package authors need to take care to create the package to be compatible with both project types.

NOTE

Explicit assembly references are related to .NET assemblies. It is not a method to distribute native assemblies that are P/Invoked by a managed assembly.

`PackageReference` support

When a project uses a package with `PackageReference` and the package contains a `ref\<tfm>\` directory, NuGet will classify those assemblies as compile-time assets, while the `lib\<tfm>\` assemblies are classified as runtime assets. Assemblies in `ref\<tfm>\` are not used at runtime. This means it is necessary for any assembly in `ref\<tfm>\` to have a matching assembly in either `lib\<tfm>\` or a relevant `runtime\` directory, otherwise runtime errors will likely occur. Since assemblies in `ref\<tfm>\` are not used at runtime, they may be [metadata-only assemblies](#) to reduce package size.

IMPORTANT

If a package contains the nuspec `<references>` element (used by `packages.config`, see below) and does not contain assemblies in `ref\<tfm>\`, NuGet will advertise the assemblies listed in the nuspec `<references>` element as both the compile and runtime assets. This means there will be runtime exceptions when the referenced assemblies need to load any other assembly in the `lib\<tfm>\` directory.

NOTE

If the package contains a `runtime\` directory, NuGet may not use the assets in the `lib\` directory.

`packages.config` support

Projects using `packages.config` to manage NuGet packages normally add references to all assemblies in the `lib\<tfm>\` directory. The `ref\` directory was added to support `PackageReference` and therefore isn't considered when using `packages.config`. To explicitly set which assemblies are referenced for projects using `packages.config`, the package must use the [`<references>` element in the nuspec file](#). For example:

```
<references>
  <group targetFramework="net45">
    <reference file="MyLibrary.dll" />
  </group>
</references>
```

NOTE

`packages.config` project use a process called `ResolveAssemblyReference` to copy assemblies to the `bin\<configuration>\` output directory. Your project's assembly is copied, then the build system looks at the assembly manifest for referenced assemblies, then copies those assemblies and recursively repeats for all assemblies. This means that if any of the assemblies in your `lib\<tfm>\` directory are not listed in any other assembly's manifest as a dependency (if the assembly is loaded at runtime using `Assembly.Load`, MEF or another dependency injection framework), then it may not be copied to your project's `bin\<configuration>\` output directory despite being in `bin\<tfm>\`.

Example

My package will contain three assemblies, `MyLib.dll`, `MyHelpers.dll` and `MyUtilities.dll`, which are targeting the .NET Framework 4.7.2. `MyUtilities.dll` contains classes intended to be used only by the other two assemblies, so I don't want to make those classes available in IntelliSense or at compile time to projects using my package. My `nuspec` file needs to contain the following XML elements:

```
<references>
  <group targetFramework="net472">
    <reference file="MyLib.dll" />
    <reference file="MyHelpers.dll" />
  </group>
</references>
```

and the files in the package will be:

```
lib\net472\MyLib.dll
lib\net472\MyHelpers.dll
lib\net472\MyUtilities.dll
ref\net472\MyLib.dll
ref\net472\MyHelpers.dll
```

Set a NuGet package type

7/12/2019 • 2 minutes to read • [Edit Online](#)

With NuGet 3.5+, packages can be marked with a specific *package type* to indicate its intended use. Packages not marked with a type, including all packages created with earlier versions of NuGet, default to the `Dependency` type.

- `Dependency` type packages add build- or run-time assets to libraries and applications, and can be installed in any project type (assuming they are compatible).
- `DotnetCliTool` type packages are extensions to the [dotnet CLI](#) and are invoked from the command line. Such packages can be installed only in .NET Core projects and have no effect on restore operations. More details about these per-project extensions are available in the [.NET Core extensibility](#) documentation.
- Custom type packages use an arbitrary type identifier that conforms to the same format rules as package IDs. Any type other than `Dependency` and `DotnetCliTool`, however, are not recognized by the NuGet Package Manager in Visual Studio.

Package types are set in the `.nuspec` file. It's best for backwards compatibility to *not* explicitly set the `Dependency` type and to instead rely on NuGet assuming this type when no type is specified.

- `.nuspec` : Indicate the package type within a `packageTypes\packageType` node under the `<metadata>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<package xmlns="http://schemas.microsoft.com/packaging/2012/06/nuspec.xsd">
  <metadata>
    <!-- ... -->
    <packageTypes>
      <packageType name="DotnetCliTool" />
    </packageTypes>
  </metadata>
</package>
```

Creating localized NuGet packages

11/5/2019 • 4 minutes to read • [Edit Online](#)

There are two ways to create localized versions of a library:

1. Include all localized resources assemblies in a single package.
2. Create separate localized satellite packages by following a strict set of conventions.

Both methods have their advantages and disadvantages, as described in the following sections.

Localized resource assemblies in a single package

Including localized resource assemblies in a single package is typically the simplest approach. To do this, create folders within `lib` for supported language other than the package default (assumed to be `en-us`). In these folders you can place resource assemblies and localized IntelliSense XML files.

For example, the following folder structure supports, German (de), Italian (it), Japanese (ja), Russian (ru), Chinese (Simplified) (zh-Hans), and Chinese (Traditional) (zh-Hant):

```
lib
└── net40
    ├── Contoso.Utilities.dll
    └── Contoso.Utilities.xml

    ├── de
    │   ├── Contoso.Utilities.resources.dll
    │   └── Contoso.Utilities.xml

    ├── it
    │   ├── Contoso.Utilities.resources.dll
    │   └── Contoso.Utilities.xml

    ├── ja
    │   ├── Contoso.Utilities.resources.dll
    │   └── Contoso.Utilities.xml

    ├── ru
    │   ├── Contoso.Utilities.resources.dll
    │   └── Contoso.Utilities.xml

    ├── zh-Hans
    │   ├── Contoso.Utilities.resources.dll
    │   └── Contoso.Utilities.xml

    └── zh-Hant
        ├── Contoso.Utilities.resources.dll
        └── Contoso.Utilities.xml
```

You can see that the languages are all listed underneath the `net40` target framework folder. If you're [supporting multiple frameworks](#), then you have a folder under `lib` for each variant.

With these folders in place, you then reference all the files in your `.nuspec`:

```

<?xml version="1.0"?>
<package>
  <metadata>...
  </metadata>
  <files>
    <file src="lib\**" target="lib" />
  </files>
</package>

```

One example package that uses this approach is [Microsoft.Data.OData 5.4.0](#).

Advantages and disadvantages (localized resource assemblies)

Bundling all languages in a single package has a few disadvantages:

1. **Shared metadata:** Because a NuGet package can only contain a single `.nuspec` file, you can provide metadata for only a single language. That is, NuGet does not support localized metadata.
2. **Package size:** Depending on the number of languages you support, the library can become considerably large, which slows installing and restoring the package.
3. **Simultaneous releases:** Bundling localized files into a single package requires that you release all assets in that package simultaneously, rather than being able to release each localization separately. Furthermore, any update to any one localization requires a new version of the entire package.

However, it also has a few benefits:

1. **Simplicity:** Consumers of the package get all supported languages in a single install, rather than having to install each language separately. A single package is also easier to find on [nuget.org](#).
2. **Coupled versions:** Because all of the resource assemblies are in the same package as the primary assembly, they all share the same version number and don't run a risk of getting erroneously decoupled.

Localized satellite packages

Similar to how .NET Framework supports satellite assemblies, this method separates localized resources and IntelliSense XML files into satellite packages.

To do this, your primary package uses the naming convention `{identifier}.{version}.nupkg` and contains the assembly for the default language (such as en-US). For example, `ContosoUtilities.1.0.0.nupkg` would contain the following structure:

```

lib
└──net40
    ContosoUtilities.dll
    ContosoUtilities.xml

```

A satellite assembly then uses the naming convention `{identifier}.{language}.{version}.nupkg`, such as `ContosoUtilities.de.1.0.0.nupkg`. The identifier **must** exactly match that of the primary package.

Because this is a separate package, it has its own `.nuspec` file that contains localized metadata. Be mindful that the language in the `.nuspec` **must** match the one used in the filename.

The satellite assembly **must** also declare an exact version of the primary package as a dependency, using the `[]` version notation (see [Package versioning](#)). For example, `ContosoUtilities.de.1.0.0.nupkg` must declare a dependency on `ContosoUtilities.1.0.0.nupkg` using the `[1.0.0]` notation. The satellite package can, of course, have a different version number than the primary package.

The satellite package's structure must then include the resource assembly and XML IntelliSense file in a subfolder that matches `{language}` in the package filename:

```
lib
└── net40
    └── de
        ContosoUtilities.resources.dll
        ContosoUtilities.xml
```

Note: unless specific subcultures such as `ja-JP` are necessary, always use the higher level language identifier, like `ja`.

In a satellite assembly, NuGet will recognize **only** those files in the folder that matches the `{language}` in the filename. All others are ignored.

When all of these conventions are met, NuGet will recognize the package as a satellite package and install the localized files into the primary package's `lib` folder, as if they had been originally bundled. Uninstalling the satellite package will remove its files from that same folder.

You would create additional satellite assemblies in the same way for each supported language. For an example, examine the set of ASP.NET MVC packages:

- [Microsoft.AspNet.Mvc](#) (English primary)
- [Microsoft.AspNet.Mvc.de](#) (German)
- [Microsoft.AspNet.Mvc.ja](#) (Japanese)
- [Microsoft.AspNet.Mvc.zh-Hans](#) (Chinese (Simplified))
- [Microsoft.AspNet.Mvc.zh-Hant](#) (Chinese (Traditional))

Summary of required conventions

- Primary package must be named `{identifier}.{version}.nupkg`
- A satellite package must be named `{identifier}.{language}.{version}.nupkg`
- A satellite package's `.nuspec` must specify its language to match the filename.
- A satellite package must declare a dependency on an exact version of the primary using the `[]` notation in its `.nuspec` file. Ranges are not supported.
- A satellite package must place files in the `lib\[{framework}\]\{language}` folder that exactly matches `{language}` in the filename.

Advantages and disadvantages (satellite packages)

Using satellite packages has a few benefits:

1. **Package size:** The overall footprint of the primary package is minimized, and consumers only incur the costs of each language they want to use.
2. **Separate metadata:** Each satellite package has its own `.nuspec` file and thus its own localized metadata because. This can allow some consumers to find packages more easily by searching nuget.org with localized terms.
3. **Decoupled releases:** Satellite assemblies can be released over time, rather than all at once, allowing you to spread out your localization efforts.

However, satellite packages have their own set of disadvantages:

1. **Clutter:** Instead of a single package, you have many packages that can lead to cluttered search results on nuget.org and a long list of references in a Visual Studio project.
2. **Strict conventions.** Satellite packages must follow the conventions exactly or the localized versions won't be picked up properly.
3. **Versioning:** Each satellite package must have an exact version dependency on the primary package. This means that updating the primary package may require updating all satellite packages as well, even if the resources didn't change.

Create UWP packages

10/15/2019 • 5 minutes to read • [Edit Online](#)

The [Universal Windows Platform \(UWP\)](#) provides a common app platform for every device that runs Windows 10. Within this model, UWP apps can call both the WinRT APIs that are common to all devices, and also APIs (including Win32 and .NET) that are specific to the device family on which the app is running.

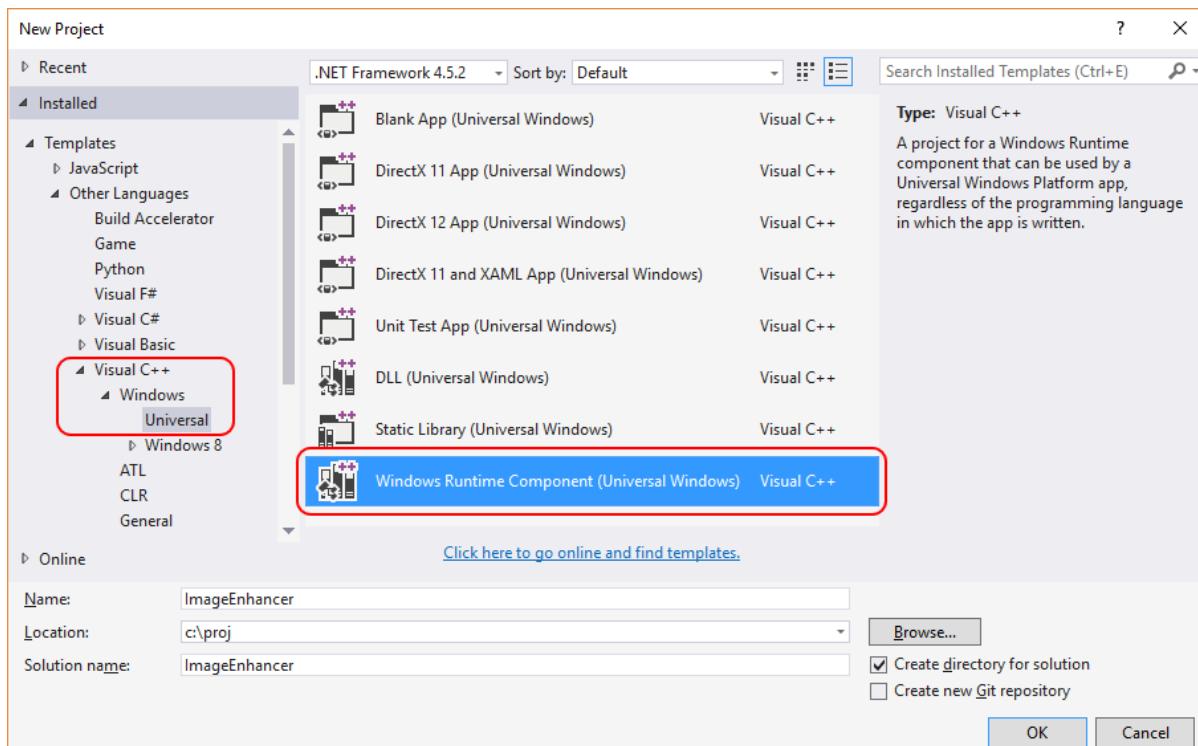
In this walkthrough you create a NuGet package with a native UWP component (including a XAML control) that can be used in both Managed and Native projects.

Prerequisites

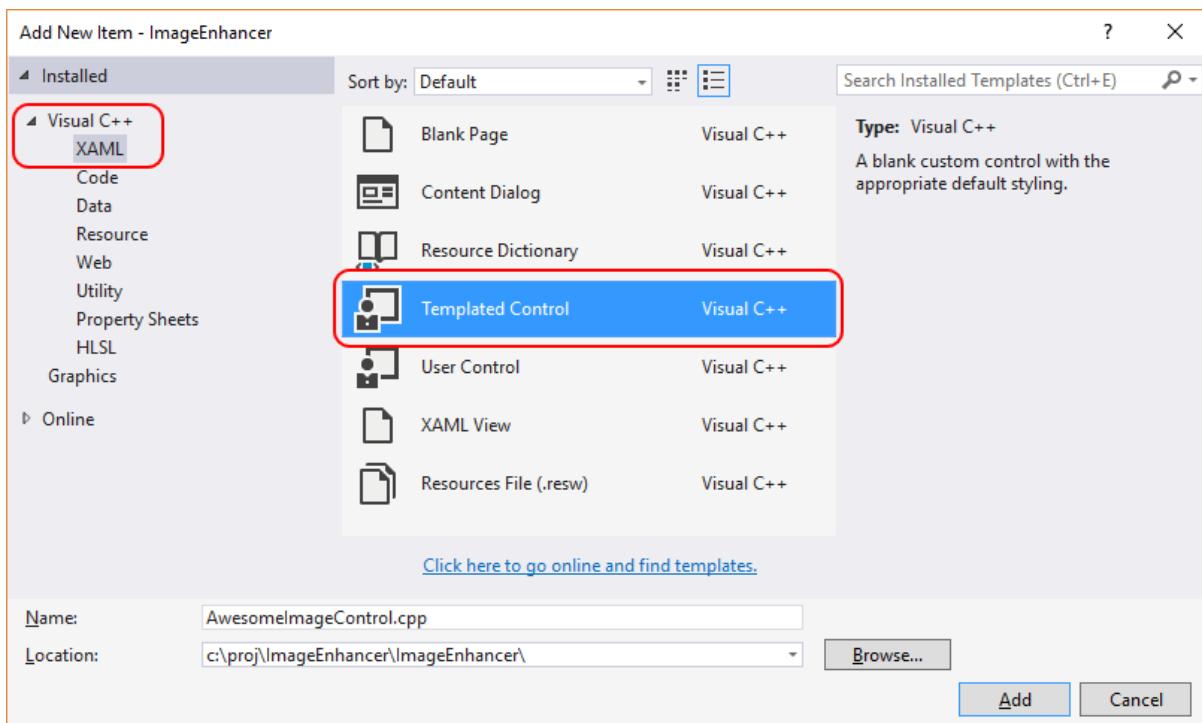
1. Visual Studio 2017 or Visual Studio 2015. Install the 2017 Community edition for free from [visualstudio.com](#); you can use the Professional and Enterprise editions as well.
2. NuGet CLI. Download the latest version of `nuget.exe` from [nuget.org/downloads](#), saving it to a location of your choice (the download is the `.exe` directly). Then add that location to your PATH environment variable if it isn't already.

Create a UWP Windows Runtime component

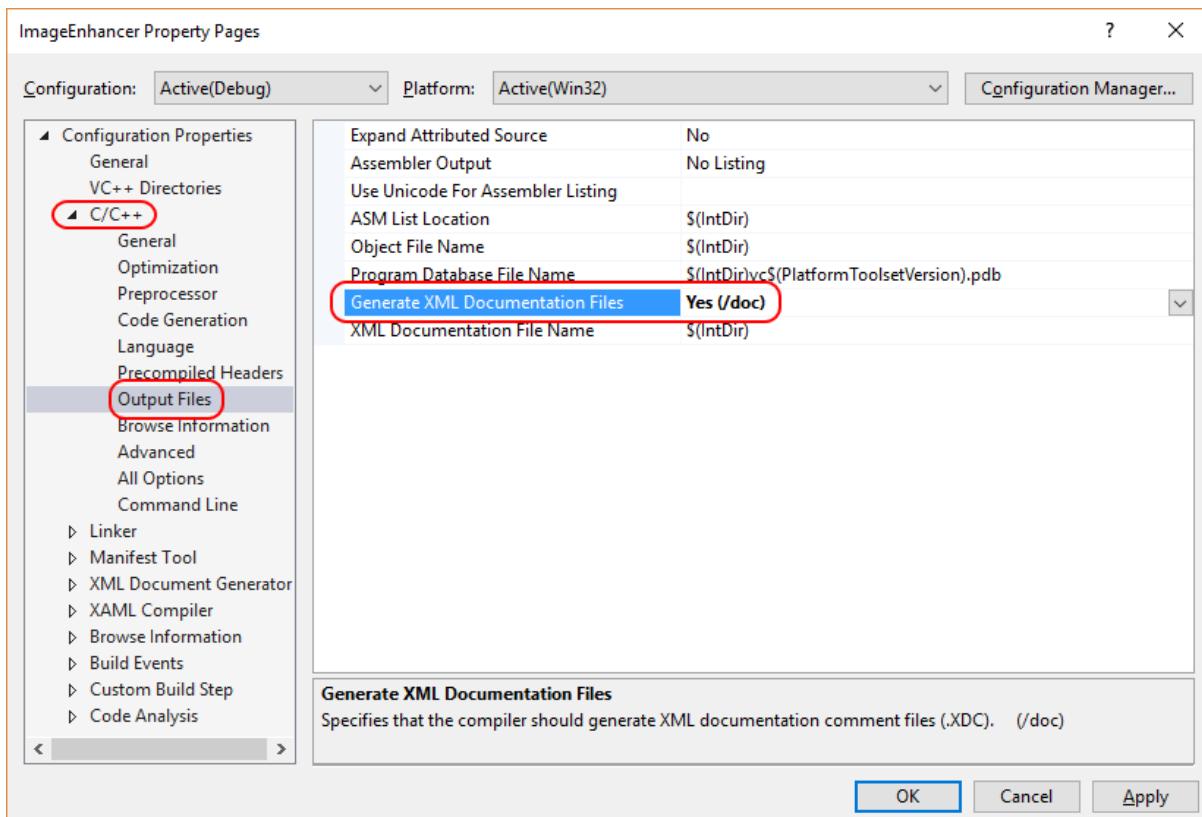
1. In Visual Studio, choose **File > New > Project**, expand the **Visual C++ > Windows > Universal** node, select the **Windows Runtime Component (Universal Windows)** template, change the name to ImageEnhancer, and click OK. Accept the default values for Target Version and Minimum Version when prompted.



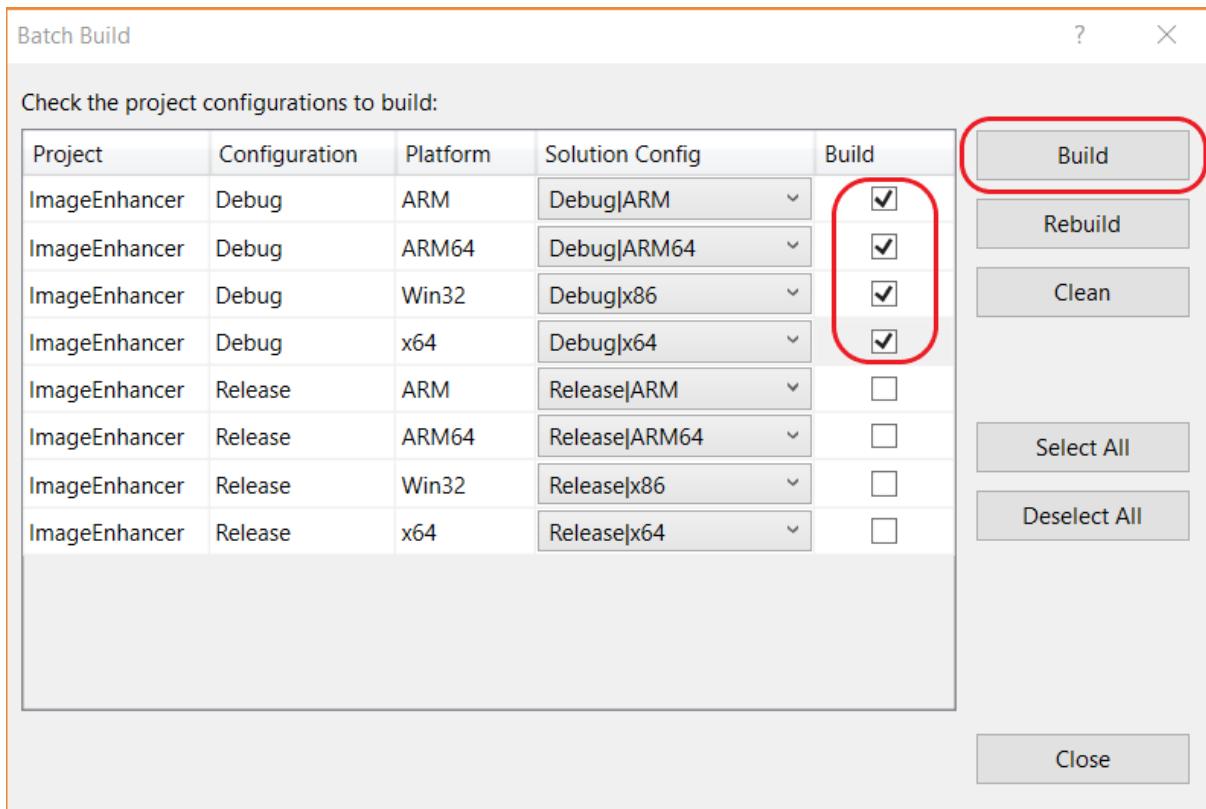
2. Right click the project in Solution Explorer, select **Add > New Item**, click the **Visual C++ > XAML** node, select **Templated Control**, change the name to AwesomeImageControl.cpp, and click **Add**:



3. Right-click the project in Solution Explorer and select **Properties**. In the Properties page, expand **Configuration Properties > C/C++** and click **Output Files**. In the pane on the right, change the value for **Generate XML Documentation Files** to Yes:



4. Right click the *solution* now, select **Batch Build**, check the three Debug boxes in the dialog as shown below. This makes sure that when you do a build, you generate a full set of artifacts for each of the target systems that Windows supports.



5. In the Batch Build dialog, and click **Build** to verify the project and create the output files that you need for the NuGet package.

NOTE

In this walkthrough you use the Debug artifacts for the package. For non-debug package, check the Release options in the Batch Build dialog instead, and refer to the resulting Release folders in the steps that follow.

Create and update the .nuspec file

To create the initial `.nuspec` file, do the three steps below. The sections that follow then guide you through other necessary updates.

1. Open a command prompt and navigate to the folder containing `ImageEnhancer.vcxproj` (this will be a subfolder below where the solution file is).
2. Run the NuGet `spec` command to generate `ImageEnhancer.nuspec` (the name of the file is taken from the name of the `.vcxproj` file):

```
nuget spec
```

3. Open `ImageEnhancer.nuspec` in an editor and update it to match the following, replacing `YOUR_NAME` with an appropriate value. The `<id>` value, specifically, must be unique across nuget.org (see the naming conventions described in [Creating a package](#)). Also note that you must also update the author and description tags or you get an error during the packing step.

```
<?xml version="1.0"?>
<package>
  <metadata>
    <id>ImageEnhancer.YOUR_NAME</id>
    <version>1.0.0</version>
    <title>ImageEnhancer</title>
    <authors>YOUR_NAME</authors>
    <owners>YOUR_NAME</owners>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>Awesome Image Enhancer</description>
    <releaseNotes>First release</releaseNotes>
    <copyright>Copyright 2016</copyright>
    <tags>image enhancer imageenhancer</tags>
  </metadata>
</package>
```

NOTE

For packages built for public consumption, pay special attention to the `<tags>` element, as these tags help others find your package and understand what it does.

Adding Windows metadata to the package

A Windows Runtime Component requires metadata that describes all of its publicly available types, which makes it possible for other apps and libraries to consume the component. This metadata is contained in a .winmd file, which is created when you compile the project and must be included in your NuGet package. An XML file with IntelliSense data is also built at the same time, and should be included as well.

Add the following `<files>` node to the `.nuspec` file:

```
<package>
  <metadata>
    ...
  </metadata>

  <files>
    <!-- WinMd and IntelliSense files -->
    <file src="..\Debug\ImageEnhancer\ImageEnhancer.winmd" target="lib\uap10.0"/>
    <file src="..\Debug\ImageEnhancer\ImageEnhancer.xml" target="lib\uap10.0"/>
  </files>
</package>
```

Adding XAML content

To include a XAML control with your component, you need to add the XAML file that has the default template for the control (as generated by the project template). This also goes in the `<files>` section:

```
<?xml version="1.0"?>
<package>
  <metadata>
    ...
  </metadata>
  <files>
    ...

    <!-- XAML controls -->
    <file src="Themes\Generic.xaml" target="lib\uap10.0\Themes"/>

  </files>
</package>
```

Adding the native implementation libraries

Within your component, the core logic of the `ImageEnhancer` type is in native code, which is contained in the various `ImageEnhancer.dll` assemblies that are generated for each target runtime (ARM, x86, and x64). To include these in the package, reference them in the `<files>` section along with their associated `.pri` resource files:

```
<?xml version="1.0"?>
<package>
  <metadata>
    ...
  </metadata>
  <files>
    ...

    <!-- DLLs and resources -->
    <file src="..\ARM\Debug\ImageEnhancer\ImageEnhancer.dll" target="runtimes\win10-arm\native"/>
    <file src="..\ARM\Debug\ImageEnhancer\ImageEnhancer.pri" target="runtimes\win10-arm\native"/>

    <file src="..\ARM64\Debug\ImageEnhancer\ImageEnhancer.dll" target="runtimes\win10-arm64\native"/>
    <file src="..\ARM64\Debug\ImageEnhancer\ImageEnhancer.pri" target="runtimes\win10-arm64\native"/>

    <file src="..\x64\Debug\ImageEnhancer\ImageEnhancer.dll" target="runtimes\win10-x64\native"/>
    <file src="..\x64\Debug\ImageEnhancer\ImageEnhancer.pri" target="runtimes\win10-x64\native"/>

    <file src="..\Debug\ImageEnhancer\ImageEnhancer.dll" target="runtimes\win10-x86\native"/>
    <file src="..\Debug\ImageEnhancer\ImageEnhancer.pri" target="runtimes\win10-x86\native"/>

  </files>
</package>
```

Adding .targets

Next, C++ and JavaScript projects that might consume your NuGet package need a `.targets` file to identify the necessary assembly and `winmd` files. (C# and Visual Basic projects do this automatically.) Create this file by copying the text below into `ImageEnhancer.targets` and save it in the same folder as the `.nuspec` file. *Note:* This `.targets` file needs to be the same name as the package ID (e.g. the `<Id>` element in the `.nuspec` file):

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <ImageEnhancer-Platform Condition="‘$(Platform)’ == ‘Win32’">x86</ImageEnhancer-Platform>
    <ImageEnhancer-Platform Condition="‘$(Platform)’ != ‘Win32’">$(Platform)</ImageEnhancer-Platform>
  </PropertyGroup>
  <ItemGroup Condition="‘$(TargetPlatformIdentifier)’ == ‘UAP’">
    <Reference Include="‘$(MSBuildThisFileDirectory)..\..\lib\uap10.0\ImageEnhancer.winmd’">
      <Implementation>ImageEnhancer.dll</Implementation>
    </Reference>
    <ReferenceCopyLocalPaths Include="‘$(MSBuildThisFileDirectory)..\..\runtimes\win10-$(ImageEnhancer-Platform)\native\ImageEnhancer.dll’" />
  </ItemGroup>
</Project>
```

Then refer to `ImageEnhancer.targets` in your `.nuspec` file:

```

<?xml version="1.0"?>
<package>
  <metadata>
    ...
  </metadata>
  <files>
    ...
    <!-- .targets -->
    <file src="ImageEnhancer.targets" target="build\native"/>
  </files>
</package>

```

Final .nuspec

Your final `.nuspec` file should now look like the following, where again YOUR_NAME should be replaced with an appropriate value:

```

<?xml version="1.0"?>
<package>
  <metadata>
    <id>ImageEnhancer.YOUR_NAME</id>
    <version>1.0.0</version>
    <title>ImageEnhancer</title>
    <authors>YOUR_NAME</authors>
    <owners>YOUR_NAME</owners>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>Awesome Image Enhancer</description>
    <releaseNotes>First Release</releaseNotes>
    <copyright>Copyright 2016</copyright>
    <tags>image enhancer imageenhancer</tags>
  </metadata>
  <files>
    <!-- WinMd and IntelliSense -->
    <file src="..\Debug\ImageEnhancer\ImageEnhancer.winmd" target="lib\uap10.0"/>
    <file src="..\Debug\ImageEnhancer\ImageEnhancer.xml" target="lib\uap10.0"/>

    <!-- XAML controls -->
    <file src="Themes\Generic.xaml" target="lib\uap10.0\Themes"/>

    <!-- DLLs and resources -->
    <file src="..\ARM\Debug\ImageEnhancer\ImageEnhancer.dll" target="runtimes\win10-arm\native"/>
    <file src="..\ARM\Debug\ImageEnhancer\ImageEnhancer.pri" target="runtimes\win10-arm\native"/>
    <file src="..\ARM64\Debug\ImageEnhancer\ImageEnhancer.dll" target="runtimes\win10-arm64\native"/>
    <file src="..\ARM64\Debug\ImageEnhancer\ImageEnhancer.pri" target="runtimes\win10-arm64\native"/>
    <file src="..\x64\Debug\ImageEnhancer\ImageEnhancer.dll" target="runtimes\win10-x64\native"/>
    <file src="..\x64\Debug\ImageEnhancer\ImageEnhancer.pri" target="runtimes\win10-x64\native"/>
    <file src="..\Debug\ImageEnhancer\ImageEnhancer.dll" target="runtimes\win10-x86\native"/>
    <file src="..\Debug\ImageEnhancer\ImageEnhancer.pri" target="runtimes\win10-x86\native"/>

    <!-- .targets -->
    <file src="ImageEnhancer.targets" target="build\native"/>
  </files>
</package>

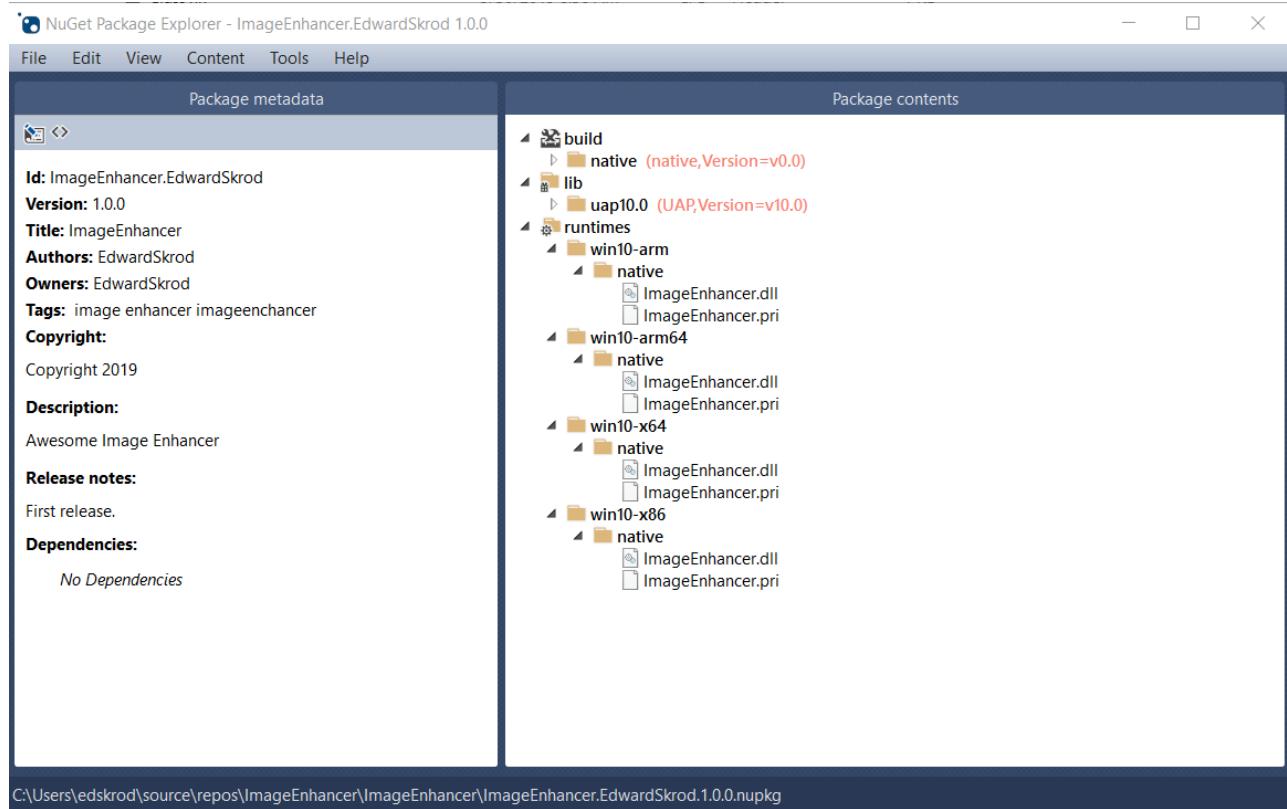
```

Package the component

With the completed `.nuspec` referencing all the files you need to include in the package, you're ready to run the `pack` command:

```
nuget pack ImageEnhancer.nuspec
```

This generates `ImageEnhancer.YOUR_NAME.1.0.0.nupkg`. Opening this file in a tool like the [NuGet Package Explorer](#) and expanding all the nodes, you see the following contents:



TIP

A `.nupkg` file is just a ZIP file with a different extension. You can also examine package contents, then, by changing `.nupkg` to `.zip`, but remember to restore the extension before uploading a package to [nuget.org](#).

To make your package available to other developers, follow the instructions on [Publish a package](#).

Related topics

- [.nuspec Reference](#)
- [Symbol packages](#)
- [Package versioning](#)
- [Supporting Multiple .NET Framework Versions](#)
- [Include MSBuild props and targets in a package](#)
- [Creating Localized Packages](#)

Creating native packages

8/15/2019 • 2 minutes to read • [Edit Online](#)

A native package contains native binaries instead of managed assemblies, allowing it to be used within C++ (or similar) projects. (See [Native C++ Packages](#) in the Consume section.)

To be consumable in a C++ project, a package must target the `native` framework. At present there are not any version numbers associated with this framework as NuGet treats all C++ projects the same.

NOTE

Be sure to include `native` in the `<tags>` section of your `.nuspec` to help other developers find your package by searching on that tag.

Native NuGet packages targeting `native` then provide files in `\build`, `\content`, and `\tools` folders; `\lib` is not used in this case (NuGet cannot directly add references to a C++ project). A package may also include targets and props files in `\build` that NuGet will automatically import into projects that consume the package. Those files must be named the same as the package ID with the `.targets` and/or `.props` extensions. For example, the [cpprestsdk](#) package includes a `cpprestsdk.targets` file in its `\build` folder.

The `\build` folder can be used for all NuGet packages and not just native packages. The `\build` folder respects target frameworks just like the `\content`, `\lib`, and `\tools` folders. This means you can create a `\build\net40` folder and a `\build\net45` folder and NuGet will import the appropriate props and targets files into the project. (Use of PowerShell scripts to import MSBuild targets is not needed.)

Creating UI controls as NuGet packages

11/5/2019 • 5 minutes to read • [Edit Online](#)

Starting with Visual Studio 2017, you can take advantage of added capabilities for UWP and WPF controls that you deliver in NuGet packages. This guide walks you through these capabilities in context of UWP controls using the [ExtensionSDKasNuGetPackage sample](#). The same applies to WPF controls unless mentioned otherwise.

Prerequisites

1. Visual Studio 2017
2. Understanding of how to [Create UWP Packages](#)

Generate Library Layout

NOTE

This is applicable only to UWP controls.

Setting the `GenerateLibraryLayout` property ensures that the project build output is generated in a layout that is ready to be packaged without the need for individual file entries in the nuspec.

From the project properties, go to the build tab and check the "Generate Library Layout" check box. This will modify the project file and set the `GenerateLibraryLayout` flag to true for your currently selected build configuration and platform.

Alternately, edit the the project file to add `<GenerateLibraryLayout>true</GenerateLibraryLayout>` to the first unconditional property group. This would apply the property irrespective of the build configuration and platform.

Add toolbox/assets pane support for XAML controls

To have a XAML control appear in the XAML designer's toolbox in Visual Studio and the Assets pane of Blend, create a `VisualStudioToolsManifest.xml` file in the root of the `tools` folder of your package project. This file is not required if you don't need the control to appear in the toolbox or Assets pane.

```
\build
\lib
\tools
  VisualStudioToolsManifest.xml
```

The structure of the file is as follows:

```

<FileList>
  <File Reference = "your_package_file">
    <ToolboxItems VSCategory="vs_category" BlendCategory="blend_category">
      <Item Type="type_full_name_1" />

      <!-- Any number of additional Items -->
      <Item Type="type_full_name_2" />
      <Item Type="type_full_name_3" />
    </ToolboxItems>
  </File>
</FileList>

```

where:

- *your_package_file*: the name of your control file, such as `ManagedPackage.winmd` ("ManagedPackage" is an arbitrary name used for this example and has no other meaning).
- *vs_category*: The label for the group in which the control should appear in the Visual Studio designer's toolbox. A `vsCategory` is necessary for the control to appear in the toolbox.
- *blend_category*: The label for the group in which the control should appear in the Blend designer's Assets pane. A `BlendCategory` is necessary for the control to appear in Assets.
- *type_full_name_n*: The fully-qualified name for each control, including the namespace, such as `ManagedPackage.MyCustomControl`. Note that the dot format is used for both managed and native types.

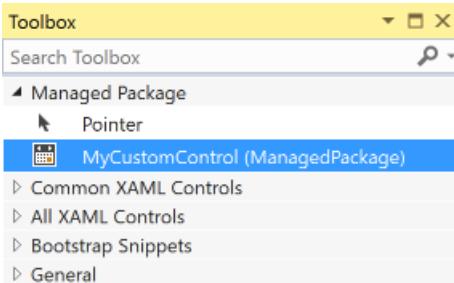
In more advanced scenarios, you can also include multiple `<File>` elements within `<FileList>` when a single package contains multiple control assemblies. You can also have multiple `<ToolboxItems>` nodes within a single `<File>` if you want to organize your controls into separate categories.

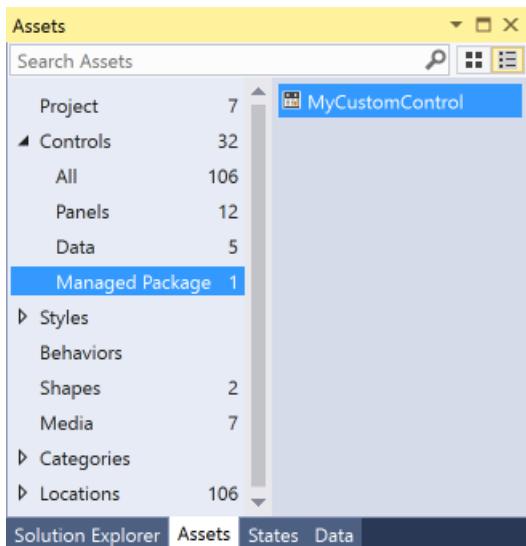
In the following example, the control implemented in `ManagedPackage.winmd` will appear in Visual Studio and Blend in a group named "Managed Package", and "MyCustomControl" will appear in that group. All these names are arbitrary.

```

<FileList>
  <File Reference = "ManagedPackage.winmd">
    <ToolboxItems VSCategory="Managed Package" BlendCategory="Managed Package">
      <Item Type="ManagedPackage.MyCustomControl" />
    </ToolboxItems>
  </File>
</FileList>

```





NOTE

You must explicitly specify every control that you would like to see in the toolbox/assets pane. Ensure you specify them in the format `Namespace.ControlName`.

Add custom icons to your controls

To display a custom icon in the toolbox/assets pane, add an image to your project or the corresponding `design.dll` project with the name "Namespace.ControlName.extension" and set the build action to "Embedded Resource". You must also ensure that the associated `AssemblyInfo.cs` specifies the `ProvideMetadata` attribute -

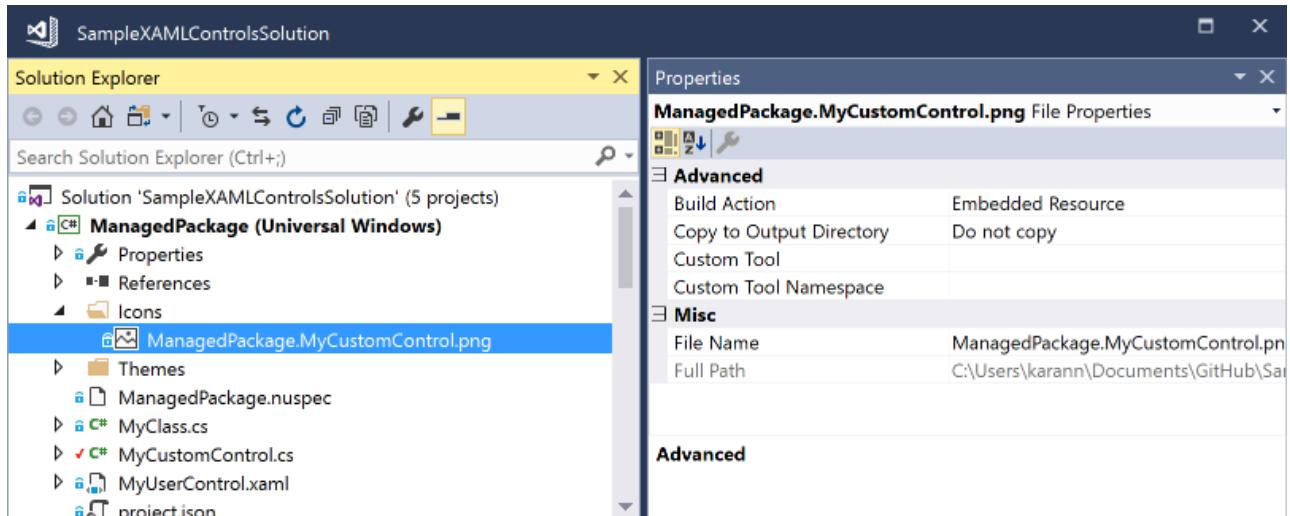
`[assembly: ProvideMetadata(typeof(RegisterMetadata))]`. See this [sample](#).

Supported formats are `.png`, `.jpg`, `.jpeg`, `.gif`, and `.bmp`. The recommended format is BMP24 in 16 pixels by 16 pixels.



The pink background is replaced at runtime. The icons are recolored when the Visual Studio theme is changed and that background color is expected. For more information, please reference [Images and Icons for Visual Studio](#).

In the example below, the project contains an image file named "ManagedPackage.MyCustomControl.png".



NOTE

For native controls, you must put the icon as a resource in the `design.dll` project.

Support specific Windows platform versions

UWP packages have a `TargetPlatformVersion` (TPV) and `TargetPlatformMinVersion` (TPMinV) that define the upper and lower bounds of the OS version where the app can be installed. TPV further specifies the version of the SDK against which the app is built. Be mindful of these properties when authoring a UWP package: using APIs outside the bounds of the platform versions defined in the app will cause either the build to fail or the app to fail at runtime.

For example, let's say you've set the TPMinV for your controls package to Windows 10 Anniversary Edition (10.0; Build 14393), so you want to ensure that the package is consumed only by UWP projects that match that lower bound. To allow your package to be consumed by UWP projects, you must package your controls with the following folder names:

```
\lib\uap10.0.14393\*  
\ref\uap10.0.14393\*
```

NuGet will automatically check the TPMinV of the consuming project, and fail installation if it is lower than Windows 10 Anniversary Edition (10.0; Build 14393)

In case of WPF, let's say you would like your WPF controls package to be consumed by projects targeting .NET Framework v4.6.1 or higher. To enforce that, you must package your controls with the following folder names:

```
\lib\net461\*  
\ref\net461\*
```

Add design-time support

To configure where the control properties show up in the property inspector, add custom adorners, etc., place your `design.dll` file inside the `lib\uap10.0.14393\Design` folder as appropriate to the target platform. Also, to ensure that the **Edit Template > Edit a Copy** feature works, you must include the `Generic.xaml` and any resource dictionaries that it merges in the `<your_assembly_name>\Themes` folder (again, using your actual assembly name). (This file has no impact on the runtime behavior of a control.) The folder structure would thus appear as follows:

```
\lib  
\uap10.0.14393  
\Design  
\MyControl.design.dll  
\your_assembly_name  
\Themes  
  Generic.xaml
```

For WPF, continuing with the example where you would like your WPF controls package to be consumed by projects targeting .NET Framework v4.6.1 or higher:

```
\lib
  \net461
    \Design
      \MyControl.design.dll
    \your_assembly_name
      \Themes
        Generic.xaml
```

NOTE

By default, control properties will show up under the Miscellaneous category in the property inspector.

Use strings and resources

You can embed string resources (`.resw`) in your package that can be used by your control or the consuming UWP project, set the **Build Action** property of the `.resw` file to **PRIResource**.

For an example, refer to [MyCustomControl.cs](#) in the ExtensionSDKasNuGetPackage sample.

NOTE

This is applicable only to UWP controls.

See also

- [Create UWP Packages](#)
- [ExtensionSDKasNuGetPackage sample](#)

Analyzer NuGet formats

10/26/2019 • 3 minutes to read • [Edit Online](#)

The .NET Compiler Platform (also known as "Roslyn") allows developers to create [analyzers](#) that examine the syntax tree and semantics of code as it's being written. This provides developers with a way to create domain-specific analysis tools, such as those that would help guide the use of a particular API or library. You can find more information on the [.NET/Roslyn](#) GitHub wiki. Also see the article, [Use Roslyn to Write a Live Code Analyzer for your API](#) in MSDN Magazine.

Analyzers themselves are typically packaged and distributed as part of the NuGet packages that implement the API or library in question.

For a good example, see the [System.Runtime.Analyzers](#) package, which has the following contents:

- analyzers\dotnet\System.Runtime.Analyzers.dll
- analyzers\dotnet\cs\System.Runtime.CSharp.Analyzers.dll
- analyzers\dotnet\vb\System.Runtime.VisualBasic.Analyzers.dll
- build\System.Runtime.Analyzers.Common.props
- build\System.Runtime.Analyzers.props
- build\System.Runtime.CSharp.Analyzers.props
- build\System.Runtime.VisualBasic.Analyzers.props
- tools\install.ps1
- tools\uninstall.ps1

As you can see, you place the analyzer DLLs into an `analyzers` folder in the package.

Props files, which are included to disable legacy FxCop rules in favor of the analyzer implementation, are placed in the `build` folder.

Install and uninstall scripts that support projects using `packages.config` are placed in `tools`.

Also note that because this package has no platform-specific requirements, the `platform` folder is omitted.

Analyzers path format

The use of the `analyzers` folder is similar to that used for [target frameworks](#), except the specifiers in the path describe development host dependencies instead of build-time. The general format is as follows:

```
$/analyzers/{framework_name}{version}/{supported_architecture}/{supported_language}/{analyzer_name}.dll
```

- **framework_name** and **version**: the *optional* API surface area of the .NET Framework that the contained DLLs need to run. `dotnet` is presently the only valid value because Roslyn is the only host that can run analyzers. If no target is specified, DLLs are assumed to apply to *all* targets.
- **supported_language**: the language for which the DLL applies, one of `cs` (C#) and `vb` (Visual Basic), and `fs` (F#). The language indicates that the analyzer should be loaded only for a project using that language. If no language is specified then the DLL is assumed to apply to *all* languages that support analyzers.
- **analyzer_name**: specifies the DLLs of the analyzer. If you need additional files beyond DLLs, they must be included through a targets or properties files.

Install and uninstall scripts

If the user's project is using `packages.config`, the MSBuild script that picks up the analyzer does not come into play, so you should place `install.ps1` and `uninstall.ps1` in the `tools` folder with the contents that are described below.

install.ps1 file contents

```
param($installPath, $toolsPath, $package, $project)

$analyzersPaths = Join-Path (Join-Path (Split-Path -Path $toolsPath -Parent) "analyzers" ) * -Resolve

foreach($analyzersPath in $analyzersPaths)
{
    # Install the language agnostic analyzers.
    if (Test-Path $analyzersPath)
    {
        foreach ($analyzerFilePath in Get-ChildItem $analyzersPath -Filter *.dll)
        {
            if($project.Object.AnalyzerReferences)
            {
                $project.Object.AnalyzerReferences.Add($analyzerFilePath.FullName)
            }
        }
    }
}

$project.Type # gives the language name like (C# or VB.NET)
$languageFolder = ""
if($project.Type -eq "C#")
{
    $languageFolder = "cs"
}
if($project.Type -eq "VB.NET")
{
    $languageFolder = "vb"
}
if($languageFolder -eq "")
{
    return
}

foreach($analyzersPath in $analyzersPaths)
{
    # Install language specific analyzers.
    $languageAnalyzersPath = join-path $analyzersPath $languageFolder
    if (Test-Path $languageAnalyzersPath)
    {
        foreach ($analyzerFilePath in Get-ChildItem $languageAnalyzersPath -Filter *.dll)
        {
            if($project.Object.AnalyzerReferences)
            {
                $project.Object.AnalyzerReferences.Add($analyzerFilePath.FullName)
            }
        }
    }
}
```

uninstall.ps1 file contents

```

param($installPath, $toolsPath, $package, $project)

$analyzersPaths = Join-Path (Join-Path (Split-Path -Path $toolsPath -Parent) "analyzers" ) * -Resolve

foreach($analyzersPath in $analyzersPaths)
{
    # Uninstall the language agnostic analyzers.
    if (Test-Path $analyzersPath)
    {
        foreach ($analyzerFilePath in Get-ChildItem $analyzersPath -Filter *.dll)
        {
            if($project.Object.AnalyzerReferences)
            {
                $project.Object.AnalyzerReferences.Remove($analyzerFilePath.FullName)
            }
        }
    }
}

$project.Type # gives the language name like (C# or VB.NET)
$languageFolder = ""
if($project.Type -eq "C#")
{
    $languageFolder = "cs"
}
if($project.Type -eq "VB.NET")
{
    $languageFolder = "vb"
}
if($languageFolder -eq "")
{
    return
}

foreach($analyzersPath in $analyzersPaths)
{
    # Uninstall language specific analyzers.
    $languageAnalyzersPath = join-path $analyzersPath $languageFolder
    if (Test-Path $languageAnalyzersPath)
    {
        foreach ($analyzerFilePath in Get-ChildItem $languageAnalyzersPath -Filter *.dll)
        {
            if($project.Object.AnalyzerReferences)
            {
                try
                {
                    $project.Object.AnalyzerReferences.Remove($analyzerFilePath.FullName)
                }
                catch
                {
                }
            }
        }
    }
}
}

```

Create packages for Xamarin with Visual Studio 2015

8/15/2019 • 6 minutes to read • [Edit Online](#)

A package for Xamarin contains code that uses native APIs on iOS, Android, and Windows, depending on the run-time operating system. Although this is straightforward to do, it's preferable to let developers consume the package from a PCL or .NET Standard libraries through a common API surface area.

In this walkthrough you use Visual Studio 2015 create a cross-platform NuGet package that can be used in mobile projects on iOS, Android, and Windows.

1. [Prerequisites](#)
2. [Create the project structure and abstraction code](#)
3. [Write your platform-specific code](#)
4. [Create and update the .nuspec file](#)
5. [Package the component](#)
6. [Related topics](#)

Prerequisites

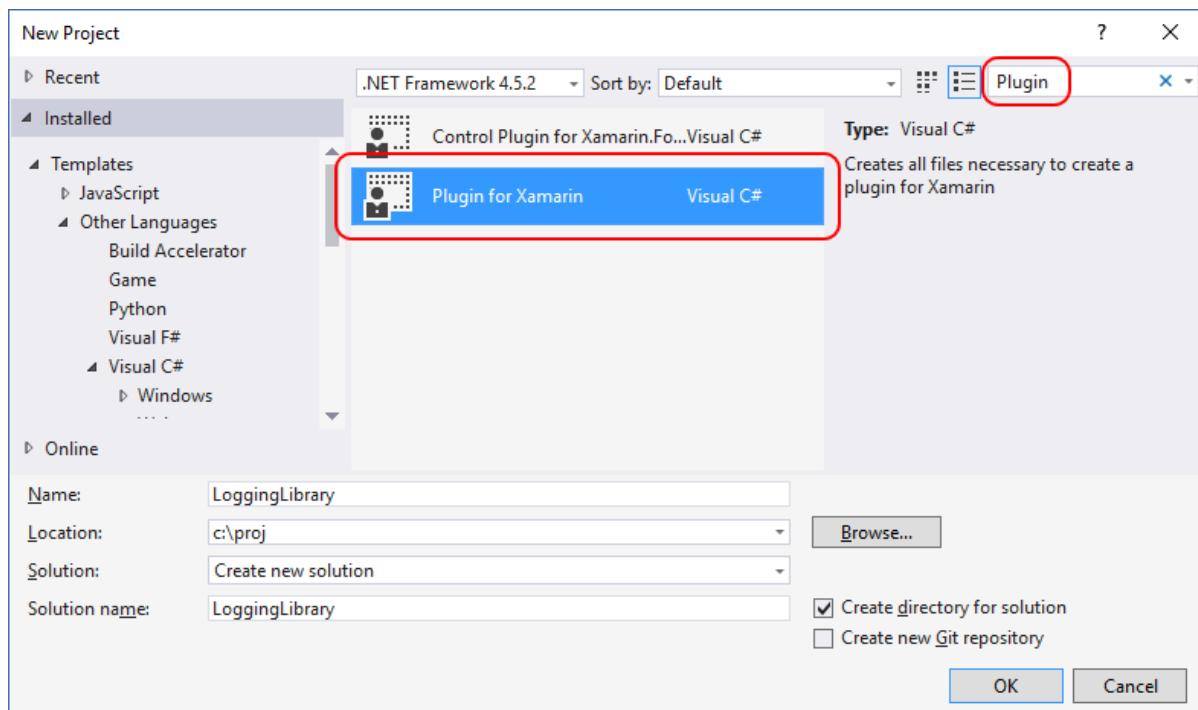
1. Visual Studio 2015 with Universal Windows Platform (UWP) and Xamarin. Install the Community edition for free from [visualstudio.com](#); you can use the Professional and Enterprise editions as well, of course. To include UWP and Xamarin tools, select a Custom install and check the appropriate options.
2. NuGet CLI. Download the latest version of nuget.exe from [nuget.org/downloads](#), saving it to a location of your choice. Then add that location to your PATH environment variable if it isn't already.

NOTE

nuget.exe is the CLI tool itself, not an installer, so be sure to save the downloaded file from your browser instead of running it.

Create the project structure and abstraction code

1. Download and run the [Plugin for Xamarin Templates extension](#) for Visual Studio. These templates will make it easy to create the necessary project structure for this walkthrough.
2. In Visual Studio, **File > New > Project**, search for `Plugin`, select the **Plugin for Xamarin** template, change the name to LoggingLibrary, and click OK.



The resulting solution contains two PCL projects, along with a variety of platform-specific projects:

- The PCL named `Plugin.LoggingLibrary.Abstractions (Portable)`, defines the public interface (the API surface area) of the component, in this case the `ILoggingLibrary` interface contained in the `ILoggingLibrary.cs` file. This is where you define the interface to your library.
- The other PCL, `Plugin.LoggingLibrary (Portable)`, contains code in `CrossLoggingLibrary.cs` that will locate a platform-specific implementation of the abstract interface at run time. You typically don't need to modify this file.
- The platform-specific projects, such as `Plugin.LoggingLibrary.Android`, each contain contain a native implementation of the interface in their respective `LoggingLibraryImplementation.cs` files. This is where you build out your library's code.

By default, the `ILoggingLibrary.cs` file of the `Abstractions` project contains an interface definition, but no methods. For the purposes of this walkthrough, add a `Log` method as follows:

```
using System;

namespace Plugin.LoggingLibrary.Abstractions
{
    /// <summary>
    /// Interface for LoggingLibrary
    /// </summary>
    public interface ILoggingLibrary
    {
        /// <summary>
        /// Log a message
        /// </summary>
        void Log(string text);
    }
}
```

Write your platform-specific code

To implement a platform-specific implementation of the `ILoggingLibrary` interface and its methods, do the following:

1. Open the `LoggingLibraryImplementation.cs` file of each platform project and add the necessary code. For

example (using the `Plugin.LoggingLibrary.Android` project):

```
using Plugin.LoggingLibrary.Abstractions;
using System;

namespace Plugin.LoggingLibrary
{
    /// <summary>
    /// Implementation for Feature
    /// </summary>
    public class LoggingLibraryImplementation : ILoggingLibrary
    {
        /// <summary>
        /// Log a message
        /// </summary>
        public void Log(string text)
        {
            throw new NotImplementedException("Called Log on Android");
        }
    }
}
```

2. Repeat this implementation in the projects for each platform you want to support.
3. Right-click the iOS project, select **Properties**, click the **Build** tab, and remove "\iPhone" from the **Output path** and **XML documentation file** settings. This is just for later convenience in this walkthrough. Save the file when done.
4. Right-click the solution, select **Configuration Manager...**, and check the **Build** boxes for the PCls and each platform you're supporting.
5. Right-click the solution and select **Build Solution** to check your work and produce the artifacts that you package next. If you get errors about missing references, right-click the solution, select **Restore NuGet Packages** to install dependencies, and rebuild.

NOTE

To build for iOS you need a networked Mac connected to Visual Studio as described on [Introduction to Xamarin.iOS for Visual Studio](#). If you don't have a Mac available, clear the iOS project in the configuration manager (step 3 above).

Create and update the .nuspec file

1. Open a command prompt, navigate to the `LoggingLibrary` folder that's one level below where the `.sln` file is, and run the NuGet `spec` command to create the initial `Package.nuspec` file:

```
nuget spec
```

2. Rename this file to `LoggingLibrary.nuspec` and open it in an editor.
3. Update the file to match the following, replacing YOUR_NAME with an appropriate value. The `<id>` value, specifically, must be unique across nuget.org (see the naming conventions described in [Creating a package](#)). Also note that you must also update the author and description tags or you get an error during the packing step.

```

<?xml version="1.0"?>
<package >
  <metadata>
    <id>LoggingLibrary.YOUR_NAME</id>
    <version>1.0.0</version>
    <title>LoggingLibrary</title>
    <authors>YOUR_NAME</authors>
    <owners>YOUR_NAME</owners>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>Awesome application logging utility</description>
    <releaseNotes>First release</releaseNotes>
    <copyright>Copyright 2016</copyright>
    <tags>logger logging logs</tags>
  </metadata>
</package>

```

TIP

You can suffix your package version with `-alpha` , `-beta` or `-rc` to mark your package as pre-release, check [Pre-release versions](#) for more information about pre-release versions.

Add reference assemblies

To include platform-specific reference assemblies, add the following to the `<files>` element of `LoggingLibrary.nuspec` as appropriate for your supported platforms:

```

<!-- Insert below <metadata> element -->
<files>
  <!-- Cross-platform reference assemblies -->
  <file src="Plugin.LoggingLibrary\bin\Release\Plugin.LoggingLibrary.dll"
target="lib\netstandard1.4\Plugin.LoggingLibrary.dll" />
  <file src="Plugin.LoggingLibrary\bin\Release\Plugin.LoggingLibrary.xml"
target="lib\netstandard1.4\Plugin.LoggingLibrary.xml" />
  <file src="Plugin.LoggingLibrary.Abstractions\bin\Release\Plugin.LoggingLibrary.Abstractions.dll"
target="lib\netstandard1.4\Plugin.LoggingLibrary.Abstractions.dll" />
  <file src="Plugin.LoggingLibrary.Abstractions\bin\Release\Plugin.LoggingLibrary.Abstractions.xml"
target="lib\netstandard1.4\Plugin.LoggingLibrary.Abstractions.xml" />

  <!-- iOS reference assemblies -->
  <file src="Plugin.LoggingLibrary.iOS\bin\Release\Plugin.LoggingLibrary.dll"
target="lib\Xamarin.iOS10\Plugin.LoggingLibrary.dll" />
  <file src="Plugin.LoggingLibrary.iOS\bin\Release\Plugin.LoggingLibrary.xml"
target="lib\Xamarin.iOS10\Plugin.LoggingLibrary.xml" />

  <!-- Android reference assemblies -->
  <file src="Plugin.LoggingLibrary.Android\bin\Release\Plugin.LoggingLibrary.dll"
target="lib\MonoAndroid10\Plugin.LoggingLibrary.dll" />
  <file src="Plugin.LoggingLibrary.Android\bin\Release\Plugin.LoggingLibrary.xml"
target="lib\MonoAndroid10\Plugin.LoggingLibrary.xml" />

  <!-- UWP reference assemblies -->
  <file src="Plugin.LoggingLibrary.UWP\bin\Release\Plugin.LoggingLibrary.dll"
target="lib\UAP10\Plugin.LoggingLibrary.dll" />
  <file src="Plugin.LoggingLibrary.UWP\bin\Release\Plugin.LoggingLibrary.xml"
target="lib\UAP10\Plugin.LoggingLibrary.xml" />
</files>

```

NOTE

To shorten the names of the DLL and XML files, right-click on any given project, select the **Library** tab, and change the assembly names.

Add dependencies

If you have specific dependencies for native implementations, use the `<dependencies>` element with `<group>` elements to specify them, for example:

```
<!-- Insert within the <metadata> element -->
<dependencies>
    <group targetFramework="MonoAndroid">
        <!--MonoAndroid dependencies go here-->
    </group>
    <group targetFramework="Xamarin.iOS10">
        <!--Xamarin.iOS10 dependencies go here-->
    </group>
    <group targetFramework="uap">
        <!--uap dependencies go here-->
    </group>
</dependencies>
```

For example, the following would set `iTextSharp` as a dependency for the UAP target:

```
<dependencies>
    <group targetFramework="uap">
        <dependency id="iTextSharp" version="5.5.9" />
    </group>
</dependencies>
```

Final .nuspec

Your final `.nuspec` file should now look like the following, where again `YOUR_NAME` should be replaced with an appropriate value:

```

<?xml version="1.0"?>
<package>
  <metadata>
    <id>LoggingLibrary.YOUR_NAME</id>
    <version>1.0.0</version>
    <title>LoggingLibrary</title>
    <authors>YOUR_NAME</authors>
    <owners>YOUR_NAME</owners>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>Awesome application logging utility</description>
    <releaseNotes>First release</releaseNotes>
    <copyright>Copyright 2016</copyright>
    <tags>logger logging logs</tags>
    <dependencies>
      <group targetFramework="MonoAndroid">
        <!--MonoAndroid dependencies go here-->
      </group>
      <group targetFramework="Xamarin.iOS10">
        <!--Xamarin.iOS10 dependencies go here-->
      </group>
      <group targetFramework="uap">
        <dependency id="iTextSharp" version="5.5.9" />
      </group>
    </dependencies>
  </metadata>
  <files>
    <!-- Cross-platform reference assemblies -->
    <file src="Plugin.LoggingLibrary\bin\Release\Plugin.LoggingLibrary.dll" target="lib\netstandard1.4\Plugin.LoggingLibrary.dll" />
    <file src="Plugin.LoggingLibrary\bin\Release\Plugin.LoggingLibrary.xml" target="lib\netstandard1.4\Plugin.LoggingLibrary.xml" />
    <file src="Plugin.LoggingLibrary.Abstractions\bin\Release\Plugin.LoggingLibrary.Abstractions.dll" target="lib\netstandard1.4\Plugin.LoggingLibrary.Abstractions.dll" />
    <file src="Plugin.LoggingLibrary.Abstractions\bin\Release\Plugin.LoggingLibrary.Abstractions.xml" target="lib\netstandard1.4\Plugin.LoggingLibrary.Abstractions.xml" />
    <!-- iOS reference assemblies -->
    <file src="Plugin.LoggingLibrary.iOS\bin\Release\Plugin.LoggingLibrary.dll" target="lib\Xamarin.iOS10\Plugin.LoggingLibrary.dll" />
    <file src="Plugin.LoggingLibrary.iOS\bin\Release\Plugin.LoggingLibrary.xml" target="lib\Xamarin.iOS10\Plugin.LoggingLibrary.xml" />
    <!-- Android reference assemblies -->
    <file src="Plugin.LoggingLibrary.Android\bin\Release\Plugin.LoggingLibrary.dll" target="lib\MonoAndroid10\Plugin.LoggingLibrary.dll" />
    <file src="Plugin.LoggingLibrary.Android\bin\Release\Plugin.LoggingLibrary.xml" target="lib\MonoAndroid10\Plugin.LoggingLibrary.xml" />
    <!-- UWP reference assemblies -->
    <file src="Plugin.LoggingLibrary.UWP\bin\Release\Plugin.LoggingLibrary.dll" target="lib\UAP10\Plugin.LoggingLibrary.dll" />
    <file src="Plugin.LoggingLibrary.UWP\bin\Release\Plugin.LoggingLibrary.xml" target="lib\UAP10\Plugin.LoggingLibrary.xml" />
  </files>
</package>

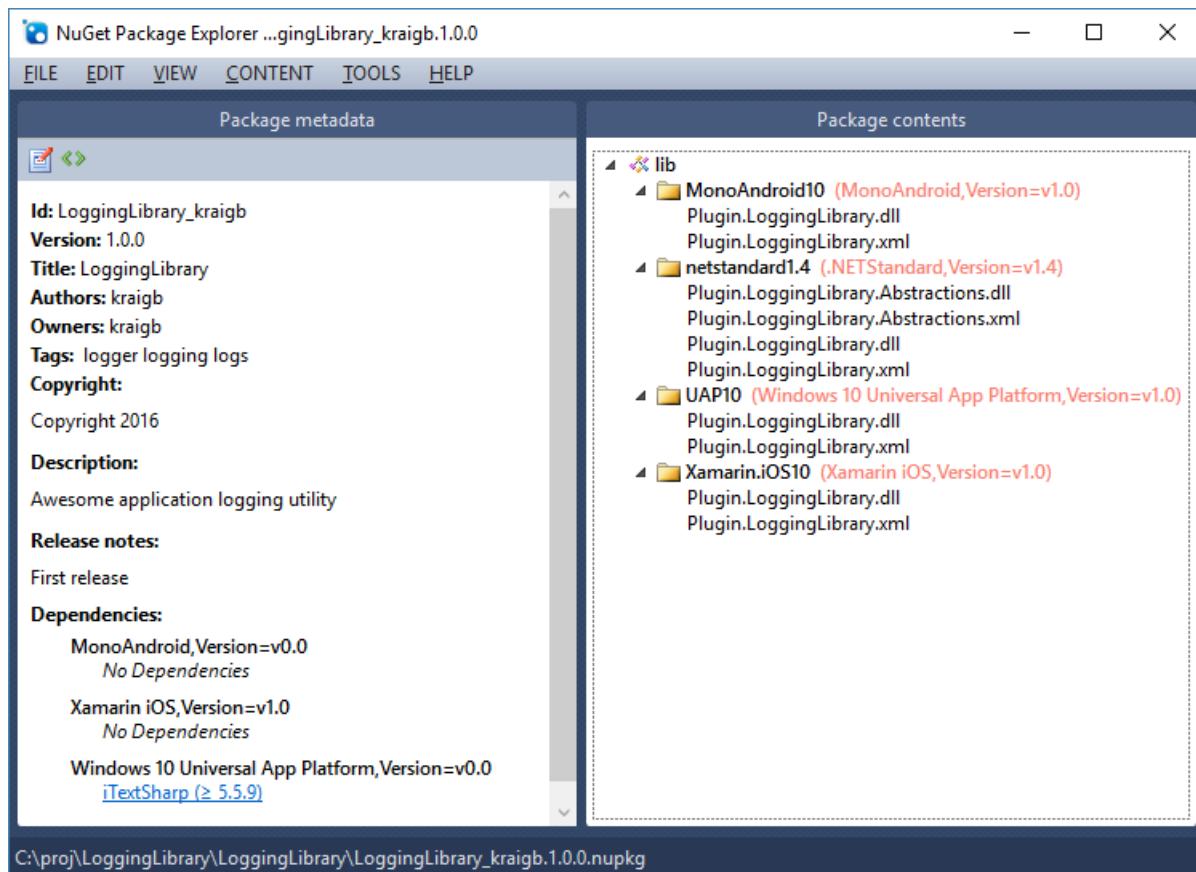
```

Package the component

With the completed `.nuspec` referencing all the files you need to include in the package, you're ready to run the `pack` command:

```
nuget pack LoggingLibrary.nuspec
```

This will generate `LoggingLibrary.YOUR_NAME.1.0.0.nupkg`. Opening this file in a tool like the [NuGet Package Explorer](#) and expanding all the nodes, you see the following contents:



C:\proj\LoggingLibrary\LoggingLibrary\LoggingLibrary_kraigb.1.0.0.nupkg

TIP

A `.nupkg` file is just a ZIP file with a different extension. You can also examine package contents, then, by changing `.nupkg` to `.zip`, but remember to restore the extension before uploading a package to [nuget.org](#).

To make your package available to other developers, follow the instructions on [Publish a package](#).

Related topics

- [Nuspec Reference](#)
- [Symbol packages](#)
- [Package versioning](#)
- [Supporting Multiple .NET Framework Versions](#)
- [Include MSBuild props and targets in a package](#)
- [Creating Localized Packages](#)

Create NuGet packages that contain COM interop assemblies

7/12/2019 • 2 minutes to read • [Edit Online](#)

Packages that contain COM interop assemblies must include an appropriate [targets file](#) so that the correct `EmbedInteropTypes` metadata is added to projects using the `PackageReference` format. By default, the `EmbedInteropTypes` metadata is always false for all assemblies when `PackageReference` is used, so the targets file adds this metadata explicitly. To avoid conflicts, the target name should be unique; ideally, use a combination of your package name and the assembly being embedded, replacing the `{InteropAssemblyName}` in the example below with that value. (Also see [NuGet.Samples.Interop](#) for an example.)

```
<Target Name="Embedding**AssemblyName**From**PackageId**" AfterTargets="ResolveReferences"
BeforeTargets="FindReferenceAssembliesForReferences">
  <ItemGroup>
    <ReferencePath Condition=" '%(FileName)' == '{InteropAssemblyName}' AND '%(ReferencePath.NuGetPackageId)'
    == '$(MSBuildThisFileName)' ">
      <EmbedInteropTypes>true</EmbedInteropTypes>
    </ReferencePath>
  </ItemGroup>
</Target>
```

Note that when using the `packages.config` management format, adding references to the assemblies from the packages causes NuGet and Visual Studio to check for COM interop assemblies and set the `EmbedInteropTypes` to true in the project file. In this case the targets are overridden.

Additionally, by default the [build assets do not flow transitively](#). Packages authored as described here work differently when they are pulled as a transitive dependency from a project to project reference. The package consumer can allow them to flow by modifying the `PrivateAssets` default value to not include `build`.

Signing NuGet Packages

9/26/2019 • 3 minutes to read • [Edit Online](#)

Signed packages allows for content integrity verification checks which provides protection against content tampering. The package signature also serves as the single source of truth about the actual origin of the package and bolsters package authenticity for the consumer. This guide assumes you have already [created a package](#).

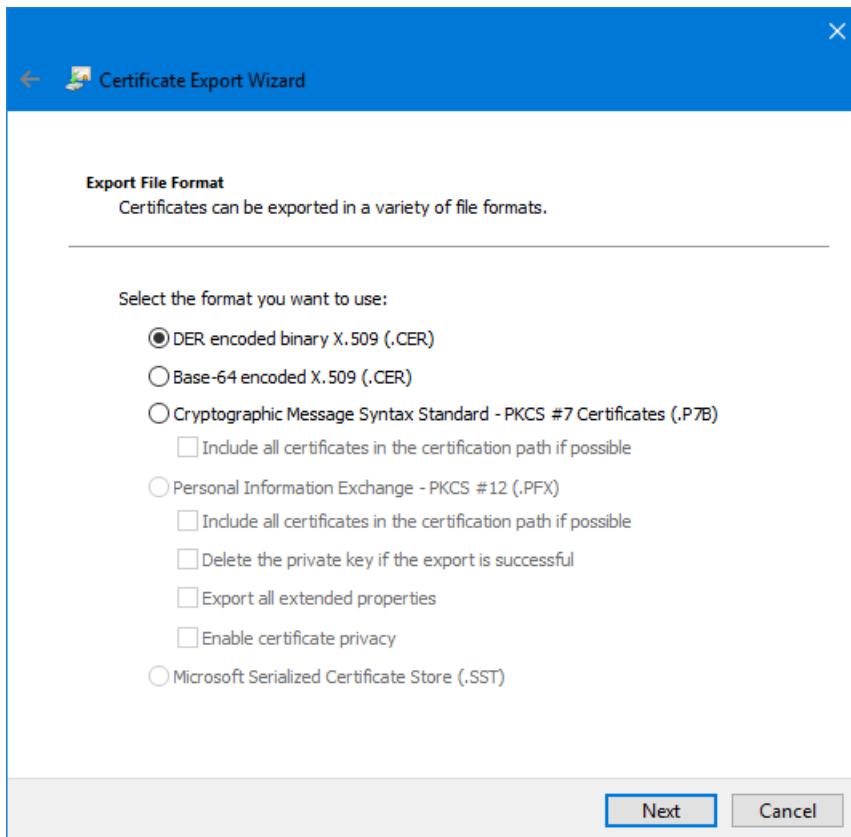
Get a code signing certificate

Valid certificates may be obtained from a public certificate authority such as [Symantec](#), [DigiCert](#), [Go Daddy](#), [Global Sign](#), [Comodo](#), [Certum](#), etc. The complete list of certification authorities trusted by Windows can be obtained from <http://aka.ms/trustcertpartners>.

You can use self-issued certificates for testing purposes. However, packages signed using self-issued certificates are not accepted by NuGet.org. Learn more about [creating a test certificate](#)

Export the certificate file

- You can export an existing certificate to a binary DER format by using the Certificate Export Wizard.



- You can also export the certificate using the [Export-Certificate PowerShell command](#).

Sign the package

NOTE

Requires nuget.exe 4.6.0 or later. dotnet.exe support is coming soon - [#7939](#)

Sign the package using [nuget sign](#):

```
nuget sign MyPackage.nupkg -CertificatePath <PathToTheCertificate> -Timestamper <TimestampServiceURL>
```

TIP

The certificate provider often also provides a timestamping server URL which you can use for the `Timestamper` optional argument shown above. Consult with your provider's documentation and/or support for that service URL.

- You can use a certificate available in the certificate store or use a certificate from a file. See CLI reference for [nuget sign](#).
- Signed packages should include a timestamp to make sure the signature remains valid when the signing certificate has expired. Else the sign operation will produce a [warning](#).
- You can see the signature details of a given package using [nuget verify](#).

Register the certificate on NuGet.org

To publish a signed package, you must first register the certificate with NuGet.org. You need the certificate as a `.cer` file in a binary DER format.

1. [Sign in](#) to NuGet.org.
2. Go to [Account settings](#) (or [Manage Organization](#) > [Edit Organization](#)) if you would like to register the certificate with an Organization account).
3. Expand the [Certificates](#) section and select [Register new](#).
4. Browse and select the certificate file that was exported earlier.

The screenshot shows the 'Certificates' section of the NuGet.org account settings. It displays two registered certificates with their respective fingerprints:

Fingerprint	Action
043b20ebf91097c531315ee70bad94a04093fb6a	Remove
fceb9b6a7cc59a113565cf6637da882854f51356	Remove

At the bottom of the section is a blue 'Register new' button.

Note

- One user can submit multiple certificates and the same certificate can be registered by multiple users.
- Once a user has a certificate registered, all future package submissions **must** be signed with one of the certificates. See [Manage signing requirements for your package on NuGet.org](#)
- Users can also remove a registered certificate from the account. Once a certificate is removed, new packages signed with that certificate will fail at submission. Existing packages aren't affected.

Publish the package

You are now ready to publish the package to NuGet.org. See [Publishing packages](#).

Create a test certificate

You can use self-issued certificates for testing purposes. To create a self-issued certificate, use the [New-](#)

SelfSignedCertificate PowerShell command.

```
New-SelfSignedCertificate -Subject "CN=NuGet Test Developer, OU=Use for testing purposes ONLY" `  
-FriendlyName "NuGetTestDeveloper" `  
-Type CodeSigning `  
-KeyUsage DigitalSignature `  
-KeyLength 2048 `  
-KeyAlgorithm RSA `  
-HashAlgorithm SHA256 `  
-Provider "Microsoft Enhanced RSA and AES Cryptographic Provider" `  
-CertStoreLocation "Cert:\CurrentUser\My"
```

This command creates a testing certificate available in the current user's personal certificate store. You can open the certificate store by running `certmgr.msc` to see the newly created certificate.

WARNING

NuGet.org does not accept packages signed with self-issued certificates.

Manage signing requirements for your package on NuGet.org

1. [Sign in](#) to NuGet.org.

2. Go to `Manage Packages`

✓ Published Packages 2 packages / 10 downloads

Package ID	Owners	Signing Owner	Downloads	Latest Version	Actions
 Contoso.Controls	 Contoso, JohnSmith	Any	10	1.0.0	  
 Contoso.Extensions.Data	 Contoso	Contoso (1 certificate)	0	0.1.2	  

- If you are the sole owner of a package, you are the required signer i.e. you can use any of the registered certificates to sign and publish your packages to NuGet.org.
- If a package has multiple owners, by default, "Any" owner's certificates can be used to sign the package. As a co-owner of the package, you can override "Any" with yourself or any other co-owner to be the required signer. If you make an owner who does not have any certificate registered, then unsigned packages will be allowed.
- Similarly, if the default "Any" option is selected for a package where one owner has a certificate registered and another owner does not have any certificate registered, then NuGet.org accepts either a signed package with a signature registered by one of its owners or an unsigned package (because one of the owners does not have any certificate registered).

Related articles

- [Manage package trust boundaries](#)
- [Signed Packages Reference](#)

Signed packages

7/18/2019 • 2 minutes to read • [Edit Online](#)

NuGet 4.6.0+ and Visual Studio 2017 version 15.6 and later

NuGet packages can include a digital signature that provides protection against tampered content. This signature is produced from an X.509 certificate that also adds authenticity proofs to the actual origin of the package.

Signed packages provide the strongest end-to-end validation. There are two different types of NuGet signatures:

- **Author Signature.** An author signature guarantees that the package has not been modified since the author signed the package, no matter from which repository or what transport method the package is delivered. Additionally, author-signed packages provide an extra authentication mechanism to the nuget.org publishing pipeline because the signing certificate must be registered ahead of time. For more information, see [Register certificates](#).
- **Repository Signature.** Repository signatures provide an integrity guarantee for **all** packages in a repository whether they are author signed or not, even if those packages are obtained from a different location than the original repository where they were signed.

For details on creating an author signed package, see [Signing Packages](#) and the [nuget sign command](#).

IMPORTANT

Package signing is currently supported only when using nuget.exe on Windows. Verification of signed packages is currently supported only when using nuget.exe or Visual Studio on Windows.

Certificate requirements

Package signing requires a code signing certificate, which is a special type of certificate that is valid for the `id-kp-codeSigning` purpose [\[RFC 5280 section 4.2.1.12\]](#). Additionally, the certificate must have an RSA public key length of 2048 bits or higher.

Timestamp requirements

Signed packages should include an RFC 3161 timestamp to ensure signature validity beyond the package signing certificate's validity period. The certificate used to sign the timestamp must be valid for the `id-kp-timeStamping` purpose [\[RFC 5280 section 4.2.1.12\]](#). Additionally, the certificate must have an RSA public key length of 2048 bits or higher.

Additional technical details can be found in the [package signature technical specs](#) (GitHub).

Signature requirements on NuGet.org

nuget.org has additional requirements for accepting a signed package:

- The primary signature must be an author signature.
- The primary signature must have a single valid timestamp.
- The X.509 certificates for both the author signature and its timestamp signature:
 - Must have an RSA public key 2048 bits or greater.
 - Must be within its validity period per current UTC time at time of package validation on nuget.org.

- Must chain to a trusted root authority that is trusted by default on Windows. Packages signed with self-issued certificates are rejected.
- Must be valid for its purpose:
 - The author signing certificate must be valid for code signing.
 - The timestamp certificate must be valid for timestamping.
- Must not be revoked at signing time. (This may not be knowable at submission time, so nuget.org periodically rechecks revocation status).

Related articles

- [Signing NuGet Packages](#)
- [Manage package trust boundaries](#)

Publishing packages

11/5/2019 • 6 minutes to read • [Edit Online](#)

Once you have created a package and have your `.nupkg` file in hand, it's a simple process to make it available to other developers, either publicly or privately:

- Public packages are made available to all developers globally through [nuget.org](#) as described in this article (requires NuGet 4.1.0+).
- Private packages are available to only a team or organization, by hosting them either a file share, a private NuGet server, [Azure Artifacts](#), or a third-party repository such as myget, ProGet, Nexus Repository, and Artifactory. For additional details, see [Hosting Packages Overview](#).

This article covers publishing to nuget.org; for publishing to Azure Artifacts, see [Package Management](#).

Publish to nuget.org

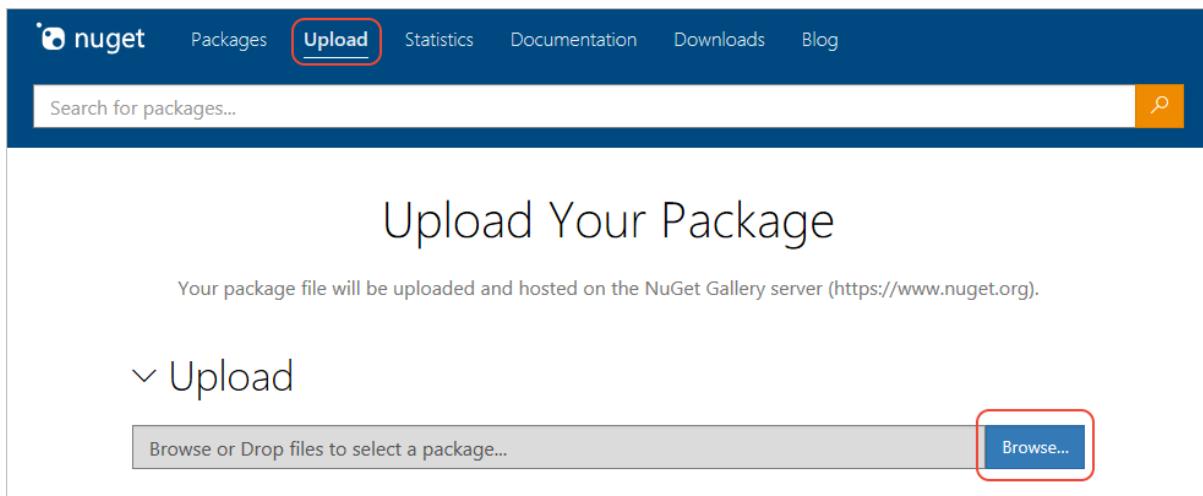
For nuget.org, you must sign in with a Microsoft account, with which you'll be asked to register the account with nuget.org. You can also sign in with a nuget.org account created using older versions of the portal.



Next, you can either upload the package through the nuget.org web portal, push to nuget.org from the command line (requires `nuget.exe` 4.1.0+), or publish as part of a CI/CD process through Azure DevOps Services, as described in the following sections.

Web portal: use the Upload Package tab on nuget.org

1. Select **Upload** on the top menu of nuget.org and browse to the package location.



2. nuget.org tells you if the package name is available. If it isn't, change the package identifier in your project, rebuild, and try the upload again.
3. If the package name is available, nuget.org opens a **Verify** section in which you can review the metadata from the package manifest. To change any of the metadata, edit your project (project file or `.nuspec` file), rebuild, recreate the package, and upload again.
4. Under **Import Documentation** you can paste Markdown, point to your docs with a URL, or upload a documentation file.

5. When all the information is ready, select the **Submit** button

Command line

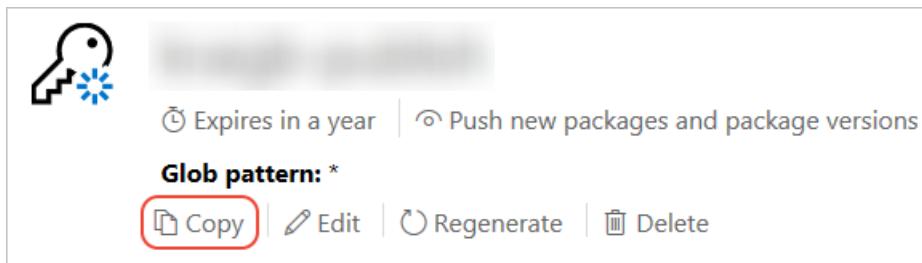
To push packages to nuget.org you must use [nuget.exe v4.1.0 or above](#), which implements the required [NuGet protocols](#). You also need an API key, which is created on nuget.org.

Create API keys

1. [Sign into your nuget.org account](#) or create an account if you don't have one already.

For more information on creating your account, see [Individual accounts](#).

2. Select your user name (on the upper right), then select **API Keys**.
3. Select **Create**, provide a name for your key, select **Select Scopes > Push**. Enter * for **Glob pattern**, then select **Create**. (See below for more about scopes.)
4. Once the key is created, select **Copy** to retrieve the access key you need in the CLI:



5. **Important:** Save your key in a secure location because you cannot copy the key again later on. If you return to the API key page, you need to regenerate the key to copy it. You can also remove the API key if you no longer want to push packages via the CLI.

Scoping allows you to create separate API keys for different purposes. Each key has its expiration timeframe and can be scoped to specific packages (or glob patterns). Each key is also scoped to specific operations: push of new packages and updates, push of updates only, or delisting. Through scoping, you can create API keys for different people who manage packages for your organization such that they have only the permissions they need. For more information, see [scoped API keys](#).

Publish with dotnet nuget push

1. Change to the folder containing the `.nupkg` file.
2. Run the following command, specifying your package name (unique package ID) and replacing the key value with your API key:

```
dotnet nuget push AppLogger.1.0.0.nupkg -k qz2jga8pl3dvn2akksyquwcs9ygggg4exypy3bhxy6w6x6 -s https://api.nuget.org/v3/index.json
```

3. dotnet displays the results of the publishing process:

```
info : Pushing AppLogger.1.0.0.nupkg to 'https://www.nuget.org/api/v2/package'...
info : PUT https://www.nuget.org/api/v2/package/
info : Created https://www.nuget.org/api/v2/package/ 12620ms
info : Your package was pushed.
```

See [dotnet nuget push](#).

Publish with nuget push

1. At a command prompt, run the following command, replacing `<your_API_key>` with the key obtained from nuget.org:

```
nuget setApiKey <your_API_key>
```

This command stores your API key in your NuGet configuration so that you don't need to repeat this step again on the same computer.

2. Push your package to NuGet Gallery using the following command:

```
nuget push YourPackage.nupkg -Source https://api.nuget.org/v3/index.json
```

Publish signed packages

To submit signed packages, you must first [register the certificate](#) used for signing the packages.

WARNING

nuget.org rejects packages that don't satisfy the [signed package requirements](#).

Package validation and indexing

Packages pushed to nuget.org undergo several validations, such as virus checks. (All packages on nuget.org are periodically scanned.)

When the package has passed all validation checks, it might take a while for it to be indexed and appear in search results. Once indexing is complete, you receive an email confirming that the package was successfully published. If the package fails a validation check, the package details page will update to display the associated error and you also receive an email notifying you about it.

Package validation and indexing usually takes under 15 minutes. If the package publishing is taking longer than expected, visit [status.nuget.org](#) to check if nuget.org is experiencing any interruptions. If all systems are operational and the package hasn't been successfully published within an hour, please login to nuget.org and contact us using the Contact Support link on the package page.

To see the status of a package, select **Manage packages** under your account name on nuget.org. You receive a confirmation email when validation is complete.

Note that it might take a while for your package to be indexed and appear in search results where others can find it, during which time you see the following message on your package page:

⚠ This package has not been published yet. It will appear in search results and will be available for install/restore after both validation and indexing are complete. Package validation and indexing may take up to an hour. [Read more](#).

Azure DevOps Services (CI/CD)

If you push packages to nuget.org using Azure DevOps Services as part of your Continuous Integration/Deployment process, you must use `nuget.exe` 4.1 or above in the NuGet tasks. Details can be found on [Using the latest NuGet in your build](#) (Microsoft DevOps blog).

Managing package owners on nuget.org

Although each NuGet package's `.nuspec` file defines the package's authors, the nuget.org gallery does not use that metadata to define ownership. Instead, nuget.org assigns initial ownership to the person who publishes the package. This is either the logged-in user who uploaded the package through the nuget.org UI, or the users whose API key was used with `nuget SetApiKey` or `nuget push`.

All package owners have full permissions for the package, including adding and removing other owners, and

publishing updates.

To change ownership of a package, do the following:

1. Sign in to nuget.org with the account that is the current owner of the package.
2. Select your account name, select **Manage packages**, and expand **Published Packages**.
3. Select on the package you want to manage, then on the right side select **Manage owners**.

From here you have several options:

1. Remove any owner listed under **Current Owners**.
2. Add an owner under **Add Owner** by entering their user name, a message, and selecting **Add**. This action sends an email to that new co-owner with a confirmation link. Once confirmed, that person has full permissions to add and remove owners. (Until confirmed, the **Current Owners** section indicates pending approval for that person.)
3. To transfer ownership (as when ownership changes or a package was published under the wrong account), add the new owner, and once they've confirmed ownership they can remove you from the list.

To assign ownership to a company or group, create a nuget.org account using an email alias that is forwarded to the appropriate team members. For example, various Microsoft ASP.NET packages are co-owned by the [microsoft](#) and [aspnet](#) accounts, which simply such aliases.

Recovering package ownership

Occasionally, a package may not have an active owner. For example, the original owner may have left the company that produces the package, nuget.org credentials are lost, or earlier bugs in the gallery left a package ownerless.

If you are the rightful owner of a package and need to regain ownership, use the [contact form](#) on nuget.org to explain your situation to the NuGet team. We then follow a process to verify your ownership of the package, including trying to locate the existing owner through the package's Project URL, Twitter, email, or other means. But if all else fails, we can send you a new invite to become an owner.

Scoped API keys

6/28/2019 • 6 minutes to read • [Edit Online](#)

To make NuGet a more secure environment for package distribution, you can take control of the API keys by adding scopes.

The ability to provide scope to your API keys give you better control on your APIs. You can:

- Create multiple scoped API keys that can be used for different packages with varying expiration timeframes.
- Obtain API keys securely.
- Edit existing API keys to change package applicability.
- Refresh or delete existing API keys without hampering operations using other keys.

Why do we support scoped API keys?

We support scopes for API keys to allow you to have more fine-grained permissions. Previously, NuGet offered a single API key for an account, and that approach had several drawbacks:

- **One API key to control all packages.** With a single API key that is used to manage all packages, it is difficult to securely share the key when multiple developers are involved with different packages, and when they share a publisher account.
- **All permissions or none.** Anyone with access to the API key has all permissions (publish, push and un-list) on the packages. This is often not desirable in environment with multiple teams.
- **Single point of failure.** A single API key also means a single point of failure. If the key is compromised, all packages associated with the account could potentially be compromised. Refreshing the API key is the only way to plug the leak and avoid an interruption to your CI/CD workflow. In addition, there may be cases when you want to revoke access to the API key for an individual (for example, when an employee leaves the organization). There isn't a clean way to handle this today.

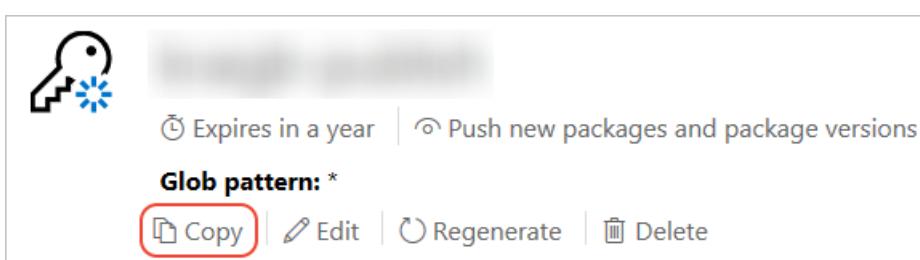
With scoped API keys, we try to address these problems while making sure that none of the existing workflows break.

Acquire an API key

1. [Sign into your nuget.org account](#) or create an account if you don't have one already.

For more information on creating your account, see [Individual accounts](#).

2. Select your user name (on the upper right), then select **API Keys**.
3. Select **Create**, provide a name for your key, select **Select Scopes > Push**. Enter * for **Glob pattern**, then select **Create**. (See below for more about scopes.)
4. Once the key is created, select **Copy** to retrieve the access key you need in the CLI:



5. **Important:** Save your key in a secure location because you cannot copy the key again later on. If you return to the API key page, you need to regenerate the key to copy it. You can also remove the API key if you no longer want to push packages via the CLI.

Scoping allows you to create separate API keys for different purposes. Each key has its expiration timeframe and can be scoped to specific packages (or glob patterns). Each key is also scoped to specific operations: push of new packages and updates, push of updates only, or delisting. Through scoping, you can create API keys for different people who manage packages for your organization such that they have only the permissions they need. For more information, see [scoped API keys](#).

Create scoped API keys

You can create multiple API keys based on your requirements. An API key can apply to one or more packages, have varying scopes that grant specific privileges, and have an expiration date associated with it.

In the following example, you have an API key named `Contoso service CI` that can be used to push packages for specific `Contoso.Service` packages, and is valid for 365 days. This is a typical scenario where different teams within the same organization work on different packages, and the members of the team are provided the key that grants them privileges only for the package they are working on. The expiration serves as a mechanism to prevent stale or forgotten keys.

Manage API Keys

API Keys

Key name: Contoso service CI (REQUIRED)

Expires in: 365 days

Select scopes:

- Push
- Push new packages and package versions
- Push only new package versions
- Unlist package

Select packages:

To select which packages to associate with a key use a glob pattern, select individual packages, or both. Example glob patterns.

Glob pattern: Example: The wildcard '*' will select all packages.

Available packages:

- Contoso.Service.API
- Contoso.Service.Extensions
- Contoso.Service.Framework
- Contoso.Service.Integration
- Contoso.UI.Extensions
- Contoso.UI.Framework
- Fabrikam.Service.API
- Fabrikam.Service.Extensions
- Fabrikam.UI.Extensions
- Fabrikam.UI.Framework

2/10 selected

Add key

Use glob patterns

If you are working on multiple packages and have a large list of packages to manage, you can choose to use globbing patterns to select multiple packages together. For example, if you wish to grant specific scopes to a key for all packages whose ID starts with `Fabrikam.Service`, you could do this by specifying `fabrikam.service.*` in the **Glob pattern** text box.

Select packages

To select which packages to associate with a key use a glob pattern, select individual packages, or both. [Example glob patterns](#).

Glob pattern

Available packages

- Contoso.Service.API
- Contoso.Service.Extensions
- Contoso.Service.Framework
- Contoso.Service.Integration
- Contoso.UI.Extensions
- Contoso.UI.Framework
- Fabrikam.Service.API
- Fabrikam.Service.Extensions
- Fabrikam.UI.Extensions
- Fabrikam.UI.Framework

2/10 selected

Add key

A glob pattern allows you to replace any sequence of characters with `*`.

Example glob patterns

Pattern	Result
<code>*</code>	Select all packages.
<code>Alpha.*</code>	Select any packages that begin with <code>Alpha</code> .

Using glob patterns to determine API key permissions also applies to new packages matching the glob pattern. For example, if you try to push a new package named `Fabrikam.Service.Framework`, you can do that with the key created previously, since the package matches the glob pattern `fabrikam.service.*`.

Obtain API keys securely

For security, a newly created key is never shown on the screen and is only available using the **Copy** button. Similarly, the key is not accessible after the page is refreshed.

Manage API Keys

API Keys [New API key](#)

Keys

! New API key has been created. Make sure to copy your new API key now using the  copy key button. You won't be able to do so again.

 Contoso service CI Expires: in a year Scopes: Push new packages and package versions Packages: Contoso.Service.API, Contoso.Service.Extensions	  
 Full access API key Expires: in a year Scopes: All Packages: All	  

Edit existing API keys

You may also want to update the key permissions and scopes without changing the key itself. If you have a key with specific scope(s) for a single package, you can choose to apply the same scope(s) on one or many other packages.

Keys

 Contoso service CI
Expires: in a year
Scopes: Push new packages and package versions
Packages: Contoso.Service.API, Contoso.Service.Extensions



Refresh or delete existing API keys

The account owner can choose to refresh the key, in which case the permission (on packages), scope, and expiry remain the same, but a new key is issued making the old key unusable. This is helpful in managing stale keys or where there is any potential for an API key leakage.

Keys

 Contoso service CI
Expires: in a year
Scopes: Push new packages and package versions
Packages: Contoso.Service.API, Contoso.Service.Extensions



You may also choose to delete these keys if they are not needed anymore. Deleting a key removes the key and makes it unusable.

FAQs

What happens to my old (legacy) API key?

Your old API key (legacy) continues to work and can work as long as you want it to work. However, these keys will be retired if they have not been used for more than 365 days to push a package. For more details, see the blog post [Changes to expiring API keys](#). You can no longer refresh this key. You need to delete the legacy key and create a new scoped key instead.

NOTE

This key has all permissions on all the packages and it never expires. You should consider deleting this key and creating new keys with scoped permissions and definite expiry.

How many API keys can I create?

There is no limit on the number of API keys you can create. However, we advise you to keep it to a manageable count so that you do not end up having many stale keys with no knowledge of where and who is using them.

Can I delete my legacy API key or discontinue using now?

Yes. You can--and you probably should--delete your legacy API key.

Can I get back my API key that I deleted by mistake?

No. Once deleted, you can only create new keys. There is no recovery possible for accidentally deleted keys.

Does the old API key continue to work upon API key refresh?

No. Once you refresh a key, a new key gets generated that has the same scope, permission, and expiry as the old one. The old key ceases to exist.

Can I give more permissions to an existing API key?

You cannot modify the scope, but you can edit the package list it is applicable to.

How do I know if any of my keys expired or are getting expired?

If any key expires, we will let you know through a warning message at the top of the page. We also send a warning e-mail to the account holder ten days before the expiration of the key so that you can act on it well in advance.

Hosting your own NuGet feeds

11/5/2019 • 2 minutes to read • [Edit Online](#)

Instead of making packages publicly available, you might want to release packages to only a limited audience, such as your organization or workgroup. In addition, some companies may want to restrict which third-party libraries their developers may use, and thus direct those developers to draw from a limited package source rather than nuget.org.

For all such purposes, NuGet supports setting up private package sources in the following ways:

- Local feed: Packages are simply placed on a suitable network file share, ideally using `nuget init` and `nuget add` to create a hierarchical folder structure (NuGet 3.3+). For details, see [Local Feeds](#).
- NuGet.Server: Packages are made available through a local HTTP server. For details, see [NuGet.Server](#).
- NuGet Gallery: Packages are hosted on an Internet server using the [NuGet Gallery Project](#) (github.com). NuGet Gallery provides user management and features such as an extensive web UI that allows searching and exploring packages from within the browser, similar to nuget.org.

There are also several other NuGet hosting products such as [Azure Artifacts](#) and [GitHub package registry](#) that support remote private feeds. Below is a list of such products:

- [Artifactory](#) from JFrog.
- [Azure Artifacts](#), which is also available on Team Foundation Server 2017 and later.
- [BaGet](#), an open-source implementation of NuGet V3 server built on ASP.NET Core
- [Cloudsmith](#), a fully managed package management SaaS
- [GitHub package registry](#)
- [LiGet](#), an open-source implementation of NuGet V2 server that runs on kestrel in docker
- [MyGet](#)
- [Nexus](#) from Sonatype.
- [NuGet Server \(Open Source\)](#), an open-source implementation similar to Inedo's NuGet Server
- [NuGet Server](#), a community project from Inedo
- [ProGet](#) from Inedo
- [Sleet](#), an open-source NuGet V3 static feed generator
- [TeamCity](#) from JetBrains.

Regardless of how packages are hosted, you access them by adding them to the list of available sources in `NuGet.Config`. This can be done in Visual Studio as described in [Package Sources](#), or from the command line using `nuget sources`. The path to a source can be a local folder pathname, a network name, or a URL.

NuGet.Server

11/5/2019 • 4 minutes to read • [Edit Online](#)

NuGet.Server is a package provided by the .NET Foundation that creates an ASP.NET application that can host a package feed on any server that runs IIS. Simply said, NuGet.Server makes a folder on the server available through HTTP(S) (specifically OData). It's easy to set up and is best for simple scenarios.

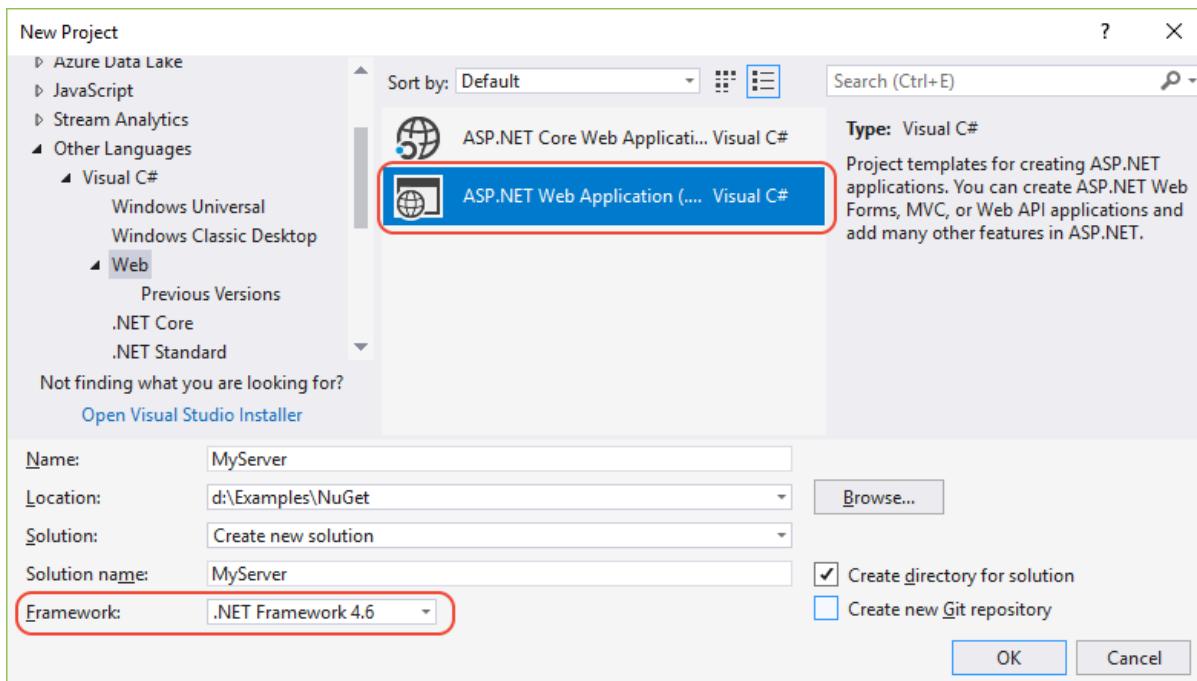
1. Create an empty ASP.NET Web application in Visual Studio and add the NuGet.Server package to it.
2. Configure the `Packages` folder in the application and add packages.
3. Deploy the application to a suitable server.

The following sections walk through this process in detail, using C#.

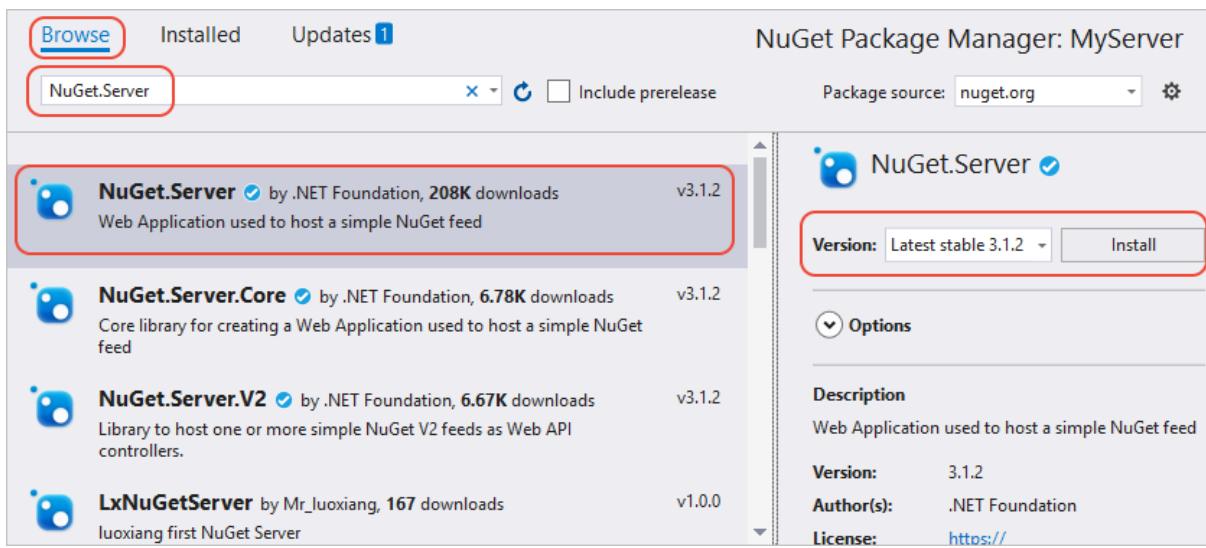
If you have further questions about NuGet.Server, create an issue on <https://github.com/nuget/NuGetGallery/issues>.

Create and deploy an ASP.NET Web application with NuGet.Server

1. In Visual Studio, select **File > New > Project**, search for "ASP.NET", select the **ASP.NET Web Application (.NET Framework)** template for C#, and set **Framework** to ".NET Framework 4.6":



2. Give the application a suitable name *other* than NuGet.Server, select **OK**, and in the next dialog select the **Empty** template, then select **OK**.
3. Right-click the project, select **Manage NuGet Packages**.
4. In the Package Manager UI, select the **Browse** tab, then search and install the latest version of the NuGet.Server package if you're targeting .NET Framework 4.6. (You can also install it from the Package Manager Console with `Install-Package NuGet.Server`.) Accept the license terms if prompted.

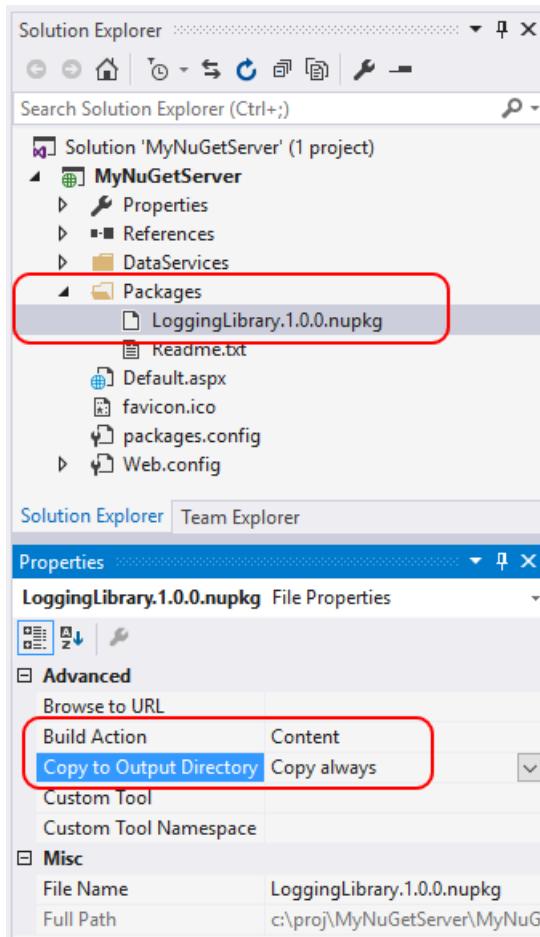


5. Installing NuGet.Server converts the empty Web application into a package source. It installs a variety of other packages, creates a `Packages` folder in the application, and modifies `web.config` to include additional settings (see the comments in that file for details).

IMPORTANT

Carefully inspect `web.config` after the NuGet.Server package has completed its modifications to that file. NuGet.Server may not overwrite existing elements but instead create duplicate elements. Those duplicates will cause an "Internal Server Error" when you later try to run the project. For example, if your `web.config` contains `<compilation debug="true" targetFramework="4.5.2" />` before installing NuGet.Server, the package doesn't overwrite it but inserts a second `<compilation debug="true" targetFramework="4.6" />`. In that case, delete the element with the older framework version.

6. To make packages available in the feed when you publish the application to a server, add each `.nupkg` files to the `Packages` folder in Visual Studio, then set each one's **Build Action** to **Content** and **Copy to Output Directory to Copy always**:



7. Run the site locally in Visual Studio (using **Debug > Start Without Debugging** or **Ctrl+F5**). The home page provides the package feed URLs as shown below. If you see errors, carefully inspect your `web.config` for duplicate elements are noted earlier with step 5.

You are running NuGet.Server v2.11.3.0

[Click here to view your packages.](#)

Repository URLs

In the package manager settings, add the following URL to the list of Package Sources:

`http://localhost:53673/nuget`

To enable pushing packages to this feed using the [NuGet command line tool](#) (nuget.exe), set the api key appSetting in `web.config`.

`nuget.exe push {package file} {apikey} -Source http://localhost:53673/api/v2/package`

Adding packages

To add packages to the feed put package files (.nupkg files) in the folder `c:\proj\WebApplication2\WebApplication2\ Packages`

[Click here to clear the package cache.](#)

8. Click on **here** in the area outlined above to see the OData feed of packages.
9. The first time you run the application, NuGet.Server restructures the `Packages` folder to contain a folder for each package. This matches the [local storage layout](#) introduced with NuGet 3.3 to improve performance. When adding more packages, continue to follow this structure.
10. Once you've tested your local deployment, deploy the application to any other internal or external site as needed.
11. Once deployed to `http://<domain>`, the URL that you use for the package source will be

`http://<domain>/nuget`.

Configuring the Packages folder

With `NuGet.Server` 1.5 and later, you can more specifically configure the package folder using the `appSetting/packagesPath` value in `web.config`:

```
<appSettings>
  <!-- Set the value here to specify your custom packages folder. -->
  <add key="packagesPath" value="C:\MyPackages" />
</appSettings>
```

`packagesPath` can be an absolute or virtual path.

When `packagesPath` is omitted or left blank, the packages folder is the default `~/Packages`.

Adding packages to the feed externally

Once a NuGet.Server site is running, you can add packages using `nuget push` provided that you set an API key value in `web.config`.

After installing the NuGet.Server package, `web.config` contains an empty `appSetting/apiKey` value:

```
<appSettings>
  <add key="apiKey" value="" />
</appSettings>
```

When `apiKey` is omitted or blank, pushing packages to the feed is disabled.

To enable this capability, set the `apiKey` to a value (ideally a strong password) and add a key called `appSettings/requireApiKey` with the value of `true`:

```
<appSettings>
  <!-- Sets whether an API Key is required to push/delete packages -->
  <add key="requireApiKey" value="true" />

  <!-- Set a shared password (for all users) to push/delete packages -->
  <add key="apiKey" value="" />
</appSettings>
```

If your server is already secured or you do not otherwise require an API key (for example, when using a private server on a local team network), you can set `requireApiKey` to `false`. All users with access to the server can then push packages.

Removing packages from the feed

With NuGet.Server, the `nuget delete` command removes a package from the repository provided that you include the API key with the comment.

If you want to change the behavior to delist the package instead (leaving it available for package restore), change the `enableDelisting` key in `web.config` to true.

NuGet.Server support

For additional help using NuGet.Server, create an issue on <https://github.com/nuget/NuGetGallery/issues>.

Local feeds

7/18/2019 • 2 minutes to read • [Edit Online](#)

Local NuGet package feeds are simply hierarchical folder structures on your local network (or even just your own computer) in which you place packages. These feeds can then be used as package sources with all other NuGet operations using the CLI, the Package Manager UI, and the Package Manager Console.

To enable the source, add its pathname (such as `\myserver\packages`) to the list of sources using the [Package Manager UI](#) or the `nuget sources` command.

NOTE

Hierarchical folder structures are supported in NuGet 3.3+. Older versions of NuGet use only a single folder containing packages, with which performance is much lower than the hierarchical structure.

Initializing and maintaining hierarchical folders

The hierarchical versioned folder tree has the following general structure:

```
\myserver\packages
  └<packageID>
    └<version>
      └<packageID>.<version>.nupkg
      └<other files>
```

NuGet creates this structure automatically when you use the `nuget add` command to copy a package to the feed:

```
nuget add new_package.1.0.0.nupkg -source \myserver\packages
```

The `nuget add` command works with one package at a time, which can be inconvenient when setting up a feed with multiple packages.

In such cases, use the `nuget init` command to copy all packages in a folder to the feed as if you ran `nuget add` on each one individually. For example, the following command copies all packages from `c:\packages` to a hierarchical tree on `\myserver\packages`:

```
nuget init c:\packages \myserver\packages
```

As with the `add` command, `init` creates a folder for each package identifier, each of which contains a version number folder, within which is the appropriate package.

What happens when a NuGet package is installed?

8/15/2019 • 2 minutes to read • [Edit Online](#)

Simply said, the different NuGet tools typically create a reference to a package in the project file or `packages.config`, then perform a package restore, which effectively installs the package. The exception is `nuget install`, which only expands the package into a `packages` folder and does not modify any other files.

The general process is as follows:

1. (All tools except `nuget.exe`) Record the package identifier and version into the project file or `packages.config`.

If the installation tool is Visual Studio or the dotnet CLI, the tool first attempts to install the package. If it's incompatible, the package is not added to the project file or `packages.config`.

2. Acquire the package:

- Check if the package (by exact identifier and version number) is already installed in the *global-packages* folder as described on [Managing the global packages and cache folders](#).
- If the package is not in the *global-packages* folder, attempt to retrieve it from the sources listed in the [configuration files](#). For online sources, attempt first to retrieve the package from the HTTP cache unless `-NoCache` is specified with `nuget.exe` commands or `--no-cache` is specified with `dotnet restore`. (Visual Studio and `dotnet add package` always use the cache.) If a package is used from the cache, "CACHE" appears in the output. The cache has an expiration time of 30 minutes.
- If the package is not in the HTTP cache, attempt to download it from the sources listed in the configuration. If a package is downloaded, "GET" and "OK" appear in the output. NuGet logs http traffic on normal verbosity.
- If the package cannot be successfully acquired from any sources, installation fails at this point with an error such as [NU1103](#). Note that errors from `nuget.exe` commands show only the last source checked, but implies that the package wasn't available from any source.

When acquiring the package, the order of sources in the NuGet configuration may apply:

- NuGet checks sources local folder and network shares before checking HTTP sources.
3. Save a copy of the package and other information in the *http-cache* folder as described on [Managing the global packages and cache folders](#).
 4. If downloaded, install the package into the per-user *global-packages* folder. NuGet creates a subfolder for each package identifier, then creates subfolders for each installed version of the package.
 5. NuGet installs package dependencies as required. This process might update package versions in the process, as described in [Dependency Resolution](#).
 6. Update other project files and folders:
 - For projects using `PackageReference`, update the package dependency graph stored in `obj/project.assets.json`. Package contents themselves are not copied into any project folder.
 - Update `app.config` and/or `web.config` if the package uses [source and config file transformations](#).
 7. (Visual Studio only) Display the package's `readme` file, if available, in a Visual Studio window.

Enjoy your productive coding with NuGet packages!

Package versioning

11/5/2019 • 7 minutes to read • [Edit Online](#)

A specific package is always referred to using its package identifier and an exact version number. For example, [Entity Framework](#) on nuget.org has several dozen specific packages available, ranging from version 4.1.10311 to version 6.1.3 (the latest stable release) and a variety of pre-release versions like 6.2.0-beta1.

When creating a package, you assign a specific version number with an optional pre-release text suffix. When consuming packages, on the other hand, you can specify either an exact version number or a range of acceptable versions.

In this topic:

- [Version basics](#) including pre-release suffixes.
- [Version ranges and wildcards](#)
- [Normalized version numbers](#)

Version basics

A specific version number is in the form *Major.Minor.Patch[-Suffix]*, where the components have the following meanings:

- *Major*: Breaking changes
- *Minor*: New features, but backwards compatible
- *Patch*: Backwards compatible bug fixes only
- *-Suffix* (optional): a hyphen followed by a string denoting a pre-release version (following the [Semantic Versioning or SemVer 1.0 convention](#)).

Examples:

```
1.0.1
6.11.1231
4.3.1-rc
2.2.44-beta1
```

IMPORTANT

nuget.org rejects any package upload that lacks an exact version number. The version must be specified in the `.nuspec` or project file used to create the package.

Pre-release versions

Technically speaking, package creators can use any string as a suffix to denote a pre-release version, as NuGet treats any such version as pre-release and makes no other interpretation. That is, NuGet displays the full version string in whatever UI is involved, leaving any interpretation of the suffix's meaning to the consumer.

That said, package developers generally follow recognized naming conventions:

- `-alpha` : Alpha release, typically used for work-in-progress and experimentation.
- `-beta` : Beta release, typically one that is feature complete for the next planned release, but may contain known bugs.

- `-rc` : Release candidate, typically a release that's potentially final (stable) unless significant bugs emerge.

NOTE

NuGet 4.3.0+ supports [SemVer 2.0.0](#), which supports pre-release numbers with dot notation, as in `1.0.1-build.23`. Dot notation is not supported with NuGet versions before 4.3.0. You can use a form like `1.0.1-build23`.

When resolving package references and multiple package versions differ only by suffix, NuGet chooses a version without a suffix first, then applies precedence to pre-release versions in reverse alphabetical order. For example, the following versions would be chosen in the exact order shown:

```
1.0.1
1.0.1-zzz
1.0.1-rc
1.0.1-open
1.0.1-beta
1.0.1-alpha2
1.0.1-alpha
1.0.1-aaa
```

Semantic Versioning 2.0.0

With NuGet 4.3.0+ and Visual Studio 2017 version 15.3+, NuGet supports [Semantic Versioning 2.0.0](#).

Certain semantics of SemVer v2.0.0 are not supported in older clients. NuGet considers a package version to be SemVer v2.0.0 specific if either of the following statements is true:

- The pre-release label is dot-separated, for example, `1.0.0-alpha.1`
- The version has build-metadata, for example, `1.0.0+githash`

For nuget.org, a package is defined as a SemVer v2.0.0 package if either of the following statements is true:

- The package's own version is SemVer v2.0.0 compliant but not SemVer v1.0.0 compliant, as defined above.
- Any of the package's dependency version ranges has a minimum or maximum version that is SemVer v2.0.0 compliant but not SemVer v1.0.0 compliant, defined above; for example, `[1.0.0-alpha.1,)`.

If you upload a SemVer v2.0.0-specific package to nuget.org, the package is invisible to older clients and available to only the following NuGet clients:

- NuGet 4.3.0+
- Visual Studio 2017 version 15.3+
- Visual Studio 2015 with [NuGet VSIX v3.6.0](#)
- dotnet
 - `dotnetcore.exe` (.NET SDK 2.0.0+)

Third-party clients:

- JetBrains Rider
- Paket version 5.0+

Version ranges and wildcards

When referring to package dependencies, NuGet supports using interval notation for specifying version ranges, summarized as follows:

NOTATION	APPLIED RULE	DESCRIPTION
1.0	$x \geq 1.0$	Minimum version, inclusive
(1.0,)	$x > 1.0$	Minimum version, exclusive
[1.0]	$x == 1.0$	Exact version match
(,1.0]	$x \leq 1.0$	Maximum version, inclusive
(1.0,)	$x < 1.0$	Maximum version, exclusive
[1.0,2.0]	$1.0 \leq x \leq 2.0$	Exact range, inclusive
(1.0,2.0)	$1.0 < x < 2.0$	Exact range, exclusive
[1.0,2.0)	$1.0 \leq x < 2.0$	Mixed inclusive minimum and exclusive maximum version
(1.0)	invalid	invalid

When using the `PackageReference` format, NuGet also supports using a wildcard notation, `*`, for Major, Minor, Patch, and pre-release suffix parts of the number. Wildcards are not supported with the `packages.config` format.

NOTE

Version ranges in `PackageReference` include pre-release versions. By design, floating versions do not resolve prerelease versions unless opted into. For the status of the related feature request, see [issue 6434](#).

Examples

Always specify a version or version range for package dependencies in project files, `packages.config` files, and `.nuspec` files. Without a version or version range, NuGet 2.8.x and earlier chooses the latest available package version when resolving a dependency, whereas NuGet 3.x and later chooses the lowest package version. Specifying a version or version range avoids this uncertainty.

References in project files (`PackageReference`)

```

<!-- Accepts any version 6.1 and above. -->
<PackageReference Include="ExamplePackage" Version="6.1" />

<!-- Accepts any 6.x.y version. -->
<PackageReference Include="ExamplePackage" Version="6.*" />
<PackageReference Include="ExamplePackage" Version="[6,7)" />

<!-- Accepts any version above, but not including 4.1.3. Could be
     used to guarantee a dependency with a specific bug fix. -->
<PackageReference Include="ExamplePackage" Version="(4.1.3,)" />

<!-- Accepts any version up below 5.x, which might be used to prevent pulling in a later
     version of a dependency that changed its interface. However, this form is not
     recommended because it can be difficult to determine the lowest version. -->
<PackageReference Include="ExamplePackage" Version="(<,5.0)" />

<!-- Accepts any 1.x or 2.x version, but not 0.x or 3.x and higher. -->
<PackageReference Include="ExamplePackage" Version="[1,3)" />

<!-- Accepts 1.3.2 up to 1.4.x, but not 1.5 and higher. -->
<PackageReference Include="ExamplePackage" Version="[1.3.2,1.5)" />

```

References in `packages.config`:

In `packages.config`, every dependency is listed with an exact `version` attribute that's used when restoring packages. The `allowedVersions` attribute is used only during update operations to constrain the versions to which the package might be updated.

```

<!-- Install/restore version 6.1.0, accept any version 6.1.0 and above on update. -->
<package id="ExamplePackage" version="6.1.0" allowedVersions="6.1.0" />

<!-- Install/restore version 6.1.0, and do not change during update. -->
<package id="ExamplePackage" version="6.1.0" allowedVersions="[6.1.0]" />

<!-- Install/restore version 6.1.0, accept any 6.x version during update. -->
<package id="ExamplePackage" version="6.1.0" allowedVersions="[6,7)" />

<!-- Install/restore version 4.1.4, accept any version above, but not including, 4.1.3.
     Could be used to guarantee a dependency with a specific bug fix. -->
<package id="ExamplePackage" version="4.1.4" allowedVersions="(4.1.3,)" />

<!-- Install/restore version 3.1.2, accept any version up below 5.x on update, which might be
     used to prevent pulling in a later version of a dependency that changed its interface.
     However, this form is not recommended because it can be difficult to determine the lowest version. -->
<package id="ExamplePackage" version="3.1.2" allowedVersions="(<,5.0)" />

<!-- Install/restore version 1.1.4, accept any 1.x or 2.x version on update, but not
     0.x or 3.x and higher. -->
<package id="ExamplePackage" version="1.1.4" allowedVersions="[1,3)" />

<!-- Install/restore version 1.3.5, accepts 1.3.2 up to 1.4.x on update, but not 1.5 and higher. -->
<package id="ExamplePackage" version="1.3.5" allowedVersions="[1.3.2,1.5)" />

```

References in `.nuspec` files

The `version` attribute in a `<dependency>` element describes the range versions that are acceptable for a dependency.

```
<!-- Accepts any version 6.1 and above. -->
<dependency id="ExamplePackage" version="6.1" />

<!-- Accepts any version above, but not including 4.1.3. Could be
     used to guarantee a dependency with a specific bug fix. -->
<dependency id="ExamplePackage" version="(4.1.3,)" />

<!-- Accepts any version up below 5.x, which might be used to prevent pulling in a later
     version of a dependency that changed its interface. However, this form is not
     recommended because it can be difficult to determine the lowest version. -->
<dependency id="ExamplePackage" version="(<5.0)" />

<!-- Accepts any 1.x or 2.x version, but not 0.x or 3.x and higher. -->
<dependency id="ExamplePackage" version="[1,3)" />

<!-- Accepts 1.3.2 up to 1.4.x, but not 1.5 and higher. -->
<dependency id="ExamplePackage" version="[1.3.2,1.5)" />
```

Normalized version numbers

NOTE

This is a breaking change for NuGet 3.4 and later.

When obtaining packages from a repository during install, reinstall, or restore operations, NuGet 3.4+ treats version numbers as follows:

- Leading zeroes are removed from version numbers:

```
1.00 is treated as 1.0
1.01.1 is treated as 1.1.1
1.00.0.1 is treated as 1.0.0.1
```

- A zero in the fourth part of the version number will be omitted

```
1.0.0.0 is treated as 1.0.0
1.0.01.0 is treated as 1.0.1
```

`pack` and `restore` operations normalize versions whenever possible. For packages already built, this normalization does not affect the version numbers in the packages themselves; it affects only how NuGet matches versions when resolving dependencies.

However, NuGet package repositories must treat these values in the same way as NuGet to prevent package version duplication. Thus a repository that contains version *1.0* of a package should not also host version *1.0.0* as a separate and different package.

How NuGet resolves package dependencies

11/5/2019 • 8 minutes to read • [Edit Online](#)

Any time a package is installed or reinstalled, which includes being installed as part of a [restore](#) process, NuGet also installs any additional packages on which that first package depends.

Those immediate dependencies might then also have dependencies on their own, which can continue to an arbitrary depth. This produces what's called a *dependency graph* that describes the relationships between packages at all levels.

When multiple packages have the same dependency, then the same package ID can appear in the graph multiple times, potentially with different version constraints. However, only one version of a given package can be used in a project, so NuGet must choose which version is used. The exact process depends on the package management format being used.

Dependency resolution with PackageReference

When installing packages into projects using the `PackageReference` format, NuGet adds references to a flat package graph in the appropriate file and resolves conflicts ahead of time. This process is referred to as *transitive restore*. Reinstalling or restoring packages is then a process of downloading the packages listed in the graph, resulting in faster and more predictable builds. You can also take advantage of wildcard (floating) versions, such as `2.8.*`, avoiding expensive and error prone calls to `nuget update` on the client machines and build servers.

When the NuGet restore process runs prior to a build, it resolves dependencies first in memory, then writes the resulting graph to a file called `project.assets.json`. It also writes the resolved dependencies to a lock file named `packages.lock.json`, if the [lock file functionality is enabled](#). The assets file is located at `MSBuildProjectExtensionsPath`, which defaults to the project's 'obj' folder. MSBuild then reads this file and translates it into a set of folders where potential references can be found, and then adds them to the project tree in memory.

The `project.assets.json` file is temporary and should not be added to source control. It's listed by default in both `.gitignore` and `.tfignore`. See [Packages and source control](#).

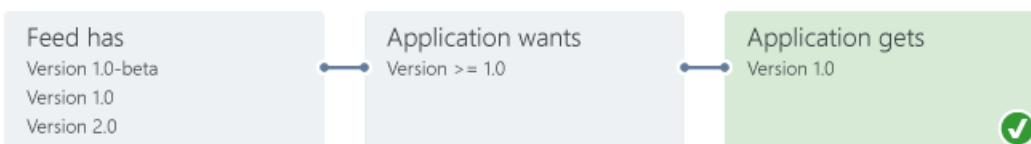
Dependency resolution rules

Transitive restore applies four main rules to resolve dependencies: lowest applicable version, floating versions, nearest-wins, and cousin dependencies.

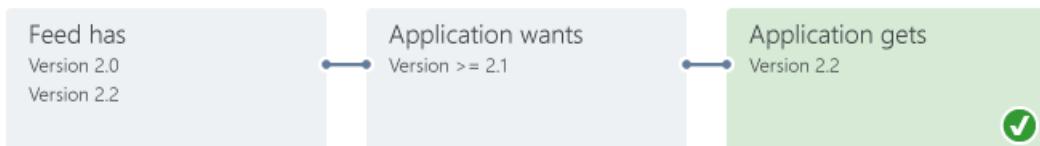
Lowest applicable version

The lowest applicable version rule restores the lowest possible version of a package as defined by its dependencies. It also applies to dependencies on the application or the class library unless declared as [floating](#).

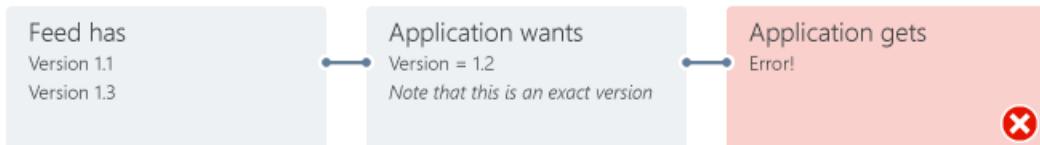
In the following figure, for example, 1.0-beta is considered lower than 1.0 so NuGet chooses the 1.0 version:



In the next figure, version 2.1 is not available on the feed but because the version constraint is ≥ 2.1 NuGet picks the next lowest version it can find, in this case 2.2:



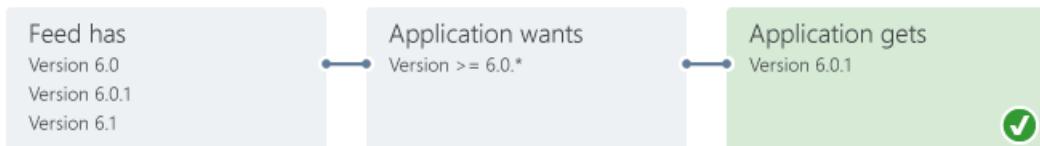
When an application specifies an exact version number, such as 1.2, that is not available on the feed, NuGet fails with an error when attempting to install or restore the package:



Floating (wildcard) versions

A floating or wildcard dependency version is specified with the `*` wildcard, as with `6.0.*`. This version specification says "use the latest 6.0.x version"; `4.*` means "use the latest 4.x version." Using a wildcard allows a dependency package to continue evolving without requiring a change to the consuming application (or package).

When using a wildcard, NuGet resolves the highest version of a package that matches the version pattern, for example `6.0.*` gets the highest version of a package that starts with 6.0:



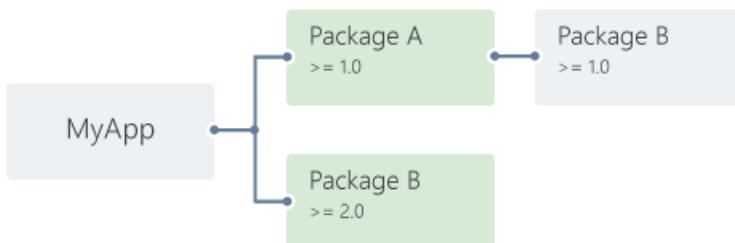
NOTE

For information on the behavior of wildcards and pre-release versions, see [Package versioning](#).

Nearest wins

When the package graph for an application contains different versions of the same package, NuGet chooses the package that's closest to the application in the graph and ignores all others. This behavior allows an application to override any particular package version in the dependency graph.

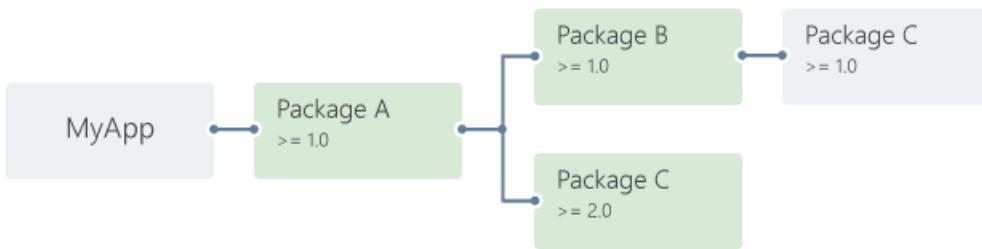
In the example below, the application depends directly on Package B with a version constraint of `>=2.0`. The application also depends on Package A which in turn also depends on Package B, but with a `>=1.0` constraint. Because the dependency on Package B 2.0 is nearer to the application in the graph, that version is used:



WARNING

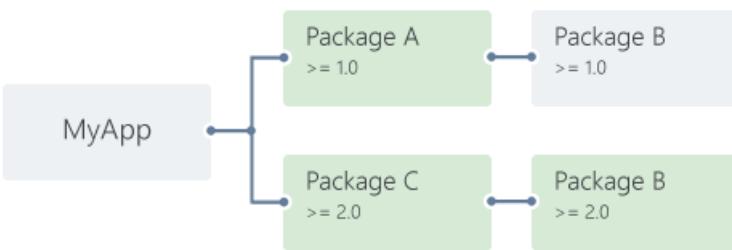
The Nearest Wins rule can result in a downgrade of the package version, thus potentially breaking other dependencies in the graph. Hence this rule is applied with a warning to alert the user.

This rule also results in greater efficiency with a large dependency graph (such as those with the BCL packages) because once a given dependency is ignored, NuGet also ignores all remaining dependencies on that branch of the graph. In the diagram below, for example, because Package C 2.0 is used, NuGet ignores any branches in the graph that refer to an older version of Package C:

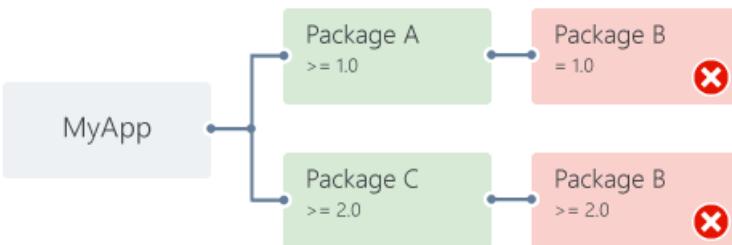


Cousin dependencies

When different package versions are referred to at the same distance in the graph from the application, NuGet uses the lowest version that satisfies all version requirements (as with the [lowest applicable version](#) and [floating versions](#) rules). In the image below, for example, version 2.0 of Package B satisfies the other ≥ 1.0 constraint, and is thus used:



In some cases, it's not possible to meet all version requirements. As shown below, if Package A requires exactly Package B 1.0 and Package C requires Package B ≥ 2.0 , then NuGet cannot resolve the dependencies and gives an error.



In these situations, the top-level consumer (the application or package) should add its own direct dependency on Package B so that the [Nearest Wins](#) rule applies.

Dependency resolution with packages.config

With `packages.config`, a project's dependencies are written to `packages.config` as a flat list. Any dependencies of those packages are also written in the same list. When packages are installed, NuGet might also modify the `.csproj` file, `app.config`, `web.config`, and other individual files.

With `packages.config`, NuGet attempts to resolve dependency conflicts during the installation of each individual package. That is, if Package A is being installed and depends on Package B, and Package B is already listed in `packages.config` as a dependency of something else, NuGet compares the versions of Package B being requested and attempts to find a version that satisfies all version constraints. Specifically, NuGet selects the lower *major.minor* version that satisfies dependencies.

By default, NuGet 2.8 looks for the lowest patch version (see [NuGet 2.8 release notes](#)). You can control this setting

through the `DependencyVersion` attribute in `Nuget.Config` and the `-DependencyVersion` switch on the command line.

The `packages.config` process for resolving dependencies gets complicated for larger dependency graphs. Each new package installation requires a traversal of the whole graph and raises the chance for version conflicts. When a conflict occurs, installation is stopped, leaving the project in an indeterminate state, especially with potential modifications to the project file itself. This is not an issue when using other package management formats.

Managing dependency assets

When using the `PackageReference` format, you can control which assets from dependencies flow into the top-level project. For details, see [PackageReference](#).

When the top-level project is itself a package, you also have control over this flow by using the `include` and `exclude` attributes with dependencies listed in the `.nuspec` file. See [.nuspec Reference - Dependencies](#).

Excluding references

There are scenarios in which assemblies with the same name might be referenced more than once in a project, producing design-time and build-time errors. Consider a project that contains a custom version of `c.dll`, and references Package C that also contains `c.dll`. At the same time, the project also depends on Package B which also depends on Package C and `c.dll`. As a result, NuGet can't determine which `c.dll` to use, but you can't just remove the project's dependency on Package C because Package B also depends on it.

To resolve this, you must directly reference the `c.dll` you want (or use another package that references the right one), and then add a dependency on Package C that excludes all its assets. This is done as follows depending on the package management format in use:

- `PackageReference`: add `ExcludeAssets="All"` in the dependency:

```
<PackageReference Include="PackageC" Version="1.0.0" ExcludeAssets="All" />
```

- `packages.config`: remove the reference to PackageC from the `.csproj` file so that it references only the version of `c.dll` that you want.

Dependency updates during package install

If a dependency version is already satisfied, the dependency isn't updated during other package installations. For example, consider package A that depends on package B and specifies 1.0 for the version number. The source repository contains versions 1.0, 1.1, and 1.2 of package B. If A is installed in a project that already contains B version 1.0, then B 1.0 remains in use because it satisfies the version constraint. However, if package A had requests version 1.1 or higher of B, then B 1.2 would be installed.

Resolving incompatible package errors

During a package restore operation, you may see the error "One or more packages are not compatible..." or that a package "is not compatible" with the project's target framework.

This error occurs when one or more of the packages referenced in your project do not indicate that they support the project's target framework; that is, the package does not contain a suitable DLL in its `lib` folder for a target framework that is compatible with the project. (See [Target frameworks](#) for a list.)

For example, if a project targets `netstandard1.6` and you attempt to install a package that contains DLLs in only the `lib\net20` and `\lib\net45` folders, then you see messages like the following for the package and possibly its dependents:

```
Restoring packages for myproject.csproj...
Package ContosoUtilities 2.1.2.3 is not compatible with netstandard1.6 (.NETStandard,Version=v1.6). Package
ContosoUtilities 2.1.2.3 supports:
- net20 (.NETFramework,Version=v2.0)
- net45 (.NETFramework,Version=v4.5)
Package ContosoCore 0.86.0 is not compatible with netstandard1.6 (.NETStandard,Version=v1.6). Package
ContosoCore 0.86.0 supports:
- 11 (11,Version=v0.0)
- net20 (.NETFramework,Version=v2.0)
- sl3 (Silverlight,Version=v3.0)
- sl4 (Silverlight,Version=v4.0)
One or more packages are incompatible with .NETStandard,Version=v1.6.
Package restore failed. Rolling back package changes for 'MyProject'.
```

To resolve incompatibilities, do one of the following:

- Retarget your project to a framework that is supported by the packages you want to use.
- Contact the author of the packages and work with them to add support for your chosen framework. Each package listing page on nuget.org has a **Contact Owners** link for this purpose.

.nuspec reference

11/14/2019 • 21 minutes to read • [Edit Online](#)

A `.nuspec` file is an XML manifest that contains package metadata. This manifest is used both to build the package and to provide information to consumers. The manifest is always included in a package.

In this topic:

- [General form and schema](#)
- [Replacement tokens](#) (when used with a Visual Studio project)
- [Dependencies](#)
- [Explicit assembly references](#)
- [Framework assembly references](#)
- [Including assembly files](#)
- [Including content files](#)
- [Example nuspec files](#)

Project type compatibility

- Use `.nuspec` with `nuget.exe pack` for non-SDK-style projects that use `packages.config`.
- A `.nuspec` file is not required to create packages for [SDK-style projects](#) (typically .NET Core and .NET Standard projects that use the [SDK attribute](#)). (Note that a `.nuspec` is generated when you create the package.)

If you are creating a package using `dotnet.exe pack` or `msbuild pack target`, we recommend that you [include all the properties](#) that are usually in the `.nuspec` file in the project file instead. However, you can instead choose to [use a `.nuspec` file to pack using `dotnet.exe` or `msbuild pack target`](#).

- For projects migrated from `packages.config` to [PackageReference](#), a `.nuspec` file is not required to create the package. Instead, use `msbuild -t:pack`.

General form and schema

The current `nuspec.xsd` schema file can be found in the [NuGet GitHub repository](#).

Within this schema, a `.nuspec` file has the following general form:

```
<?xml version="1.0" encoding="utf-8"?>
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
  <metadata>
    <!-- Required elements-->
    <id></id>
    <version></version>
    <description></description>
    <authors></authors>

    <!-- Optional elements -->
    <!-- ... -->
  </metadata>
  <!-- Optional 'files' node -->
</package>
```

For a clear visual representation of the schema, open the schema file in Visual Studio in Design mode and click on the **XML Schema Explorer** link. Alternately, open the file as code, right-click in the editor, and select **Show XML Schema Explorer**. Either way you get a view like the one below (when mostly expanded):



Required metadata elements

Although the following elements are the minimum requirements for a package, you should consider adding the [optional metadata elements](#) to improve the overall experience developers have with your package.

These elements must appear within a `<metadata>` element.

id

The case-insensitive package identifier, which must be unique across nuget.org or whatever gallery the package resides in. IDs may not contain spaces or characters that are not valid for a URL, and generally follow .NET namespace rules. See [Choosing a unique package identifier](#) for guidance.

version

The version of the package, following the *major.minor.patch* pattern. Version numbers may include a pre-release suffix as described in [Package versioning](#).

description

A description of the package for UI display.

authors

A comma-separated list of packages authors, matching the profile names on nuget.org. These are displayed in the NuGet Gallery on nuget.org and are used to cross-reference packages by the same authors.

Optional metadata elements

owners

A comma-separated list of the package creators using profile names on nuget.org. This is often the same list as in `authors`, and is ignored when uploading the package to nuget.org. See [Managing package owners on nuget.org](#).

projectUrl

A URL for the package's home page, often shown in UI displays as well as nuget.org.

licenseUrl

IMPORTANT

licenseUrl is deprecated. Use license instead.

A URL for the package's license, often shown in UIs like nuget.org.

license

An SPDX license expression or path to a license file within the package, often shown in UIs like nuget.org. If you're licensing the package under a common license, like MIT or BSD-2-Clause, use the associated [SPDX license identifier](#). For example:

```
<license type="expression">MIT</license>
```

NOTE

NuGet.org only accepts license expressions that are approved by the Open Source Initiative or the Free Software Foundation.

If your package is licensed under multiple common licenses, you can specify a composite license using the [SPDX expression syntax version 2.0](#). For example:

```
<license type="expression">BSD-2-Clause OR MIT</license>
```

If you use a custom license that isn't supported by license expressions, you can package a `.txt` or `.md` file with the license's text. For example:

```
<package>
  <metadata>
    ...
    <license type="file">LICENSE.txt</license>
    ...
  </metadata>
  <files>
    ...
    <file src="licenses\LICENSE.txt" target="" />
    ...
  </files>
</package>
```

For the MSBuild equivalent, take a look at [Packing a license expression or a license file](#).

The exact syntax of NuGet's license expressions is described below in [ABNF](#).

```
license-id          = <short form license identifier from https://spdx.org/spdx-specification-21-web-version#h.luq9dgcle9mo>

license-exception-id = <short form license exception identifier from https://spdx.org/spdx-specification-21-web-version#h.ruv3y18g6czd>

simple-expression = license-id / license-id""

compound-expression = 1*1(simple-expression /
  simple-expression "WITH" license-exception-id /
  compound-expression "AND" compound-expression /
  compound-expression "OR" compound-expression ) /
  "(" compound-expression ")" )

license-expression = 1*1(simple-expression / compound-expression / UNLICENSED)
```

iconUrl

IMPORTANT

iconUrl is deprecated. Use icon instead.

A URL for a 64x64 image with transparency background to use as the icon for the package in UI display. Be sure this element contains the *direct image URL* and not the URL of a web page containing the image. For example, to use an image from GitHub, use the raw file URL like

<https://github.com/<username>/<repository>/raw/<branch>/<logo.png>>.

icon

It is a path to an image file within the package, often shown in UIs like nuget.org as the package icon. Image file size is limited to 1 MB. Supported file formats include JPEG and PNG. We recommend an image resolution of 64x64.

For example, you would add the following to your nuspec when creating a package using nuget.exe:

```
<package>
  <metadata>
    ...
    <icon>images\icon.png</icon>
    ...
  </metadata>
  <files>
    ...
    <file src="..\icon.png" target="images\" />
    ...
  </files>
</package>
```

[Package Icon nuspec sample.](#)

For the MSBuild equivalent, take a look at [Packing an icon image file](#).

TIP

You can specify both `icon` and `iconUrl` to maintain backward compatibility with sources that do not support `icon`. Visual Studio will support `icon` for packages coming from a folder-based source in a future release.

requireLicenseAcceptance

A Boolean value specifying whether the client must prompt the consumer to accept the package license before installing the package.

developmentDependency

(2.8+) A Boolean value specifying whether the package is be marked as a development-only-dependency, which prevents the package from being included as a dependency in other packages. With `PackageReference` (NuGet 4.8+), this flag also means that it will exclude compile-time assets from compilation. See [DevelopmentDependency support for PackageReference](#)

summary

IMPORTANT

`summary` is being deprecated. Use `description` instead.

A short description of the package for UI display. If omitted, a truncated version of `description` is used.

releaseNotes

(1.5+) A description of the changes made in this release of the package, often used in UI like the **Updates** tab of the Visual Studio Package Manager in place of the package description.

copyright

(1.5+) Copyright details for the package.

language

The locale ID for the package. See [Creating localized packages](#).

tags

A space-delimited list of tags and keywords that describe the package and aid discoverability of packages through search and filtering.

serviceable

(3.3+) For internal NuGet use only.

repository

Repository metadata, consisting of four optional attributes: `type` and `url` (4.0+), and `branch` and `commit` (4.6+). These attributes allow you to map the `.nupkg` to the repository that built it, with the potential to get as detailed as the individual branch name and / or commit SHA-1 hash that built the package. This should be a publicly available url that can be invoked directly by a version control software. It should not be an html page as this is meant for the computer. For linking to project page, use the `projectUrl` field, instead.

For example:

```
<?xml version="1.0"?>
<package xmlns="http://schemas.microsoft.com/packaging/2016/06/nuspec.xsd">
  <metadata>
    ...
    <repository type="git" url="https://github.com/NuGet/NuGet.Client.git" branch="dev"
    commit="e1c65e4524cd70ee6e22abe33e6cb6ec73938cb3" />
    ...
  </metadata>
</package>
```

title

A human-friendly title of the package which may be used in some UI displays. (nuget.org and the Package Manager in Visual Studio do not show title)

Collection elements

packageTypes

(3.5+) A collection of zero or more `<packageType>` elements specifying the type of the package if other than a traditional dependency package. Each packageType has attributes of *name* and *version*. See [Setting a package type](#).

dependencies

A collection of zero or more `<dependency>` elements specifying the dependencies for the package. Each dependency has attributes of *id*, *version*, *include* (3.x+), and *exclude* (3.x+). See [Dependencies](#) below.

frameworkAssemblies

(1.2+) A collection of zero or more `<frameworkAssembly>` elements identifying .NET Framework assembly references that this package requires, which ensures that references are added to projects consuming the package. Each frameworkAssembly has *assemblyName* and *targetFramework* attributes. See [Specifying framework assembly references GAC](#) below.

references

(1.5+) A collection of zero or more `<reference>` elements naming assemblies in the package's `lib` folder that are added as project references. Each reference has a *file* attribute. `<references>` can also contain a `<group>`

element with a `targetFramework` attribute, that then contains `<reference>` elements. If omitted, all references in `lib` are included. See [Specifying explicit assembly references](#) below.

contentFiles

(3.3+) A collection of `<files>` elements that identify content files to include in the consuming project. These files are specified with a set of attributes that describe how they should be used within the project system. See [Specifying files to include in the package](#) below.

files

The `<package>` node may contain a `<files>` node as a sibling to `<metadata>`, and a `<contentFiles>` child under `<metadata>`, to specify which assembly and content files to include in the package. See [Including assembly files](#) and [Including content files](#) later in this topic for details.

metadata attributes

minClientVersion

Specifies the minimum version of the NuGet client that can install this package, enforced by `nuget.exe` and the Visual Studio Package Manager. This is used whenever the package depends on specific features of the `.nuspec` file that were added in a particular version of the NuGet client. For example, a package using the `developmentDependency` attribute should specify "2.8" for `minClientVersion`. Similarly, a package using the `contentFiles` element (see the next section) should set `minClientVersion` to "3.3". Note also that because NuGet clients prior to 2.5 do not recognize this flag, they *always* refuse to install the package no matter what `minClientVersion` contains.

```
<?xml version="1.0" encoding="utf-8"?>
<package xmlns="http://schemas.microsoft.com/packaging/2013/01/nuspec.xsd">
  <metadata minClientVersion="100.0.0.1">
    <id>dasdas</id>
    <version>2.0.0</version>
    <title />
    <authors>dsadas</authors>
    <owners />
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>My package description.</description>
  </metadata>
  <files>
    <file src="content\one.txt" target="content\one.txt" />
  </files>
</package>
```

Replacement tokens

When creating a package, the `nuget pack` command replaces \$-delimited tokens in the `.nuspec` file's `<metadata>` node with values that come from either a project file or the `pack` command's `-properties` switch.

On the command line, you specify token values with `nuget pack -properties <name>=<value>;<name>=<value>`. For example, you can use a token such as `$owners$` and `$desc$` in the `.nuspec` and provide the values at packing time as follows:

```
nuget pack MyProject.csproj -properties
  owners=janedoe,harikm,kimo,xiaop;desc="Awesome app logger utility"
```

To use values from a project, specify the tokens described in the table below (AssemblyInfo refers to the file in `Properties` such as `AssemblyInfo.cs` or `AssemblyInfo.vb`).

To use these tokens, run `nuget pack` with the project file rather than just the `.nuspec`. For example, when using the following command, the `id` and `$version$` tokens in a `.nuspec` file are replaced with the project's

`AssemblyName` and `AssemblyVersion` values:

```
nuget pack MyProject.csproj
```

Typically, when you have a project, you create the `.nuspec` initially using `nuget spec MyProject.csproj` which automatically includes some of these standard tokens. However, if a project lacks values for required `.nuspec` elements, then `nuget pack` fails. Furthermore, if you change project values, be sure to rebuild before creating the package; this can be done conveniently with the pack command's `build` switch.

With the exception of `$configuration$`, values in the project are used in preference to any assigned to the same token on the command line.

TOKEN	VALUE SOURCE	VALUE
<code>\$id\$</code>	Project file	AssemblyName (title) from the project file
<code>\$version\$</code>	AssemblyInfo	AssemblyInformationalVersion if present, otherwise AssemblyVersion
<code>\$author\$</code>	AssemblyInfo	AssemblyCompany
<code>\$title\$</code>	AssemblyInfo	AssemblyTitle
<code>\$description\$</code>	AssemblyInfo	AssemblyDescription
<code>\$copyright\$</code>	AssemblyInfo	AssemblyCopyright
<code>\$configuration\$</code>	Assembly DLL	Configuration used to build the assembly, defaulting to Debug. Note that to create a package using a Release configuration, you always use <code>-properties Configuration=Release</code> on the command line.

Tokens can also be used to resolve paths when you include [assembly files](#) and [content files](#). The tokens have the same names as the MSBuild properties, making it possible to select files to be included depending on the current build configuration. For example, if you use the following tokens in the `.nuspec` file:

```
<files>
  <file src="bin\$configuration$\$id$.pdb" target="lib\net40" />
</files>
```

And you build an assembly whose `AssemblyName` is `LoggingLibrary` with the `Release` configuration in MSBuild, the resulting lines in the `.nuspec` file in the package is as follows:

```
<files>
  <file src="bin\Release\LoggingLibrary.pdb" target="lib\net40" />
</files>
```

Dependencies element

The `<dependencies>` element within `<metadata>` contains any number of `<dependency>` elements that identify

other packages upon which the top-level package depends. The attributes for each `<dependency>` are as follows:

ATTRIBUTE	DESCRIPTION
<code>id</code>	(Required) The package ID of the dependency, such as "EntityFramework" and "NUnit", which is the name of the package nuget.org shows on a package page.
<code>version</code>	(Required) The range of versions acceptable as a dependency. See Package versioning for exact syntax. Wildcard (floating) versions are not supported.
<code>include</code>	A comma-delimited list of include/exclude tags (see below) indicating of the dependency to include in the final package. The default value is <code>all</code> .
<code>exclude</code>	A comma-delimited list of include/exclude tags (see below) indicating of the dependency to exclude in the final package. The default value is <code>build, analyzers</code> , which can be over-written. But <code>content/ ContentFiles</code> are also implicitly excluded in the final package which can't be over-written. Tags specified with <code>exclude</code> take precedence over those specified with <code>include</code> . For example, <code>include="runtime, compile" exclude="compile"</code> is the same as <code>include="runtime"</code> .

INCLUDE/EXCLUDE TAG	AFFECTED FOLDERS OF THE TARGET
<code>contentFiles</code>	Content
<code>runtime</code>	Runtime, Resources, and FrameworkAssemblies
<code>compile</code>	lib
<code>build</code>	build (MSBuild props and targets)
<code>native</code>	native
<code>none</code>	No folders
<code>all</code>	All folders

For example, the following lines indicate dependencies on `PackageA` version 1.1.0 or higher, and `PackageB` version 1.x.

```
<dependencies>
  <dependency id="PackageA" version="1.1.0" />
  <dependency id="PackageB" version="[1,2)" />
</dependencies>
```

The following lines indicate dependencies on the same packages, but specify to include the `contentFiles` and `build` folders of `PackageA` and everything but the `native` and `compile` folders of `PackageB`:

```
<dependencies>
  <dependency id="PackageA" version="1.1.0" include="contentFiles, build" />
  <dependency id="PackageB" version="[1,2)" exclude="native, compile" />
</dependencies>
```

IMPORTANT

When creating a `.nuspec` from a project using `nuget spec`, dependencies that exist in that project are not automatically included in the resulting `.nuspec` file. Instead, use `nuget pack myproject.csproj`, and get the `.nuspec` file from within the generated `.nupkg` file. This `.nuspec` contains the dependencies.

Dependency groups

Version 2.0+

As an alternative to a single flat list, dependencies can be specified according to the framework profile of the target project using `<group>` elements within `<dependencies>`.

Each group has an attribute named `targetFramework` and contains zero or more `<dependency>` elements. Those dependencies are installed together when the target framework is compatible with the project's framework profile.

The `<group>` element without a `targetFramework` attribute is used as the default or fallback list of dependencies. See [Target frameworks](#) for the exact framework identifiers.

IMPORTANT

The group format cannot be intermixed with a flat list.

The following example shows different variations of the `<group>` element:

```
<dependencies>
  <group>
    <dependency id="RouteMagic" version="1.1.0" />
  </group>

  <group targetFramework="net40">
    <dependency id="jQuery" version="1.6.2" />
    <dependency id="WebActivator" version="1.4.4" />
  </group>

  <group targetFramework="sl30">
  </group>
</dependencies>
```

Explicit assembly references

The `<references>` element is used by projects using `packages.config` to explicitly specify the assemblies that the target project should reference when using the package. Explicit references are typically used for design-time only assemblies. For more information, see the page on [selecting assemblies referenced by projects](#) for more information.

For example, the following `<references>` element instructs NuGet to add references to only `xunit.dll` and `xunit.extensions.dll` even if there are additional assemblies in the package:

```
<references>
  <reference file="xunit.dll" />
  <reference file="xunit.extensions.dll" />
</references>
```

Reference groups

As an alternative to a single flat list, references can be specified according to the framework profile of the target project using `<group>` elements within `<references>`.

Each group has an attribute named `targetFramework` and contains zero or more `<reference>` elements. Those references are added to a project when the target framework is compatible with the project's framework profile.

The `<group>` element without a `targetFramework` attribute is used as the default or fallback list of references.

See [Target frameworks](#) for the exact framework identifiers.

IMPORTANT

The group format cannot be intermixed with a flat list.

The following example shows different variations of the `<group>` element:

```
<references>
  <group>
    <reference file="a.dll" />
  </group>

  <group targetFramework="net45">
    <reference file="b45.dll" />
  </group>

  <group targetFramework="netcore45">
    <reference file="bcore45.dll" />
  </group>
</references>
```

Framework assembly references

Framework assemblies are those that are part of the .NET framework and should already be in the global assembly cache (GAC) for any given machine. By identifying those assemblies within the `<frameworkAssemblies>` element, a package can ensure that required references are added to a project in the event that the project doesn't have such references already. Such assemblies, of course, are not included in a package directly.

The `<frameworkAssemblies>` element contains zero or more `<frameworkAssembly>` elements, each of which specifies the following attributes:

ATTRIBUTE	DESCRIPTION
assemblyName	(Required) The fully qualified assembly name.
targetFramework	(Optional) Specifies the target framework to which this reference applies. If omitted, indicates that the reference applies to all frameworks. See Target frameworks for the exact framework identifiers.

The following example shows a reference to `System.Net` for all target frameworks, and a reference to

`System.ServiceModel` for .NET Framework 4.0 only:

```
<frameworkAssemblies>
  <frameworkAssembly assemblyName="System.Net" />

  <frameworkAssembly assemblyName="System.ServiceModel" targetFramework="net40" />
</frameworkAssemblies>
```

Including assembly files

If you follow the conventions described in [Creating a Package](#), you do not have to explicitly specify a list of files in the `.nuspec` file. The `nuget pack` command automatically picks up the necessary files.

IMPORTANT

When a package is installed into a project, NuGet automatically adds assembly references to the package's DLLs, *excluding* those that are named `.resources.dll` because they are assumed to be localized satellite assemblies. For this reason, avoid using `.resources.dll` for files that otherwise contain essential package code.

To bypass this automatic behavior and explicitly control which files are included in a package, place a `<files>` element as a child of `<package>` (and a sibling of `<metadata>`), identifying each file with a separate `<file>` element. For example:

```
<files>
  <file src="bin\Debug\*.dll" target="lib" />
  <file src="bin\Debug\*.pdb" target="lib" />
  <file src="tools\**\*.*" exclude="**\*.log" />
</files>
```

With NuGet 2.x and earlier, and projects using `packages.config`, the `<files>` element is also used to include immutable content files when a package is installed. With NuGet 3.3+ and projects `PackageReference`, the `<contentFiles>` element is used instead. See [Including content files](#) below for details.

File element attributes

Each `<file>` element specifies the following attributes:

ATTRIBUTE	DESCRIPTION
<code>src</code>	The location of the file or files to include, subject to exclusions specified by the <code>exclude</code> attribute. The path is relative to the <code>.nuspec</code> file unless an absolute path is specified. The wildcard character <code>*</code> is allowed, and the double wildcard <code>**</code> implies a recursive folder search.
<code>target</code>	The relative path to the folder within the package where the source files are placed, which must begin with <code>lib</code> , <code>content</code> , <code>build</code> , or <code>tools</code> . See Creating a .nuspec from a convention-based working directory .
<code>exclude</code>	A semicolon-delimited list of files or file patterns to exclude from the <code>src</code> location. The wildcard character <code>*</code> is allowed, and the double wildcard <code>**</code> implies a recursive folder search.

Examples

Single assembly

```
Source file:  
  library.dll  
  
.nuspec entry:  
  <file src="library.dll" target="lib" />  
  
Packaged result:  
  lib\library.dll
```

Single assembly specific to a target framework

```
Source file:  
  library.dll  
  
.nuspec entry:  
  <file src="assemblies\net40\library.dll" target="lib\net40" />  
  
Packaged result:  
  lib\net40\library.dll
```

Set of DLLs using a wildcard

```
Source files:  
  bin\release\libraryA.dll  
  bin\release\libraryB.dll  
  
.nuspec entry:  
  <file src="bin\release\*.dll" target="lib" />  
  
Packaged result:  
  lib\libraryA.dll  
  lib\libraryB.dll
```

DLLs for different frameworks

```
Source files:  
  lib\net40\library.dll  
  lib\net20\library.dll  
  
.nuspec entry (using ** recursive search):  
  <file src="lib\**" target="lib" />  
  
Packaged result:  
  lib\net40\library.dll  
  lib\net20\library.dll
```

Excluding files

```

Source files:
  \tools\fileA.bak
  \tools\fileB.bak
  \tools\fileA.log
  \tools\build\fileB.log

.nuspec entries:
  <file src="tools\*.*" target="tools" exclude="tools\*.bak" />
  <file src="tools\**\*.*" target="tools" exclude="**\*.log" />

Package result:
  (no files)

```

Including content files

Content files are immutable files that a package needs to include in a project. Being immutable, they are not intended to be modified by the consuming project. Example content files include:

- Images that are embedded as resources
- Source files that are already compiled
- Scripts that need to be included with the build output of the project
- Configuration files for the package that need to be included in the project but don't need any project-specific changes

Content files are included in a package using the `<files>` element, specifying the `content` folder in the `target` attribute. However, such files are ignored when the package is installed in a project using `PackageReference`, which instead uses the `<contentFiles>` element.

For maximum compatibility with consuming projects, a package ideally specifies the content files in both elements.

Using the `files` element for content files

For content files, simply use the same format as for assembly files, but specify `content` as the base folder in the `target` attribute as shown in the following examples.

Basic content files

```

Source files:
  css\mobile\style1.css
  css\mobile\style2.css

.nuspec entry:
  <file src="css\mobile\*.css" target="content\css\mobile" />

Packaged result:
  content\css\mobile\style1.css
  content\css\mobile\style2.css

```

Content files with directory structure

```
Source files:  
  css\mobile\style.css  
  css\mobile\wp7\style.css  
  css\browser\style.css  
  
.nuspec entry:  
  <file src="css\**\*.css" target="content\css" />  
  
Packaged result:  
  content\css\mobile\style.css  
  content\css\mobile\wp7\style.css  
  content\css\browser\style.css
```

Content file specific to a target framework

```
Source file:  
  css\cool\style.css  
  
.nuspec entry:  
  <file src="css\cool\style.css" target="Content" />  
  
Packaged result:  
  content\style.css
```

Content file copied to a folder with dot in name

In this case, NuGet sees that the extension in `target` does not match the extension in `src` and thus treats that part of the name in `target` as a folder:

```
Source file:  
  images\picture.png  
  
.nuspec entry:  
  <file src="images\picture.png" target="Content\images\package.icons" />  
  
Packaged result:  
  content\images\package.icons\picture.png
```

Content files without extensions

To include files without an extension, use the `*` or `**` wildcards:

```
Source file:  
  flags\installed  
  
.nuspec entry:  
  <file src="flags\**" target="flags" />  
  
Packaged result:  
  flags\installed
```

Content files with deep path and deep target

In this case, because the file extensions of the source and target match, NuGet assumes that the target is a file name and not a folder:

```

Source file:
  css\cool\style.css

.nuspec entry:
  <file src="css\cool\style.css" target="Content\css\cool" />
  or:
  <file src="css\cool\style.css" target="Content\css\cool\style.css" />

Packaged result:
  content\css\cool\style.css

```

Renaming a content file in the package

```

Source file:
  ie\css\style.css

.nuspec entry:
  <file src="ie\css\style.css" target="Content\css\ie.css" />

Packaged result:
  content\css\ie.css

```

Excluding files

```

Source file:
  docs\*.txt (multiple files)

.nuspec entry:
  <file src="docs\*.txt" target="content\docs" exclude="docs\admin.txt" />
  or
  <file src="*.txt" target="content\docs" exclude="admin.txt;log.txt" />

Packaged result:
  All .txt files from docs except admin.txt (first example)
  All .txt files from docs except admin.txt and log.txt (second example)

```

Using the `contentFiles` element for content files

NuGet 4.0+ with `PackageReference`

By default, a package places content in a `contentFiles` folder (see below) and `nuget pack` included all files in that folder using default attributes. In this case it's not necessary to include a `contentFiles` node in the `.nuspec` at all.

To control which files are included, the `<contentFiles>` element specifies is a collection of `<files>` elements that identify the exact files include.

These files are specified with a set of attributes that describe how they should be used within the project system:

ATTRIBUTE	DESCRIPTION
include	(Required) The location of the file or files to include, subject to exclusions specified by the <code>exclude</code> attribute. The path is relative to the <code>contentFiles</code> folder unless an absolute path is specified. The wildcard character <code>*</code> is allowed, and the double wildcard <code>**</code> implies a recursive folder search.

ATTRIBUTE	DESCRIPTION
exclude	A semicolon-delimited list of files or file patterns to exclude from the <code>src</code> location. The wildcard character <code>*</code> is allowed, and the double wildcard <code>**</code> implies a recursive folder search.
buildAction	The build action to assign to the content item for MSBuild, such as <code>Content</code> , <code>None</code> , <code>Embedded Resource</code> , <code>Compile</code> , etc. The default is <code>Compile</code> .
copyToOutput	A Boolean indicating whether to copy content items to the build (or publish) output folder. The default is false.
flatten	A Boolean indicating whether to copy content items to a single folder in the build output (true), or to preserve the folder structure in the package (false). This flag only works when <code>copyToOutput</code> flag is set to true. The default is false.

When installing a package, NuGet applies the child elements of `<contentFiles>` from top to bottom. If multiple entries match the same file then all entries are applied. The top-most entry overrides the lower entries if there is a conflict for the same attribute.

Package folder structure

The package project should structure content using the following pattern:

```
/contentFiles/{codeLanguage}/{TxM}/{any?}
```

- `codeLanguages` may be `cs`, `vb`, `fs`, `any`, or the lowercase equivalent of a given `$(ProjectLanguage)`
- `TxM` is any legal target framework moniker that NuGet supports (see [Target frameworks](#)).
- Any folder structure may be appended to the end of this syntax.

For example:

```
Language- and framework-agnostic:
/contentFiles/any/any/config.xml

net45 content for all languages
/contentFiles/any/net45/config.xml

C#-specific content for net45 and up
/contentFiles/cs/net45/sample.cs
```

Empty folders can use `.` to opt out of providing content for certain combinations of language and TxM, for example:

```
/contentFiles/vb/any/code.vb
/contentFiles/cs/any/.
```

Example contentFiles section

```

<?xml version="1.0" encoding="utf-8"?>
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
    <metadata>
        ...
        <contentFiles>
            <!-- Embed image resources -->
            <files include="any/any/images/dnf.png" buildAction="EmbeddedResource" />
            <files include="any/any/images/ui.png" buildAction="EmbeddedResource" />

            <!-- Embed all image resources under contentFiles/cs/ -->
            <files include="cs/**/*.png" buildAction="EmbeddedResource" />

            <!-- Copy config.xml to the root of the output folder -->
            <files include="cs/uap/config/config.xml" buildAction="None" copyToOutput="true" flatten="true" />

            <!-- Copy run.cmd to the output folder and keep the directory structure -->
            <files include="cs/commands/run.cmd" buildAction="None" copyToOutput="true" flatten="false" />

            <!-- Include everything in the scripts folder except exe files -->
            <files include="cs/net45/scripts/*" exclude="**/*.exe" buildAction="None" copyToOutput="true" />
        </contentFiles>
        </metadata>
    </package>

```

Example nuspec files

A simple `.nuspec` that does not specify dependencies or files

```

<?xml version="1.0" encoding="utf-8"?>
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
    <metadata>
        <id>sample</id>
        <version>1.2.3</version>
        <authors>Kim Abercrombie, Franck Halmaert</authors>
        <description>Sample exists only to show a sample .nuspec file.</description>
        <language>en-US</language>
        <projectUrl>http://xunit.codeplex.com/</projectUrl>
        <license type="expression">MIT</license>
    </metadata>
</package>

```

A `.nuspec` with dependencies

```

<?xml version="1.0" encoding="utf-8"?>
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
    <metadata>
        <id>sample</id>
        <version>1.0.0</version>
        <authors>Microsoft</authors>
        <dependencies>
            <dependency id="another-package" version="3.0.0" />
            <dependency id="yet-another-package" version="1.0.0" />
        </dependencies>
    </metadata>
</package>

```

A `.nuspec` with files

```

<?xml version="1.0"?>
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
  <metadata>
    <id>routedebugger</id>
    <version>1.0.0</version>
    <authors>Jay Hamlin</authors>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>Route Debugger is a little utility I wrote...</description>
  </metadata>
  <files>
    <file src="bin\Debug\*.dll" target="lib" />
  </files>
</package>

```

A `.nuspec` with framework assemblies

```

<?xml version="1.0"?>
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
  <metadata>
    <id>PackageWithGacReferences</id>
    <version>1.0</version>
    <authors>Author here</authors>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>
      A package that has framework assemblyReferences depending
      on the target framework.
    </description>
    <frameworkAssemblies>
      <frameworkAssembly assemblyName="System.Web" targetFramework="net40" />
      <frameworkAssembly assemblyName="System.Net" targetFramework="net40-client, net40" />
      <frameworkAssembly assemblyName="Microsoft.Devices.Sensors" targetFramework="sl4-wp" />
      <frameworkAssembly assemblyName="System.Json" targetFramework="sl3" />
    </frameworkAssemblies>
  </metadata>
</package>

```

In this example, the following are installed for specific project targets:

- .NET4 -> `System.Web`, `System.Net`
- .NET4 Client Profile -> `System.Net`
- Silverlight 3 -> `System.Json`
- WindowsPhone -> `Microsoft.Devices.Sensors`

nuget.config reference

8/14/2019 • 8 minutes to read • [Edit Online](#)

NuGet behavior is controlled by settings in different `NuGet.Config` files as described in [Common NuGet configurations](#).

`nuget.config` is an XML file containing a top-level `<configuration>` node, which then contains the section elements described in this topic. Each section contains zero or more items. See the [examples config file](#). Setting names are case-insensitive, and values can use [environment variables](#).

config section

Contains miscellaneous configuration settings, which can be set using the `nuget config` command.

`dependencyVersion` and `repositoryPath` apply only to projects using `packages.config`. `globalPackagesFolder` applies only to projects using the PackageReference format.

KEY	VALUE
<code>dependencyVersion</code> (<code>packages.config</code> only)	The default <code>DependencyVersion</code> value for package install, restore, and update, when the <code>-DependencyVersion</code> switch is not specified directly. This value is also used by the NuGet Package Manager UI. Values are <code>Lowest</code> , <code>HighestPatch</code> , <code>HighestMinor</code> , <code>Highest</code> .
<code>globalPackagesFolder</code> (projects using PackageReference only)	The location of the default global packages folder. The default is <code>%userprofile%\.nuget\packages</code> (Windows) or <code>~/.nuget/packages</code> (Mac/Linux). A relative path can be used in project-specific <code>nuget.config</code> files. This setting is overridden by the <code>NUGET_PACKAGES</code> environment variable, which takes precedence.
<code>repositoryPath</code> (<code>packages.config</code> only)	The location in which to install NuGet packages instead of the default <code>\$(Solutiondir)/packages</code> folder. A relative path can be used in project-specific <code>nuget.config</code> files. This setting is overridden by the <code>NUGET_PACKAGES</code> environment variable, which takes precedence.
<code>defaultPushSource</code>	Identifies the URL or path of the package source that should be used as the default if no other package sources are found for an operation.
<code>http_proxy</code> <code>http_proxy.user</code> <code>http_proxy.password</code> <code>no_proxy</code>	Proxy settings to use when connecting to package sources; <code>http_proxy</code> should be in the format <code>http://<username>:<password>@<domain></code> . Passwords are encrypted and cannot be added manually. For <code>no_proxy</code> , the value is a comma-separated list of domains the bypass the proxy server. You can alternately use the <code>http_proxy</code> and <code>no_proxy</code> environment variables for those values. For additional details, see NuGet proxy settings (skolima.blogspot.com).

KEY	VALUE
signatureValidationMode	Specifies the validation mode used to verify package signatures for package install, and restore. Values are <code>accept</code> , <code>require</code> . Defaults to <code>accept</code> .

Example:

```
<config>
  <add key="dependencyVersion" value="Highest" />
  <add key="globalPackagesFolder" value="c:\packages" />
  <add key="repositoryPath" value="c:\installed_packages" />
  <add key="http_proxy" value="http://company-squid:3128@contoso.com" />
  <add key="signatureValidationMode" value="require" />
</config>
```

bindingRedirects section

Configures whether NuGet does automatic binding redirects when a package is installed.

KEY	VALUE
skip	A Boolean indicating whether to skip automatic binding redirects. The default is false.

Example:

```
<bindingRedirects>
  <add key="skip" value="True" />
</bindingRedirects>
```

packageRestore section

Controls package restore during builds.

KEY	VALUE
enabled	A Boolean indicating whether NuGet can perform automatic restore. You can also set the <code>EnableNuGetPackageRestore</code> environment variable with a value of <code>True</code> instead of setting this key in the config file.
automatic	A Boolean indicating whether NuGet should check for missing packages during a build.

Example:

```
<packageRestore>
  <add key="enabled" value="true" />
  <add key="automatic" value="true" />
</packageRestore>
```

solution section

Controls whether the `packages` folder of a solution is included in source control. This section works only in `nuget.config` files in a solution folder.

KEY	VALUE
disableSourceControlIntegration	A Boolean indicating whether to ignore the packages folder when working with source control. The default value is false.

Example:

```
<solution>
  <add key="disableSourceControlIntegration" value="true" />
</solution>
```

Package source sections

The `packageSources`, `packageSourceCredentials`, `apikeys`, `activePackageSource`, `disabledPackageSources` and `trustedSigners` all work together to configure how NuGet works with package repositories during install, restore, and update operations.

The [nuget sources command](#) is generally used to manage these settings, except for `apikeys` which is managed using the [nuget setapikey command](#), and `trustedSigners` which is managed using the [nuget trusted-signers command](#).

Note that the source URL for nuget.org is <https://api.nuget.org/v3/index.json>.

packageSources

Lists all known package sources. The order is ignored during restore operations and with any project using the `PackageReference` format. NuGet respects the order of sources for install and update operations with projects using `packages.config`.

KEY	VALUE
(name to assign to the package source)	The path or URL of the package source.

Example:

```
<packageSources>
  <add key="nuget.org" value="https://api.nuget.org/v3/index.json" protocolVersion="3" />
  <add key="Contoso" value="https://contoso.com/packages/" />
  <add key="Test Source" value="c:\packages" />
</packageSources>
```

packageSourceCredentials

Stores usernames and passwords for sources, typically specified with the `-username` and `-password` switches with [nuget sources](#). Passwords are encrypted by default unless the `-storepasswordinplaintext` option is also used.

KEY	VALUE
username	The user name for the source in plain text.
password	The encrypted password for the source.

KEY	VALUE
cleartextpassword	The unencrypted password for the source.

Example:

In the config file, the `<packageSourceCredentials>` element contains child nodes for each applicable source name (spaces in the name are replaced with `_x0020_`). That is, for sources named "Contoso" and "Test Source", the config file contains the following when using encrypted passwords:

```
<packageSourceCredentials>
  <Contoso>
    <add key="Username" value="user@contoso.com" />
    <add key="Password" value="..." />
  </Contoso>
  <Test_x0020_Source>
    <add key="Username" value="user" />
    <add key="Password" value="..." />
  </Test_x0020_Source>
</packageSourceCredentials>
```

When using unencrypted passwords:

```
<packageSourceCredentials>
  <Contoso>
    <add key="Username" value="user@contoso.com" />
    <add key="ClearTextPassword" value="33f!!lloppa" />
  </Contoso>
  <Test_x0020_Source>
    <add key="Username" value="user" />
    <add key="ClearTextPassword" value="hal+9ooo_da!sY" />
  </Test_x0020_Source>
</packageSourceCredentials>
```

apikeys

Stores keys for sources that use API key authentication, as set with the [nuget setapikey](#) command.

KEY	VALUE
(source URL)	The encrypted API key.

Example:

```
<apikeys>
  <add key="https://MyRepo/ES/api/v2/package" value="encrypted_api_key" />
</apikeys>
```

disabledPackageSources

Identified currently disabled sources. May be empty.

KEY	VALUE
(name of source)	A Boolean indicating whether the source is disabled.

Example:

```

<disabledPackageSources>
  <add key="Contoso" value="true" />
</disabledPackageSources>

<!-- Empty list -->
<disabledPackageSources />

```

activePackageSource

(2.x only; deprecated in 3.x+)

Identifies to the currently active source or indicates the aggregate of all sources.

KEY	VALUE
(name of source) or <code>All</code>	If key is the name of a source, the value is the source path or URL. If <code>All</code> , value should be <code>(Aggregate source)</code> to combine all package sources that are not otherwise disabled.

Example:

```

<activePackageSource>
  <!-- Only one active source-->
  <add key="nuget.org" value="https://api.nuget.org/v3/index.json" />

  <!-- All non-disabled sources are active -->
  <add key="All" value="(Aggregate source)" />
</activePackageSource>

```

trustedSigners section

Stores trusted signers used to allow package while installing or restoring. This list cannot be empty when the user sets `signatureValidationMode` to `require`.

This section can be updated with the `nuget trusted-signers` command.

Schema:

A trusted signer has a collection of `certificate` items that enlist all the certificates that identify a given signer. A trusted signer can be either an `Author` or a `Repository`.

A trusted `repository` also specifies the `serviceIndex` for the repository (which has to be a valid `https` uri) and can optionally specify a semi-colon delimited list of `owners` to restrict even more who is trusted from that specific repository.

The supported hash algorithms used for a certificate fingerprint are `SHA256`, `SHA384` and `SHA512`.

If a `certificate` specifies `allowUntrustedRoot` as `true` the given certificate is allowed to chain to an untrusted root while building the certificate chain as part of the signature verification.

Example:

```

<trustedSigners>
  <author name="microsoft">
    <certificate fingerprint="3F9001EA83C560D712C24CF213C3D312CB3BFF51EE89435D3430BD06B5D0EECE"
hashAlgorithm="SHA256" allowUntrustedRoot="false" />
  </author>
  <repository name="nuget.org" serviceIndex="https://api.nuget.org/v3/index.json">
    <certificate fingerprint="0E5F38F57DC1BCC806D8494F4F90FBCEDD988B46760709CBEEC6F4219AA6157D"
hashAlgorithm="SHA256" allowUntrustedRoot="false" />
    <owners>microsoft;aspnet;nuget</owners>
  </repository>
</trustedSigners>

```

fallbackPackageFolders section

(3.5+) Provides a way to preinstall packages so that no work needs to be done if the package is found in the fallback folders. Fallback package folders have the exact same folder and file structure as the global package folder: `.nupkg` is present, and all files are extracted.

The lookup logic for this configuration is:

- Look in global package folder to see if the package/version is already downloaded.
- Look in the fallback folders for a package/version match.

If either lookup is successful, then no download is necessary.

If a match is not found, then NuGet checks file sources, and then http sources, and then it downloads the packages.

KEY	VALUE
(name of fallback folder)	Path to fallback folder.

Example:

```

<fallbackPackageFolders>
  <add key="XYZ Offline Packages" value="C:\somePath\someFolder\"/>
</fallbackPackageFolders>

```

packageManagement section

Sets the default package management format, either `packages.config` or `PackageReference`. SDK-style projects always use `PackageReference`.

KEY	VALUE
format	A Boolean indicating the default package management format. If <code>1</code> , format is <code>PackageReference</code> . If <code>0</code> , format is <code>packages.config</code> .
disabled	A Boolean indicating whether to show the prompt to select a default package format on first package install. <code>False</code> hides the prompt.

Example:

```
<packageManagement>
  <add key="format" value="1" />
  <add key="disabled" value="False" />
</packageManagement>
```

Using environment variables

You can use environment variables in `nuget.config` values (NuGet 3.4+) to apply settings at run time.

For example, if the `HOME` environment variable on Windows is set to `c:\users\username`, then the value of `%HOME%\NuGetRepository` in the configuration file resolves to `c:\users\username\NuGetRepository`.

Similarly, if `HOME` on Mac/Linux is set to `/home/myStuff`, then `%HOME%\NuGetRepository` in the configuration file resolves to `/home/myStuff/NuGetRepository`.

If an environment variable is not found, NuGet uses the literal value from the configuration file.

Example config file

Below is an example `nuget.config` file that illustrates a number of settings:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <config>
    <!--
        Used to specify the default location to expand packages.
        See: nuget.exe help install
        See: nuget.exe help update

        In this example, %PACKAGEHOME% is an environment variable. On Mac/Linux,
        use $PACKAGE_HOME/External as the value.
    -->
    <add key="repositoryPath" value="%PACKAGEHOME%\External" />

    <!--
        Used to specify default source for the push command.
        See: nuget.exe help push
    -->

    <add key="defaultPushSource" value="https://MyRepo/ES/api/v2/package" />

    <!-- Proxy settings -->
    <add key="http_proxy" value="host" />
    <add key="http_proxy.user" value="username" />
    <add key="http_proxy.password" value="encrypted_password" />
  </config>

  <packageRestore>
    <!-- Allow NuGet to download missing packages -->
    <add key="enabled" value="True" />

    <!-- Automatically check for missing packages during build in Visual Studio -->
    <add key="automatic" value="True" />
  </packageRestore>

  <!--
      Used to specify the default Sources for list, install and update.
      See: nuget.exe help list
      See: nuget.exe help install
      See: nuget.exe help update
  -->
  <packageSources>
    <add key="NuGet official package source" value="https://api.nuget.org/v3/index.json" />
```

```
    <add key="MyRepo - ES" value="https://MyRepo/ES/nuget" />
  </packageSources>

  <!-- Used to store credentials -->
  <packageSourceCredentials />

  <!-- Used to disable package sources -->
  <disabledPackageSources />

  <!--
    Used to specify default API key associated with sources.
    See: nuget.exe help setApiKey
    See: nuget.exe help push
    See: nuget.exe help mirror
  -->
  <apikeys>
    <add key="https://MyRepo/ES/api/v2/package" value="encrypted_api_key" />
  </apikeys>

  <!--
    Used to specify trusted signers to allow during signature verification.
    See: nuget.exe help trusted-signers
  -->
  <trustedSigners>
    <author name="microsoft">
      <certificate fingerprint="3F9001EA83C560D712C24CF213C3D312CB3BFF51EE89435D3430BD06B5D0EECE"
hashAlgorithm="SHA256" allowUntrustedRoot="false" />
    </author>
    <repository name="nuget.org" serviceIndex="https://api.nuget.org/v3/index.json">
      <certificate fingerprint="0E5F38F57DC1BCC806D8494F4F90FBCEDD988B46760709CBEEC6F4219AA6157D"
hashAlgorithm="SHA256" allowUntrustedRoot="false" />
      <owners>microsoft;aspnet;nuget</owners>
    </repository>
  </trustedSigners>
</configuration>
```

Target frameworks

10/15/2019 • 5 minutes to read • [Edit Online](#)

NuGet uses target framework references in a variety of places to specifically identify and isolate framework-dependent components of a package:

- **project file:** For SDK-style projects, the `.csproj` contains the target framework references.
- **.nuspec manifest:** A package can indicate distinct packages to be included in a project depending on the project's target framework.
- **.nupkg folder name:** The folders inside a package's `lib` folder can be named according to the target framework, each of which contains the DLLs and other content appropriate to that framework.
- **packages.config:** The `targetframework` attribute of a dependency specifies the variant of a package to install.

NOTE

The NuGet client source code that calculates the tables below is found in the following locations:

- Supported framework names: [FrameworkConstants.cs](#)
- Framework precedence and mapping: [DefaultFrameworkMappings.cs](#)

Supported frameworks

A framework is typically referenced by a short target framework moniker or TFM. In .NET Standard this is also generalized to *TxM* to allow a single reference to multiple frameworks.

The NuGet clients support the frameworks in the table below. Equivalents are shown within brackets `[]`. Note that some tools, such as `dotnet`, might use variations of canonical TFMs in some files. For example,

`dotnet pack` uses `.NETCoreApp2.0` in a `.nuspec` file rather than `netcoreapp2.0`. The various NuGet client tools handle these variations properly, but you should always use canonical TFMs when editing files directly.

NAME	ABBREVIATION	TFMS/TXMS
.NET Framework	net	net11
		net20
		net35
		net40
		net403
		net45
		net451
		net452
		net46

NAME	ABBREVIATION	TFMS/TXMS
		net461
		net462
		net47
		net471
		net472
		net48
Microsoft Store (Windows Store)	netcore	netcore [netcore45]
		netcore45 [win, win8]
		netcore451 [win81]
		netcore50
.NET MicroFramework	netmf	netmf
Windows	win	win [win8, netcore45]
		win8 [netcore45, win]
		win81 [netcore451]
		win10 (not supported by Windows 10 Platform)
Silverlight	sl	sl4
		sl5
Windows Phone (SL)	wp	wp [wp7]
		wp7
		wp75
		wp8
		wp81
Windows Phone (UWP)		wpa81
Universal Windows Platform	uap	uap [uap10.0]
		uap10.0

NAME	ABBREVIATION	TFMS/TXMS
		uap10.0.xxxxx (where 10.0.xxxxx is the target platform min version of the consuming app)
.NET Standard	netstandard	netstandard1.0
		netstandard1.1
		netstandard1.2
		netstandard1.3
		netstandard1.4
		netstandard1.5
		netstandard1.6
		netstandard2.0
.NET Core App	netcoreapp	netcoreapp1.0
		netcoreapp1.1
		netcoreapp2.0
		netcoreapp2.1
		netcoreapp2.2
		netcoreapp3.0
Tizen	tizen	tizen3
		tizen4

Deprecated frameworks

The following frameworks are deprecated. Packages targeting these frameworks should migrate to the indicated replacements.

DEPRECATED FRAMEWORK	REPLACEMENT
aspnet50	netcoreapp
aspnetcore50	
dnxcore50	
dnx	

DEPRECATED FRAMEWORK	REPLACEMENT
dnx45	
dnx451	
dnx452	
dotnet	netstandard
dotnet50	
dotnet51	
dotnet52	
dotnet53	
dotnet54	
dotnet55	
dotnet56	
winrt	win

Precedence

A number of frameworks are related to and compatible with one another, but not necessarily equivalent:

FRAMEWORK	CAN USE
uap (Universal Windows Platform)	win81
	wpa81
	netcore50
win (Microsoft Store)	winrt

NET Standard

[.NET Standard](#) simplifies references between binary-compatible frameworks, allowing a single target framework to reference a combination of others. (For background, see the [.NET Primer](#).)

The [NuGet Get Nearest Framework Tool](#) simulates what NuGet uses to select one framework from many available framework assets in a package based on the project's framework.

The `dotnet` series of monikers should be used in NuGet 3.3 and earlier; the `netstandard` moniker syntax should be used in v3.4 and later.

Portable Class Libraries

WARNING

PCLs are not recommended. Although PCLs are supported, package authors should support netstandard instead. The .NET Platform Standard is an evolution of PCLs and represents binary portability across platforms using a single moniker that isn't tied to a static library like *portable-a+b+c* monikers.

To define a target framework that refers to multiple child-target-frameworks, the `portable` keyword is used to prefix the list of referenced frameworks. Avoid artificially including extra frameworks that are not directly compiled against because it can lead to unintended side-effects in those frameworks.

Additional frameworks defined by third parties provide compatibility with other environments that are accessible in this manner. Additionally, there are shorthand profile numbers that are available to reference these combinations of related frameworks as `Profile#`, but this is not a recommended practice to use these numbers as it reduces the readability of the folders and `.nuspec`.

PROFILE #	FRAMEWORKS	FULL NAME	.NET STANDARD
Profile2	.NETFramework 4.0	portable-net40+win8+sl4+wp7	
	Windows 8.0		
	Silverlight 4.0		
	WindowsPhone 7.0		
Profile3	.NETFramework 4.0	portable-net40+sl4	
	Silverlight 4.0		
Profile4	.NETFramework 4.5	portable-net45+sl4+win8+wp7	
	Silverlight 4.0		
	Windows 8.0		
	WindowsPhone 7.0		
Profile5	.NETFramework 4.0	portable-net40+win8	
	Windows 8.0		
Profile6	.NETFramework 4.0.3	portable-net403+win8	
	Windows 8.0		
Profile7	.NETFramework 4.5	portable-net45+win8	netstandard1.1
	Windows 8.0		

PROFILE #	FRAMEWORKS	FULL NAME	.NET STANDARD
Profile14	.NETFramework 4.0	portable-net40+sl5	
	Silverlight 5.0		
Profile18	.NETFramework 4.0.3	portable-net403+sl4	
	Silverlight 4.0		
Profile19	.NETFramework 4.0.3	portable-net403+sl5	
	Silverlight 5.0		
Profile23	.NETFramework 4.5	portable-net45+sl4	
	Silverlight 4.0		
Profile24	.NETFramework 4.5	portable-net45+sl5	
	Silverlight 5.0		
Profile31	Windows 8.1	portable-win81+wp81	netstandard1.0
	WindowsPhone 8.1 (SL)		
Profile32	Windows 8.1	portable-win81+wpa81	netstandard1.2
	WindowsPhone 8.1 (UWP)		
Profile36	.NETFramework 4.0	portable-net40+sl4+win8+wp8	
	Silverlight 4.0		
	Windows 8.0		
	WindowsPhone 8.0 (SL)		
Profile37	.NETFramework 4.0	portable-net40+sl5+win8	
	Silverlight 5.0		
	Windows 8.0		
Profile41	.NETFramework 4.0.3	portable-net403+sl4+win8	
	Silverlight 4.0		
	Windows 8.0		
Profile42	.NETFramework 4.0.3	portable-net403+sl5+win8	

PROFILE #	FRAMEWORKS	FULL NAME	.NET STANDARD
	Silverlight 5.0		
	Windows 8.0		
Profile44	.NETFramework 4.5.1	portable-net451+win81	netstandard1.2
	Windows 8.1		
Profile46	.NETFramework 4.5	portable-net45+sl4+win8	
	Silverlight 4.0		
	Windows 8.0		
Profile47	.NETFramework 4.5	portable-net45+sl5+win8	
	Silverlight 5.0		
	Windows 8.0		
Profile49	.NETFramework 4.5	portable-net45+wp8	netstandard1.0
	WindowsPhone 8.0 (SL)		
Profile78	.NETFramework 4.5	portable-net45+win8+wp8	netstandard1.0
	Windows 8.0		
	WindowsPhone 8.0 (SL)		
Profile84	WindowsPhone 8.1	portable-wp81+wpa81	netstandard1.0
	WindowsPhone 8.1 (UWP)		
Profile88	.NETFramework 4.0	portable-net40+sl4+win8+wp75	
	Silverlight 4.0		
	Windows 8.0		
	WindowsPhone 7.5		
Profile92	.NETFramework 4.0	portable-net40+win8+wpa81	
	Windows 8.0		
	WindowsPhone 8.1 (UWP)		

PROFILE #	FRAMEWORKS	FULL NAME	.NET STANDARD
Profile95	.NETFramework 4.0.3	portable-net403+sl4+win8+wp7	
	Silverlight 4.0		
	Windows 8.0		
	WindowsPhone 7.0		
Profile96	.NETFramework 4.0.3	portable-net403+sl4+win8+wp75	
	Silverlight 4.0		
	Windows 8.0		
	WindowsPhone 7.5		
Profile102	.NETFramework 4.0.3	portable-net403+win8+wpa81	
	Windows 8.0		
	WindowsPhone 8.1 (UWP)		
Profile104	.NETFramework 4.5	portable-net45+sl4+win8+wp75	
	Silverlight 4.0		
	Windows 8.0		
	WindowsPhone 7.5		
Profile111	.NETFramework 4.5	portable-net45+win8+wpa81	netstandard1.1
	Windows 8.0		
	WindowsPhone 8.1 (UWP)		
Profile136	.NETFramework 4.0	portable-net40+sl5+win8+wp8	
	Silverlight 5.0		
	Windows 8.0		
	WindowsPhone 8.0 (SL)		

PROFILE #	FRAMEWORKS	FULL NAME	.NET STANDARD
Profile143	.NETFramework 4.0.3	portable-net403+sl4+win8+wp8	
	Silverlight 4.0		
	Windows 8.0		
	WindowsPhone 8.0 (SL)		
Profile147	.NETFramework 4.0.3	portable-net403+sl5+win8+wp8	
	Silverlight 5.0		
	Windows 8.0		
	WindowsPhone 8.0 (SL)		
Profile151	NETFramework 4.5.1	portable-net451+win81+wpa81	netstandard1.2
	Windows 8.1		
	WindowsPhone 8.1 (UWP)		
Profile154	.NETFramework 4.5	portable-net45+sl4+win8+wp8	
	Silverlight 4.0		
	Windows 8.0		
	WindowsPhone 8.0 (SL)		
Profile157	Windows 8.1	portable-win81+wp81+wpa81	netstandard1.0
	WindowsPhone 8.1 (SL)		
	WindowsPhone 8.1 (UWP)		
Profile158	.NETFramework 4.5	portable-net45+sl5+win8+wp8	
	Silverlight 5.0		
	Windows 8.0		
	WindowsPhone 8.0 (SL)		

PROFILE #	FRAMEWORKS	FULL NAME	.NET STANDARD
Profile225	.NETFramework 4.0	portable-net40+sl5+win8+wpa81	
	Silverlight 5.0		
	Windows 8.0		
	WindowsPhone 8.1 (UWP)		
Profile240	.NETFramework 4.0.3	portable-net403+sl5+win8+wpa8	
	Silverlight 5.0		
	Windows 8.0		
	WindowsPhone 8.1 (UWP)		
Profile255	.NETFramework 4.5	portable-net45+sl5+win8+wpa81	
	Silverlight 5.0		
	Windows 8.0		
	WindowsPhone 8.1 (UWP)		
Profile259	.NETFramework 4.5	portable-net45+win8+wpa81+wp8	netstandard1.0
	Windows 8.0		
	WindowsPhone 8.1 (UWP)		
	WindowsPhone 8.0 (SL)		
Profile328	.NETFramework 4.0	portable-net40+sl5+win8+wpa81+wp8	
	Silverlight 5.0		
	Windows 8.0		
	WindowsPhone 8.1 (UWP)		
	WindowsPhone 8.0 (SL)		
Profile336	.NETFramework 4.0.3	portable-net403+sl5+win8+wpa81+wp8	

PROFILE #	FRAMEWORKS	FULL NAME	.NET STANDARD
	Silverlight 5.0		
	Windows 8.0		
	WindowsPhone 8.1 (UWP)		
	WindowsPhone 8.0 (SL)		
Profile344	.NETFramework 4.5	portable-net45+sl5+win8+wpa81+wp8	
	Silverlight 5.0		
	Windows 8.0		
	WindowsPhone 8.1 (UWP)		
	WindowsPhone 8.0 (SL)		

Additionally, NuGet packages targeting Xamarin can use additional Xamarin-defined frameworks. See [Creating NuGet packages for Xamarin](#).

NAME	DESCRIPTION	.NET STANDARD
monoandroid	Mono Support for Android OS	netstandard1.4
monotouch	Mono Support for iOS	netstandard1.4
monomac	Mono Support for OSX	netstandard1.4
xamarinios	Support for Xamarin for iOS	netstandard1.4
xamarinmac	Supports for Xamarin for Mac	netstandard1.4
xamarinpsthree	Support for Xamarin on Playstation 3	netstandard1.4
xamarinpsthreefour	Support for Xamarin on Playstation 4	netstandard1.4
xamarinpsvita	Support for Xamarin on PS Vita	netstandard1.4
xamarinwatchos	Xamarin for Watch OS	netstandard1.4
xamarintvos	Xamarin for TV OS	netstandard1.4
xamarinxboxthreesixty	Xamarin for XBox 360	netstandard1.4
xamarinxboxone	Xamarin for XBox One	netstandard1.4

NOTE

Stephen Cleary has created a tool that lists the supported PCLs, which you can find on his post, [Framework profiles in .NET](#).

NuGet pack and restore as MSBuild targets

11/14/2019 • 13 minutes to read • [Edit Online](#)

NuGet 4.0+

With the [PackageReference](#) format, NuGet 4.0+ can store all manifest metadata directly within a project file rather than using a separate `.nuspec` file.

With MSBuild 15.1+, NuGet is also a first-class MSBuild citizen with the `pack` and `restore` targets as described below. These targets allow you to work with NuGet as you would with any other MSBuild task or target. For instructions creating a NuGet package using MSBuild, see [Create a NuGet package using MSBuild](#). (For NuGet 3.x and earlier, you use the `pack` and `restore` commands through the NuGet CLI instead.)

Target build order

Because `pack` and `restore` are MSBuild targets, you can access them to enhance your workflow. For example, let's say you want to copy your package to a network share after packing it. You can do that by adding the following in your project file:

```
<Target Name="CopyPackage" AfterTargets="Pack">
  <Copy
    SourceFiles="$(OutputPath)..\$(PackageId).$(PackageVersion).nupkg"
    DestinationFolder="\myshare\packageshare\"
  />
</Target>
```

Similarly, you can write an MSBuild task, write your own target and consume NuGet properties in the MSBuild task.

NOTE

`$(OutputPath)` is relative and expects that you are running the command from the project root.

pack target

For .NET Standard projects using the [PackageReference](#) format, using `msbuild -t:pack` draws inputs from the project file to use in creating a NuGet package.

The table below describes the MSBuild properties that can be added to a project file within the first `<PropertyGroup>` node. You can make these edits easily in Visual Studio 2017 and later by right-clicking the project and selecting **Edit {project_name}** on the context menu. For convenience the table is organized by the equivalent property in a [.nuspec](#) file.

Note that the `Owners` and `Summary` properties from `.nuspec` are not supported with MSBuild.

ATTRIBUTE/NUSPEC VALUE	MSBUILD PROPERTY	DEFAULT	NOTES
<code>Id</code>	<code>PackagId</code>	<code>AssemblyName</code>	<code>\$(AssemblyName)</code> from MSBuild

ATTRIBUTE/NUSPEC VALUE	MSBUILD PROPERTY	DEFAULT	NOTES
Version	PackageVersion	Version	This is semver compatible, for example "1.0.0", "1.0.0-beta", or "1.0.0-beta-00345"
VersionPrefix	PackageVersionPrefix	empty	Setting PackageVersion overwrites PackageVersionPrefix
VersionSuffix	PackageVersionSuffix	empty	\$(VersionSuffix) from MSBuild. Setting PackageVersion overwrites PackageVersionSuffix
Authors	Authors	Username of the current user	
Owners	N/A	Not present in NuSpec	
Title	Title	The PackageId	
Description	Description	"Package Description"	
Copyright	Copyright	empty	
RequireLicenseAcceptance	PackageRequireLicenseAcceptance	false	
license	PackageLicenseExpression	empty	Corresponds to <code><license type="expression"></code>
license	PackageLicenseFile	empty	Corresponds to <code><license type="file"></code> . You need to explicitly pack the referenced license file.
LicenseUrl	PackageLicenseUrl	empty	<code>PackageLicenseUrl</code> is deprecated, use the <code>PackageLicenseExpression</code> or <code>PackageLicenseFile</code> property
ProjectUrl	PackageProjectUrl	empty	
Icon	PackageIcon	empty	You need to explicitly pack the referenced icon image file.
IconUrl	PackageIconUrl	empty	For the best downlevel experience, <code>PackageIconUrl</code> should be specified in addition to <code>PackageIcon</code> . Longer term, <code>PackageIconUrl</code> will be deprecated.

ATTRIBUTE/NUSPEC VALUE	MSBUILD PROPERTY	DEFAULT	NOTES
Tags	PackageTags	empty	Tags are semi-colon delimited.
ReleaseNotes	PackageReleaseNotes	empty	
Repository/Url	RepositoryUrl	empty	Repository URL used to clone or retrieve source code. Example: https://github.com/NuGet/NuGet.Client.git
Repository/Type	RepositoryType	empty	Repository type. Examples: <i>git</i> , <i>tfs</i> .
Repository/Branch	RepositoryBranch	empty	Optional repository branch information. <i>RepositoryUrl</i> must also be specified for this property to be included. Example: <i>master</i> (NuGet 4.7.0+)
Repository/Commit	RepositoryCommit	empty	Optional repository commit or changeset to indicate which source the package was built against. <i>RepositoryUrl</i> must also be specified for this property to be included. Example: <i>0e4d1b598f350b3dc675018d539114d1328189ef</i> (NuGet 4.7.0+)
PackageType	<pre><PackageType>DotNetCliTool,1.0.0.0;Dependency,2.0.0.0</PackageType></pre>		
Summary	Not supported		

pack target inputs

- IsPackable
- SuppressDependenciesWhenPacking
- PackageVersion
- PackageId
- Authors
- Description
- Copyright
- PackageRequireLicenseAcceptance
- DevelopmentDependency
- PackageLicenseExpression
- PackageLicenseFile
- PackageLicenseUrl
- PackageProjectUrl
- PackageIconUrl

- `PackageReleaseNotes`
- `PackageTags`
- `PackageOutputPath`
- `IncludeSymbols`
- `IncludeSource`
- `PackageTypes`
- `IsTool`
- `RepositoryUrl`
- `RepositoryType`
- `RepositoryBranch`
- `RepositoryCommit`
- `NoPackageAnalysis`
- `MinClientVersion`
- `IncludeBuildOutput`
- `IncludeContentInPack`
- `BuildOutputTargetFolder`
- `ContentTargetFolders`
- `NuspecFile`
- `NuspecBasePath`
- `NuspecProperties`

pack scenarios

Suppress dependencies

To suppress package dependencies from generated NuGet package, set `SuppressDependenciesWhenPacking` to `true` which will allow skipping all the dependencies from generated nupkg file.

PackageIconUrl

`PackageIconUrl` will be deprecated in favor of the new `PackageIcon` property.

Starting with NuGet 5.3 & Visual Studio 2019 version 16.3, `pack` will raise [NU5048](#) warning if the package metadata only specifies `PackageIconUrl`.

PackageIcon

TIP

You should specify both `PackageIcon` and `PackageIconUrl` to maintain backward compatibility with clients and sources that do not yet support `PackageIcon`. Visual Studio will support `PackageIcon` for packages coming from a folder-based source in a future release.

Packing an icon image file

When packing an icon image file, you need to use `PackageIcon` property to specify the package path, relative to the root of the package. In addition, you need to make sure that the file is included in the package. Image file size is limited to 1 MB. Supported file formats include JPEG and PNG. We recommend an image resolution of 64x64.

For example:

```

<PropertyGroup>
  ...
  <PackageIcon>icon.png</PackageIcon>
  ...
</PropertyGroup>

<ItemGroup>
  ...
  <None Include="images\icon.png" Pack="true" PackagePath="\"/>
  ...
</ItemGroup>

```

Package Icon sample.

For the nuspec equivalent, take a look at [nuspec reference for icon](#).

Output assemblies

`nuget pack` copies output files with extensions `.exe`, `.dll`, `.xml`, `.winmd`, `.json`, and `.pri`. The output files that are copied depend on what MSBuild provides from the `BuiltOutputProjectGroup` target.

There are two MSBuild properties that you can use in your project file or command line to control where output assemblies go:

- `IncludeBuildOutput` : A boolean that determines whether the build output assemblies should be included in the package.
- `BuildOutputTargetFolder` : Specifies the folder in which the output assemblies should be placed. The output assemblies (and other output files) are copied into their respective framework folders.

Package references

See [Package References in Project Files](#).

Project to project references

Project to project references are considered by default as nuget package references, for example:

```
<ProjectReference Include="..\UwpLibrary2\UwpLibrary2.csproj"/>
```

You can also add the following metadata to your project reference:

```

<IncludeAssets>
<ExcludeAssets>
<PrivateAssets>

```

Including content in a package

To include content, add extra metadata to the existing `<Content>` item. By default everything of type "Content" gets included in the package unless you override with entries like the following:

```

<Content Include="..\win7-x64\libuv.txt">
  <Pack>false</Pack>
</Content>

```

By default, everything gets added to the root of the `content` and `contentFiles\any\<target_framework>` folder within a package and preserves the relative folder structure, unless you specify a package path:

```
<Content Include="..\win7-x64\libuv.txt">
  <Pack>true</Pack>
  <PackagePath>content\myfiles\</PackagePath>
</Content>
```

If you want to copy all your content to only a specific root folder(s) (instead of `content` and `contentFiles` both), you can use the MSBuild property `ContentTargetFolders`, which defaults to "content;contentFiles" but can be set to any other folder names. Note that just specifying "contentFiles" in `ContentTargetFolders` puts files under `contentFiles\any\<target_framework>` or `contentFiles\<language>\<target_framework>` based on `buildAction`.

`PackagePath` can be a semicolon-delimited set of target paths. Specifying an empty package path would add the file to the root of the package. For example, the following adds `libuv.txt` to `content\myfiles`, `content\samples`, and the package root:

```
<Content Include="..\win7-x64\libuv.txt">
  <Pack>true</Pack>
  <PackagePath>content\myfiles;content\sample;;</PackagePath>
</Content>
```

There is also an MSBuild property `$(IncludeContentInPack)`, which defaults to `true`. If this is set to `false` on any project, then the content from that project are not included in the nuget package.

Other pack specific metadata that you can set on any of the above items includes `<PackageCopyToOutput>` and `<PackageFlatten>` which sets `CopyToOutput` and `Flatten` values on the `contentFiles` entry in the output nuspec.

NOTE

Apart from Content items, the `<Pack>` and `<PackagePath>` metadata can also be set on files with a build action of Compile, EmbeddedResource, ApplicationDefinition, Page, Resource, SplashScreen, DesignData, DesignWithDataWithDesignTimeCreateableTypes, CodeAnalysisDictionary, AndroidAsset, AndroidResource, BundleResource or None.

For pack to append the filename to your package path when using globbing patterns, your package path must end with the folder separator character, otherwise the package path is treated as the full path including the file name.

IncludeSymbols

When using `MSBuild -t:pack -p:IncludeSymbols=true`, the corresponding `.pdb` files are copied along with other output files (`.dll`, `.exe`, `.winmd`, `.xml`, `.json`, `.pri`). Note that setting `IncludeSymbols=true` creates a regular package and a symbols package.

IncludeSource

This is the same as `IncludeSymbols`, except that it copies source files along with `.pdb` files as well. All files of type `Compile` are copied over to `src\<ProjectName>\` preserving the relative path folder structure in the resulting package. The same also happens for source files of any `ProjectReference` which has `TreatAsPackageReference` set to `false`.

If a file of type `Compile`, is outside the project folder, then it's just added to `src\<ProjectName>\`.

Packing a license expression or a license file

When using a license expression, the `PackageLicenseExpression` property should be used. [License expression sample](#).

```
<PropertyGroup>
  <PackageLicenseExpression>MIT</PackageLicenseExpression>
</PropertyGroup>
```

[Learn more about license expressions and licenses that are accepted by NuGet.org.](#)

When packing a license file, you need to use `PackageLicenseFile` property to specify the package path, relative to the root of the package. In addition, you need to make sure that the file is included in the package. For example:

```
<PropertyGroup>
  <PackageLicenseFile>LICENSE.txt</PackageLicenseFile>
</PropertyGroup>

<ItemGroup>
  <None Include="licenses\LICENSE.txt" Pack="true" PackagePath="" />
</ItemGroup>
```

[License file sample.](#)

IsTool

When using `MSBuild -t:pack -p:IsTool=true`, all output files, as specified in the [Output Assemblies](#) scenario, are copied to the `tools` folder instead of the `lib` folder. Note that this is different from a `DotNetCliTool` which is specified by setting the `PackageType` in `.csproj` file.

Packing using a .nuspec

Although it is recommended that you [include all the properties](#) that are usually in the `.nuspec` file in the project file instead, you can choose to use a `.nuspec` file to pack your project. For a non-SDK-style project that uses `PackageReference`, you must import `NuGet.Build.Tasks.Pack.targets` so that the pack task can be executed. You still need to restore the project before you can pack a nuspec file. (An SDK-style project includes the pack targets by default.)

The target framework of the project file is irrelevant and not used when packing a nuspec. The following three MSBuild properties are relevant to packing using a `.nuspec`:

1. `NuspecFile` : relative or absolute path to the `.nuspec` file being used for packing.
2. `NuspecProperties` : a semicolon-separated list of key=value pairs. Due to the way MSBuild command-line parsing works, multiple properties must be specified as follows:
`-p:NuspecProperties=\"key1=value1;key2=value2\"`.
3. `NuspecbasePath` : Base path for the `.nuspec` file.

If using `dotnet.exe` to pack your project, use a command like the following:

```
dotnet pack <path to .csproj file> -p:NuspecFile=<path to nuspec file> -p:NuspecProperties=<> -p:NuspecbasePath=<Base path>
```

If using MSBuild to pack your project, use a command like the following:

```
msbuild -t:pack <path to .csproj file> -p:NuspecFile=<path to nuspec file> -p:NuspecProperties=<> -p:NuspecbasePath=<Base path>
```

Please note that packing a nuspec using `dotnet.exe` or `msbuild` also leads to building the project by default. This can be avoided by passing `--no-build` property to `dotnet.exe`, which is the equivalent of setting `<NoBuild>true</NoBuild>` in your project file, along with setting `<IncludeBuildOutput>false</IncludeBuildOutput>` in the project file.

An example of a `.csproj` file to pack a nuspec file is:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <NoBuild>true</NoBuild>
    <IncludeBuildOutput>false</IncludeBuildOutput>
    <NuspecFile>PATH_TO_NUSPEC_FILE</NuspecFile>
    <NuspecProperties>add nuspec properties here</NuspecProperties>
    <NuspecBasePath>optional to provide</NuspecBasePath>
  </PropertyGroup>
</Project>
```

Advanced extension points to create customized package

The `pack` target provides two extension points that run in the inner, target framework specific build. The extension points support including target framework specific content and assemblies into a package:

- `TargetsForTfmSpecificBuildOutput` target: Use for files inside the `lib` folder or a folder specified using `BuildOutputTargetFolder`.
- `TargetsForTfmSpecificContentInPackage` target: Use for files outside the `BuildOutputTargetFolder`.

TargetsForTfmSpecificBuildOutput

Write a custom target and specify it as the value of the `$(TargetsForTfmSpecificBuildOutput)` property. For any files that need to go into the `BuildOutputTargetFolder` (lib by default), the target should write those files into the ItemGroup `BuildOutputInPackage` and set the following two metadata values:

- `FinalOutputPath`: The absolute path of the file; if not provided, the Identity is used to evaluate source path.
- `TargetPath`: (Optional) Set when the file needs to go into a subfolder within `lib\<TargetFramework>`, like satellite assemblies that go under their respective culture folders. Defaults to the name of the file.

Example:

```
<PropertyGroup>

  <TargetsForTfmSpecificBuildOutput>$(TargetsForTfmSpecificBuildOutput);GetMyPackageFiles</TargetsForTfmSpecificBuildOutput>
</PropertyGroup>

<Target Name="GetMyPackageFiles">
  <ItemGroup>
    <BuildOutputInPackage Include="$(OutputPath)cs\$(AssemblyName).resources.dll">
      <TargetPath>cs</TargetPath>
    </BuildOutputInPackage>
  </ItemGroup>
</Target>
```

TargetsForTfmSpecificContentInPackage

Write a custom target and specify it as the value of the `$(TargetsForTfmSpecificContentInPackage)` property. For any files to include in the package, the target should write those files into the ItemGroup `TfmSpecificPackageFile` and set the following optional metadata:

- `PackagePath`: Path where the file should be output in the package. NuGet issues a warning if more than one file is added to the same package path.
- `BuildAction`: The build action to assign to the file, required only if the package path is in the `contentFiles` folder. Defaults to "None".

An example:

```

<PropertyGroup>

<TargetsForTfmSpecificContentInPackage>$({TargetsForTfmSpecificContentInPackage});CustomContentTarget</Targets
ForTfmSpecificContentInPackage>
</PropertyGroup>

<Target Name="CustomContentTarget">
<ItemGroup>
<TfmSpecificPackageFile Include="abc.txt">
<PackagePath>mycontent/${TargetFramework}</PackagePath>
</TfmSpecificPackageFile>
<TfmSpecificPackageFile Include="Extensions/ext.txt" Condition="`${TargetFramework}` == 'net46'">
<PackagePath>net46content</PackagePath>
</TfmSpecificPackageFile>
</ItemGroup>
</Target>

```

restore target

`MSBuild -t:restore` (which `nuget restore` and `dotnet restore` use with .NET Core projects), restores packages referenced in the project file as follows:

1. Read all project to project references
2. Read the project properties to find the intermediate folder and target frameworks
3. Pass MSBuild data to NuGet.Build.Tasks.dll
4. Run restore
5. Download packages
6. Write assets file, targets, and props

The `restore` target works **only** for projects using the `PackageReference` format. It does **not** work for projects using the `packages.config` format; use [nuget restore](#) instead.

Restore properties

Additional restore settings may come from MSBuild properties in the project file. Values can also be set from the command line using the `-p:` switch (see Examples below).

PROPERTY	DESCRIPTION
RestoreSources	Semicolon-delimited list of package sources.
RestorePackagesPath	User packages folder path.
RestoreDisableParallel	Limit downloads to one at a time.
RestoreConfigFile	Path to a <code>Nuget.Config</code> file to apply.
RestoreNoCache	If true, avoids using cached packages. See Managing the global packages and cache folders .
RestoreIgnoreFailedSources	If true, ignores failing or missing package sources.
RestoreFallbackFolders	Fallback folders, used in the same way the user packages folder is used.
RestoreAdditionalProjectSources	Additional sources to use during restore.

PROPERTY	DESCRIPTION
RestoreAdditionalProjectFallbackFolders	Additional fallback folders to use during restore.
RestoreAdditionalProjectFallbackFoldersExcludes	Excludes fallback folders specified in <code>RestoreAdditionalProjectFallbackFolders</code>
RestoreTaskAssemblyFile	Path to <code>NuGet.Build.Tasks.dll</code> .
RestoreGraphProjectInput	Semicolon-delimited list of projects to restore, which should contain absolute paths.
RestoreUseSkipNonexistentTargets	When the projects are collected via MSBuild it determines whether they are collected using the <code>SkipNonexistentTargets</code> optimization. When not set, defaults to <code>true</code> . The consequence is a fail-fast behavior when a project's targets cannot be imported.
MSBuildProjectExtensionsPath	Output folder, defaulting to <code>BaseIntermediateOutputPath</code> and the <code>obj</code> folder.
RestoreForce	In <code>PackageReference</code> based projects, forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is similar to deleting the <code>project.assets.json</code> file. This does not bypass the http-cache.
RestorePackagesWithLockFile	Opts into the usage of a lock file.
RestoreLockedMode	Run restore in locked mode. This means that restore will not reevaluate the dependencies.
NuGetLockFilePath	A custom location for the lock file. The default location is next to the project and is named <code>packages.lock.json</code> .
RestoreForceEvaluate	Forces restore to recompute the dependencies and update the lock file without any warning.

Examples

Command line:

```
msbuild -t:restore -p:RestoreConfigFile=<path>
```

Project file:

```
<PropertyGroup>
  <RestoreIgnoreFailedSource>true</RestoreIgnoreFailedSource>
</PropertyGroup>
```

Restore outputs

Restore creates the following files in the build `obj` folder:

FILE	DESCRIPTION
project.assets.json	Contains the dependency graph of all package references.
{projectName}.projectFileExtension.nuget.g.props	References to MSBuild props contained in packages
{projectName}.projectFileExtension.nuget.g.targets	References to MSBuild targets contained in packages

Restoring and building with one MSBuild command

Due to the fact that NuGet can restore packages that bring down MSBuild targets and props, the restore and build evaluations are run with different global properties. This means that the following will have an unpredictable and often incorrect behavior.

```
msbuild -t:restore,build
```

Instead the recommended approach is:

```
msbuild -t:build -restore
```

The same logic applies to other targets similar to `build`.

PackageTargetFallback

The `PackageTargetFallback` element allows you to specify a set of compatible targets to be used when restoring packages. It's designed to allow packages that use a dotnet TxM to work with compatible packages that don't declare a dotnet TxM. That is, if your project uses the dotnet TxM, then all the packages it depends on must also have a dotnet TxM, unless you add the `<PackageTargetFallback>` to your project in order to allow non-dotnet platforms to be compatible with dotnet.

For example, if the project is using the `netstandard1.6` TxM, and a dependent package contains only `lib/net45/a.dll` and `lib/portable-net45+win81/a.dll`, then the project will fail to build. If what you want to bring in is the latter DLL, then you can add a `PackageTargetFallback` as follows to say that the `portable-net45+win81` DLL is compatible:

```
<PackageTargetFallback Condition="$(TargetFramework) == 'netstandard1.6'>
  portable-net45+win81
</PackageTargetFallback>
```

To declare a fallback for all targets in your project, leave off the `Condition` attribute. You can also extend any existing `PackageTargetFallback` by including `$(PackageTargetFallback)` as shown here:

```
<PackageTargetFallback>
  $(PackageTargetFallback);portable-net45+win81
</PackageTargetFallback >
```

Replacing one library from a restore graph

If a restore is bringing the wrong assembly, it's possible to exclude that packages default choice, and replace it with your own choice. First with a top level `PackageReference`, exclude all assets:

```
<PackageReference Include="Newtonsoft.Json" Version="9.0.1">
  <ExcludeAssets>All</ExcludeAssets>
</PackageReference>
```

Next, add your own reference to the appropriate local copy of the DLL:

```
<Reference Include="Newtonsoft.Json.dll" />
```

dotnet CLI commands

7/18/2019 • 2 minutes to read • [Edit Online](#)

The `dotnet` command-line interface (CLI), which runs on Windows, Mac OS X, and Linux, provides a number of essential commands such as installing, restoring, and publishing packages. If dotnet satisfies your needs, it's not necessary to use `nuget.exe`.

For examples of using these commands to consume packages, see [Install and manage packages using the dotnet CLI](#). For examples of using these commands to create packages, see [Create and publish a package using the dotnet CLI](#).

For the complete command reference on `dotnet` CLI, see [.NET Core command-line interface \(CLI\) tools](#).

Package consumption

- **dotnet add package**: Adds a package reference to the project file, then runs `dotnet restore` to install the package.
- **dotnet remove package**: Removes a package reference from the project file.
- **dotnet restore**: Restores the dependencies and tools of a project. As of NuGet 4.0, this runs the same code as `nuget restore`.
- **dotnet nuget locals**: Lists locations of the *global-packages*, *http-cache*, and *temp* folders and clears the contents of those folders.
- **dotnet new nugetconfig**: Creates a `nuget.config` file to configure NuGet's behavior.

Package creation

- **dotnet pack**: Packs the code into a NuGet package.
- **dotnet nuget push**: Publishes a package to a NuGet server. Applicable to nuget.org, Azure Artifacts, and [third-party NuGet servers](#).
- **dotnet nuget delete**: Deletes or unlists a package from a NuGet server. Applicable to nuget.org, Azure Artifacts, and [third-party NuGet servers](#).

NuGet CLI reference

7/18/2019 • 3 minutes to read • [Edit Online](#)

The NuGet Command Line Interface (CLI), `nuget.exe`, provides the full extent of NuGet functionality to install, create, publish, and manage packages without making any changes to project files.

To use any command, open a command window or bash shell, then run `nuget` followed by the command and appropriate options, such as `nuget help pack` (to view help on the pack command).

This documentation reflects the latest version of the NuGet CLI. For exact details for any given version that you're using, run `nuget help` for the desired command.

To learn how to use basic commands with the `nuget.exe` CLI, see [Install and use packages using the nuget.exe CLI](#).

Installing nuget.exe

Windows

NOTE

NuGet.exe 5.0 and later require .NET Framework 4.7.2 or later to execute.

1. Visit [nuget.org/downloads](#) and select NuGet 3.3 or higher (2.8.6 is not compatible with Mono). The latest version is always recommended, and 4.1.0+ is required to publish packages to nuget.org.
2. Each download is the `nuget.exe` file directly. Instruct your browser to save the file to a folder of your choice. The file is *not* an installer; you won't see anything if you run it directly from the browser.
3. Add the folder where you placed `nuget.exe` to your PATH environment variable to use the CLI tool from anywhere.

macOS/Linux

Behaviors may vary slightly by OS distribution.

1. Install [Mono 4.4.2 or later](#).
2. Execute the following command at a shell prompt:

```
# Download the latest stable `nuget.exe` to `/usr/local/bin`  
sudo curl -o /usr/local/bin/nuget.exe https://dist.nuget.org/win-x86-commandline/latest/nuget.exe
```

3. Create an alias by adding the following script to the appropriate file for your OS (typically `~/.bash_aliases` or `~/.bash_profile`):

```
# Create as alias for nuget  
alias nuget="mono /usr/local/bin/nuget.exe"
```

4. Reload the shell. Test the installation by entering `nuget` with no parameters. NuGet CLI help should display.

TIP

To make the NuGet CLI available within the Package Manager Console in Visual Studio, see [Using the nuget.exe CLI in the console](#).

Availability

See [feature availability](#) for exact details.

- All commands are available on Windows.
- All commands work with nuget.exe running on Mono except where indicated for `pack`, `restore`, and `update`.
- The `pack`, `restore`, `delete`, `locals`, and `push` commands are also available on Mac and Linux through the dotnet CLI.

Commands and applicability

Available commands and applicability to package creation, package consumption, and/or publishing a package to a host:

COMMON COMMANDS	APPLICABLE ROLES	NUGET VERSION	DESCRIPTION
pack	Creation	2.7+	Creates a NuGet package from a <code>.nuspec</code> or project file. When running on Mono, creating a package from a project file is not supported.
push	Publishing	All	Publishes a package to a package source.
config	All	All	Gets or sets NuGet configuration values.
help or ?	All	All	Displays help information or help for a command.
locals	Consumption	3.3+	Lists locations of the <i>global-packages</i> , <i>http-cache</i> , and <i>temp</i> folders and clears the contents of those folders.
restore	Consumption	2.7+	Restores all packages referenced by the package management format in use. When running on Mono, restoring packages using the <code>PackageReference</code> format is not supported.
setapikey	Publishing, Consumption	All	Saves an API key for a given package source when that package source requires a key for access.

COMMON COMMANDS	APPLICABLE ROLES	NUGET VERSION	DESCRIPTION
spec	Creation	All	Generates a <code>.nuspec</code> file, using tokens if generating the file from a Visual Studio project.
SECONDARY COMMANDS	APPLICABLE ROLES	NUGET VERSION	DESCRIPTION
add	Publishing	3.3+	Adds a package to a non-HTTP package source using hierarchical layout. For HTTP sources, use <code>push</code> .
delete	Publishing	All	Removes or unlists a package from a package source.
init	Creation	3.3+	Adds packages from a folder to a package source using hierarchical layout.
install	Consumption	All	Installs a package into the current project but does not modify projects or reference files.
list	Consumption, perhaps Publishing	All	Displays packages from a given source.
mirror	Publishing	Deprecated in 3.2+	Mirrors a package and its dependencies from a source to a target repository.
sources	Consumption, Publishing	All	Manages package sources in configuration files.
update	Consumption	All	Updates a project's packages to the latest available versions. Not supported when running on Mono.

Different commands make use of various [Environment variables](#).

NuGet CLI commands by applicable roles:

ROLE	COMMANDS
Consumption	<code>config</code> , <code>help</code> , <code>install</code> , <code>list</code> , <code>locals</code> , <code>restore</code> , <code>setapikey</code> , <code>sources</code> , <code>update</code>
Creation	<code>config</code> , <code>help</code> , <code>init</code> , <code>pack</code> , <code>spec</code>
Publishing	<code>add</code> , <code>config</code> , <code>delete</code> , <code>help</code> , <code>list</code> , <code>push</code> , <code>setapikey</code> , <code>sources</code>

Developers concerned only with consuming packages, for example, need only understand that subset of NuGet commands.

NOTE

Command option names are case-insensitive. Options that are deprecated are not included in this reference, such as

`NoPrompt` (replaced by `NonInteractive`) and `Verbose` (replaced by `Verbosity`).

add command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package publishing • **Supported versions:** 3.3+

Adds a specified package to a non-HTTP package source (a folder or UNC path) in a hierarchical layout, wherein folders are created for the package ID and version number. For example:

```
\myserver\packages
└<packageID>
  └<version>
    |<packageID>.<version>.nupkg
    |<packageID>.<version>.nupkg.sha512
    └<packageID>.nuspec
```

When restoring or updating against the package source, hierarchical layout provides significantly better performance.

To expand all the files in the package to the destination package source, use the `-Expand` switch. This typically results in additional subfolders appearing in the destination, such as `tools` and `lib`.

Usage

```
nuget add <packagePath> -Source <sourcePath> [options]
```

where `<packagePath>` is the pathname to the package to add, and `<sourcePath>` specifies the folder-based package source to which the package will be added. HTTP sources are not supported.

Options

OPTION	DESCRIPTION
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
Expand	Adds all the files in the package to the package source.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the command.
NonInteractive	Suppresses prompts for user input or confirmations.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal</i> , <i>quiet</i> , <i>detailed</i> .

Also see [Environment variables](#)

Examples

```
nuget add foo.nupkg -Source c:\bar\  
nuget add foo.nupkg -Source \\bar\packages\
```

config command (NuGet CLI)

7/24/2019 • 2 minutes to read • [Edit Online](#)

Applies to: all • **Supported versions:** all

Gets or sets NuGet configuration values. For additional usage, see [Common NuGet configurations](#). For details on allowable key names, refer to the [NuGet config file reference](#).

Usage

```
nuget config -Set <name>=[<value>] [<name>=<value> ...] [options]  
nuget config -AsPath <name> [options]
```

where `<name>` and `<value>` specify a key-value pair to be set in the configuration. You can specify as many pairs as desired. To remove a value, specify the name and the `=` sign but no value.

For allowable key names, see the [NuGet config file reference](#).

In NuGet 3.4+, `<value>` can use [environment variables](#).

Options

OPTION	DESCRIPTION
AsPath	Returns the config value as a path, ignored when <code>-Set</code> is used.
ConfigFile	The NuGet configuration file to modify. If not specified, the default file is used - <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.config/NuGet/NuGet.Config</code> (Mac/Linux) or <code>~/.nuget/NuGet/NuGet.Config</code> (varies by OS distribution).
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the command.
NonInteractive	Suppresses prompts for user input or confirmations.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal, quiet, detailed</i> .

Also see [Environment variables](#)

Examples

```
nuget config -Set repositoryPath=c:\packages -configfile c:\my.config  
nuget config -Set repositoryPath=  
nuget config -Set repositoryPath=%PACKAGE_REPO% -configfile %ProgramData%\NuGet\NuGetDefaults.Config  
nuget config -Set HTTP_PROXY=http://127.0.0.1 -set HTTP_PROXY.USER=domain\user
```

delete command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package publishing • **Supported versions:** all

Deletes or unlists a package from a package source. For nuget.org, the delete command [unlists the package](#).

Usage

```
nuget delete <packageID> <packageVersion> [options]
```

where `<packageID>` and `<packageVersion>` identify the exact package to delete or unlist. The exact behavior depends on the source. For local folders, for instance, the package is deleted; for nuget.org the package is unlisted.

Options

OPTION	DESCRIPTION
ApiKey	The API key for the target repository. If not present, the one specified in the config file is used.
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the command.
NonInteractive	Suppresses prompts for user input or confirmations.
Source	Specifies the server URL. The URL for nuget.org is <code>https://api.nuget.org/v3/index.json</code> . For private feeds, substitute the host name, for example, <code>%hostname%/api/v3</code> .
Verbosity	Specifies the amount of detail displayed in the output: <i>normal</i> , <i>quiet</i> , <i>detailed</i> .

Also see [Environment variables](#)

Examples

```
nuget delete MyPackage 1.0

nuget delete MyPackage 1.0 -Source http://package.contoso.com/source -apikey A1B2C3
```

help or ? command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: all • **Supported versions:** all

Displays general help information and help information about specific commands.

Usage

```
nuget help [command] [options]  
nuget ? [command] [options]
```

where [command] identifies a specific command for which to display help.

WARNING

With some commands, be mindful to specify *help* first, as with `nuget help install`, because there is a package named "help" on nuget.org. If you give the command `nuget install help`, you won't get help on the install command but will instead install the package named help.

Options

OPTION	DESCRIPTION
All	Print detailed help for all available commands; ignored if a specific command is given.
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the help command itself.
Markdown	Print detailed help in markdown format when used with <code>-All</code> . Ignored otherwise.
NonInteractive	Suppresses prompts for user input or confirmations.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal</i> , <i>quiet</i> , <i>detailed</i> .

Also see [Environment variables](#)

Examples

```
nuget help
nuget help push
nuget ?
nuget push -?
nuget help -All -Markdown
```

init command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package creation • **Supported versions:** 3.3+

Copies all the packages from a flat folder to a destination folder using a hierarchical layout as described for the [add command](#). That is, using `init` is equivalent to using the `add` command on each package in the folder.

As with `add`, the destination must be either a local folder or a UNC path; HTTP package repositories such as [nuget.org](#) or private servers are not supported.

Usage

```
nuget init <source> <destination> [options]
```

where `<source>` is the folder containing packages and `<destination>` is the local folder or UNC pathname to which the packages are copied.

Options

OPTION	DESCRIPTION
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Expand	Adds all files in each package that's added to the package source; same as <code>-Expand</code> with the <code>add</code> command.
Help	Displays help information for the command.
NonInteractive	Suppresses prompts for user input or confirmations.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal</i> , <i>quiet</i> , <i>detailed</i> .

Also see [Environment variables](#)

Examples

```
nuget init c:\foo c:\bar
nuget init \\foo\packages \\bar\packages -Expand
```

install command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package consumption • **Supported versions:** all

Downloads and installs a package into a project, defaulting to the current folder, using specified package sources.

TIP

To download a package directly outside the context of a project, visit the package's page on [nuget.org](#) and select the **Download** link.

If no sources are specified, those listed in the global configuration file, `%appdata%\NuGet\NuGet.Config` (Windows) or `~/.nuget/NuGet.Config` (Mac/Linux), are used. See [Common NuGet configurations](#) for additional details.

If no specific packages are specified, `install` installs all packages listed in the project's `packages.config` file, making it similar to `restore`.

The `install` command does not modify a project file or `packages.config`; in this way it's similar to `restore` in that it only adds packages to disk but does not change a project's dependencies.

To add a dependency, either add a package through the Package Manager UI or Console in Visual Studio, or modify `packages.config` and then run either `install` or `restore`.

Usage

```
nuget install <packageID | configFilePath> [options]
```

where `<packageID>` names the package to install (using the latest version), or `<configFilePath>` identifies the `packages.config` file that lists the packages to install. You can indicate a specific version with the `-Version` option.

Options

OPTION	DESCRIPTION
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet.Config</code> (Mac/Linux) is used.
DependencyVersion	(4.4+) The version of the dependency packages to use, which can be one of the following: <ul style="list-style-type: none">• <i>Lowest</i> (default): the lowest version• <i>HighestPatch</i>: the version with the lowest major, lowest minor, highest patch• <i>HighestMinor</i>: the version with the lowest major, highest minor, highest patch• <i>Highest</i>: the highest version
DisableParallelProcessing	Disables installing multiple packages in parallel.

OPTION	DESCRIPTION
ExcludeVersion	Installs the package to a folder named with only the package name and not the version number.
FallbackSource	(3.2+) A list of package sources to use as fallbacks in case the package isn't found in the primary or default source.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Framework	(4.4+) Target framework used for selecting dependencies. Defaults to 'Any' if not specified.
Help	Displays help information for the command.
NoCache	Prevents NuGet from using cached packages. See Managing the global packages and cache folders .
NonInteractive	Suppresses prompts for user input or confirmations.
OutputDirectory	Specifies the folder in which packages are installed. If no folder is specified, the current folder is used.
PackageSaveMode	Specifies the types of files to save after package installation: one of <code>nuspec</code> , <code>nupkg</code> , or <code>nuspec;nupkg</code> .
PreRelease	Allows prerelease packages to be installed. This flag is not required when restoring packages with <code>packages.config</code> .
RequireConsent	Verifies that restoring packages is enabled before downloading and installing the packages. For details, see Package Restore .
SolutionDirectory	Specifies root folder of the solution for which to restore packages.
Source	Specifies the list of package sources (as URLs) to use. If omitted, the command uses the sources provided in configuration files, see Common NuGet configurations .
Verbosity	Specifies the amount of detail displayed in the output: <i>normal, quiet, detailed</i> .
Version	Specifies the version of the package to install.

Also see [Environment variables](#)

Examples

```
nuget install elmah
nuget install packages.config
nuget install ninject -OutputDirectory c:\proj
```

list command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package consumption, publishing • **Supported versions:** all

Displays a list of packages from a given source. If no sources are specified, all sources defined in the global configuration file, `%AppData%\NuGet\NuGet.Config` (Windows) or `~/.nuget/NuGet/NuGet.Config`, are used. If `NuGet.Config` specifies no sources, then `list` uses the default feed (nuget.org).

Usage

```
nuget list [search terms] [options]
```

where the optional search terms will filter the displayed list. Search terms are applied to the names of packages, tags, and package descriptions just as they are when using them on nuget.org.

Options

OPTION	DESCRIPTION
AllVersions	List all versions of a package. By default, only the latest package version is displayed.
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the command.
IncludeDelisted	(3.2+) Display unlisted packages.
NonInteractive	Suppresses prompts for user input or confirmations.
PreRelease	Includes prerelease packages in the list.
Source	Specifies a list of packages sources to search.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal</i> , <i>quiet</i> , <i>detailed</i> .

Also see [Environment variables](#)

Examples

```
nuget list

nuget list chinese korean -Verbosity detailed

nuget list couchbase -AllVersions
```

locals command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package consumption • **Supported versions:** 3.3+

Clears or lists local NuGet resources such as the *http-cache*, *global-packages* folder, and the *temp* folder. The `locals` command can also be used to display a list of those locations. For more information, see [Managing the global packages and cache folders](#).

Usage

```
nuget locals <folder> [options]
```

where `<folder>` is one of `all`, `http-cache`, `packages-cache` (3.5 and earlier), `global-packages`, `temp` (3.4+), and `plugins-cache` (4.8+).

Options

OPTION	DESCRIPTION
Clear	Clears the specified folder.
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the command.
List	Lists the location of the specified folder, or the locations of all folders when used with <code>all</code> .
NonInteractive	Suppresses prompts for user input or confirmations.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal</i> , <i>quiet</i> , <i>detailed</i> .

Also see [Environment variables](#)

Examples

```
nuget locals all -list
nuget locals http-cache -clear
```

For additional examples, see [Managing the global packages and cache folders](#).

mirror command (NuGet CLI)

8/12/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package publishing • **Supported versions:** deprecated in 3.2+

Mirrors a package and its dependencies from the specified source repositories to the target repository.

NOTE

NuGet.ServerExtensions.dll and NuGet-Signed.exe that previously supported this command in NuGet 2.x (by renaming NuGet-Signed.exe to nuget.exe) are no longer available for download. To use a command similar to this, try [NuGetMirror](#).

Usage

```
nuget mirror <packageID | configFilePath> <listUrlTarget> <publishUrlTarget> [options]
```

where `<packageID>` is the package to mirror, or `<configFilePath>` identifies the `packages.config` file that lists the packages to mirror.

The `<listUrlTarget>` specifies the source repository, and `<publishUrlTarget>` specifies the target repository.

If your target repository is on `https://machine/repo` that's running [NuGet.Server](#), the list and push urls will be `https://machine/repo/nuget` and `https://machine/repo/api/v2/package`, respectively.

Options

OPTION	DESCRIPTION
ApiKey	The API key for the target repository. If not present, the one specified in the config file is used (<code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux)).
Help	Displays help information for the command.
NoCache	Prevents NuGet from using cached packages. See Managing the global packages and cache folders .
Noop	Logs what would be done but does not perform the actions; assumes success for push operations.
PreRelease	Includes prerelease packages in the mirroring operation.
Source	A list of package sources to mirror. If no sources are specified, the ones defined in the config file (see ApiKey above) are used, defaulting to nuget.org if none are specified.
Timeout	Specifies the timeout, in seconds, for pushing to a server. The default is 300 seconds (5 minutes).

OPTION	DESCRIPTION
Version	The version of the package to install. If not specified, the latest version is mirrored.

Also see [Environment variables](#)

Examples

```
nuget mirror packages.config https://MyRepo/nuget https://MyRepo/api/v2/package -source  
https://nuget.org/api/v2 -apikey myApiKey -nocache  
  
nuget mirror Microsoft.AspNet.Mvc https://MyRepo/nuget https://MyRepo/api/v2/package -version 4.0.20505.0  
  
nuget mirror Microsoft.Net.Http https://MyRepo/nuget https://MyRepo/api/v2/package -prerelease
```

pack command (NuGet CLI)

9/24/2019 • 4 minutes to read • [Edit Online](#)

Applies to: package creation • **Supported versions:** 2.7+

Creates a NuGet package based on the specified `.nuspec` or project file. The `dotnet pack` command (see [dotnet Commands](#)) and `msbuild -t:pack` (see [MSBuild targets](#)) may be used as alternates.

IMPORTANT

Under Mono, creating a package from a project file is not supported. You also need to adjust non-local paths in the `.nuspec` file to Unix-style paths, as `nuget.exe` doesn't convert Windows pathnames itself.

Usage

```
nuget pack <nuspecPath | projectPath> [options] [-Properties ...]
```

where `<nuspecPath>` and `<projectPath>` specify the `.nuspec` or project file, respectively.

Options

OPTION	DESCRIPTION
BasePath	Sets the base path of the files defined in the <code>.nuspec</code> file.
Build	Specifies that the project should be built before building the package.
Exclude	Specifies one or more wildcard patterns to exclude when creating a package. To specify more than one pattern, repeat the <code>-Exclude</code> flag. See example below.
ExcludeEmptyDirectories	Prevents inclusion of empty directories when building the package.
ForceEnglishOutput	(3.5+) Forces <code>nuget.exe</code> to run using an invariant, English-based culture.
ConfigFile	Specify the configuration file for the pack command.
Help	Displays help information for the command.
IncludeReferencedProjects	Indicates that the built package should include referenced projects either as dependencies or as part of the package. If a referenced project has a corresponding <code>.nuspec</code> file that has the same name as the project, then that referenced project is added as a dependency. Otherwise, the referenced project is added as part of the package.

OPTION	DESCRIPTION
MinClientVersion	Set the <i>minClientVersion</i> attribute for the created package. This value will override the value of the existing <i>minClientVersion</i> attribute (if any) in the <code>.nuspec</code> file.
MSBuildPath	(4.0+) Specifies the path of MSBuild to use with the command, taking precedence over <code>-MSBuildVersion</code> .
MSBuildVersion	(3.2+) Specifies the version of MSBuild to be used with this command. Supported values are 4, 12, 14, 15.1, 15.3, 15.4, 15.5, 15.6, 15.7, 15.8, 15.9. By default the MSBuild in your path is picked, otherwise it defaults to the highest installed version of MSBuild.
NoDefaultExcludes	Prevents default exclusion of NuGet package files and files and folders starting with a dot, such as <code>.svn</code> and <code>.gitignore</code> .
NoPackageAnalysis	Specifies that pack should not run package analysis after building the package.
OutputDirectory	Specifies the folder in which the created package is stored. If no folder is specified, the current folder is used.
Properties	Should appear last on the command line after other options. Specifies a list of properties that override values in the project file; see Common MSBuild Project Properties for property names. The Properties argument here is a list of token=value pairs, separated by semicolons, where each occurrence of <code>\$token\$</code> in the <code>.nuspec</code> file will be replaced with the given value. Values can be strings in quotation marks. Note that for the "Configuration" property, the default is "Debug". To change to a Release configuration, use <code>-Properties Configuration=Release</code> .
Suffix	(3.4.4+) Appends a suffix to the internally generated version number, typically used for appending build or other pre-release identifiers. For example, using <code>-suffix nightly</code> will create a package with a version number like <code>1.2.3-nightly</code> . Suffixes must start with a letter to avoid warnings, errors, and potential incompatibilities with different versions of NuGet and the NuGet Package Manager.
Symbols	Specifies that the package contains sources and symbols. When used with a <code>.nuspec</code> file, this creates a regular NuGet package file and the corresponding symbols package. By default it creates a legacy symbol package . The new recommended format for symbol packages is <code>.snupkg</code> . See Creating symbol packages (.snupkg) .
Tool	Specifies that the output files of the project should be placed in the <code>tool</code> folder.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal, quiet, detailed</i> .
Version	Overrides the version number from the <code>.nuspec</code> file.

Also see [Environment variables](#)

Excluding development dependencies

Some NuGet packages are useful as development dependencies, which help you author your own library, but aren't necessarily needed as actual package dependencies.

The `pack` command will ignore `package` entries in `packages.config` that have the `developmentDependency` attribute set to `true`. These entries will not be included as dependencies in the created package.

For example, consider the following `packages.config` file in the source project:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="jQuery" version="1.5.2" />
  <package id="netfx-Guard" version="1.3.3.2" developmentDependency="true" />
  <package id="microsoft-web Helpers" version="1.15" />
</packages>
```

For this project, the package created by `nuget pack` will have a dependency on `jquery` and `microsoft-web Helpers` but not `netfx-Guard`.

Examples

```
nuget pack

nuget pack foo.nuspec

nuget pack foo.csproj

nuget pack foo.csproj -Properties Configuration=Release

nuget pack foo.csproj -Build -Symbols -Properties owners=janedoe,xiaop;version="1.0.5"

# Create a package from project foo.csproj, using MSBuild version 12 to build the project
nuget pack foo.csproj -Build -Symbols -MSBuildVersion 12 -Properties owners=janedoe,xiaop;version="1.0.5"

# Create a package from project foo.nuspec and the corresponding symbol package using the new recommended
# format .snupkg
nuget pack foo.nuspec -Symbols -SymbolPackageFormat snupkg

nuget pack foo.nuspec -Version 2.1.0

nuget pack foo.nuspec -Version 1.0.0 -MinClientVersion 2.5

nuget pack Package.nuspec -exclude "*.exe" -exclude "*.bat"
```

push command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package publishing • **Supported versions:** all; 4.1.0+ required for nuget.org

IMPORTANT

To push packages to nuget.org you must use nuget.exe v4.1.0+, which implements the required [NuGet protocols](#).

Pushes a package to a package source and publishes it.

NuGet's default configuration is obtained by loading `%AppData%\NuGet\NuGet.Config` (Windows) or `~/.nuget/NuGet.Config` (Mac/Linux), then loading any `Nuget.Config` or `.nuget\Nuget.Config` files starting from root of drive and ending in current directory (see [Common NuGet configurations](#))

Usage

```
nuget push <packagePath> [options]
```

where `<packagePath>` identifies the package to push to the server.

Options

OPTION	DESCRIPTION
ApiKey	The API key for the target repository. If not present, the one specified in the config file is used.
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet.Config</code> (Mac/Linux) is used.
DisableBuffering	Disables buffering when pushing to an HTTP(s) server to decrease memory usages. Caution: when this option is used, integrated Windows authentication might not work.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the command.
NonInteractive	Suppresses prompts for user input or confirmations.
NoSymbols	(3.5+) If a symbols package exists, it will not be pushed to a symbol server.

OPTION	DESCRIPTION
Source	Specifies the server URL. NuGet identifies a UNC or local folder source and simply copies the file there instead of pushing it using HTTP. Also, starting with NuGet 3.4.2, this is a mandatory parameter unless the <code>NuGet.Config</code> file specifies a <code>DefaultPushSource</code> value (see Configuring NuGet behavior).
SkipDuplicate	(5.1+) If a package and version already exists, skip it and continue with the next package in the push, if any.
SymbolSource	(3.5+) Specifies the symbol server URL; <code>nuget.smbsrc.net</code> is used when pushing to <code>nuget.org</code>
SymbolApiKey	(3.5+) Specifies the API key for the URL specified in <code>-SymbolSource</code> .
Timeout	Specifies the timeout, in seconds, for pushing to a server. The default is 300 seconds (5 minutes).
Verbosity	Specifies the amount of detail displayed in the output: <i>normal, quiet, detailed</i> .

Also see [Environment variables](#)

Examples

```
nuget push foo.nupkg

nuget push foo.symbols.nupkg

nuget push foo.nupkg -Timeout 360

nuget push *.nupkg

nuget.exe push -source \\mycompany\repo\ mypackage.1.0.0.nupkg

nuget push foo.nupkg 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a -Source https://api.nuget.org/v3/index.json

nuget push foo.nupkg 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a

nuget push foo.nupkg 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a -src https://customsource/

:: In the example below -SkipDuplicate will skip pushing the package if package "Foo" version "5.0.2" already
exists on NuGet.org
nuget push Foo.5.0.2.nupkg 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a -src https://api.nuget.org/v3/index.json -
SkipDuplicate
```

restore command (NuGet CLI)

10/15/2019 • 4 minutes to read • [Edit Online](#)

Applies to: package consumption • **Supported versions:** 2.7+

Downloads and installs any packages missing from the `packages` folder. When used with NuGet 4.0+ and the `PackageReference` format, generates a `<project>.nuget.props` file, if needed, in the `obj` folder. (The file can be omitted from source control.)

On Mac OSX and Linux with the CLI on Mono, restoring packages is not supported with `PackageReference`.

Usage

```
nuget restore <projectPath> [options]
```

where `<projectPath>` specifies the location of a solution or a `packages.config` file. See [Remarks](#) below for behavioral details.

Options

OPTION	DESCRIPTION
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
DirectDownload	(4.0+) Downloads packages directly without populating caches with any binaries or metadata.
DisableParallelProcessing	Disables restoring multiple packages in parallel.
FallbackSource	(3.2+) A list of package sources to use as fallbacks in case the package isn't found in the primary or default source. Use a semicolon to separate list entries.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the command.
MSBuildPath	(4.0+) Specifies the path of MSBuild to use with the command, taking precedence over <code>-MSBuildVersion</code> .
MSBuildVersion	(3.2+) Specifies the version of MSBuild to be used with this command. Supported values are 4, 12, 14, 15.1, 15.3, 15.4, 15.5, 15.6, 15.7, 15.8, 15.9. By default the MSBuild in your path is picked, otherwise it defaults to the highest installed version of MSBuild.

OPTION	DESCRIPTION
NoCache	Prevents NuGet from using cached packages. See Managing the global packages and cache folders .
NonInteractive	Suppresses prompts for user input or confirmations.
OutputDirectory	Specifies the folder in which packages are installed. If no folder is specified, the current folder is used. Required when restoring with a <code>packages.config</code> file unless <code>PackagesDirectory</code> or <code>SolutionDirectory</code> is used.
PackageSaveMode	Specifies the types of files to save after package installation: one of <code>nuspec</code> , <code>nupkg</code> , or <code>nuspec;nupkg</code> .
PackagesDirectory	Same as <code>OutputDirectory</code> . Required when restoring with a <code>packages.config</code> file unless <code>OutputDirectory</code> or <code>SolutionDirectory</code> is used.
Project2ProjectTimeOut	Timeout in seconds for resolving project-to-project references.
Recursive	(4.0+) Restores all references projects for UWP and .NET Core projects. Does not apply to projects using <code>packages.config</code> .
RequireConsent	Verifies that restoring packages is enabled before downloading and installing the packages. For details, see Package Restore .
SolutionDirectory	Specifies the solution folder. Not valid when restoring packages for a solution. Required when restoring with a <code>packages.config</code> file unless <code>PackagesDirectory</code> or <code>OutputDirectory</code> is used.
Source	Specifies the list of package sources (as URLs) to use for the restore. If omitted, the command uses the sources provided in configuration files, see Configuring NuGet behavior . Use a semicolon to separate list entries.
Force	In <code>PackageReference</code> based projects, forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is similar to deleting the <code>project.assets.json</code> file. This does not bypass the http-cache.
Verbosity	Specifies the amount of detail displayed in the output: <code>normal</code> , <code>quiet</code> , <code>detailed</code> .

Also see [Environment variables](#)

Remarks

The restore command performs the following steps:

1. Determine the operation mode of the restore command.

PROJECTPATH FILE TYPE	BEHAVIOR
Solution (folder)	NuGet looks for a <code>.sln</code> file and uses that if found; otherwise gives an error. <code>\$(SolutionDir)\.nuget</code> is used as the starting folder.
<code>.sln</code> file	Restore packages identified by the solution; gives an error if <code>-SolutionDirectory</code> is used. <code>\$(SolutionDir)\.nuget</code> is used as the starting folder.
<code>packages.config</code> or project file	Restore packages listed in the file, resolving and installing dependencies.
Other file type	File is assumed to be a <code>.sln</code> file as above; if it's not a solution, NuGet gives an error.
(projectPath not specified)	<ul style="list-style-type: none"> NuGet looks for solution files in the current folder. If a single file is found, that one is used to restore packages; if multiple solutions are found, NuGet gives an error. If there are no solution files, NuGet looks for a <code>packages.config</code> and uses that to restore packages. If no solution or <code>packages.config</code> file is found, NuGet gives an error.

2. Determine the packages folder using the following priority order (NuGet gives an error if none of these folders are found):

- The folder specified with `-PackagesDirectory`.
- The `repositoryPath` value in `Nuget.Config`
- The folder specified with `-SolutionDirectory`
- `$(SolutionDir)\packages`

3. When restoring packages for a solution, NuGet does the following:

- Loads the solution file.
- Restores solution level packages listed in `$(SolutionDir)\.nuget\packages.config` into the `packages` folder.
- Restore packages listed in `$(ProjectDir)\packages.config` into the `packages` folder. For each package specified, restore the package in parallel unless `-DisableParallelProcessing` is specified.

Examples

```
# Restore packages for a solution file
nuget restore a.sln

# Restore packages for a solution file, using MSBuild version 14.0 to load the solution and its project(s)
nuget restore a.sln -MSBuildVersion 14

# Restore packages for a project's packages.config file, with the packages folder at the parent
nuget restore proj1\packages.config -PackagesDirectory ..\packages

# Restore packages for the solution in the current folder, specifying package sources
nuget restore -source "https://api.nuget.org/v3/index.json;https://www.myget.org/F/nuget"
```

setapikey command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package consumption, publishing • **Supported versions:** all

Saves an API key for a given server URL into `NuGet.Config` so that it doesn't need to be entered for subsequent commands.

Usage

```
nuget setapikey <key> -Source <url> [options]
```

where `<source>` identifies the server and `<key>` is the key or password to save. If `<source>` is omitted, `nuget.org` is assumed.

Options

OPTION	DESCRIPTION
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the command.
NonInteractive	Suppresses prompts for user input or confirmations.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal, quiet, detailed</i> .

Also see [Environment variables](#)

Examples

```
nuget setapikey 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a
```

```
nuget setapikey 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a -source https://example.com/nugetfeed
```

sign command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package creation • **Supported versions:** 4.6+

Signs all the packages matching the first argument with a certificate. The certificate with the private key can be obtained from a file or from a certificate installed in a certificate store by providing a subject name or a thumbprint.

Package signing is not yet supported in .NET Core, under Mono, or on non-Windows platforms.

Usage

```
nuget sign <package(s)> [options]
```

where `<package(s)>` is one or more `.nupkg` files.

Options

OPTION	DESCRIPTION
<code>CertificateFingerprint</code>	Specifies the SHA-1 fingerprint of the certificate used to search a local certificate store for the certificate.
<code>CertificatePassword</code>	Specifies the certificate password, if needed. If a certificate is password protected but no password is provided, the command will prompt for a password at run time, unless the <code>-NonInteractive</code> option is passed.
<code>CertificatePath</code>	Specifies the file path to the certificate to be used in signing the package.
<code>CertificateStoreLocation</code>	Specifies the name of the X.509 certificate store use to search for the certificate. Defaults to "CurrentUser", the X.509 certificate store used by the current user. This option should be used when specifying the certificate via <code>-CertificateSubjectName</code> or <code>-CertificateFingerprint</code> options.
<code>CertificateStoreName</code>	Specifies the name of the X.509 certificate store to use to search for the certificate. Defaults to "My", the X.509 certificate store for personal certificates. This option should be used when specifying the certificate via <code>-CertificateSubjectName</code> or <code>-CertificateFingerprint</code> options.
<code>CertificateSubjectName</code>	Specifies the subject name of the certificate used to search a local certificate store for the certificate. The search is a case-insensitive string comparison using the supplied value, which will find all certificates with the subject name containing that string, regardless of other subject values. The certificate store can be specified by <code>-CertificateStoreName</code> and <code>-CertificateStoreLocation</code> options.

OPTION	DESCRIPTION
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
ForceEnglishOutput	Forces nuget.exe to run using an invariant, English-based culture.
HashAlgorithm	Hash algorithm to be used to sign the package. Defaults to SHA256.
Help	Displays help information for the command.
NonInteractive	Suppresses prompts for user input or confirmations.
OutputDirectory	Specifies the directory where the signed package should be saved. By default the original package is overwritten by the signed package.
Overwrite	Switch to indicate if the current signature should be overwritten. By default the command will fail if the package already has a signature.
Timestamper	URL to an RFC 3161 timestamping server.
TimestampHashAlgorithm	Hash algorithm to be used by the RFC 3161 timestamp server. Defaults to SHA256.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal, quiet, detailed</i> .

Examples

```
nuget sign MyPackage.nupkg -Timestamper http://timestamp.test

nuget sign ..\MyPackage.nupkg -Timestamper http://timestamp.test -OutputDirectory ..\Signed
```

sources command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package consumption, publishing • **Supported versions:** all

Manages the list of sources located in the user scope configuration file or a specified configuration file. The user scope configuration file is located at `%appdata%\NuGet\NuGet.Config` (Windows) and `~/.nuget/NuGet/NuGet.Config` (Mac/Linux).

Note that the source URL for nuget.org is <https://api.nuget.org/v3/index.json>.

Usage

```
nuget sources <operation> -Name <name> -Source <source>
```

where `<operation>` is one of *List*, *Add*, *Remove*, *Enable*, *Disable*, or *Update*, `<name>` is the name of the source, and `<source>` is the source's URL. You can operate on only one source at a time.

Options

OPTION	DESCRIPTION
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Format	Applies to the <code>list</code> action and can be <code>Detailed</code> (the default) or <code>Short</code> .
Help	Displays help information for the command.
NonInteractive	Suppresses prompts for user input or confirmations.
Password	Specifies the password for authenticating with the source.
StorePasswordInClearText	Indicates to store the password in unencrypted text instead of the default behavior of storing an encrypted form.
UserName	Specifies the user name for authenticating with the source.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal</i> , <i>quiet</i> , <i>detailed</i> .

NOTE

Make sure to add the sources' password under the same user context as the nuget.exe is later used to access the package source. The password will be stored encrypted in the config file and can only be decrypted in the same user context as it was encrypted. So for example when you use a build server to restore NuGet packages the password must be encrypted with the same Windows user under which the build server task will run.

Also see [Environment variables](#)

Examples

```
nuget sources Add -Name "MyServer" -Source \\myserver\packages

nuget sources Disable -Name "MyServer"

nuget sources Enable -Name "nuget.org"

nuget sources add -name foo.bar -source C:\NuGet\local -username foo -password bar -StorePasswordInClearText
-configfile %AppData%\NuGet\my.config
```

spec command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package creation • **Supported versions:** all

Generates a `.nuspec` file for a new package. If run in the same folder as a project file (`.csproj`, `.vbproj`, `.fsproj`), `spec` creates a tokenized `.nuspec` file. For additional information, see [Creating a Package](#).

Usage

```
nuget spec [<packageID>] [options]
```

where `<packageID>` is an optional package identifier to save in the `.nuspec` file.

Options

OPTION	DESCRIPTION
AssemblyPath	Specifies the path to the assembly to use for metadata.
Force	Overwrites any existing <code>.nuspec</code> file.
ForceEnglishOutput	(3.5+) Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the command.
NonInteractive	Suppresses prompts for user input or confirmations.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal</i> , <i>quiet</i> , <i>detailed</i> .

Also see [Environment variables](#)

Examples

```
nuget spec  
nuget spec MyPackage  
nuget spec -AssemblyPath MyAssembly.dll
```

update command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package consumption • **Supported versions:** all

Updates all packages in a project (using `packages.config`) to their latest available versions. It is recommended to run '`restore`' before running the `update`. (To update an individual package, use `nuget install` without specifying a version number, in which case NuGet installs the latest version.)

Note: `update` does not work with the CLI running under Mono (Mac OSX or Linux) or when using the `PackageReference` format.

The `update` command also updates assembly references in the project file, provided those references already exist. If an updated package has an added assembly, a new reference is *not* added. New package dependencies also don't have their assembly references added. To include these operations as part of an update, update the package in Visual Studio using the Package Manager UI or the Package Manager Console.

This command can also be used to update `nuget.exe` itself using the `-self` flag.

Usage

```
nuget update <configPath> [options]
```

where `<configPath>` identifies either a `packages.config` or solution file that lists the project's dependencies.

Options

OPTION	DESCRIPTION
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
FileConflictAction	Specifies the action to take when asked to overwrite or ignore existing files referenced by the project. Values are <i>overwrite</i> , <i>ignore</i> , <i>none</i> .
ForceEnglishOutput	(3.5+) Forces <code>nuget.exe</code> to run using an invariant, English-based culture.
Help	Displays help information for the command.
Id	Specifies a list of package IDs to update.
MSBuildPath	(4.0+) Specifies the path of MSBuild to use with the command, taking precedence over <code>-MSBuildVersion</code> .

OPTION	DESCRIPTION
MSBuildVersion	(3.2+) Specifies the version of MSBuild to be used with this command. Supported values are 4, 12, 14, 15.1, 15.3, 15.4, 15.5, 15.6, 15.7, 15.8, 15.9. By default the MSBuild in your path is picked, otherwise it defaults to the highest installed version of MSBuild.
NonInteractive	Suppresses prompts for user input or confirmations.
PreRelease	Allows updating to prerelease versions. This flag is not required when updating prerelease packages that are already installed.
RepositoryPath	Specifies the local folder where packages are installed.
Safe	Specifies that only updates with the highest version available within the same major and minor version as the installed package will be installed.
Self	Updates nuget.exe to the latest version; all other arguments are ignored.
Source	Specifies the list of package sources (as URLs) to use for the updates. If omitted, the command uses the sources provided in configuration files, see Common NuGet configurations .
Verbosity	Specifies the amount of detail displayed in the output: <i>normal</i> , <i>quiet</i> , <i>detailed</i> .
Version	When used with one package ID, specifies the version of the package to update.

Also see [Environment variables](#)

Examples

```

nuget update

# update packages installed in solution.sln, using MSBuild version 14.0 to load the solution and its
# project(s).
nuget update solution.sln -MSBuildVersion 14

nuget update -safe

nuget update -self

```

verify command (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: package consumption • **Supported versions:** 4.6+

Verifies a package.

Verification of signed packages is not yet supported in .NET Core, under Mono, or on non-Windows platforms.

Usage

```
nuget verify <-All|-Signatures> <package(s)> [options]
```

where `<package(s)>` is one or more `.nupkg` files.

nuget verify -All

Specifies that all verifications possible should be performed on the package(s).

nuget verify -Signatures

Specifies that package signature verification should be performed.

Options for "verify -Signatures"

OPTION	DESCRIPTION
CertificateFingerprint	Specifies one or more SHA-256 certificate fingerprints of certificates(s) which signed packages must be signed with. A certificate SHA-256 fingerprint is a SHA-256 hash of the certificate. Multiple inputs should be semicolon separated.

Options

OPTION	DESCRIPTION
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
ForceEnglishOutput	Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the command.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal</i> , <i>quiet</i> , <i>detailed</i> .

Examples

```
nuget verify -Signatures .\..\MyPackage.nupkg -CertificateFingerprint  
"CE40881FF5F0AD3E58965DA20A9F571EF1651A56933748E1BF1C99E537C4E039;5F874AAF47BCB268A19357364E7FBB09D6BF9E8A93E1  
229909AC5CAC865802E2" -Verbosity detailed

nuget verify -Signatures c:\packages\MyPackage.nupkg -CertificateFingerprint  
CE40881FF5F0AD3E58965DA20A9F571EF1651A56933748E1BF1C99E537C4E039

nuget verify -Signatures MyPackage.nupkg -Verbosity quiet

nuget verify -Signatures .\*.nupkg

nuget verify -All .\*.nupkg
```

trusted-signers command (NuGet CLI)

11/5/2019 • 3 minutes to read • [Edit Online](#)

Applies to: package consumption • **Supported versions:** 4.9.1+

Gets or sets trusted signers to the NuGet configuration. For additional usage, see [Common NuGet configurations](#). For details on how the nuget.config schema looks like, refer to the [NuGet config file reference](#).

Usage

```
nuget trusted-signers <list|add|remove|sync> [options]
```

if none of `list|add|remove|sync` is specified, the command will default to `list`.

nuget trusted-signers list

Lists all the trusted signers in the configuration. This option will include all the certificates (with fingerprint and fingerprint algorithm) each signer has. If a certificate has a preceding `[U]`, it means that certificate entry has `allowUntrustedRoot` set as `true`.

Below is an example output from this command:

```
$ nuget trusted-signers
Registered trusted signers:

1. nuget.org [repository]
  Service Index: https://api.nuget.org/v3/index.json
  Certificate fingerprint(s):
    SHA256 - 0E5F38F57DC1BCC806D8494F4F90FBCEDD988B46760709CBEEC6F4219AA6157D

2. microsoft [author]
  Certificate fingerprint(s):
    SHA256 - 3F9001EA83C560D712C24CF213C3D312CB3BFF51EE89435D3430BD06B5D0EECE

3. myUntrustedAuthorSignature [author]
  Certificate fingerprint(s):
    [U] SHA256 - 518F9CF082C0872025EFB2587B6A6AB198208F63EA58DD54D2B9FF6735CA4434
```

nuget trusted-signers add [options]

Adds a trusted signer with the given name to the config. This option has different gestures to add a trusted author or repository.

Options for add based on a package

```
nuget trusted-signers add <package(s)> -Name <name> [options]
```

where `<package(s)>` is one or more `.nupkg` files.

OPTION	DESCRIPTION
Author	Specifies that the author signature of the package(s) should be trusted.
Repository	Specifies that the repository signature or countersignature of the package(s) should be trusted.
AllowUntrustedRoot	Specifies if the certificate for the trusted signer should be allowed to chain to an untrusted root.
Owners	Semi-colon separated list of trusted owners to further restrict the trust of a repository. Only valid when using the <code>-Repository</code> option.

Providing both `-Author` and `-Repository` at the same time is not supported.

Options for add based on a service index

```
nuget trusted-signers add -Name <name> [options]
```

Note: This option will only add trusted repositories.

OPTION	DESCRIPTION
ServiceIndex	Specifies the V3 service index of the repository to be trusted. This repository has to support the repository signatures resource. If not provided, the command will look for a package source with the same <code>-Name</code> and get the service index from there.
AllowUntrustedRoot	Specifies if the certificate for the trusted signer should be allowed to chain to an untrusted root.
Owners	Semi-colon separated list of trusted owners to further restrict the trust of a repository.

Options for add based on the certificate information

```
nuget trusted-signers add -Name <name> [options]
```

Note: If a trusted signer with the given name already exists, the certificate item will be added to that signer. Otherwise a trusted author will be created with a certificate item from given certificate information.

OPTION	DESCRIPTION
CertificateFingerprint	Specifies a certificate fingerprints of a certificate which signed packages must be signed with. A certificate fingerprint is a hash of the certificate. The hash algorithm used for calculating this hash should be specified in the <code>FingerprintAlgorithm</code> option.

OPTION	DESCRIPTION
FingerprintAlgorithm	Specifies the hash algorithm used to calculate the certificate fingerprint. Defaults to <code>SHA256</code> . Values supported are <code>SHA256</code> , <code>SHA384</code> and <code>SHA512</code>
AllowUntrustedRoot	Specifies if the certificate for the trusted signer should be allowed to chain to an untrusted root.

nuget trusted-signers remove -Name <name>

Removes any trusted signers that match the given name.

nuget trusted-signers sync -Name <name>

Requests the latest list of certificates used in a currently trusted repository to update the the existing certificate list in the trusted signer.

Note: This gesture will delete the current list of certificates and replace them with an up-to-date list from the repository.

Options

OPTION	DESCRIPTION
ConfigFile	The NuGet configuration file to apply. If not specified, <code>%AppData%\NuGet\NuGet.Config</code> (Windows) or <code>~/.nuget/NuGet/NuGet.Config</code> (Mac/Linux) is used.
ForceEnglishOutput	Forces nuget.exe to run using an invariant, English-based culture.
Help	Displays help information for the command.
Verbosity	Specifies the amount of detail displayed in the output: <i>normal, quiet, detailed</i> .

Examples

```
nuget trusted-signers list

nuget trusted-signers Add -Name existingSource

nuget trusted-signers Add -Name trustedRepo -ServiceIndex https://trustedRepo.test/v3ServiceIndex

nuget trusted-signers Add -Name author1 -CertificateFingerprint
CE40881FF5F0AD3E58965DA20A9F571EF1651A56933748E1BF1C99E537C4E039 -FingerprintAlgorithm SHA256

nuget trusted-signers Add -Repository ..\..\MyRepositorySignedPackage.nupkg -Name TrustedRepo

nuget-trusted-signers Remove -Name TrustedRepo

nuget-trusted-signers Sync -Name TrustedRepo
```

NuGet CLI environment variables

7/18/2019 • 2 minutes to read • [Edit Online](#)

The behavior of the nuget.exe CLI can be configured through a number of environment variables, which affect nuget.exe on computer-wide, user, or process levels. Environment variables always override any settings in `NuGet.Config` files, allowing build servers to change appropriate settings without modifying any files.

In general, options specified directly on the command line or in NuGet configuration files have precedence, but there are a few exceptions such as `FORCE_NUGET_EXE_INTERACTIVE`. If you find that nuget.exe behaves differently between different computers, an environment variable could be the cause. For example, Azure Web Apps Kudu (used during deployment) has `NUGET_XMLDOC_MODE` set to `skip` to speed up package restore performance and save disk space.

The NuGet CLI uses MSBuild to read the project files. All environment variables are available as [properties](#) during the MSBuild evaluation. The list of properties documented in [NuGet pack and restore as MSBuild targets](#) can also be set as environment variables.

VARIABLE	DESCRIPTION	REMARKS
http_proxy	Http proxy used for NuGet HTTP operations.	This would be specified as <code>http://<username>:<password>@proxy.com</code>
no_proxy	Configures domains to bypass from using proxy.	Specified as domains separated by comma (,).
EnableNuGetPackageRestore	Flag for if NuGet should implicitly grant consent if that's required by package on restore.	Specified flag is treated as <code>true</code> or <code>1</code> , any other value treated as flag not set.
NUGET_EXE_NO_PROMPT	Prevents the exe for prompting for credentials.	Any value except null or empty string will be treated as this flag set/true.
FORCE_NUGET_EXE_INTERACTIVE	Global environment variable to force interactive mode.	Any value except null or empty string will be treated as this flag set/true.
NUGET_PACKAGES	Path to use for the <i>global-packages</i> folder as described on Managing the global packages and cache folders .	Specified as absolute path.
NUGET_FALLBACK_PACKAGES	Global fallback packages folders.	Absolute folder paths separated by semicolon (;).
NUGET_HTTP_CACHE_PATH	Path to use for the <i>http-cache</i> folder as described on Managing the global packages and cache folders .	Specified as absolute path.
NUGET_PERSIST_DG	Flag indicating if dg files (data collected from MSBuild) should be persisted.	Specified as <code>true</code> or <code>false</code> (default), if <code>NUGET_PERSIST_DG_PATH</code> not set will be stored to temporary directory (NuGetScratch folder in current environment temp directory).

VARIABLE	DESCRIPTION	REMARKS
NUGET_PERSIST_DG_PATH	Path to persist dg files.	Specified as absolute path, this option is only used when <i>NUGET_PERSIST_DG</i> is set to true.
NUGET_RESTORE_MSBUILD_ARGS	Sets additional MSBuild arguments.	Pass arguments identical to how you would pass them to msbuild.exe. An example of setting a project property <i>Foo</i> from the command line to value <i>Bar</i> would be <code>/p:Foo=Bar</code>
NUGET_RESTORE_MSBUILD_VERBOSITY	Sets the MSBuild log verbosity.	Default is <i>quiet</i> (" <code>/v:q</code> "). Possible values <i>quiet</i> , <i>m[inimal]</i> , <i>n[ormal]</i> , <i>d[etailed]</i> , and <i>diag[nostic]</i> .
NUGET_SHOW_STACK	Determines whether the full exception (including stack trace) should be displayed to the user.	Specified as <i>true</i> or <i>false</i> (default).
NUGET_XMLDOC_MODE	Determines how assemblies XML documentation file extraction should be handled.	Supported modes are <i>skip</i> (do not extract XML documentation files), <i>compress</i> (store XML doc files as a zip archive) or <i>none</i> (default, treat XML doc files as regular files).
NUGET_CERT_REVOCATION_MODE	Determines how the revocation status check of the certificate used to sign a package, is performed when a signed package is installed or restored. When not set, defaults to <code>online</code> .	Possible values <i>online</i> (default), <i>offline</i> . Related to NU3028

Long Path Support (NuGet CLI)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Applies to: all • **Supported versions:** 4.8+

NuGet.exe 4.8 and later support long paths for files and directories for scenarios like Pack, Restore, Install, and most other scenarios that need file paths.

Required Operating System

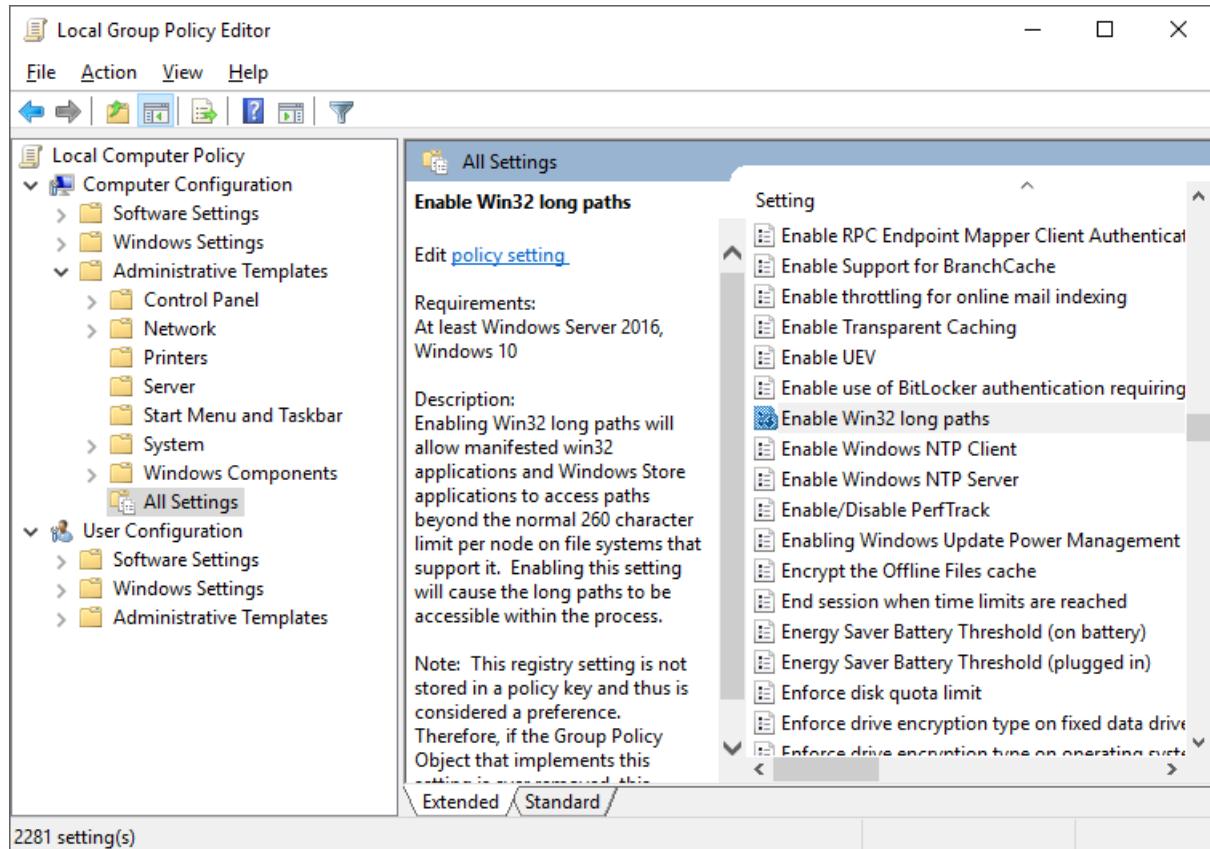
- Windows 10 (version 1607 or later)
- Windows 10 (July 2015 release or version 1511) if you upgrade .NET Framework to versions 4.6.2 or later.
- Windows Server 2016 (all versions)

Enable "Win32 Long Paths" Group Policy

One needs to enable long path support on those systems by setting a group policy.

Steps:

1. Launch **Group Policy Editor** - Type "Edit group policy" in the Start search bar or Run "gpedit.msc" from the Run command (Windows-R).
2. In the **Local Group Policy Editor**, enable "Local Computer Policy/Computer Configuration/Administrative Templates/All Settings/Enable Win32 long paths".



NOTE

Enabling Other NuGet Tools to Support Long Paths

- Dotnet CLI supports long paths regardless of the operating system or version.
- Visual Studio or msbuild -t:restore does not yet support long paths.
- Software that uses NuGet Libraries to execute restore and other commands, will support long paths on the same systems that NuGet.exe works on, if they also set longPathAware in their windows manifest and configure UseLegacyPathHandling to false via App.Config [See more information](#)

PowerShell reference

7/18/2019 • 2 minutes to read • [Edit Online](#)

The Package Manager Console provides a PowerShell interface within Visual Studio on Windows to interact with NuGet through the specific commands listed below. (The console is not presently available in Visual Studio for Mac.) For a guide to using the console, see [Install and manage packages using Package Manager Console](#) topic.

TIP

All PowerShell commands relate only to package consumption. No PowerShell commands relate to creating and publishing packages except to the extent that a package can also be a consumer of other packages.

IMPORTANT

The commands listed here are specific to the Package Manager Console in Visual Studio, and differ from the [Package Management module commands](#) that are available in a general PowerShell environment. Specifically, each environment has commands that are not available in the other, and commands with the same name may also differ in their specific arguments. When using the Package Management Console in Visual Studio, the commands and arguments documented in this present topic apply.

COMMON COMMANDS	DESCRIPTION	NUGET VERSION
Install-Package	Installs a package and its dependencies into the project.	All
Update-Package	Updates a package and its dependencies, or all packages in a project.	All
Find-Package	Searches a package source using a package ID or keywords.	3.0+
Get-Package	Retrieves the list of packages installed in the local repository, or lists packages available from a package source.	All
SECONDARY COMMANDS	DESCRIPTION	NUGET VERSION
Add-BindingRedirect	Examines all assemblies within the output path for a project and adds binding redirects to the <code>app.config</code> or <code>web.config</code> where necessary.	All
Get-Project	Displays information about the default or specified project.	3.0+
Open-PackagePage	Launches the default browser with the project, license, or report abuse URL for the specified package.	Deprecated in 3.0+

SECONDARY COMMANDS	DESCRIPTION	NUGET VERSION
Register-TabExpansion	Registers a tab expansion for the parameters of a command, allowing you to create customized expansions for commonly-used parameter values.	All
Sync-Package	Get the version of installed package from specified project and syncs the version to the rest of projects in the solution.	3.0+
Uninstall-Package	Removes a package from a project, optionally removing its dependencies.	All

For complete, detailed help on any of these commands within the console, just run the following with the command name in question:

```
Get-Help <command> -full
```

All Package Manager Console commands support the following [common PowerShell parameters](#):

- `Debug`
- `ErrorAction`
- `ErrorVariable`
- `OutBuffer`
- `OutVariable`
- `PipelineVariable`
- `Verbose`
- `WarningAction`
- `WarningVariable`

For details, refer to [about_CommonParameters](#) in the PowerShell documentation.

Add-BindingRedirect (Package Manager Console in Visual Studio)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Available only within the [Package Manager Console](#) in Visual Studio on Windows.

Examines all assemblies within the output path for a project and adds binding redirects to the application or web configuration file where necessary. This command is run automatically when installing a package.

For details on binding redirects and why they are used, see [Redirecting Assembly Versions](#) in the .NET documentation.

Syntax

```
Add-BindingRedirect [-ProjectName] <string> [<CommonParameters>]
```

Parameters

PARAMETER	DESCRIPTION
ProjectName	(Required) The project to which to add binding redirects. The -ProjectName switch itself is optional.

None of these parameters accept pipeline input or wildcard characters.

Common Parameters

`Add-BindingRedirect` supports the following [common PowerShell parameters](#): Debug, Error Action, ErrorVariable, OutBuffer, OutVariable, PipelineVariable, Verbose, WarningAction, and WarningVariable.

Examples

```
Add-BindingRedirect MyProject
Add-BindingRedirect -ProjectName MyProject
```

Find-Package (Package Manager Console in Visual Studio)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Version 3.0+; this topic describes the command within the [Package Manager Console](#) in Visual Studio on Windows. For the generic PowerShell `Find-Package` command, see the [PowerShell PackageManagement](#) reference.

Gets the set of remote packages with specified ID or keywords from the package source.

Syntax

```
Find-Package [-Id] <keywords> -Source <string> [-AllVersions] [-First [<int>]]  
[-Skip <int>] [-IncludePrerelease] [-ExactMatch] [-StartWith] [<CommonParameters>]
```

Parameters

PARAMETER	DESCRIPTION
<code>Id <keywords></code>	(Required) Keywords to use when searching the package source. Use <code>-ExactMatch</code> to return only those packages whose package ID matches the keywords. If no keywords are given, <code>Find-Package</code> returns a list of the top 20 packages by downloads, or the number specified by <code>-First</code> . Note that <code>-Id</code> is optional and a no-op.
<code>Source</code>	The URL or folder path for the package source to search. Local folder paths can be absolute, or relative to the current folder. If omitted, <code>Find-Package</code> searches the currently selected package source.
<code>AllVersions</code>	Displays all available versions of each package instead of only the latest version.
<code>First</code>	The number of packages to return from the beginning of the list; the default is 20.
<code>Skip</code>	Omits the first <code><int></code> packages from the displayed list.
<code>IncludePrerelease</code>	Includes prerelease packages in the results.
<code>ExactMatch</code>	Specified to use <code><keywords></code> as a case-sensitive package ID.
<code>StartWith</code>	Returns packages whose package ID begins with <code><keywords></code> .

None of these parameters accept pipeline input or wildcard characters.

Common Parameters

`Find-Package` supports the following [common PowerShell parameters](#): `Debug`, `Error Action`, `ErrorVariable`, `OutBuffer`, `OutVariable`, `PipelineVariable`, `Verbose`, `WarningAction`, and `WarningVariable`.

Examples

```
# Find packages containing keywords
Find-Package elmah
Find-Package logging

# List packages whose ID begins with Elmah
Find-Package Elmah -StartWith

# By default, Get-Package returns a list of 20 packages; use -First to show more
Find-Package logging -First 100

# List all versions of the package with the ID of "jquery"
Find-Package jquery -AllVersions -ExactMatch
```

Get-Package (Package Manager Console in Visual Studio)

7/18/2019 • 2 minutes to read • [Edit Online](#)

This topic describes the command within the [Package Manager Console](#) in Visual Studio on Windows. For the generic PowerShell `Get-Package` command, see the [PowerShell PackageManagement](#) reference.

Retrieves the list of packages installed in the local repository, lists packages available from a package source when used with the `-ListAvailable` switch, or lists available updates when used with the `-Update` switch.

Syntax

```
Get-Package -Source <string> [-ListAvailable] [-Updates] [-ProjectName <string>]
  [-Filter <string>] [-First <int>] [-Skip <int>] [-AllVersions] [-IncludePrerelease]
  [-PageSize] [<CommonParameters>]
```

With no parameters, `Get-Package` displays the list of packages installed in the default project.

Parameters

PARAMETER	DESCRIPTION
Source	The URL or folder path for the package . Local folder paths can be absolute, or relative to the current folder. If omitted, <code>Get-Package</code> searches the currently selected package source. When used with <code>-ListAvailable</code> , defaults to <code>nuget.org</code> .
ListAvailable	Lists packages available from a package source, defaulting to <code>nuget.org</code> . Shows a default of 50 packages unless <code>-PageSize</code> and/or <code>-First</code> are specified.
Updates	Lists packages that have an update available from the package source.
ProjectName	The project from which to get installed packages. If omitted, returns installed projects for the entire solution.
Filter	A filter string used to narrow down the list of packages by applying it to the package ID, description, and tags.
First	The number of packages to return from the beginning of the list. If not specified, defaults to 50.
Skip	Omits the first <int> packages from the displayed list.
AllVersions	Displays all available versions of each package instead of only the latest version.
IncludePrerelease	Includes prerelease packages in the results.

PARAMETER	DESCRIPTION
PageSize	(3.0+) When used with -ListAvailable (required), the number of packages to list before giving a prompt to continue.

None of these parameters accept pipeline input or wildcard characters.

Common Parameters

`Get-Package` supports the following [common PowerShell parameters](#): Debug, Error Action, ErrorVariable, OutBuffer, OutVariable, PipelineVariable, Verbose, WarningAction, and WarningVariable.

Examples

```
# Lists the packages installed in the current solution
Get-Package

# Lists the packages installed in a project
Get-Package -ProjectName MyProject

# Lists packages available in the current package source
Get-Package -ListAvailable

# Lists 30 packages at a time from the current source, and prompts to continue if more are available
Get-Package -ListAvailable -PageSize 30

# Lists packages with the Ninject keyword in the current source, up to 50
Get-Package -ListAvailable -Filter Ninject

# List all versions of packages matching the filter "jquery"
Get-Package -ListAvailable -Filter jquery -AllVersions

# Lists packages installed in the solution that have available updates
Get-Package -Updates

# Lists packages installed in a specific project that have available updates
Get-Package -Updates -ProjectName MyProject
```

Get-Project (Package Manager Console in Visual Studio)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Available only within the [Package Manager Console](#) in Visual Studio on Windows.

Displays information about the default or specified project. `Get-Project` specifically returns a referent to the Visual Studio DTE (Development Tools Environment) object for the project.

Syntax

```
Get-Project [[-Name] <string>] [-All] [<CommonParameters>]
```

Parameters

PARAMETER	DESCRIPTION
Name	Specifies the project to display, defaulting to the default project selected in the Package Manager Console. The <code>-Name</code> switch is itself optional.
All	Displays information for every project in the solution; the order of projects is not deterministic.

None of these parameters accept pipeline input or wildcard characters.

Common Parameters

`Get-Project` supports the following [common PowerShell parameters](#): `Debug`, `Error Action`, `ErrorVariable`, `OutBuffer`, `OutVariable`, `PipelineVariable`, `Verbose`, `WarningAction`, and `WarningVariable`.

Examples

```
# Displays information for the default project
Get-Project

# Displays information for a project in the solution
Get-Project MyProjectName

# Displays information for all projects in the solution
Get-Project -All
```

Install-Package (Package Manager Console in Visual Studio)

7/18/2019 • 2 minutes to read • [Edit Online](#)

This topic describes the command within the [Package Manager Console](#) in Visual Studio on Windows. For the generic PowerShell `Install-Package` command, see the [PowerShell PackageManagement](#) reference.

Installs a package and its dependencies into a project.

Syntax

```
Install-Package [-Id] <string> [-IgnoreDependencies] [-ProjectName <string>] [[-Source] <string>]
    [[-Version] <string>] [-IncludePrerelease] [-FileConflictAction] [-DependencyVersion]
    [-WhatIf] [<CommonParameters>]
```

In NuGet 2.8+, `Install-Package` can downgrade an existing package in your project. For example, if you have `Microsoft.AspNet.MVC 5.1.0-rc1` installed, the following command would downgrade it to 5.0.0:

```
Install-Package Microsoft.AspNet.MVC -Version 5.0.0.
```

Parameters

PARAMETER	DESCRIPTION
<code>Id</code>	(Required) The identifier of the package to install. (3.0+) The identifier can be a path or URL of a <code>packages.config</code> file or a <code>.nupkg</code> file. The <code>-Id</code> switch itself is optional.
<code>IgnoreDependencies</code>	Install only this package and not its dependencies.
<code>ProjectName</code>	The project into which to install the package, defaulting to the default project.
<code>Source</code>	The URL or folder path for the package source to search. Local folder paths can be absolute, or relative to the current folder. If omitted, <code>Install-Package</code> searches the currently selected package source.
<code>Version</code>	The version of the package to install, defaulting to the latest version.
<code>IncludePrerelease</code>	Considers prerelease packages for the install. If omitted, only stable packages are considered.
<code>FileConflictAction</code>	The action to take when asked to overwrite or ignore existing files referenced by the project. Possible values are <code>Overwrite</code> , <code>Ignore</code> , <code>None</code> , <code>OverwriteAll</code> , and (3.0+) <code>IgnoreAll</code> .

PARAMETER	DESCRIPTION
DependencyVersion	<p>The version of the dependency packages to use, which can be one of the following:</p> <ul style="list-style-type: none"> • <i>Lowest</i> (default): the lowest version • <i>HighestPatch</i>: the version with the lowest major, lowest minor, highest patch • <i>HighestMinor</i>: the version with the lowest major, highest minor, highest patch • <i>Highest</i> (default for <code>Update-Package</code> with no parameters): the highest version <p>You can set the default value using the <code>dependencyVersion</code> setting in the <code>Nuget.Config</code> file.</p>
WhatIf	Shows what would happen when running the command without actually performing the install.

None of these parameters accept pipeline input or wildcard characters.

Common Parameters

`Install-Package` supports the following [common PowerShell parameters](#): `Debug`, `Error Action`, `ErrorVariable`, `OutBuffer`, `OutVariable`, `PipelineVariable`, `Verbose`, `WarningAction`, and `WarningVariable`.

Examples

```

# Installs the latest version of Elmah from the current source into the default project
Install-Package Elmah

# Installs Glimpse 1.0.0 into the MvcApplication1 project
Install-Package Glimpse -Version 1.0.0 -Project MvcApplication1

# Installs Ninject.Mvc3 but not its dependencies from c:\temp\packages
Install-Package Ninject.Mvc3 -IgnoreDependencies -Source c:\temp\packages

# Installs the package listed on the online packages.config into the current project
# Note: the URL must end with "packages.config"
Install-Package https://raw.githubusercontent.com/linked-data-dotnet/json-ld.net/master/.nuget/packages.config

# Installs jquery 1.10.2 package, using the .nupkg file under local path of c:\temp\packages
Install-Package c:\temp\packages\jQuery.1.10.2.nupkg

# Installs the specific online package
# Note: the URL must end with ".nupkg"
Install-Package https://globalcdn.nuget.org/packages/microsoft.aspnet.mvc.5.2.3.nupkg

```

Open-PackagePage (Package Manager Console in Visual Studio)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Deprecated in 3.0+; available only within the [Package Manager Console](#) in Visual Studio on Windows.

Launches the default browser with the project, license, or report abuse URL for the specified package.

Syntax

```
Open-PackagePage [-Id] <string> [-Version] [-Source] [-License] [-ReportAbuse]
    [-PassThru] [<CommonParameters>]
```

Parameters

PARAMETER	DESCRIPTION
Id	The package ID of the desired package. The -Id switch itself is optional.
Version	The version of the package, defaulting to the latest version.
Source	The package source, defaulting to the selected source in the source drop-down.
License	Opens the browser to the package's License URL. If neither -License nor -ReportAbuse is specified, the browser opens the package's Project URL.
ReportAbuse	Opens the browser to the package's Report Abuse URL. If neither -License nor -ReportAbuse is specified, the browser opens the package's Project URL.
PassThru	Displays the URL; use with -WhatIf to suppress opening the browser.

None of these parameters accept pipeline input or wildcard characters.

Common Parameters

`Open-PackagePage` supports the following [common PowerShell parameters](#): `Debug`, `Error Action`, `ErrorVariable`, `OutBuffer`, `OutVariable`, `PipelineVariable`, `Verbose`, `WarningAction`, and `WarningVariable`.

Examples

```
# Opens a browser with the Ninject package's project page
Open-PackagePage Ninject

# Opens a browser with the Ninject package's license page
Open-PackagePage Ninject -License

# Opens a browser with the Ninject package's report abuse page
Open-PackagePage Ninject -ReportAbuse

# Assigns the license URL to the variable, $url, without launching the browser
$url = Open-PackagePage Ninject -License -PassThru -WhatIf
```

Sync-Package (Package Manager Console in Visual Studio)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Version 3.0+; available only within the [Package Manager Console](#) in Visual Studio on Windows.

Gets the version of installed package from specified (or default) project and synchronizes the version to the rest of projects in the solution.

Syntax

```
Sync-Package [-Id] <string> [-IgnoreDependencies] [-ProjectName <string>] [[-Version] <string>]
  [[-Source] <string>] [-IncludePrerelease] [-FileConflictAction] [-DependencyVersion]
  [-WhatIf] [<CommonParameters>]
```

Parameters

PARAMETER	DESCRIPTION
Id	(Required) The identifier of the package to sync. The -Id switch itself is optional.
IgnoreDependencies	Install only this package and not its dependencies.
ProjectName	The project to sync the package from, defaulting to the default project.
Version	The version of the package to sync, defaulting to the currently installed version.
Source	The URL or folder path for the package source to search. Local folder paths can be absolute, or relative to the current folder. If omitted, <code>Sync-Package</code> searches the currently selected package source.
IncludePrerelease	Includes prerelease packages in the sync.
FileConflictAction	The action to take when asked to overwrite or ignore existing files referenced by the project. Possible values are <i>Overwrite</i> , <i>Ignore</i> , <i>None</i> , <i>OverwriteAll</i> , and (3.0+) <i>IgnoreAll</i> .

PARAMETER	DESCRIPTION
DependencyVersion	<p>The version of the dependency packages to use, which can be one of the following:</p> <ul style="list-style-type: none"> • <i>Lowest</i> (default): the lowest version • <i>HighestPatch</i>: the version with the lowest major, lowest minor, highest patch • <i>HighestMinor</i>: the version with the lowest major, highest minor, highest patch • <i>Highest</i> (default for <code>Update-Package</code> with no parameters): the highest version <p>You can set the default value using the <code>dependencyVersion</code> setting in the <code>Nuget.Config</code> file.</p>
WhatIf	Shows what would happen when running the command without actually performing the sync.

None of these parameters accept pipeline input or wildcard characters.

Common Parameters

`Sync-Package` supports the following [common PowerShell parameters](#): `Debug`, `Error Action`, `ErrorVariable`, `OutBuffer`, `OutVariable`, `PipelineVariable`, `Verbose`, `WarningAction`, and `WarningVariable`.

Examples

```

# Sync the Elmah package installed in the default project into the other projects in the solution
Sync-Package Elmah

# Sync the Elmah package installed in the ClassLibrary1 project into other projects in the solution
Sync-Package Elmah -ProjectName ClassLibrary1

# Sync Microsoft.AspNet.package but not its dependencies into the other projects in the solution
Sync-Package Microsoft.AspNet.Mvc -IgnoreDependencies

# Sync jQuery.Validation and install the highest version of jQuery (a dependency) from the package source
Sync-Package jQuery.Validation -DependencyVersion highest

```

Uninstall-Package (Package Manager Console in Visual Studio)

7/18/2019 • 2 minutes to read • [Edit Online](#)

This topic describes the command within the [Package Manager Console](#) in Visual Studio on Windows. For the generic PowerShell `Uninstall-Package` command, see the [PowerShell PackageManagement](#) reference.

Removes a package from a project, optionally removing its dependencies. If other packages depend on this package, the command will fail unless the `-Force` option is specified.

Syntax

```
Uninstall-Package [-Id] <string> [-RemoveDependencies] [-ProjectName <string>] [-Force]
  [-Version <string>] [-WhatIf] [<CommonParameters>]
```

If other packages depend on this package, the command will fail unless the `-Force` option is specified.

Parameters

PARAMETER	DESCRIPTION
<code>Id</code>	(Required) The identifier of the package to uninstall. The <code>-Id</code> switch itself is optional.
<code>Version</code>	The version of the package to uninstall, defaulting to the currently installed version.
<code>RemoveDependencies</code>	Uninstall the package and its unused dependencies. That is, if any dependency has another package that depends on it, it's skipped.
<code>ProjectName</code>	The project from which to uninstall the package, defaulting to the default project.
<code>Force</code>	Forces a package to be uninstalled, even if other packages depend on it.
<code>WhatIf</code>	Shows what would happen when running the command without actually performing the uninstall.

None of these parameters accept pipeline input or wildcard characters.

Common Parameters

`Uninstall-Package` supports the following [common PowerShell parameters](#): `Debug`, `Error Action`, `ErrorVariable`, `OutBuffer`, `OutVariable`, `PipelineVariable`, `Verbose`, `WarningAction`, and `WarningVariable`.

Examples

```
# Uninstalls the Elmah package from the default project
Uninstall-Package Elmah

# Uninstalls the Elmah package and all its unused dependencies
Uninstall-Package Elmah -RemoveDependencies

# Uninstalls the Elmah package even if another package depends on it
Uninstall-Package Elmah -Force
```

Update-Package (Package Manager Console in Visual Studio)

7/18/2019 • 2 minutes to read • [Edit Online](#)

Available only within the [NuGet Package Manager Console](#) in Visual Studio on Windows.

Updates a package and its dependencies, or all packages in a project, to a newer version.

Syntax

```
Update-Package [-Id] <string> [-IgnoreDependencies] [-ProjectName <string>] [-Version <string>]
    [-Safe] [-Source <string>] [-IncludePrerelease] [-Reinstall] [-FileConflictAction]
    [-DependencyVersion] [-ToHighestPatch] [-ToHighestMinor] [-WhatIf] [<CommonParameters>]
```

In NuGet 2.8+, `Update-Package` can be used to downgrade an existing package in your project. For example, if you have `Microsoft.AspNet.MVC` 5.1.0-rc1 installed, the following command would downgrade it to 5.0.0:

```
Update-Package Microsoft.AspNet.MVC -Version 5.0.0.
```

Parameters

PARAMETER	DESCRIPTION
Id	The identifier of the package to update. If omitted, updates all packages. The <code>-Id</code> switch itself is optional.
IgnoreDependencies	Skips updating the package's dependencies.
ProjectName	The name of the project containing the packages to update, defaulting to all projects.
Version	The version to use for the upgrade, defaulting to the latest version. In NuGet 3.0+, the version value must be one of <i>Lowest</i> , <i>Highest</i> , <i>HighestMinor</i> , or <i>HighestPatch</i> (equivalent to <code>-Safe</code>).
Safe	Constrains upgrades to only versions with the same Major and Minor version as the currently installed package.
Source	The URL or folder path for the package source to search. Local folder paths can be absolute, or relative to the current folder. If omitted, <code>Update-Package</code> searches the currently selected package source.
IncludePrerelease	Includes prerelease packages for updates.
Reinstall	Resinstalls packages using their currently installed versions. See Reinstalling and updating packages .

PARAMETER	DESCRIPTION
FileConflictAction	The action to take when asked to overwrite or ignore existing files referenced by the project. Possible values are <i>Overwrite</i> , <i>Ignore</i> , <i>None</i> , <i>OverwriteAll</i> , and <i>IgnoreAll</i> (3.0+).
DependencyVersion	<p>The version of the dependency packages to use, which can be one of the following:</p> <ul style="list-style-type: none"> • <i>Lowest</i> (default): the lowest version • <i>HighestPatch</i>: the version with the lowest major, lowest minor, highest patch • <i>HighestMinor</i>: the version with the lowest major, highest minor, highest patch • <i>Highest</i> (default for <code>Update-Package</code> with no parameters): the highest version <p>You can set the default value using the <code>dependencyVersion</code> setting in the <code>Nuget.Config</code> file.</p>
ToHighestPatch	equivalent to <code>-Safe</code> .
ToHighestMinor	Constrains upgrades to only versions with the same Major version as the currently installed package.
WhatIf	Shows what would happen when running the command without actually performing the update.

None of these parameters accept pipeline input or wildcard characters.

Common Parameters

`Update-Package` supports the following [common PowerShell parameters](#): `Debug`, `Error Action`, `ErrorVariable`, `OutBuffer`, `OutVariable`, `PipelineVariable`, `Verbose`, `WarningAction`, and `WarningVariable`.

Examples

```
# Updates all packages in every project of the solution
Update-Package

# Updates every package in the MvcApplication1 project
Update-Package -ProjectName MvcApplication1

# Updates the Elmah package in every project to the latest version
Update-Package Elmah

# Updates the Elmah package to version 1.1.0 in every project showing optional -Id usage
Update-Package -Id Elmah -Version 1.1.0

# Updates the Elmah package within the MvcApplication1 project to the highest "safe" version.
# For example, if Elmah version 1.0.0 of a package is installed, and versions 1.0.1, 1.0.2,
# and 1.1 are available in the feed, the -Safe parameter updates the package to 1.0.2 instead
# of 1.1 as it would otherwise.
Update-Package Elmah -ProjectName MvcApplication1 -Safe

# Reinstall the same version of the original package, but with the latest version of dependencies
# (subject to version constraints). If this command rolls a dependency back to an earlier version,
# use Update-Package <dependency_name> to reinstall that one dependency without affecting the
# dependent package.
Update-Package Elmah -reinstall

# Reinstall the Elmah package in just MyProject
Update-Package Elmah -ProjectName MyProject -reinstall

# Reinstall the same version of the original package without touching dependencies.
Update-Package Elmah -reinstall -ignoreDependencies
```

NuGet Server API

7/23/2019 • 5 minutes to read • [Edit Online](#)

The NuGet Server API is a set of HTTP endpoints that can be used to download packages, fetch metadata, publish new packages, and perform most other operations available in the official NuGet clients.

This API is used by the NuGet client in Visual Studio, nuget.exe, and the .NET CLI to perform NuGet operations such as `dotnet restore`, search in the Visual Studio UI, and `nuget.exe push`.

Note in some cases, nuget.org has additional requirements that are not enforced by other package sources. These differences are documented by the [nuget.org Protocols](#).

For a simple enumeration and download of available nuget.exe versions, see the [tools.json](#) endpoint.

Service index

The entry point for the API is a JSON document in a well known location. This document is called the **service index**. The location of the service index for nuget.org is <https://api.nuget.org/v3/index.json>.

This JSON document contains a list of *resources* which provide different functionality and fulfill different use cases.

Clients that support the API should accept one or more of these service index URL as the means of connecting to the respective package sources.

For more information about the service index, see [its API reference](#).

Versioning

The API is version 3 of NuGet's HTTP protocol. This protocol is sometimes referred to as "the V3 API". These reference documents will refer to this version of the protocol simply as "the API."

The service index schema version is indicated by the `version` property in the service index. The API mandates that the version string has a major version number of `3`. As non-breaking changes are made to the service index schema, the version string's minor version will be increased.

Older clients (such as nuget.exe 2.x) do not support the V3 API and only support the older V2 API, which is not documented here.

The NuGet V3 API is named as such because it's the successor of the V2 API, which was the OData-based protocol implemented by the 2.x version of the official NuGet client. The V3 API was first supported by the 3.0 version of the official NuGet client and is still the latest major protocol version supported by the NuGet client, 4.0 and on.

Non-breaking protocol changes have been made to the API since it was first released.

Resources and schema

The **service index** describes a variety of resources. The current set of supported resources are as follows:

RESOURCE NAME	REQUIRED	DESCRIPTION
Catalog	no	Full record of all package events.

RESOURCE NAME	REQUIRED	DESCRIPTION
PackageBaseAddress	yes	Get package content (.nupkg).
PackageDetailsUriTemplate	no	Construct a URL to access a package details web page.
PackagePublish	yes	Push and delete (or unlist) packages.
RegistrationsBaseUrl	yes	Get package metadata.
ReportAbuseUriTemplate	no	Construct a URL to access a report abuse web page.
RepositorySignatures	no	Get certificates used for repository signing.
SearchAutocompleteService	no	Discover package IDs and versions by substring.
SearchQueryService	yes	Filter and search for packages by keyword.
SymbolPackagePublish	no	Push symbol packages.

In general, all non-binary data returned by a API resource are serialized using JSON. The response schema returned by each resource in the service index is defined individually for that resource. For more information about each resource, see the topics listed above.

In the future, as the protocol evolves, new properties may be added to JSON responses. For the client to be future-proof, the implementation should not assume that the response schema is final and cannot include extra data. All properties that the implementation does not understand should be ignored.

NOTE

When a source does not implement `SearchAutocompleteService` any autocomplete behavior should be disabled gracefully. When `ReportAbuseUriTemplate` is not implemented, the official NuGet client falls back to nuget.org's report abuse URL (tracked by [NuGet/Home#4924](#)). Other clients may opt to simply not show a report abuse URL to the user.

Undocumented resources on nuget.org

The V3 service index on nuget.org has some resources that are not documented above. There are a few reasons for not documenting a resource.

First, we don't document resources used as an implementation detail of nuget.org. The `SearchGalleryQueryService` falls into this category. [NuGetGallery](#) uses this resource to delegate some V2 (OData) queries to our search index instead of using the database. This resource was introduced for scalability reasons and is not intended for external use.

Second, we don't document resources that never shipped in an RTM version of the official client.

`PackageDisplayMetadataUriTemplate` and `PackageVersionDisplayMetadataUriTemplate` fall into this category.

Thirdly, we don't document resources that are tightly coupled with the V2 protocol, which itself is intentionally undocumented. The `LegacyGallery` resource falls into this category. This resource allows a V3 service index to point to a corresponding V2 source URL. This resource supports the `nuget.exe list`.

If a resource is not documented here, we *strongly* recommend that you do not take a dependency on them. We may remove or change the behavior of these undocumented resources which could break your implementation in unexpected ways.

Timestamps

All timestamps returned by the API are UTC or are otherwise specified using [ISO 8601](#) representation.

HTTP methods

VERB	USE
GET	Performs a read-only operation, typically retrieving data.
HEAD	Fetches the response headers for the corresponding <code>GET</code> request.
PUT	Creates a resource that doesn't exist or, if it does exist, updates it. Some resources may not support update.
DELETE	Deletes or unlists a resource.

HTTP status codes

CODE	DESCRIPTION
200	Success, and there is a response body.
201	Success, and the resource was created.
202	Success, the request has been accepted but some work may still be incomplete and completed asynchronously.
204	Success, but there is no response body.
301	A permanent redirect.
302	A temporary redirect.
400	The parameters in the URL or in the request body aren't valid.
401	The provided credentials are invalid.
403	The action is not allowed given the provided credentials.
404	The requested resource doesn't exist.
409	The request conflicts with an existing resource.
500	The service has encountered an unexpected error.
503	The service is temporarily unavailable.

Any `GET` request made to a API endpoint may return an HTTP redirect (301 or 302). Clients should gracefully handle such redirects by observing the `Location` header and issuing a subsequent `GET`. Documentation concerning specific endpoints will not explicitly call out where redirects may be used.

In the case of a 500-level status code, the client can implement a reasonable retry mechanism. The official NuGet client retries three times when encountering any 500-level status code or TCP/DNS error.

HTTP request headers

NAME	DESCRIPTION
X-NuGet-ApiKey	Required for push and delete, see PackagePublish resource
X-NuGet-Client-Version	Deprecated and replaced by <code>X-NuGet-Protocol-Version</code>
X-NuGet-Protocol-Version	Required in certain cases only on nuget.org, see nuget.org protocols
X-NuGet-Session-Id	<i>Optional.</i> NuGet clients v4.7+ identify HTTP requests that are part of the same NuGet client session.

The `x-NuGet-Session-Id` has a single value for all operations related to a single restore in `PackageReference`. For other scenarios such as autocomplete and `packages.config` restore there may be several different session ID's due to how the code is factored.

Authentication

Authentication is left up to the package source implementation to define. For nuget.org, only the `PackagePublish` resource requires authentication via a special API key header. See [PackagePublish resource](#) for details.

Autocomplete

8/15/2019 • 3 minutes to read • [Edit Online](#)

It is possible to build a package ID and version autocomplete experience using the V3 API. The resource used for making autocomplete queries is the `SearchAutocompleteService` resource found in the [service index](#).

Versioning

The following `@type` values are used:

<code>@TYPE</code> VALUE	NOTES
SearchAutocompleteService	The initial release
SearchAutocompleteService/3.0.0-beta	Alias of <code>SearchAutocompleteService</code>
SearchAutocompleteService/3.0.0-rc	Alias of <code>SearchAutocompleteService</code>

Base URL

The base URL for the following APIs is the value of the `@id` property associated with one of the aforementioned resource `@type` values. In the following document, the placeholder base URL `{@id}` will be used.

HTTP Methods

All URLs found in the registration resource support the HTTP methods `GET` and `HEAD`.

Search for package IDs

The first autocomplete API supports searching for part of a package ID string. This is great when you want to provide a package typeahead feature in a user interface integrated with a NuGet package source.

A package with only unlisted versions will not appear in the results.

```
GET {@id}?q={QUERY}&skip={SKIP}&take={TAKE}&prerelease={PRERELEASE}&semVerLevel={SEMVERLEVEL}
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
q	URL	string	no	The string to compare against package IDs
skip	URL	integer	no	The number of results to skip, for pagination
take	URL	integer	no	The number of results to return, for pagination

NAME	IN	TYPE	REQUIRED	NOTES
prerelease	URL	boolean	no	<code>true</code> or <code>false</code> determining whether to include pre-release packages
semVerLevel	URL	string	no	A SemVer 1.0.0 version string

The autocomplete query `q` is parsed in a manner that is defined by the server implementation. nuget.org supports querying for the prefix of package ID tokens, which are pieces of the ID produced by splitting the original by camel case and symbol characters.

The `skip` parameter defaults to 0.

The `take` parameter should be an integer greater than zero. The server implementation may impose a maximum value.

If `prerelease` is not provided, pre-release packages are excluded.

The `semVerLevel` query parameter is used to opt-in to [SemVer 2.0.0 packages](#). If this query parameter is excluded, only package IDs with SemVer 1.0.0 compatible versions will be returned (with the [standard NuGet versioning](#) caveats, such as version strings with 4 integer pieces). If `semVerLevel=2.0.0` is provided, both SemVer 1.0.0 and SemVer 2.0.0 compatible packages will be returned. See the [SemVer 2.0.0 support for nuget.org](#) for more information.

Response

The response is JSON document containing up to `take` autocomplete results.

The root JSON object has the following properties:

NAME	TYPE	REQUIRED	NOTES
totalHits	integer	yes	The total number of matches, disregarding <code>skip</code> and <code>take</code>
data	array of strings	yes	The package IDs matched by the request

Sample request

```
GET https://api-v2v3search-0.nuget.org/autocomplete?q=storage&prerelease=true
```

Sample response

```
{
  "totalHits": 571,
  "data": [
    "WindowsAzure.Storage",
    "Storage.Net",
    "CK.Storage",
    "NCL.Storage",
    "DK.Storage",
    "Nine.Storage.Test",
    "Touch.Storage.Aws",
    "StorageAPIClient",
    "StorageAccess",
    "Storage.Net.Microsoft.Azure.Storage",
    "UnofficialAzure.StorageClient",
    "StorageAccess12",
    "AWSSDK.StorageGateway",
    "StorageExtensions",
    "Cloud.Storage",
    "lighthouse.storage",
    "ZU.Storage.Redis",
    "Magicodes.Storage",
    "Masticore.Storage",
    "hq.storage"
  ]
}
```

Enumerate package versions

Once a package ID is discovered using the previous API, a client can use the autocomplete API to enumerate package versions for a provided package ID.

A package version that is unlisted will not appear in the results.

```
GET {@id}?id={ID}&prerelease={PRERELEASE}&semVerLevel={SEMVERLEVEL}
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
id	URL	string	yes	The package ID to fetch versions for
prerelease	URL	boolean	no	<code>true</code> or <code>false</code> determining whether to include pre-release packages
semVerLevel	URL	string	no	A SemVer 2.0.0 version string

If `prerelease` is not provided, pre-release packages are excluded.

The `semVerLevel` query parameter is used to opt-in to SemVer 2.0.0 packages. If this query parameter is excluded, only SemVer 1.0.0 versions will be returned. If `semVerLevel=2.0.0` is provided, both SemVer 1.0.0 and SemVer 2.0.0 versions will be returned. See the [SemVer 2.0.0 support for nuget.org](#) for more information.

Response

The response is JSON document containing all package versions of the provided package ID, filtering by the given

query parameters.

The root JSON object has the following property:

NAME	TYPE	REQUIRED	NOTES
data	array of strings	yes	The package versions matched by the request

The package versions in the `data` array may contain SemVer 2.0.0 build metadata (e.g. `1.0.0+metadata`) if the `semVerLevel=2.0.0` is provided in the query string.

Sample request

```
GET https://api-v2v3search-0.nuget.org/autocomplete?id=nuget.protocol&prerelease=true
```

Sample response

```
{
  "data": [
    "4.3.0-preview3-4168",
    "4.3.0-preview4",
    "4.3.0-rtm-4324",
    "4.3.0",
    "4.4.0-preview3-4475",
    "4.4.0"
  ]
}
```

Catalog

7/1/2019 • 15 minutes to read • [Edit Online](#)

The **catalog** is a resource that records all package operations on a package source, such as creations and deletions. The catalog resource has the `Catalog` type in the [service index](#). You can use this resource to [query for all published packages](#).

NOTE

Because the catalog is not used by the official NuGet client, not all package sources implement the catalog.

NOTE

Currently, the nuget.org catalog is not available in China. For more details, see [NuGet/NuGetGallery#4949](#).

Versioning

The following `@type` value is used:

<code>@TYPE</code> VALUE	NOTES
Catalog/3.0.0	The initial release

Base URL

The entry point URL for the following APIs is the value of the `@id` property associated with the aforementioned resource `@type` values. This topic uses the placeholder URL `{@id}`.

HTTP methods

All URLs found in the catalog resource support only the HTTP methods `GET` and `HEAD`.

Catalog index

The catalog index is a document in a well-known location that has a list of catalog items, ordered chronologically. It is the entry point of the catalog resource.

The index is comprised of catalog pages. Each catalog page contains catalog items. Each catalog item represents an event concerning a single package at a point in time. A catalog item can represent a package that was created, unlisted, relisted, or deleted from the package source. By processing the catalog items in chronological order, the client can build an up-to-date view of every package that exists on the V3 package source.

In short, catalog blobs have the following hierarchical structure:

- **Index:** the entry point for the catalog.
- **Page:** a grouping of catalog items.
- **Leaf:** a document representing a catalog item, which is a snapshot of a single package's state.

Each catalog object has a property called the `commitTimestamp` representing when the item was added to the

catalog. Catalog items are added to a catalog page in batches called commits. All catalog items in the same commit have the same commit timestamp (`commitTimeStamp`) and commit ID (`commitId`). Catalog items placed in the same commit represent events that happened around the same point in time on the package source. There is no ordering within a catalog commit.

Because each package ID and version is unique, there will never be more than one catalog item in a commit. This ensures that catalog items for a single package can always be unambiguously ordered with respect to commit timestamp.

There is never be more than one commit to the catalog per `commitTimeStamp`. In other words, the `commitId` is redundant with the `commitTimeStamp`.

In contrast to the [package metadata resource](#), which is indexed by package ID, the catalog is indexed (and queryable) only by time.

Catalog items are always added to the catalog in a monotonically increasing, chronological order. This means that if a catalog commit is added at time X then no catalog commit will ever be added with a time less than or equal to X.

The following request fetches the catalog index.

```
GET {@id}
```

The catalog index is a JSON document that contains an object with the following properties:

NAME	TYPE	REQUIRED	NOTES
commitId	string	yes	A unique ID associated with the most recent commit
commitTimeStamp	string	yes	A timestamp of the most recent commit
count	integer	yes	The number of pages in the index
items	array of objects	yes	A array of objects, each object representing a page

Each element in the `items` array is an object with some minimal details about each page. These page objects do not contain the catalog leaves (items). The order of the elements in this array is not defined. Pages can be ordered by the client in memory using their `commitTimeStamp` property.

As new pages are introduced, the `count` will be incremented and new objects will appear in the `items` array.

As items are added to the catalog, the index's `commitId` will change and the `commitTimeStamp` will increase. These two properties are essentially a summary over all page `commitId` and `commitTimeStamp` values in the `items` array.

Catalog page object in the index

The catalog page objects found in the catalog index's `items` property have the following properties:

NAME	TYPE	REQUIRED	NOTES
@id	string	yes	The URL to fetch catalog page

NAME	TYPE	REQUIRED	NOTES
commitId	string	yes	A unique ID associated with the most recent commit in this page
commitTimeStamp	string	yes	A timestamp of the most recent commit in this page
count	integer	yes	The number of items in the catalog page

In contrast to the [package metadata resource](#) which in some cases inlines leaves into the index, catalog leaves are never inlined into the index and must always be fetched by using the page's `@id` URL.

Sample request

```
GET https://api.nuget.org/v3/catalog0/index.json
```

Sample response

```
{
  "commitId": "3d698852-eefb-48ed-8f55-9ee357540d20",
  "commitTimeStamp": "2017-10-31T23:33:17.0954363Z",
  "count": 3,
  "items": [
    {
      "@id": "https://api.nuget.org/v3/catalog0/page0.json",
      "commitId": "3a4df280-3d86-458e-a713-4c91ca261fef",
      "commitTimeStamp": "2015-02-01T06:30:11.7477681Z",
      "count": 540
    },
    {
      "@id": "https://api.nuget.org/v3/catalog0/page1.json",
      "commitId": "8bcd3cbf-74f0-47a2-a7ae-b7ecc50005d3",
      "commitTimeStamp": "2015-02-01T06:39:53.9553899Z",
      "count": 540
    },
    {
      "@id": "https://api.nuget.org/v3/catalog0/page2.json",
      "commitId": "3d698852-eefb-48ed-8f55-9ee357540d20",
      "commitTimeStamp": "2017-10-31T23:33:17.0954363Z",
      "count": 47
    }
  ]
}
```

Catalog page

The catalog page is a collection of catalog items. It is a document fetched using one of the `@id` values found in the catalog index. The URL to a catalog page is not intended to be predictable and should be discovered using only the catalog index.

New catalog items are added to the page in the catalog index only with the highest commit timestamp or to a new page. Once a page with a higher commit timestamp is added to the catalog, older pages are never added to or changed.

The catalog page document is a JSON object with the following properties:

NAME	TYPE	REQUIRED	NOTES
commitId	string	yes	A unique ID associated with the most recent commit in this page
commitTimeStamp	string	yes	A timestamp of the most recent commit in this page
count	integer	yes	The number of items in the page
items	array of objects	yes	The catalog items in this page
parent	string	yes	A URL to the catalog index

Each element in the `items` array is an object with some minimal details about the catalog item. These item objects do not contain all of the catalog item's data. The order of the items in the page's `items` array is not defined. Items can be ordered by the client in memory using their `commitTimeStamp` property.

The number of catalog items in a page is defined by server implementation. For nuget.org, there are at most 550 items in each page, although the actual number may be smaller for some pages depending on the size of the next commit batch at the point in time.

As new items are introduced, the `count` is incremented and new catalog item objects appear in the `items` array.

As items are added to the page, the `commitId` changes and the `commitTimeStamp` increases. These two properties are essentially a summary over all `commitId` and `commitTimeStamp` values in the `items` array.

Catalog item object in a page

The catalog item objects found in the catalog page's `items` property have the following properties:

NAME	TYPE	REQUIRED	NOTES
<code>@id</code>	string	yes	The URL to fetch the catalog item
<code>@type</code>	string	yes	The type of the catalog item
<code>commitId</code>	string	yes	The commit ID associated with this catalog item
<code>commitTimeStamp</code>	string	yes	The commit timestamp of this catalog item
<code>nuget:id</code>	string	yes	The package ID that this leaf is related to
<code>nuget:version</code>	string	yes	The package version that this leaf is related to

The `@type` value will be one of the following two values:

1. `nuget:PackageDetails` : this corresponds to `PackageDetails` type in the catalog leaf document.
2. `nuget:PackageDelete` : this corresponds to the `PackageDelete` type in the catalog leaf document.

For more details about what each type means, see the [corresponding items type](#) below.

Sample request

```
GET https://api.nuget.org/v3/catalog0/page2926.json
```

Sample response

```
{
  "commitId": "616117f5-d9dd-4664-82b9-74d87169bbe9",
  "commitTimeStamp": "2017-10-31T23:30:32.4197849Z",
  "count": 5,
  "parent": "https://api.nuget.org/v3/catalog0/index.json",
  "items": [
    {
      "@id": "https://api.nuget.org/v3/catalog0/data/2017.10.31.23.30.32/util.biz.payments.0.0.4-preview.json",
      "@type": "nuget:PackageDetails",
      "commitId": "616117f5-d9dd-4664-82b9-74d87169bbe9",
      "commitTimeStamp": "2017-10-31T23:30:32.4197849Z",
      "nuget:id": "Util.Biz.Payments",
      "nuget:version": "0.0.4-preview"
    },
    {
      "@id": "https://api.nuget.org/v3/catalog0/data/2017.10.31.23.28.02/util.biz.0.0.4-preview.json",
      "@type": "nuget:PackageDetails",
      "commitId": "820340b2-97e3-4f93-b82e-bc85550a6560",
      "commitTimeStamp": "2017-10-31T23:28:02.788239Z",
      "nuget:id": "Util.Biz",
      "nuget:version": "0.0.4-preview"
    },
    {
      "@id": "https://api.nuget.org/v3/catalog0/data/2017.10.31.22.31.22/sourcecode.clay.data.1.0.0-preview1-00258.json",
      "@type": "nuget:PackageDetails",
      "commitId": "cae34527-ffc7-4e96-884f-7cf95a32dbdd",
      "commitTimeStamp": "2017-10-31T22:31:22.5169519Z",
      "nuget:id": "SourceCode.Clay.Data",
      "nuget:version": "1.0.0-preview1-00258"
    },
    {
      "@id": "https://api.nuget.org/v3/catalog0/data/2017.10.31.22.31.22/sourcecode.clay.1.0.0-preview1-00258.json",
      "@type": "nuget:PackageDetails",
      "commitId": "cae34527-ffc7-4e96-884f-7cf95a32dbdd",
      "commitTimeStamp": "2017-10-31T22:31:22.5169519Z",
      "nuget:id": "SourceCode.Clay",
      "nuget:version": "1.0.0-preview1-00258"
    },
    {
      "@id": "https://api.nuget.org/v3/catalog0/data/2017.10.31.22.31.22/sourcecode.clay.json.1.0.0-preview1-00258.json",
      "@type": "nuget:PackageDetails",
      "commitId": "cae34527-ffc7-4e96-884f-7cf95a32dbdd",
      "commitTimeStamp": "2017-10-31T22:31:22.5169519Z",
      "nuget:id": "SourceCode.Clay.Json",
      "nuget:version": "1.0.0-preview1-00258"
    }
  ]
}
```

Catalog leaf

The catalog leaf contains metadata about a specific package ID and version at some point in time. It is a document

fetched using the `@id` value found in a catalog page. The URL to a catalog leaf is not intended to be predictable and should be discovered using only a catalog page.

The catalog leaf document is a JSON object with the following properties:

NAME	TYPE	REQUIRED	NOTES
<code>@type</code>	string or array of strings	yes	The type(s) of the catalog item
<code>catalog:commitId</code>	string	yes	A commit ID associated with this catalog item
<code>catalog:commitTimeStamp</code>	string	yes	The commit timestamp of this catalog item
<code>id</code>	string	yes	The package ID of the catalog item
<code>published</code>	string	yes	The published date of the package catalog item
<code>version</code>	string	yes	The package version of the catalog item

Item types

The `@type` property is a string or array of strings. For convenience, if the `@type` value is a string, it should be treated as any array of size one. Not all possible values for `@type` are documented. However, each catalog item has exactly one of the two following string type values:

1. `PackageDetails` : represents a snapshot of package metadata
2. `PackageDelete` : represents a package that was deleted

Package details catalog items

Catalog items with the type `PackageDetails` contain a snapshot of package metadata for a specific package (ID and version combination). A package details catalog item is produced when a package source encounters any of the following scenarios:

1. A package is **pushed**.
2. A package is **listed**.
3. A package is **unlisted**.
4. A package is **reflowed**.

A package reflow is an administrative gesture that essentially generates a fake push of an existing package with no changes to the package itself. On nuget.org, a reflow is used after fixing a bug in one of the background jobs which consume the catalog.

Clients consuming the catalog items should not attempt to determine which of these scenarios produced the catalog item. Instead, the client should simply update any maintained view or index with the metadata contained in the catalog item. Furthermore, duplicate or redundant catalog items should be handled gracefully (idempotently).

Package details catalog items have the following properties in addition to those [included on all catalog leaves](#).

NAME	TYPE	REQUIRED	NOTES
authors	string	no	
created	string	no	A timestamp of when the package was first created. Fallback property: <code>published</code> .
dependencyGroups	array of objects	no	The dependencies of the package, grouped by target framework (same format as the package metadata resource)
deprecation	object	no	The deprecation associated with the package (same format as the package metadata resource)
description	string	no	
iconUrl	string	no	
isPrerelease	boolean	no	Whether or not the package version is prerelease. Can be detected from <code>version</code> .
language	string	no	
licenseUrl	string	no	
listed	boolean	no	Whether or not the package is listed
minClientVersion	string	no	
packageHash	string	yes	The hash of the package, encoding using standard base 64
packageHashAlgorithm	string	yes	
packageSize	integer	yes	The size of the package .nupkg in bytes
projectUrl	string	no	
releaseNotes	string	no	
requireLicenseAgreement	boolean	no	Assume <code>false</code> if excluded
summary	string	no	
tags	array of strings	no	

NAME	TYPE	REQUIRED	NOTES
title	string	no	
verbatimVersion	string	no	The version string as it's originally found in the .nuspec

The package `version` property is the full version string after normalization. This means that SemVer 2.0.0 build data can be included here.

The `created` timestamp is when the package was first received by the package source, which is typically a short time before the catalog item's commit timestamp.

The `packageHashAlgorithm` is a string defined by the server implementation representing the hashing algorithm used to produce the `packageHash`. nuget.org always used the `packageHashAlgorithm` value of `SHA512`.

The `published` timestamp is the time when the package was last listed.

NOTE

On nuget.org, the `published` value is set to the year 1900 when the package is unlisted.

Sample request

GET <https://api.nuget.org/v3/catalog0/data/2015.02.01.11.18.40/windowsazure.storage.1.0.0.json>

Sample response

```
{
  "@type": [
    "PackageDetails",
    "catalog:Permalink"
  ],
  "authors": "NuGet.org Team",
  "catalog:commitId": "49fe04d8-5694-45a5-9822-3be61bda871b",
  "catalog:commitTimeStamp": "2015-02-01T11:18:40.8589193Z",
  "created": "2011-12-02T20:21:23.74Z",
  "description": "This package is an example for the V3 protocol.",
  "deprecation": {
    "reasons": [
      "Legacy",
      "HasCriticalBugs",
      "Other"
    ],
    "message": "This package is an example--it should not be used!",
    "alternatePackage": {
      "id": "Newtonsoft.JSON",
      "range": "12.0.2"
    }
  },
  "iconUrl": "https://www.nuget.org/Content/gallery/img/default-package-icon.svg",
  "id": "NuGet.Protocol.V3.Example",
  "isPrerelease": false,
  "language": "en-US",
  "licenseUrl": "http://www.opensource.org/licenses/ms-pl",
  "packageHash": "2edCwKLcbcFJpsAwa883BLtOy8bZpWwbQpiIb71E74k5t2f2WzXEGWbPwntRleUEgSrcxJrh90rm/TAmg04NQ==",
  "packageHashAlgorithm": "SHA512",
  "packageSize": 118348,
  "projectUrl": "https://github.com/NuGet/NuGetGallery",
  "published": "1900-01-01T00:00:00Z",
  "requireLicenseAcceptance": false,
  "title": "NuGet V3 Protocol Example",
  "version": "1.0.0"
}
```

```

"version": "1.0.0",
"dependencyGroups": [
{
  "@id": "https://api.nuget.org/v3/catalog0/data/2015.02.01.11.18.40/windowsazure.storage.1.0.0.json#dependencygroup",
  "@type": "PackageDependencyGroup",
  "dependencies": [
    {
      "@id": "https://api.nuget.org/v3/catalog0/data/2015.02.01.11.18.40/windowsazure.storage.1.0.0.json#dependencygroup/aspnet.suppressformsredirect",
      "@type": "PackageDependency",
      "id": "aspnet.suppressformsredirect",
      "range": "[0.0.1.4, )"
    },
    {
      "@id": "https://api.nuget.org/v3/catalog0/data/2015.02.01.11.18.40/windowsazure.storage.1.0.0.json#dependencygroup/webactivator",
      "@type": "PackageDependency",
      "id": "WebActivator",
      "range": "[1.4.4, )"
    },
    {
      "@id": "https://api.nuget.org/v3/catalog0/data/2015.02.01.11.18.40/windowsazure.storage.1.0.0.json#dependencygroup/webapi.all",
      "@type": "PackageDependency",
      "id": "WebApi.All",
      "range": "[0.5.0, )"
    }
  ],
  "targetFramework": ".NETFramework4.6"
}
],
"tags": [
  "NuGet",
  "V3",
  "Protocol",
  "Example"
]
}

```

Package delete catalog items

Catalog items with the type `PackageDelete` contain a minimal set of information indicating to catalog clients that a package has been deleted from the package source and is no longer available for any package operation (such as restore).

NOTE

It is possible for a package to be deleted and later republished using the same package ID and version. On nuget.org, this is a very rare case as it breaks the official client's assumption that a package ID and version imply a specific package content. For more information about package deletion on nuget.org, see [our policy](#).

Package delete catalog items have no additional properties in addition to those [included on all catalog leaves](#).

The `version` property is the original version string found in the package .nuspec.

The `published` property is the time when package was deleted, which is typically as short time before the catalog item's commit timestamp.

Sample request

GET https://api.nuget.org/v3/catalog0/data/2017.11.02.00.40.00/netstandard1.4_lib.1.0.0-test.json

Sample response

```
{  
  "@type": [  
    "PackageDelete",  
    "catalog:Permalink"  
  "catalog:commitId": "19fec5b4-9335-4e4b-bd50-8d5d3f734597",  
  "catalog:commitTimeStamp": "2017-11-02T00:40:00.1969812Z",  
  "id": "netstandard1.4_lib",  
  "originalId": "netstandard1.4_lib",  
  "published": "2017-11-02T00:37:43.7181952Z",  
  "version": "1.0.0-test"  
}
```

Cursor

Overview

This section describes a client concept that, although is not necessarily mandated by the protocol, should be part of any practical catalog client implementation.

Because the catalog is an append-only data structure indexed by time, the client should store a **cursor** locally, representing up to what point in time the client has processed catalog items. Note that this cursor value should never be generated using the client's machine clock. Instead, the value should come from a catalog object's `commitTimestamp` value.

Every time the client wants to process new events on the package source, it need only query the catalog for all catalog items with a commit timestamp greater than its stored cursor. After the client successfully processes all new catalog items, it records the latest commit timestamp of catalog items just processed as its new cursor value.

Using this approach, the client can be sure to never miss any package events that occurred on the package source. Additionally, the client never has to reprocess old events prior to the cursor's recorded commit timestamp.

This powerful concept of cursors is used for many of nuget.org background jobs and is used to keep the V3 API itself up-to-date.

Initial value

When the catalog client is starting for the very first time (and therefore has no cursor value), it should use a default cursor value of .NET's `System.DateTimeOffset.MinValue` or some such analogous notion of minimum representable timestamp.

Iterating over catalog items

To query for the next set of catalog items to process, the client should:

1. Fetch the recorded cursor value from a local store.
2. Download and deserialize the catalog index.
3. Find all catalog pages with a commit timestamp *greater than* the cursor.
4. Declare an empty list of catalog items to process.
5. For each catalog page matched in step 3:
 - a. Download and deserialized the catalog page.
 - b. Find all catalog items with a commit timestamp *greater than* the cursor.
 - c. Add all matching catalog items to the list declared in step 4.
6. Sort the catalog item list by commit timestamp.
7. Process each catalog item in sequence:
 - a. Download and deserialize the catalog item.

- b. React appropriately to the catalog item's type.
 - c. Process the catalog item document in a client-specific fashion.
8. Record the last catalog item's commit timestamp as the new cursor value.

With this basic algorithm, the client implementation can build up a complete view of all packages available on the package source. The client need only execute this algorithm periodically to always be aware of the latest changes to the package source.

NOTE

This is the algorithm that nuget.org uses to keep the [Package Metadata](#), [Package Content](#), [Search](#) and [Autocomplete](#) resources up to date.

Dependent cursors

Suppose there are two catalog clients that have an inherent dependency where one client's output depends on another client's output.

Example

For example, on nuget.org a newly published package should not appear in the search resource before it appears in the package metadata resource. This is because the "restore" operation performed by the official NuGet client uses the package metadata resource. If a customer discovers a package using the search service, they should be able to successfully restore that package using the package metadata resource. In other words, the search resource depends on the package metadata resource. Each resource has a catalog client background job updating that resource. Each client has its own cursor.

Since both resources are built off of the catalog, the cursor of the catalog client that updates the search resource *must not go beyond* the cursor of the package metadata catalog client.

Algorithm

To implement this restriction, simply modify the algorithm above to be:

1. Fetch the recorded cursor value from a local store.
2. Download and deserialize the catalog index.
3. Find all catalog pages with a commit timestamp *greater than* the cursor **less than or equal to the dependency's cursor**.
4. Declare an empty list of catalog items to process.
5. For each catalog page matched in step 3:
 - a. Download and deserialized the catalog page.
 - b. Find all catalog items with a commit timestamp *greater than* the cursor **less than or equal to the dependency's cursor**.
 - c. Add all matching catalog items to the list declared in step 4.
6. Sort the catalog item list by commit timestamp.
7. Process each catalog item in sequence:
 - a. Download and deserialize the catalog item.
 - b. React appropriately to the catalog item's type.
 - c. Process the catalog item document in a client-specific fashion.
8. Record the last catalog item's commit timestamp as the new cursor value.

Using this modified algorithm, you can build a system of dependent catalog clients all producing their own specific indexes, artifacts, etc.

Package Content

9/18/2019 • 3 minutes to read • [Edit Online](#)

It is possible to generate a URL to fetch an arbitrary package's content (the .nupkg file) using the V3 API. The resource used for fetching package content is the `PackageBaseAddress` resource found in the [service index](#). This resource also enables discovery of all versions of a package, listed or unlisted.

This resource is commonly referred to as either the "package base address" or as the "flat container".

Versioning

The following `@type` value is used:

<code>@TYPE</code>	<code>VALUE</code>	<code>NOTES</code>
	PackageBaseAddress/3.0.0	The initial release

Base URL

The base URL for the following APIs is the value of the `@id` property associated with the aforementioned resource `@type` value. In the following document, the placeholder base URL `{@id}` will be used.

HTTP methods

All URLs found in the registration resource support the HTTP methods `GET` and `HEAD`.

Enumerate package versions

If the client knows a package ID and wants to discover which package versions the package source has available, the client can construct a predictable URL to enumerate all package versions. This list is meant to be a "directory listing" for the package content API mentioned below.

NOTE

This list contains both listed and unlisted package versions.

```
GET {@id}/{LOWER_ID}/index.json
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
LOWER_ID	URL	string	yes	The package ID, lowercased

The `LOWER_ID` value is the desired package ID lowercased using the rules implemented by .NET's `System.String.ToLowerInvariant()` method.

Response

If the package source has no versions of the provided package ID, a 404 status code is returned.

If the package source has one or more versions, a 200 status code is returned. The response body is a JSON object with the following property:

NAME	TYPE	REQUIRED	NOTES
versions	array of strings	yes	The versions available

The strings in the `versions` array are all lowercased, [normalized NuGet version strings](#). The version strings do not contain any SemVer 2.0.0 build metadata.

The intent is that the version strings found in this array can be used verbatim for the `LOWER_VERSION` tokens found in the following endpoints.

Sample request

```
GET https://api.nuget.org/v3-flatcontainer/owin/index.json
```

Sample response

```
{
  "versions": [
    "0.5.0",
    "0.7.0",
    "0.11.0",
    "0.12.0",
    "0.14.0",
    "1.0.0"
  ]
}
```

Download package content (.nupkg)

If the client knows a package ID and version and wants to download the package content, they only need to construct the following URL:

```
GET {@id}/{LOWER_ID}/{LOWER_VERSION}/{LOWER_ID}.{LOWER_VERSION}.nupkg
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
LOWER_ID	URL	string	yes	The package ID, lowercase
LOWER_VERSION	URL	string	yes	The package version, normalized and lowercased

Both `LOWER_ID` and `LOWER_VERSION` are lowercased using the rules implemented by .NET's [System.String.ToLowerInvariant\(\)](#) method.

The `LOWER_VERSION` is the desired package version normalized using NuGet's version [normalization rules](#). This means that build metadata that is allowed by the SemVer 2.0.0 specification must be excluded in this case.

Response body

If the package exists on the package source, a 200 status code is returned. The response body will be the package content itself.

If the package does not exist on the package source, a 404 status code is returned.

Sample request

```
GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/9.0.1/newtonsoft.json.9.0.1.nupkg
```

Sample response

The binary stream that is the .nupkg for Newtonsoft.Json 9.0.1.

Download package manifest (.nuspec)

If the client knows a package ID and version and wants to download the package manifest, they only need to construct the following URL:

```
GET {@id}/{LOWER_ID}/{LOWER_VERSION}/{LOWER_ID}.nuspec
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
LOWER_ID	URL	string	yes	The package ID, lowercase
LOWER_VERSION	URL	string	yes	The package version, normalized and lowercased

Both `LOWER_ID` and `LOWER_VERSION` are lowercased using the rules implemented by .NET's `System.String.ToLowerInvariant()` method.

The `LOWER_VERSION` is the desired package version normalized using NuGet's version [normalization rules](#). This means that build metadata that is allowed by the SemVer 2.0.0 specification must be excluded in this case.

Response body

If the package exists on the package source, a 200 status code is returned. The response body will be the package manifest, which is the .nuspec contained in the corresponding .nupkg. The .nuspec is an XML document.

If the package does not exist on the package source, a 404 status code is returned.

Sample request

```
GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/6.0.4/newtonsoft.json.nuspec
```

Sample response

```
<?xml version="1.0"?>
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
  <metadata>
    <id>Newtonsoft.Json</id>
    <version>6.0.4</version>
    <title>Json.NET</title>
    <authors>James Newton-King</authors>
    <owners>James Newton-King</owners>
    <licenseUrl>https://raw.github.com/JamesNK/Newtonsoft.Json/master/LICENSE.md</licenseUrl>
    <projectUrl>http://james.newtonking.com/json</projectUrl>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>Json.NET is a popular high-performance JSON framework for .NET</description>
    <language>en-US</language>
    <tags>json</tags>
  </metadata>
</package>
```

Package details URL template

11/5/2019 • 2 minutes to read • [Edit Online](#)

It is possible for a client to build a URL that can be used by the user to see more package details in their web browser. This is useful when a package source wants to show additional information about a package that may not fit within the scope of what the NuGet client application shows.

The resource used for building this URL is the `PackageDetailsUriTemplate` resource found in the [service index](#).

Versioning

The following `@type` values are used:

<code>@TYPE</code>	<code>VALUE</code>	<code>NOTES</code>
	PackageDetailsUriTemplate/5.1.0	The initial release

URL template

The URL for the following API is the value of the `@id` property associated with one of the aforementioned resource `@type` values.

HTTP methods

Although the client is not intended to make requests to the package details URL on behalf of the user, the web page should support the `GET` method to allow a clicked URL to be easily opened in a web browser.

Construct the URL

Given a known package ID and version, the client implementation can construct a URL used to access a web interface. The client implementation should display this constructed URL (or clickable link) to the user allowing them to open a web browser to the URL and to learn more about the package. The contents of the package details page is determined by the server implementation.

The URL must be an absolute URL and the scheme (protocol) must be HTTPS.

The value of the `@id` in the service index is a URL string containing any of the following placeholder tokens:

URL placeholders

<code>NAME</code>	<code>TYPE</code>	<code>REQUIRED</code>	<code>NOTES</code>
<code>{id}</code>	string	no	The package ID to get details for
<code>{version}</code>	string	no	The package version to get details for

The server should accept `{id}` and `{version}` values with any casing. In addition, the server should not be sensitive to whether the version is [normalized](#). In other words, the server should accept also accept non-normalized versions.

For example, nuget.org's package details template looks like this:

```
https://www.nuget.org/packages/{id}/{version}
```

If the client implementation needs to display a link to the package details for NuGet.Versioning 4.3.0, it would produce the following URL and provide it to the user:

```
https://www.nuget.org/packages/NuGet.Versioning/4.3.0
```

Package metadata

11/14/2019 • 12 minutes to read • [Edit Online](#)

It is possible to fetch metadata about the packages available on a package source using the NuGet V3 API. This metadata can be fetched using the `RegistrationsBaseUrl` resource found in the [service index](#).

The collection of the documents found under `RegistrationsBaseUrl` are often called "registrations" or "registration blobs". The set of documents under a single `RegistrationsBaseUrl` is referred to as a "registration hive". A registration hive contains all metadata about every package available on a package source.

Versioning

The following `@type` values are used:

<code>@TYPE</code> VALUE	NOTES
<code>RegistrationsBaseUrl</code>	The initial release
<code>RegistrationsBaseUrl/3.0.0-beta</code>	Alias of <code>RegistrationsBaseUrl</code>
<code>RegistrationsBaseUrl/3.0.0-rc</code>	Alias of <code>RegistrationsBaseUrl</code>
<code>RegistrationsBaseUrl/3.4.0</code>	Gzipped responses
<code>RegistrationsBaseUrl/3.6.0</code>	Includes SemVer 2.0.0 packages

This represents three distinct registration hives available for various client versions.

RegistrationsBaseUrl

These registrations are not compressed (meaning they use an implied `Content-Encoding: identity`). SemVer 2.0.0 packages are **excluded** from this hive.

RegistrationsBaseUrl/3.4.0

These registrations are compressed using `Content-Encoding: gzip`. SemVer 2.0.0 packages are **excluded** from this hive.

RegistrationsBaseUrl/3.6.0

These registrations are compressed using `Content-Encoding: gzip`. SemVer 2.0.0 packages are **included** in this hive. For more information about SemVer 2.0.0, see [SemVer 2.0.0 support for nuget.org](#).

Base URL

The base URL for the following APIs is the value of the `@id` property associated with the aforementioned resource `@type` values. In the following document, the placeholder base URL `{@id}` will be used.

HTTP methods

All URLs found in the registration resource support the HTTP methods `GET` and `HEAD`.

Registration index

The registration resource groups package metadata by package ID. It is not possible to get data about more than one package ID at a time. This resource provides no way to discover package IDs. Instead the client is assumed to already know the desired package ID. Available metadata about each package version varies by server implementation. The package registration blobs have the following hierarchical structure:

- **Index**: the entry point for the package metadata, shared by all packages on a source with the same package ID.
- **Page**: a grouping of package versions. The number of package versions in a page is defined by server implementation.
- **Leaf**: a document specific to a single package version.

The URL of the registration index is predictable and can be determined by the client given a package ID and the registration resource's `@id` value from the service index. The URLs for the registration pages and leaves are discovered by inspecting the registration index.

Registration pages and leaves

Although it's not strictly required for a server implementation to store registration leafs in separate registration page documents, it's a recommended practice to conserve client-side memory. Instead of inlining all registration leaves in the index or immediately storing leaves in page documents, it's recommended that the server implementation define some heuristic to choose between the two approaches based on the number of package versions or cumulative size of package leaves.

Storing all package versions (leaves) in the registration index saves on the number of HTTP requests necessary to fetch package metadata but means that a larger document must be downloaded and more client memory must be allocated. On the other hand, if the server implementation immediately stores registration leaves in separate page documents, the client must perform more HTTP requests to get the information it needs.

The heuristic that nuget.org uses is as follows: if there are 128 or more versions of a package, break the leaves into pages of size 64. If there are less than 128 versions, inline all leaves into the registration index. Note that this means packages with 65 to 127 versions will have two pages in the index but both pages will be inlined.

```
GET {@id}/{LOWER_ID}/index.json
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
LOWER_ID	URL	string	yes	The package ID, lowercased

The `LOWER_ID` value is the desired package ID lowercased using the rules implemented by .NET's `System.String.ToLowerInvariant()` method.

Response

The response is a JSON document which has a root object with the following properties:

NAME	TYPE	REQUIRED	NOTES
count	integer	yes	The number of registration pages in the index
items	array of objects	yes	The array of registration pages

Each item in the index object's `items` array is a JSON object representing a registration page.

Registration page object

The registration page object found in the registration index has the following properties:

NAME	TYPE	REQUIRED	NOTES
@id	string	yes	The URL to the registration page
count	integer	yes	The number of registration leaves in the page
items	array of objects	no	The array of registration leaves and their associate metadata
lower	string	yes	The lowest SemVer 2.0.0 version in the page (inclusive)
parent	string	no	The URL to the registration index
upper	string	yes	The highest SemVer 2.0.0 version in the page (inclusive)

The `lower` and `upper` bounds of the page object are useful when the metadata for a specific page version is needed. These bounds can be used to fetch the only registration page needed. The version strings adhere to [NuGet's version rules](#). The version strings are normalized and do not include build metadata. As with all versions in the NuGet ecosystem, comparison of version strings is implemented using [SemVer 2.0.0's version precedence rules](#).

The `parent` property will only appear if the registration page object has the `items` property.

If the `items` property is not present in the registration page object, the URL specified in the `@id` must be used to fetch metadata about individual package versions. The `items` array is sometimes excluded from the page object as an optimization. If the number of versions of a single package ID is very large, then the registration index document will be massive and wasteful to process for a client that only cares about a specific version or small range of versions.

Note that if the `items` property is present, the `@id` property need not be used, since all of the page data is already inlined in the `items` property.

Each item in the page object's `items` array is a JSON object representing a registration leaf and its associated metadata.

Registration leaf object in a page

The registration leaf object found in a registration page has the following properties:

NAME	TYPE	REQUIRED	NOTES
@id	string	yes	The URL to the registration leaf
catalogEntry	object	yes	The catalog entry containing the package metadata

NAME	TYPE	REQUIRED	NOTES
packageContent	string	yes	The URL to the package content (.nupkg)

Each registration leaf object represents data associated with a single package version.

Catalog entry

The `catalogEntry` property in the registration leaf object has the following properties:

NAME	TYPE	REQUIRED	NOTES
@id	string	yes	The URL to the document used to produce this object
authors	string or array of strings	no	
dependencyGroups	array of objects	no	The dependencies of the package, grouped by target framework
deprecation	object	no	The deprecation associated with the package
description	string	no	
iconUrl	string	no	
id	string	yes	The ID of the package
licenseUrl	string	no	
licenseExpression	string	no	
listed	boolean	no	Should be considered as listed if absent
minClientVersion	string	no	
projectUrl	string	no	
published	string	no	A string containing a ISO 8601 timestamp of when the package was published
requireLicenseAcceptance	boolean	no	
summary	string	no	
tags	string or array of string	no	
title	string	no	

NAME	TYPE	REQUIRED	NOTES
version	string	yes	The full version string after normalization

The package `version` property is the full version string after normalization. This means that SemVer 2.0.0 build data can be included here.

The `dependencyGroups` property is an array of objects representing the dependencies of the package, grouped by target framework. If the package has no dependencies, the `dependencyGroups` property is missing, an empty array, or the `dependencies` property of all groups is empty or missing.

The value of the `licenseExpression` property complies with [NuGet license expression syntax](#).

NOTE

On nuget.org, the `published` value is set to year 1900 when the package is unlisted.

Package dependency group

Each dependency group object has the following properties:

NAME	TYPE	REQUIRED	NOTES
targetFramework	string	no	The target framework that these dependencies are applicable to
dependencies	array of objects	no	

The `targetFramework` string uses the format implemented by NuGet's .NET library [NuGet.Frameworks](#). If no `targetFramework` is specified, the dependency group applies to all target frameworks.

The `dependencies` property is an array of objects, each representing a package dependency of the current package.

Package dependency

Each package dependency has the following properties:

NAME	TYPE	REQUIRED	NOTES
id	string	yes	The ID of the package dependency
range	object	no	The allowed version range of the dependency
registration	string	no	The URL to the registration index for this dependency

If the `range` property is excluded or an empty string, the client should default to the version range `(,)`. That is, any version of the dependency is allowed. The value of `*` is not allowed for the `range` property.

Package deprecation

Each package deprecation has the following properties:

NAME	TYPE	REQUIRED	NOTES
reasons	array of strings	yes	The reasons why the package was deprecated
message	string	no	The additional details about this deprecation
alternatePackage	object	no	The alternate package that should be used instead

The `reasons` property must contain at least one string and should only contain strings from the following table:

REASON	DESCRIPTION
Legacy	The package is no longer maintained
CriticalBugs	The package has bugs which make it unsuitable for usage
Other	The package is deprecated due to a reason not on this list

If the `reasons` property contains strings that are not from the known set, they should be ignored. The strings are case-insensitive, so `legacy` should be treated the same as `Legacy`. There is no ordering restriction on the array, so the strings can be arranged in any arbitrary order. Additionally, if the property contains only strings that are not from the known set, it should be treated as if it only contained the "Other" string.

Alternate package

The alternate package object has the following properties:

NAME	TYPE	REQUIRED	NOTES
id	string	yes	The ID of the alternate package
range	object	no	The allowed version range , or <code>*</code> if any version is allowed
registration	string	no	The URL to the registration index for this alternate package

Sample request

```
GET https://api.nuget.org/v3/registration3/nuget.server.core/index.json
```

Sample response

```
{
  "count": 1,
  "items": [
    {
      "@id": "https://api.nuget.org/v3/registration3/nuget.server.core/index.json#page/3.0.0-beta/3.0.0-beta",
      "count": 1,
      "items": [
        {
          "@id": "https://api.nuget.org/v3/registration3/nuget.server.core/3.0.0-beta.json",
          "catalogEntry": {
            "@id": "https://api.nuget.org/v3/catalog0/data/2017.10.05.18.41.33/nuget.server.core.3.0.0-beta.json",
            "authors": ".NET Foundation",
            "dependencyGroups": [
              {
                "@id": "https://api.nuget.org/v3/catalog0/data/2017.10.05.18.41.33/nuget.server.core.3.0.0-beta.json#dependencygroup",
                "dependencies": [
                  {
                    "@id":
"https://api.nuget.org/v3/catalog0/data/2017.10.05.18.41.33/nuget.server.core.3.0.0-beta.json#dependencygroup/nuget.core",
                    "id": "NuGet.Core",
                    "range": "[2.14.0, )",
                    "registration": "https://api.nuget.org/v3/registration3/nuget.core/index.json"
                  }
                ]
              }
            ],
            "description": "Core library for creating a Web Application used to host a simple NuGet feed",
            "iconUrl": "",
            "id": "NuGet.Server.Core",
            "language": "",
            "licenseUrl": "https://raw.githubusercontent.com/NuGet/NuGet.Server/dev/LICENSE.txt",
            "listed": true,
            "minClientVersion": "2.6",
            "packageContent": "https://api.nuget.org/v3-flatcontainer/nuget.server.core/3.0.0-beta/nuget.server.core.3.0.0-beta.nupkg",
            "projectUrl": "https://github.com/NuGet/NuGet.Server",
            "published": "2017-10-05T18:40:32.43+00:00",
            "requireLicenseAcceptance": false,
            "summary": "",
            "tags": [ "" ],
            "title": "",
            "version": "3.0.0-beta"
          },
          "packageContent": "https://api.nuget.org/v3-flatcontainer/nuget.server.core/3.0.0-beta/nuget.server.core.3.0.0-beta.nupkg",
          "registration": "https://api.nuget.org/v3/registration3/nuget.server.core/index.json"
        }
      ],
      "lower": "3.0.0-beta",
      "upper": "3.0.0-beta"
    }
  ]
}
```

In this particular case, the registration index has the registration page inlined so no extra requests are needed to fetch metadata about individual package versions.

Registration page

The registration page contains registration leaves. The URL to fetch a registration page is determined by the `@id` property in the [registration page object](#) mentioned above. The URL is not meant to be predictable and should

always be discovered by means of the index document.

WARNING

On nuget.org, the URL for the registration page document coincidentally contains the lower and upper bound of the page. However this assumption should never be made by a client since server implementations are free to change the shape of the URL as long as the index document has a valid link.

When the `items` array is not provided in the registration index, an HTTP GET request of the `@id` value will return a JSON document which has an object as its root. The object has the following properties:

NAME	TYPE	REQUIRED	NOTES
<code>@id</code>	string	yes	The URL to the registration page
<code>count</code>	integer	yes	The number of registration leaves in the page
<code>items</code>	array of objects	yes	The array of registration leaves and their associate metadata
<code>lower</code>	string	yes	The lowest SemVer 2.0.0 version in the page (inclusive)
<code>parent</code>	string	yes	The URL to the registration index
<code>upper</code>	string	yes	The highest SemVer 2.0.0 version in the page (inclusive)

The shape of the registration leaf objects is the same as in the registration index [above](#).

Sample request

```
GET https://api.nuget.org/v3/registration3/ravendb.client/page/1.0.531/1.0.729-unstable.json
```

Sample response

```
{
  "count": 2,
  "lower": "1.0.531",
  "parent": "https://api.nuget.org/v3/registration3/nuget.protocol.v3.example/index.json",
  "upper": "1.0.729-unstable",
  "items": [
    {
      "@id": "https://api.nuget.org/v3/registration3/nuget.protocol.v3.example/1.0.531.json",
      "@type": "Package",
      "commitId": "e0b9ca79-75b5-414f-9e3e-de9534b5cf1",
      "commitTimeStamp": "2017-10-26T14:12:19.3439088Z",
      "catalogEntry": {
        "@id": "https://api.nuget.org/v3/catalog0/data/2015.02.01.11.38.37/nuget.protocol.v3.example.1.0.531.json",
        "catalogId": "catalog0",
        "id": "nuget.protocol.v3.example.1.0.531",
        "version": "1.0.531"
      }
    }
  ]
}
```

```

"@type": "PackageDetails",
"authors": "NuGet.org Team",
"iconUrl": "https://www.nuget.org/Content/gallery/img/default-package-icon.svg",
"id": "NuGet.Protocol.V3.Example",
"licenseUrl": "http://www.opensource.org/licenses/ms-pl",
"listed": false,
"packageContent": "https://api.nuget.org/v3-flatcontainer/nuget.protocol.v3.example/1.0.531/nuget.protocol.v3.example.1.0.531.nupkg",
"projectUrl": "https://github.com/NuGet/NuGetGallery",
"published": "1900-01-01T00:00:00+00:00",
"requireLicenseAcceptance": true,
"title": "NuGet V3 Protocol Example",
"version": "1.0.531"
},
"packageContent": "https://api.nuget.org/v3-flatcontainer/nuget.protocol.v3.example/1.0.531/nuget.protocol.v3.example.1.0.531.nupkg",
"registration": "https://api.nuget.org/v3/registration3/nuget.protocol.v3.example/index.json"
},
{
"@id": "https://api.nuget.org/v3/registration3/nuget.protocol.v3.example/1.0.729-unstable.json",
"@type": "Package",
"commitId": "e0b9ca79-75b5-414f-9e3e-de9534b5cf1",
"commitTimeStamp": "2017-10-26T14:12:19.3439088Z",
"catalogEntry": {
"@id": "https://api.nuget.org/v3/catalog0/data/2015.02.01.18.22.05/nuget.protocol.v3.example.1.0.729-unstable.json",
"@type": "PackageDetails",
"authors": "NuGet.org Team",
"deprecation": {
"reasons": [
"HasCriticalBugs"
],
"message": "This package is unstable and broken!",
"alternatePackage": {
"id": "Newtonsoft.JSON",
"range": "12.0.2"
}
},
"iconUrl": "https://www.nuget.org/Content/gallery/img/default-package-icon.svg",
"id": "NuGet.Protocol.V3.Example",
"licenseUrl": "http://www.opensource.org/licenses/ms-pl",
"listed": false,
"packageContent": "https://api.nuget.org/v3-flatcontainer/nuget.protocol.v3.example/1.0.729-unstable/nuget.protocol.v3.example.1.0.729-unstable.nupkg",
"projectUrl": "https://github.com/NuGet/NuGetGallery",
"published": "1900-01-01T00:00:00+00:00",
"requireLicenseAcceptance": true,
"summary": "This package is an example for the V3 protocol.",
"title": "NuGet V3 Protocol Example",
"version": "1.0.729-Unstable"
},
"packageContent": "https://api.nuget.org/v3-flatcontainer/nuget.protocol.v3.example/1.0.729-unstable/nuget.protocol.v3.example.1.0.729-unstable.nupkg",
"registration": "https://api.nuget.org/v3/registration3/nuget.protocol.v3.example/index.json"
}
]
}

```

Registration leaf

The registration leaf contains information about a specific package ID and version. The metadata about the specific version may not be available in this document. Package metadata should be fetched from the [registration index](#) or the [registration page](#) (which is discovered using the registration index).

The URL to fetch a registration leaf is obtained from the `@id` property of a registration leaf object in either a registration index or registration page. As with the page document, the URL is not meant to be predictable and

should always be discovered by means of the registration page object.

WARNING

On nuget.org, the URL for the registration leaf document coincidentally contains the package version. However this assumption should never be made by a client since server implementations are free to change the shape of the URL as long as the parent document has a valid link.

The registration leaf is a JSON document with a root object with the following properties:

NAME	TYPE	REQUIRED	NOTES
@id	string	yes	The URL to the registration leaf
catalogEntry	string	no	The URL to the catalog entry that produced these leaf
listed	boolean	no	Should be considered as listed if absent
packageContent	string	no	The URL to the package content (.nupkg)
published	string	no	A string containing a ISO 8601 timestamp of when the package was published
registration	string	no	The URL to the registration index

NOTE

On nuget.org, the `published` value is set to year 1900 when the package is unlisted.

Sample request

```
GET https://api.nuget.org/v3/registration3/nuget.versioning/4.3.0.json
```

Sample response

```
{
  "@id": "https://api.nuget.org/v3/registration3/nuget.versioning/4.3.0.json",
  "catalogEntry": "https://api.nuget.org/v3/catalog0/data/2017.08.11.18.24.22/nuget.versioning.4.3.0.json",
  "listed": true,
  "packageContent": "https://api.nuget.org/v3-flatcontainer/nuget.versioning/4.3.0/nuget.versioning.4.3.0.nupkg",
  "published": "2017-08-11T18:24:14.36+00:00",
  "registration": "https://api.nuget.org/v3/registration3/nuget.versioning/index.json"
}
```

Push and Delete

6/28/2019 • 3 minutes to read • [Edit Online](#)

It is possible to push, delete (or unlist, depending on the server implementation), and relist packages using the NuGet V3 API. These operations are based off of the `PackagePublish` resource found in the [service index](#).

Versioning

The following `@type` value is used:

<code>@TYPE</code> VALUE	NOTES
PackagePublish/2.0.0	The initial release

Base URL

The base URL for the following APIs is the value of the `@id` property of the `PackagePublish/2.0.0` resource in the package source's [service index](#). For the documentation below, nuget.org's URL is used. Consider `https://www.nuget.org/api/v2/package` as a placeholder for the `@id` value found in the service index.

Note that this URL points to the same location as the legacy V2 push endpoint since the protocol is the same.

HTTP methods

The `PUT`, `POST` and `DELETE` HTTP methods are supported by this resource. For which methods are supported on each endpoint, see below.

Push a package

NOTE

nuget.org has [additional requirements](#) for interacting with the push endpoint.

nuget.org supports pushing new packages using the following API. If the package with the provided ID and version already exists, nuget.org will reject the push. Other package sources may support replacing an existing package.

```
PUT https://www.nuget.org/api/v2/package
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
X-NuGet-ApiKey	Header	string	yes	For example, <code>X-NuGet-ApiKey: {USER_API_KEY}</code>

The API key is an opaque string gotten from the package source by the user and configured into the client. No particular string format is mandated but the length of the API key should not exceed a reasonable size for HTTP

header values.

Request body

The request body must come in the following form:

Multipart form data

The request header `Content-Type` is `multipart/form-data` and the first item in the request body is the raw bytes of the .nupkg being pushed. Subsequent items in the multipart body are ignored. The file name or any other headers of the multipart items are ignored.

Response

STATUS CODE	MEANING
201, 202	The package was successfully pushed
400	The provided package is invalid
409	A package with the provided ID and version already exists

Server implementations vary on the success status code returned when a package is successfully pushed.

Delete a package

nuget.org interprets the package delete request as an "unlist". This means that the package is still available for existing consumers of the package but the package no longer appears in search results or in the web interface. For more information about this practice, see the [Deleted Packages](#) policy. Other server implementations are free to interpret this signal as a hard delete, soft delete, or unlist. For example, [NuGet.Server](#) (a server implementation only supporting the older V2 API) supports handling this request as either an unlist or a hard delete based on a configuration option.

```
DELETE https://www.nuget.org/api/v2/package/{ID}/{VERSION}
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
ID	URL	string	yes	The ID of the package to delete
VERSION	URL	string	yes	The version of the package to delete
X-NuGet-ApiKey	Header	string	yes	For example, <code>X-NuGet-ApiKey: {USER_API_KEY}</code>

Response

STATUS CODE	MEANING
204	The package was deleted
404	No package with the provided <code>ID</code> and <code>VERSION</code> exists

Relist a package

If a package is unlisted, it is possible to make that package once again visible in search results using the "relist" endpoint. This endpoint has the same shape as the [delete \(unlist\) endpoint](#) but uses the `POST` HTTP method instead of the `DELETE` method.

If the package is already listed, the request still succeeds.

```
POST https://www.nuget.org/api/v2/package/{ID}/{VERSION}
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
ID	URL	string	yes	The ID of the package to relist
VERSION	URL	string	yes	The version of the package to relist
X-NuGet-ApiKey	Header	string	yes	For example, <code>X-NuGet-ApiKey: {USER_API_KEY}</code>

Response

STATUS CODE	MEANING
200	The package is now listed
404	No package with the provided <code>ID</code> and <code>VERSION</code> exists

Push Symbol Packages

9/3/2019 • 2 minutes to read • [Edit Online](#)

It is possible to push symbols packages ([snupkg](#)) using the NuGet V3 API. These operations are based off of the [SymbolPackagePublish](#) resource found in the [service index](#).

Versioning

The following `@type` value is used:

<code>@TYPE</code> VALUE	NOTES
SymbolPackagePublish/4.9.0	The initial release

Base URL

The base URL for the following APIs is the value of the `@id` property of the [SymbolPackagePublish/4.9.0](#) resource in the package source's [service index](#). For the documentation below, nuget.org's URL is used. Consider <https://www.nuget.org/api/v2/symbolpackage> as a placeholder for the `@id` value found in the service index.

HTTP methods

The `PUT` HTTP method is supported by this resource.

Push a symbol package

nuget.org supports pushing new symbol packages format ([snupkg](#)) using the following API.

```
PUT https://www.nuget.org/api/v2/symbolpackage
```

Symbol packages with the same ID and version can be submitted multiple times. A symbol package will be rejected in the following cases.

- A package with the same ID and version does not exist.
- A symbol package with the same ID and version was pushed but is not yet published.
- The symbol package ([snupkg](#)) is invalid (see [symbol package constraints](#)).

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
X-NuGet-ApiKey	Header	string	yes	For example, <code>X-NuGet-ApiKey: {USER_API_KEY}</code>

The API key is an opaque string gotten from the package source by the user and configured into the client. No particular string format is mandated but the length of the API key should not exceed a reasonable size for HTTP header values.

Request body

The request body for the symbol push is same as with the request body of a package push request (see [package push and delete](#)).

Response

STATUS CODE	MEANING
201	The symbol package was successfully pushed.
400	The provided symbol package is invalid.
401	The user is not authorized to perform this action.
404	A corresponding package with the provided ID and version does not exist.
409	A symbol package with the provided ID and version was pushed but it is not available yet.
413	The package is too large.

Report abuse URL template

9/4/2018 • 2 minutes to read • [Edit Online](#)

It is possible for a client to build a URL that can be used by the user to report abuse about a specific package. This is useful when a package source wants to enable all client experiences (even 3rd party) to delegate abuse reports to the package source.

The resource used for building this URL is the `ReportAbuseUriTemplate` resource found in the [service index](#).

Versioning

The following `@type` values are used:

<code>@TYPE</code>	<code>VALUE</code>	<code>NOTES</code>
	<code>ReportAbuseUriTemplate/3.0.0-beta</code>	The initial release
	<code>ReportAbuseUriTemplate/3.0.0-rc</code>	Alias of <code>ReportAbuseUriTemplate/3.0.0-beta</code>

URL template

The URL for the following API is the value of the `@id` property associated with one of the aforementioned resource `@type` values.

HTTP methods

Although the client is not intended to make requests to the report abuse URL on behalf of the user, the web page should support the `GET` method to allow a clicked URL to be easily opened in a web browser.

Construct the URL

Given a known package ID and version, the client implementation can construct a URL used to access a web interface. The client implementation should display this constructed URL (or clickable link) to the user allowing them to open a web browser to the URL and make any necessary abuse report. The implementation of the abuse report form is determined by the server implementation.

The value of the `@id` is a URL string containing any of the following placeholder tokens:

URL placeholders

<code>NAME</code>	<code>TYPE</code>	<code>REQUIRED</code>	<code>NOTES</code>
<code>{id}</code>	string	no	The package ID to report abuse for
<code>{version}</code>	string	no	The package version to report abuse for

The `{id}` and `{version}` values interpreted by the server implementation must be case insensitive and not sensitive to whether the version is normalized.

For example, nuget.org's report abuse template looks like this:

```
https://www.nuget.org/packages/{id}/{version}/ReportAbuse
```

If the client implementation needs to display a link to the report abuse form for NuGet.Versioning 4.3.0, it would produce the following URL and provide it to the user:

```
https://www.nuget.org/packages/NuGet.Versioning/4.3.0/ReportAbuse
```

Repository signatures

4/11/2019 • 3 minutes to read • [Edit Online](#)

If a package source supports adding repository signatures to published packages, it is possible for a client to determine the signing certificates that are used by the package source. This resource allows clients to detect whether a repository signed package has been tampered or has an unexpected signing certificate.

The resource used for fetching this repository signature information is the `RepositorySignatures` resource found in the [service index](#).

Versioning

The following `@type` value is used:

<code>@TYPE</code> <code>VALUE</code>	<code>NOTES</code>
RepositorySignatures/4.7.0	The initial release
RepositorySignatures/4.9.0	Supported by NuGet v4.9+ clients
RepositorySignatures/5.0.0	Allows enabling <code>allRepositorySigned</code> . Supported by NuGet v5.0+ clients

Base URL

The entry point URL for the following APIs is the value of the `@id` property associated with the aforementioned resource `@type` value. This topic uses the placeholder URL `{@id}`.

Note that unlike other resources, the `{@id}` URL is required to be served over HTTPS.

HTTP methods

All URLs found in the repository signatures resource support only the HTTP methods `GET` and `HEAD`.

Repository signatures index

The repository signatures index contains two pieces of information:

1. Whether or not all packages found on the source are repository signed by this package source.
2. The list of certificates used by the package source to sign packages.

In most cases, the list of certificates will only ever be appended to. New certificates would be added to the list when the previous signing certificate has expired and the package source needs to start using a new signing certificate. If a certificate is removed from the list, that means that all package signatures created with the removed signing certificate should no longer be considered valid by the client. In this case, the package signature (but not necessarily the package) is invalid. A client policy may allow installing the package as unsigned.

In the case of certificate revocation (e.g. key compromise), the package source is expected to re-sign all packages signed by the affected certificate. Additionally, the package source should remove the affected certificate from the signing certificate list.

The following request fetches the repository signatures index.

```
GET {@id}
```

The repository signature index is a JSON document that contains an object with the following properties:

NAME	TYPE	REQUIRED	NOTES
allRepositorySigned	boolean	yes	Must be <code>false</code> on 4.7.0 and 4.9.0 resources
signingCertificates	array of objects	yes	

The `allRepositorySigned` boolean is set to false if the package source has some packages that have no repository signature. If the boolean is set to true, all packages available on the source must have a repository signature produced by one of the signing certificates mentioned in `signingCertificates`.

WARNING

The `allRepositorySigned` boolean must be false on the 4.7.0 and 4.9.0 resources. NuGet v4.7, v4.8, and v4.9 clients cannot install packages from sources that have `allRepositorySigned` set to true.

There should be one or more signing certificates in the `signingCertificates` array if the `allRepositorySigned` boolean is set to true. If the array is empty and `allRepositorySigned` is set to true, all packages from the source should be considered invalid, although a client policy may still allow consumption of packages. Each element in this array is a JSON object with the following properties.

NAME	TYPE	REQUIRED	NOTES
contentUrl	string	yes	Absolute URL to the DER-encoded public certificate
fingerprints	object	yes	
subject	string	yes	The subject distinguished name from the certificate
issuer	string	yes	The distinguished name of the certificate's issuer
notBefore	string	yes	The starting timestamp of the certificate's validity period
notAfter	string	yes	The ending timestamp of the certificate's validity period

Note that the `contentUrl` is required to be served over HTTPS. This URL has no specific URL pattern and must be dynamically discovered using this repository signatures index document.

All properties in this object (aside from `contentUrl`) must be derivable from the certificate found at `contentUrl`. These derivable properties are provided as a convenience to minimize round trips.

The `fingerprints` object has the following properties:

NAME	TYPE	REQUIRED	NOTES
2.16.840.1.101.3.4.2.1	string	yes	The SHA-256 fingerprint

The key name `2.16.840.1.101.3.4.2.1` is the OID of the SHA-256 hash algorithm.

All hash values must be lowercase, hex-encoded string representations of the hash digest.

Sample request

```
GET https://api.nuget.org/v3-index/repository-signatures/index.json
```

Sample response

```
{
  "allRepositorySigned": false,
  "signingCertificates": [
    {
      "fingerprints": {
        "2.16.840.1.101.3.4.2.1": "0e5f38f57dc1bcc806d8494f4f90fbcedd988b46760709cbeec6f4219aa6157d"
      },
      "subject": "CN=NuGet.org Repository by Microsoft, O=NuGet.org Repository by Microsoft, L=Redmond, S=Washington, C=US",
      "issuer": "CN=DigiCert SHA2 Assured ID Code Signing CA, OU=www.digicert.com, O=DigiCert Inc, C=US",
      "notBefore": "2018-04-10T00:00:00.000000Z",
      "notAfter": "2021-04-14T12:00:00.000000Z",
      "contentUrl": "https://api.nuget.org/v3-index/repository-signatures/certificates/0e5f38f57dc1bcc806d8494f4f90fbcedd988b46760709cbeec6f4219aa6157d.crt"
    }
  ]
}
```

Search

9/3/2019 • 4 minutes to read • [Edit Online](#)

It is possible to search for packages available on a package source using the V3 API. The resource used for searching is the `SearchQueryService` resource found in the [service index](#).

Versioning

The following `@type` values are used:

<code>@TYPE</code> VALUE	NOTES
SearchQueryService	The initial release
SearchQueryService/3.0.0-beta	Alias of <code>SearchQueryService</code>
SearchQueryService/3.0.0-rc	Alias of <code>SearchQueryService</code>

Base URL

The base URL for the following API is the value of the `@id` property associated with one of the aforementioned resource `@type` values. In the following document, the placeholder base URL `{@id}` will be used.

HTTP methods

All URLs found in the registration resource support the HTTP methods `GET` and `HEAD`.

Search for packages

The search API allows a client to query for a page of packages matching a specified search query. The interpretation of the search query (e.g. the tokenization of the search terms) is determined by the server implementation but the general expectation is that the search query is used for matching package IDs, titles, descriptions, and tags. Other package metadata fields may also be considered.

An unlisted package should never appear in search results.

```
GET {@id}?q={QUERY}&skip={SKIP}&take={TAKE}&prerelease={PRERELEASE}&semVerLevel={SEMVERLEVEL}
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
q	URL	string	no	The search terms to be used to filter packages
skip	URL	integer	no	The number of results to skip, for pagination

NAME	IN	TYPE	REQUIRED	NOTES
take	URL	integer	no	The number of results to return, for pagination
prerelease	URL	boolean	no	<code>true</code> or <code>false</code> determining whether to include pre-release packages
semVerLevel	URL	string	no	A SemVer 1.0.0 version string

The search query `q` is parsed in a manner that is defined by the server implementation. nuget.org supports basic filtering on a [variety of fields](#). If no `q` is provided, all packages should be returned, within the boundaries imposed by `skip` and `take`. This enables the "Browse" tab in the NuGet Visual Studio experience.

The `skip` parameter defaults to 0.

The `take` parameter should be an integer greater than zero. The server implementation may impose a maximum value.

If `prerelease` is not provided, pre-release packages are excluded.

The `semVerLevel` query parameter is used to opt-in to [SemVer 2.0.0 packages](#). If this query parameter is excluded, only packages with SemVer 1.0.0 compatible versions will be returned (with the [standard NuGet versioning](#) caveats, such as version strings with 4 integer pieces). If `semVerLevel=2.0.0` is provided, both SemVer 1.0.0 and SemVer 2.0.0 compatible packages will be returned. See the [SemVer 2.0.0 support for nuget.org](#) for more information.

Response

The response is JSON document containing up to `take` search results. Search results are grouped by package ID.

The root JSON object has the following properties:

NAME	TYPE	REQUIRED	NOTES
totalHits	integer	yes	The total number of matches, disregarding <code>skip</code> and <code>take</code>
data	array of objects	yes	The search results matched by the request

Search result

Each item in the `data` array is a JSON object comprised of a group of package versions sharing the same package ID. The object has the following properties:

NAME	TYPE	REQUIRED	NOTES
id	string	yes	The ID of the matched package

NAME	TYPE	REQUIRED	NOTES
version	string	yes	The full SemVer 2.0.0 version string of the package (could contain build metadata)
description	string	no	
versions	array of objects	yes	All of the versions of the package matching the <code>prerelease</code> parameter
authors	string or array of strings	no	
iconUrl	string	no	
licenseUrl	string	no	
owners	string or array of strings	no	
projectUrl	string	no	
registration	string	no	The absolute URL to the associated registration index
summary	string	no	
tags	string or array of strings	no	
title	string	no	
totalDownloads	integer	no	This value can be inferred by the sum of downloads in the <code>versions</code> array
verified	boolean	no	A JSON boolean indicating whether the package is verified

On nuget.org, a verified package is one which has a package ID matching a reserved ID prefix and owned by one of the reserved prefix's owners. For more information, see the [documentation about ID prefix reservation](#).

The metadata contained in the search result object is taken from the latest package version. Each item in the `versions` array is a JSON object with the following properties:

NAME	TYPE	REQUIRED	NOTES
@id	string	yes	The absolute URL to the associated registration leaf
version	string	yes	The full SemVer 2.0.0 version string of the package (could contain build metadata)

NAME	TYPE	REQUIRED	NOTES
downloads	integer	yes	The number of downloads for this specific package version

Sample request

```
GET https://azureresearch-usnc.nuget.org/query?q=NuGet.Versioning&prerelease=false&semVerLevel=2.0.0
```

Sample response

```
{
  "totalHits": 2,
  "data": [
    {
      "registration": "https://api.nuget.org/v3/registration3/nuget.versioning/index.json",
      "id": "NuGet.Versioning",
      "version": "4.4.0",
      "description": "NuGet's implementation of Semantic Versioning.",
      "summary": "",
      "title": "NuGet.Versioning",
      "licenseUrl": "https://raw.githubusercontent.com/NuGet/NuGet.Client/dev/LICENSE.txt",
      "tags": [ "semver", "semantic", "versioning" ],
      "authors": [ "NuGet" ],
      "totalDownloads": 141896,
      "verified": true,
      "versions": [
        {
          "version": "3.3.0",
          "downloads": 50343,
          "@id": "https://api.nuget.org/v3/registration3/nuget.versioning/3.3.0.json"
        },
        {
          "version": "3.4.3",
          "downloads": 27932,
          "@id": "https://api.nuget.org/v3/registration3/nuget.versioning/3.4.3.json"
        },
        {
          "version": "4.0.0",
          "downloads": 63004,
          "@id": "https://api.nuget.org/v3/registration3/nuget.versioning/4.0.0.json"
        },
        {
          "version": "4.4.0",
          "downloads": 617,
          "@id": "https://api.nuget.org/v3/registration3/nuget.versioning/4.4.0.json"
        }
      ]
    },
    {
      "@id": "https://api.nuget.org/v3/registration3/nerdbank.gitversioning/index.json",
      "@type": "Package",
      "registration": "https://api.nuget.org/v3/registration3/nerdbank.gitversioning/index.json",
      "id": "Nerdbank.GitVersioning",
      "version": "2.0.41",
      "description": "Stamps your assemblies with semver 2.0 compliant git commit specific version information and provides NuGet versioning information as well.",
      "summary": "Stamps your assemblies with semver 2.0 compliant git commit specific version information and provides NuGet versioning information as well.",
      "title": "Nerdbank.GitVersioning",
      "licenseUrl": "https://raw.githubusercontent.com/AArnott/Nerdbank.GitVersioning/ed547462f7/LICENSE.txt",
      "projectUrl": "http://github.com/aarnott/Nerdbank.GitVersioning",
      "tags": [ "git", "commit", "versioning", "version", "assemblyinfo" ],
      "authors": [ "Andrew Arnott" ]
    }
  ]
}
```

```
  "authors": [
    {
      "name": "NerdBank"
    }
  ],
  "totalDownloads": 11906,
  "verified": false,
  "versions": [
    {
      "version": "1.6.35",
      "downloads": 10229,
      "@id": "https://api.nuget.org/v3/registration3/nerdbank.gitversioning/1.6.35.json"
    },
    {
      "version": "2.0.41",
      "downloads": 1677,
      "@id": "https://api.nuget.org/v3/registration3/nerdbank.gitversioning/2.0.41.json"
    }
  ]
}
```

Service index

1/15/2019 • 2 minutes to read • [Edit Online](#)

The service index is a JSON document that is the entry point for a NuGet package source and allows a client implementation to discover the package source's capabilities. The service index is a JSON object with two required properties: `version` (the schema version of the service index) and `resources` (the endpoints or capabilities of the package source).

nuget.org's service index is located at <https://api.nuget.org/v3/index.json>.

Versioning

The `version` value is a SemVer 2.0.0 parseable version string which indicates the schema version of the service index. The API mandates that the version string has a major version number of `3`. As non-breaking changes are made to the service index schema, the version string's minor version will be increased.

Each resource in the service index is versioned independently from the service index schema version.

The current schema version is `3.0.0`. The `3.0.0` version is functionally equivalent to the older `3.0.0-beta.1` version but should be preferred as it more clearly communicates the stable, defined schema.

HTTP methods

The service index is accessible using HTTP methods `GET` and `HEAD`.

Resources

The `resources` property contains an array of resources supported by this package source.

Resource

A resource is an object in the `resources` array. It represents a versioned capability of a package source. A resource has the following properties:

NAME	TYPE	REQUIRED	NOTES
<code>@id</code>	string	yes	The URL to the resource
<code>@type</code>	string	yes	A string constant representing the resource type
<code>comment</code>	string	no	A human readable description of the resource

The `@id` is a URL that must be absolute and must either have the HTTP or HTTPS schema.

The `@type` is used to identify the specific protocol to use when interacting with resource. The type of the resource is an opaque string but generally has the format:

```
{RESOURCE_NAME}/{RESOURCE_VERSION}
```

Clients are expected to hard code the `@type` values that they understand and look them up in a package source's service index. The exact `@type` values in use today are enumerated on the individual resource reference documents listed in the [API overview](#).

For the sake of this documentation, the documentation about different resources will essentially be grouped by the `{RESOURCE_NAME}` found in the service index which is analogous to grouping by scenario.

There is no requirement that each resource has a unique `@id` or `@type`. It is up to the client implementation to determine which resource to prefer over another. One possible implementation is that resources of the same or compatible `@type` can be used in a round-robin fashion in case of connection failure or server error.

Sample request

```
GET https://api.nuget.org/v3/index.json
```

Sample response

```
{
  "version": "3.0.0",
  "resources": [
    {
      "@id": "https://api.nuget.org/v3-flatcontainer/",
      "@type": "PackageBaseAddress/3.0.0",
      "comment": "Base URL of Azure storage where NuGet package registration info for .NET Core is stored, in the format https://api.nuget.org/v3-flatcontainer/{id-lower}/{id-lower}.{version-lower}.nupkg"
    },
    {
      "@id": "https://www.nuget.org/api/v2/package",
      "@type": "PackagePublish/2.0.0"
    },
    {
      "@id": "https://api-v2v3search-0.nuget.org/query",
      "@type": "SearchQueryService/3.0.0-rc",
      "comment": "Query endpoint of NuGet Search service (primary) used by RC clients"
    },
    {
      "@id": "https://api-v2v3search-0.nuget.org/autocomplete",
      "@type": "SearchAutocompleteService/3.0.0-rc",
      "comment": "Autocomplete endpoint of NuGet Search service (primary) used by RC clients"
    },
    {
      "@id": "https://api.nuget.org/v3/registration2/",
      "@type": "RegistrationsBaseUrl/3.0.0-rc",
      "comment": "Base URL of Azure storage where NuGet package registration info is stored used by RC clients. This base URL does not include SemVer 2.0.0 packages."
    }
  ]
}
```

Query for all packages published to nuget.org

9/4/2018 • 7 minutes to read • [Edit Online](#)

One common query pattern on the legacy OData V2 API was enumerating all packages published to nuget.org, ordered by when the package was published. Scenarios requiring this kind of query against nuget.org vary widely:

- Replicating nuget.org entirely
- Detecting when packages have new versions released
- Finding packages that depend on your package

The legacy way of doing this typically depended on sorting the OData package entity by a timestamp and paging across the massive result set using `skip` and `top` (page size) parameters. Unfortunately, this approach has some drawbacks:

- Possibility of missing packages, because the queries are being made on data that is often changing order
- Slow query response time, because the queries are not optimized (the most optimized queries are ones that support a mainline scenario for the official NuGet client)
- Use of deprecated and undocumented API, meaning the support of such queries in the future is not guaranteed
- Inability to replay history in the exact order that it transpired

For this reason, the following guide can be followed to address the aforementioned scenarios in a more reliable and future-proof way.

Overview

At the center of this guide is resource in the [NuGet API](#) called the **catalog**. The catalog is an append-only API that allows the caller to see a full history of packages added to, modified, and deleted from nuget.org. If you are interested in all or even a subset of packages published to nuget.org, the catalog is a great way to stay up-to-date with the set of currently available packages as time goes on.

This guide is intended to be a high-level walk-through but if you are interested in the fine-grain details of the catalog, see its [API reference document](#).

The following steps can be implemented in any programming language of your choice. If you want a full running sample, take a look at the [C# sample](#) mentioned below.

Otherwise, follow the guide below to build a reliable catalog reader.

Initialize a cursor

The first step in building a reliable catalog reader is implementing a cursor. For full details about the design of a catalog cursor, see the [catalog reference document](#). In short, cursor is a point in time up to which you have processed events in the catalog. Events in the catalog represent package publishes and other package changes. If you care about all packages ever published to NuGet (since the beginning of time), you would initialize your cursor to a "minimum value" timestamp (e.g. `DateTime.MinValue` in .NET). If you care only about packages published starting now, you would use the current timestamp as your initial cursor value.

For this guide, we'll initialize our cursor to a timestamp one hour ago. For now, just save that timestamp in memory.

```
DateTime cursor = DateTime.UtcNow.AddHours(-1);
```

Determine catalog index URL

The location of every resource (endpoint) in the NuGet API should be discovered using the [service index](#). Because this guide focuses on nuget.org, we'll be using nuget.org's service index.

```
GET https://api.nuget.org/v3/index.json
```

The service document is JSON document containing all of the resources on nuget.org. Look for the resource having the `@type` property value of `Catalog/3.0.0`. The associated `@id` property value is the URL to the catalog index itself.

Find new catalog leaves

Using the `@id` property value found in the previous step, download the catalog index:

```
GET https://api.nuget.org/v3/catalog0/index.json
```

Deserialize the [catalog index](#). Filter out all [catalog page objects](#) with `commitTimeStamp` less than or equal to your current cursor value.

For each remaining catalog page, download the full document using the `@id` property.

```
GET https://api.nuget.org/v3/catalog0/page2926.json
```

Deserialize the [catalog page](#). Filter out all [catalog leaf objects](#) with `commitTimeStamp` less than or equal to your current cursor value.

After you have downloaded all of the catalog pages not filtered out, you have a set of catalog leaf objects representing packages that have been published, unlisted, listed, or deleted in the time between your cursor timestamp and now.

Process catalog leaves

At this point, you can perform any custom processing you'd like on the catalog items. If all you need is the ID and version of the package, you can inspect the `nuget:id` and `nuget:version` properties on the catalog item objects found in the pages. Make sure to look at the `@type` property to know if the catalog item concerns an existing package or a deleted package.

If you are interested in the metadata about the package (such as the description, dependencies, .nupkg size, etc), you can fetch the [catalog leaf document](#) using the `@id` property.

```
GET https://api.nuget.org/v3/catalog0/data/2015.02.01.11.18.40/windowsazure.storage.1.0.0.json
```

This document has all of the metadata included in the [package metadata resource](#), and more!

This step is where you implement your custom logic. The other steps in this guide are implemented in pretty much the same way not matter what you are doing with the catalog leaves.

Downloading the .nupkg

If you are interested in downloading the .nupkg's for packages found in the catalog, you can use the [package content resource](#). However, note that there is a short delay between when a package is found in catalog and when it's available in the package content resource. Therefore, if you encounter `404 Not Found` when attempting to download a .nupkg for a package that you found in the catalog, simply retry a short time later. Fixing this delay is tracked by GitHub issue [NuGet/NuGetGallery#3455](#).

Move the cursor forward

Once you have successfully processed the catalog items, you need to determine the new cursor value to save. To do this, find the maximum (latest chronologically) `commitTimeStamp` of all catalog items that you processed. This is your new cursor value. Save it to some persistent store, like a database, file system, or blob storage. When you want to get more catalog items, simply start from the [first step](#) by initializing your cursor value from this persistent store.

If your application throws an exception or faults, don't move the cursor forward. Moving the cursor forward has the meaning that you never again need to process catalog items before your cursor.

If, for some reason, you have a bug in how you process catalog leaves, you can simply move your cursor backward in time and allow your code to reprocess the old catalog items.

C# sample code

Because the catalog is a set of JSON documents available over HTTP, it can be interacted with using any programming language that has an HTTP client and JSON deserializer.

C# samples are available in the [NuGet/Samples repository](#).

```
git clone https://github.com/NuGet/Samples.git
```

Catalog SDK

The easiest way to consume the catalog is to use the pre-release .NET catalog SDK package: [NuGet.Protocol.Catalog](#). This package is available on the `nuget-build` MyGet feed, for which you use the NuGet package source URL <https://dotnet.myget.org/F/nuget-build/api/v3/index.json>.

You can install this package to a project compatible with `netstandard1.3` or greater (such as .NET Framework 4.6).

A sample using this package is available on GitHub in the [NuGet.Protocol.Catalog.Sample project](#).

Sample output

```
2017-11-10T22:16:44.8689025+00:00: Found package details leaf for xSkrape.APIWrapper.REST 1.0.2.
2017-11-10T22:16:54.6972769+00:00: Found package details leaf for xSkrape.APIWrapper.REST 1.0.1.
2017-11-10T22:19:20.6385542+00:00: Found package details leaf for Platform.EnUnity 1.0.8.
...
2017-11-10T23:05:04.9695890+00:00: Found package details leaf for xSkrape.APIWrapper.Base 1.0.1.
2017-11-10T23:05:04.9695890+00:00: Found package details leaf for xSkrape.APIWrapper.Base 1.0.2.
2017-11-10T23:07:23.1303569+00:00: Found package details leaf for VeiculoX.Model 0.0.15.
Processing the catalog leafs failed. Retrying.
fail: NuGet.Protocol.Catalog.LoggerCatalogLeafProcessor[0]
    10 catalog commits have been processed. We will now simulate a failure.
warn: NuGet.Protocol.Catalog.CatalogProcessor[0]
    Failed to process leaf
https://api.nuget.org/v3/catalog0/data/2017.11.10.23.07.23/veiculox.model.0.0.15.json (VeiculoX.Model 0.0.15, PackageDetails).
warn: NuGet.Protocol.Catalog.CatalogProcessor[0]
    13 out of 59 leaves were left incomplete due to a processing failure.
warn: NuGet.Protocol.Catalog.CatalogProcessor[0]
    1 out of 1 pages were left incomplete due to a processing failure.
2017-11-10T23:07:23.1303569+00:00: Found package details leaf for VeiculoX.Model 0.0.15.
2017-11-10T23:07:33.0212446+00:00: Found package details leaf for VeiculoX.Model 0.0.14.
2017-11-10T23:07:41.6621837+00:00: Found package details leaf for VeiculoX.Model 0.0.13.
2017-11-10T23:09:58.5728614+00:00: Found package details leaf for CreaSoft.Composition.Web.Extensions 1.1.0.
2017-11-10T23:09:58.5728614+00:00: Found package details leaf for DarkXaHTeP.Extensions.Configuration.Consul 0.0.4.
2017-11-10T23:09:58.5728614+00:00: Found package details leaf for xSkrape.APIWrapper.REST.Sample 1.0.3.
2017-11-10T23:10:09.0574930+00:00: Found package details leaf for Microsoft.VisualStudio.Imaging 15.4.27004.
2017-11-10T23:10:09.0574930+00:00: Found package details leaf for Microsoft.VisualStudio.Imaging.Interop.14.0.DesignTime 14.3.25407.
2017-11-10T23:10:09.0574930+00:00: Found package details leaf for Microsoft.VisualStudio.Language.Intellisense 15.4.27004.
2017-11-10T23:10:09.0574930+00:00: Found package details leaf for Microsoft.VisualStudio.Language.StandardClassification 15.4.27004.
2017-11-10T23:10:09.0574930+00:00: Found package details leaf for Microsoft.VisualStudio.ManagedInterfaces 8.0.50727.
2017-11-10T23:10:09.0574930+00:00: Found package details leaf for xSkrape.APIWrapper.REST.Sample 1.0.2.
2017-11-10T23:10:09.0574930+00:00: Found package details leaf for xSkrape.APIWrapper.REST.Sample 1.0.3.
```

Minimal sample

For an example with fewer dependencies that illustrates the interaction with the catalog in more detail, see the [CatalogReaderExample sample project](#). The project targets `netcoreapp2.0` and depends on the [NuGet.Protocol 4.4.0](#) (for resolving the service index) and [Newtonsoft.Json 9.0.1](#) (for JSON deserialization).

The main logic of the code is visible in the [Program.cs file](#).

Sample output

```
No cursor found. Defaulting to 11/2/2017 9:41:28 PM.
Fetched catalog index https://api.nuget.org/v3/catalog0/index.json.
Fetched catalog page https://api.nuget.org/v3/catalog0/page2935.json.
Processing 69 catalog leaves.
11/2/2017 9:32:35 PM: DotVVM.Compiler.Light 1.1.7 (type is nuget:PackageDetails)
11/2/2017 9:32:35 PM: Momentum.Pm.Api 5.12.181-beta (type is nuget:PackageDetails)
11/2/2017 9:32:44 PM: Momentum.Pm.PortalApi 5.12.181-beta (type is nuget:PackageDetails)
11/2/2017 9:35:14 PM: Genesys.Extensions.Standard 3.17.11.40 (type is nuget:PackageDetails)
11/2/2017 9:35:14 PM: Genesys.Extensions.Core 3.17.11.40 (type is nuget:PackageDetails)
11/2/2017 9:35:14 PM: Halforbit.DataStores.FileStores.Serialization.Bond 1.0.4 (type is nuget:PackageDetails)
11/2/2017 9:35:14 PM: Halforbit.DataStores.FileStores.AmazonS3 1.0.4 (type is nuget:PackageDetails)
11/2/2017 9:35:14 PM: Halforbit.DataStores.DocumentStores.DocumentDb 1.0.6 (type is nuget:PackageDetails)
11/2/2017 9:35:14 PM: Halforbit.DataStores.FileStores.BlobStorage 1.0.5 (type is nuget:PackageDetails)
...
11/2/2017 10:23:54 PM: Cake.GitPackager 0.1.2 (type is nuget:PackageDetails)
11/2/2017 10:23:54 PM: UtilPack.NuGet 2.0.0 (type is nuget:PackageDetails)
11/2/2017 10:23:54 PM: UtilPack.NuGet.AssemblyLoading 2.0.0 (type is nuget:PackageDetails)
11/2/2017 10:26:26 PM: UtilPack.NuGet.Deployment 2.0.0 (type is nuget:PackageDetails)
11/2/2017 10:26:26 PM: UtilPack.NuGet.Common.MSBuild 2.0.0 (type is nuget:PackageDetails)
11/2/2017 10:26:36 PM: InstaClient 1.0.2 (type is nuget:PackageDetails)
11/2/2017 10:26:36 PM: SecureStrConvertor.VARUN_RUSIYA 1.0.0.5 (type is nuget:PackageDetails)
Writing cursor value: 11/2/2017 10:26:36 PM.
```

Rate Limits

4/25/2019 • 2 minutes to read • [Edit Online](#)

The NuGet.org API enforces rate limiting to prevent abuse. Requests that exceed the rate limit return the following error:

```
{  
  "statusCode": 429,  
  "message": "Rate limit is exceeded. Try again in 56 seconds."  
}
```

In addition to request throttling using rate limits, some APIs also enforce quota. Requests that exceed the quota return the following error:

```
{  
  "statusCode": 403,  
  "message": "Quota exceeded."  
}
```

The following tables list the rate limits for the NuGet.org API.

Package search

NOTE

We recommend using NuGet.org's [V3 APIs](#) for search that are performant and do not have any limit currently. For V1 and V2 search APIs, the followins limits apply:

API	LIMIT TYPE	LIMIT VALUE	API USECASE
GET <code>/api/v1/Packages</code>	IP	1000 / minute	Query NuGet package metadata via v1 OData <code>Packages</code> collection
GET <code>/api/v1/Search()</code>	IP	3000 / minute	Search for NuGet packages via v1 Search endpoint
GET <code>/api/v2/Packages</code>	IP	20000 / minute	Query NuGet package metadata via v2 OData <code>Packages</code> collection
GET <code>/api/v2/Packages/\$count</code>	IP	100 / minute	Query NuGet package count via v2 OData <code>Packages</code> collection

Package Push and Unlist

API	LIMIT TYPE	LIMIT VALUE	API USECASE
PUT <code>/api/v2/package</code>	API Key	350 / hour	Upload a new NuGet package (version) via v2 push endpoint
DELETE <code>/api/v2/package/{id}/{version}</code>	API Key	250 / hour	Unlist a NuGet package (version) via v2 endpoint

nuget.org protocols

9/4/2018 • 2 minutes to read • [Edit Online](#)

To interact with nuget.org, clients need to follow certain protocols. Because these protocols keep evolving, clients must identify the protocol version they use when calling specific nuget.org APIs. This allows nuget.org to introduce changes in a non-breaking way for the old clients.

NOTE

The APIs documented on this page are specific to nuget.org and there is no expectation for other NuGet server implementations to introduce these APIs.

For information about the NuGet API implemented broadly across the NuGet ecosystem, see the [API overview](#).

This topic lists various protocols as and when they come to existence.

NuGet protocol version 4.1.0

The 4.1.0 protocol specifies usage of verify-scope keys to interact with services other than nuget.org, to validate a package against a nuget.org account. Note that the `4.1.0` version number is an opaque string but happens to coincide with the first version of the official NuGet client that supported this protocol.

Validation ensures that the user-created API keys are used only with nuget.org, and that other verification or validation from a third-party service is handled through a one-time use verify-scope keys. These verify-scope keys can be used to validate that the package belongs to a particular user (account) on nuget.org.

Client requirement

Clients are required to pass the following header when they make API calls to **push** packages to nuget.org:

```
X-NuGet-Protocol-Version: 4.1.0
```

Note that the `X-NuGet-Client-Version` header has similar semantics but is reserved to only be used by the official NuGet client. Third party clients should use the `X-NuGet-Protocol-Version` header and value.

The **push** protocol itself is described in the documentation for the [PackagePublish resource](#).

If a client interacts with external services and needs to validate whether a package belongs to a particular user (account), it should use the following protocol and use the verify-scope keys and not the API keys from nuget.org.

API to request a verify-scope key

This API is used to get a verify-scope key for a nuget.org author to validate a package owned by him/her.

```
POST api/v2/package/create-verification-key/{ID}/{VERSION}
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES

NAME	IN	TYPE	REQUIRED	NOTES
ID	URL	string	yes	The package identifier for which the verify scope key is requested
VERSION	URL	string	no	The package version
X-NuGet-ApiKey	Header	string	yes	For example, X-NuGet-ApiKey: {USER_API_KEY}

Response

```
{
  "Key": "{Verify scope key from nuget.org}",
  "Expires": "{Date}"
}
```

API to verify the verify scope key

This API is used to validate a verify-scope key for package owned by the nuget.org author.

```
GET api/v2/verifykey/{ID}/{VERSION}
```

Request parameters

NAME	IN	TYPE	REQUIRED	NOTES
ID	URL	string	yes	The package identifier for which the verify scope key is requested
VERSION	URL	string	no	The package version
X-NuGet-ApiKey	Header	string	yes	For example, X-NuGet-ApiKey: {VERIFY_SCOPE_KEY}

NOTE

This verify scope API key expires in a day's time or on first use, whichever occurs first.

Response

STATUS CODE	MEANING
200	The API key is valid
403	The API key is invalid or not authorized to push against the package
404	The package referred to by <code>ID</code> and <code>VERSION</code> (optional) does not exist

tools.json for discovering nuget.exe versions

11/5/2019 • 2 minutes to read • [Edit Online](#)

Today, there are a few ways to get the latest version of nuget.exe on your machine in a scriptable fashion. For example, you can download and extract the `NuGet.CommandLine` package from nuget.org. This has some complexity since it either requires that you already have nuget.exe (for `nuget.exe install`) or you have to unzip the .nupkg using a basic unzip tool and find the binary inside.

If you already have nuget.exe, you can also use `nuget.exe update -self`, however this also requires having an existing copy of nuget.exe. This approach also updates you to the latest version. It does not allow the use of a specific version.

The `tools.json` endpoint is available to both solve the bootstrapping problem and to give control of the version of nuget.exe that you download. This can be used in CI/CD environments or in custom scripts to discover and download any released version of nuget.exe.

The `tools.json` endpoint can be fetched using an unauthenticated HTTP request (e.g. `Invoke-WebRequest` in PowerShell or `wget`). It can be parsed using a JSON deserializer and subsequent nuget.exe download URLs can also be fetched using unauthenticated HTTP requests.

The endpoint can be fetched using the `GET` method:

```
GET https://dist.nuget.org/tools.json
```

The [JSON schema](#) for the endpoint is available here:

```
GET https://dist.nuget.org/tools.schema.json
```

Response

The response is a JSON document containing all of the available versions of nuget.exe.

The root JSON object has the following property:

NAME	TYPE	REQUIRED
nuget.exe	array of objects	yes

Each object in the `nuget.exe` array has the following properties:

NAME	TYPE	REQUIRED	NOTES
version	string	yes	A SemVer 2.0.0 string
url	string	yes	An absolute URL for downloading this version of nuget.exe
stage	string	yes	An enum string

NAME	TYPE	REQUIRED	NOTES
uploaded	string	yes	An approximate ISO 8601 timestamp of when the version was made available

The items in the array will be sorted in descending, SemVer 2.0.0 order. This guarantee is meant to reduce the burden of a client that is interested in highest version number. However this does mean that the list is not sorted in chronological order. For example, if a lower major version is serviced at a date later than a higher major version, this serviced version will not appear at the top of the list. If you want the latest version released by *timestamp*, simply sort the array by the `uploaded` string. This works because the `uploaded` timestamp is in the [ISO 8601](#) format which can be sorted chronologically by using a lexicographical sort (i.e. a simple string sort).

The `stage` property indicates how vetted this version of the tool is.

STAGE	MEANING
EarlyAccessPreview	Not yet visible on the download web page and should be validated by partners
Released	Available on the download site but is not yet recommended for wide-spread consumption
ReleasedAndBlessed	Available on the download site and is recommended for consumption

One simple approach for having the latest, recommended version is to take the first version in the list that has the `stage` value of `ReleasedAndBlessed`. This works because the versions are sorted in SemVer 2.0.0 order.

The `NuGet.CommandLine` package on nuget.org is typically only updated with `ReleasedAndBlessed` versions.

Sample request

```
GET https://dist.nuget.org/tools.json
```

Sample response

```
{  
  "nuget.exe": [  
    {  
      "version": "4.8.0-preview3",  
      "url": "https://dist.nuget.org/win-x86-commandline/v4.8.0-preview3/nuget.exe",  
      "stage": "EarlyAccessPreview",  
      "uploaded": "2018-07-06T23:00:00.0000000Z"  
    },  
    {  
      "version": "4.7.1",  
      "url": "https://dist.nuget.org/win-x86-commandline/v4.7.1/nuget.exe",  
      "stage": "ReleasedAndBlessed",  
      "uploaded": "2018-08-10T23:00:00.0000000Z"  
    },  
    {  
      "version": "4.6.1",  
      "url": "https://dist.nuget.org/win-x86-commandline/v4.6.1/nuget.exe",  
      "stage": "Released",  
      "uploaded": "2018-03-22T23:00:00.0000000Z"  
    },  
    {  
      "version": "3.5.0",  
      "url": "https://dist.nuget.org/win-x86-commandline/v3.5.0/nuget.exe",  
      "stage": "ReleasedAndBlessed",  
      "uploaded": "2016-12-19T15:30:00.0000000-08:00"  
    },  
    {  
      "version": "2.8.6",  
      "url": "https://dist.nuget.org/win-x86-commandline/v2.8.6/nuget.exe",  
      "stage": "ReleasedAndBlessed",  
      "uploaded": "2015-09-01T12:30:00.0000000-07:00"  
    }  
  ]  
}
```

NuGet Client SDK

10/26/2019 • 2 minutes to read • [Edit Online](#)

The *NuGet Client SDK* refers to a group of NuGet packages centered around [NuGet.Commands](#), [NuGet.Packaging](#), and [NuGet.Protocol](#). These packages replace the earlier [NuGet.Core](#) library.

NOTE

For documentation on the NuGet server protocol, please refer to the [NuGet Server API](#).

Source code

The source code is published on GitHub in the project [NuGet/NuGet.Client](#).

Third-party documentation

You can find examples and documentation for some of the API in the following blog series by Dave Glick, published 2016:

- [Exploring the NuGet v3 Libraries, Part 1: Introduction and concepts](#)
- [Exploring the NuGet v3 Libraries, Part 2: Searching for packages](#)
- [Exploring the NuGet v3 Libraries, Part 3: Installing packages](#)

NOTE

These blog posts were written shortly after the **3.4.3** version of the NuGet client SDK packages were released. Newer versions of the packages may be incompatible with the information in the blog posts.

Martin Björkström did a follow-up blog post to Dave Glick's blog series where he introduces a different approach on using the NuGet Client SDK for installing NuGet packages:

- [Revisiting the NuGet v3 Libraries](#)

Errors and warnings

9/26/2019 • 2 minutes to read • [Edit Online](#)

In NuGet 4.3.0+, errors and warnings are numbered as described in this topic and provide detailed information to help you address the issues involved.

The errors and warnings listed here are available only with [PackageReference-based](#) projects and NuGet 4.3.0+. NuGet also honors MSBuild properties to suppress warnings or elevate them to errors. For more information, see [How to: Suppress Compiler Warnings](#) in the Visual Studio documentation.

Errors

GROUP	ERROR NUMBERS
Invalid input errors	NU1001 , NU1002 , NU1003
Missing package and project errors	NU1100 , NU1101 , NU1102 , NU1103 , NU1104 , NU1105 , NU1106 , NU1107 , NU1108
Compatibility errors	NU1201 , NU1202 , NU1203 , NU1401
NuGet internal errors	NU1000
Signed packages errors (creation and verification)	NU3001 , NU3004 , NU3005 , NU3008 , NU3034
Pack Errors	NU5000 , NU5001 , NU5002 , NU5003 , NU5004 , NU5005 , NU5007 , NU5008 , NU5009 , NU5010 , NU5011 , NU5012 , NU5013 , NU5014 , NU5015 , NU5016 , NU5017 , NU5018 , NU5019 , NU5020 , NU5021 , NU5022 , NU5023 , NU5024 , NU5025 , NU5026 , NU5027 , NU5028 , NU5029 , NU5036
License specific Pack Errors	NU5030 , NU5031 , NU5032 , NU5033 , NU5034 , NU5035

Warnings

GROUP	WARNING NUMBERS
Invalid input warnings	NU1501 , NU1502 , NU1503
Unexpected package version warnings	NU1601 , NU1602 , NU1603 , NU1604 , NU1605 , NU1606 , NU1607
Resolver conflict warnings	NU1608
Package fallback warnings	NU1701
Feed warnings	NU1801
NuGet internal warnings	NU1500

GROUP	WARNING NUMBERS
Signed packages warnings (creation and verification)	NU3000 , NU3002 , NU3003 , NU3006 , NU3007 , NU3009 , NU3010 , NU3011 , NU3012 , NU3013 , NU3014 , NU3015 , NU3016 , NU3017 , NU3018 , NU3019 , NU3020 , NU3021 , NU3022 , NU3023 , NU3024 , NU3025 , NU3026 , NU3027 , NU3028 , NU3029 , NU3030 , NU3031 , NU3032 , NU3033 , NU3035 , NU3036 , NU3037 , NU3038 , NU3040
Pack Warnings	NU5100 , NU5101 , NU5102 , NU5103 , NU5104 , NU5105 , NU5106 , NU5107 , NU5108 , NU5109 , NU5110 , NU5111 , NU5112 , NU5114 , NU5115 , NU5116 , NU5117 , NU5118 , NU5119 , NU5120 , NU5121 , NU5122 , NU5123 , NU5127 , NU5128 , NU5129 , NU5130 , NU5131 , NU5500
License specific Pack Warnings	NU5124 , NU5125
Icon specific Pack Warnings	NU5046 , NU5047 , NU5048

NuGet Error NU1000

9/4/2018 • 2 minutes to read • [Edit Online](#)

Issue

The NuGet operation had a problem. NU1000 is used when we haven't yet assigned a unique error code for that issue. So we can improve, please feel free to [file the issue](#) with details of your error.

Solution

Check the output window (in Visual Studio) or console output (with NuGet command line tools) for more information.

NuGet Error NU1001

9/4/2018 • 2 minutes to read • [Edit Online](#)

The project 'Project' does not specify any target frameworks in 'ProjectFile'

Issue

The project doesn't contain one or more frameworks.

Solution

Add a `TargetFramework` or `TargetFrameworks` property to the specified project file.

NuGet Error NU1002

9/4/2018 • 2 minutes to read • [Edit Online](#)

'CLEAR' cannot be used in conjunction with other values

Issue

Invalid combination of inputs along with a CLEAR keyword.

Solution

Use CLEAR by itself and omit all other inputs.

NuGet Error NU1003

9/4/2018 • 2 minutes to read • [Edit Online](#)

`PackageTargetFallback` and `AssetTargetFallback` cannot be used together. Remove `PackageTargetFallback`(deprecated) references from the project environment.

Issue

`PackageTargetFallback` and `AssetTargetFallback` provide different behavior for selecting assets and cannot be used together.

Solution

Remove the deprecated `PackageTargetFallback` element from the project.

NuGet Error NU1100

9/4/2018 • 2 minutes to read • [Edit Online](#)

Unable to resolve 'Dependency dll' for 'TargetFramework'

Issue

A dependency group not be resolved. This is a generic issue for types that are not packages or projects.

Solution

Open the project file and examine the list of its dependencies. Check that each dependency exists on the package sources you're using, and that the package supports the project's target framework.

NuGet Error NU1101

8/15/2019 • 2 minutes to read • [Edit Online](#)

```
Unable to find package 'PackageName'. No packages exist with this id in source(s): 'sourceA',  
'sourceB', 'sourceC'
```

Issue

The package cannot be found on any sources.

Solution

Examine the project's dependencies in Visual Studio to be sure you're using the correct package identifier and version number. Also check that the [NuGet configuration](#) identifies the package sources you are expected to be using. If you use packages that have [Semantic Versioning 2.0.0](#), please make sure that you are using the V3 feed, <https://api.nuget.org/v3/index.json>, in the [NuGet configuration](#).

NuGet Error NU1102

11/30/2018 • 2 minutes to read • [Edit Online](#)

```
Unable to find package 'PackageName' with version (>= 9.0.1)
- Found 30 version(s) in 'sourceA' [ Nearest version: '4.0.0' ]
- Found 10 version(s) in 'sourceB' [ Nearest version: '4.0.0-rc-2129' ]
- Found 9 version(s) in 'sourceC' [ Nearest version: '3.0.0-beta-00032' ]
- Found 0 version(s) in 'sourceD'
- Found 0 version(s) in 'sourceE'
```

Issue

The package identifier is found but a version within the specified dependency range cannot be found on any of the sources. The range might be specified by a package and not the user.

Solution

Edit the project file to correct the package version. Also check that the [NuGet configuration](#) identifies the package sources you expect to be using. You may need to change the requested version if this package is referenced by the project directly.

NuGet Error NU1103

8/15/2019 • 2 minutes to read • [Edit Online](#)

```
Unable to find a stable package 'PackageName' with version (>= 3.0.0)
- Found 10 version(s) in 'sourceA' [ Nearest version: '4.0.0-rc-2129' ]
- Found 9 version(s) in 'sourceB' [ Nearest version: '3.0.0-beta-00032' ]
- Found 0 version(s) in 'sourceC'
- Found 0 version(s) in 'sourceD'
```

Issue

The project specified a stable version for the dependency range, but no stable versions were found in that range. Pre-release versions were found but are not allowed.

Solution

Edit the version range in the project file to include pre-release versions. See [Package versioning](#).

NuGet Error NU1104

9/4/2018 • 2 minutes to read • [Edit Online](#)

Project reference does not exist 'ProjectFile'. Check that the project reference is valid and that the project file exists.

Issue

A ProjectReference points to a file that doesn't exist.

Solution

Edit the project file to either correct the path to the referenced project or to remove the reference altogether if it's no longer needed.

NuGet Error NU1105

9/4/2018 • 2 minutes to read • [Edit Online](#)

Unable to read project information for 'ProjectFile'. The project file may be invalid or missing targets required for restore.

Issue

The project file exists but no restore information was provided for it.

Solution

In Visual Studio, the error could mean that the project is unloaded, in which case reload the project. From the command line this could mean that the file is corrupt or that it doesn't contain the custom "after imports" target needed for restore to read the project. Check that the project file is valid and contains an "after imports" target.

NuGet Error NU1106

9/4/2018 • 2 minutes to read • [Edit Online](#)

Unable to satisfy conflicting requests for 'PackageId': 'Conflict path' Framework: 'Target graph'

Issue

Dependency constraints cannot be resolved.

Solution

Edit the project file to specify more open-ended ranges for the dependency rather than an exact version.

NuGet Error NU1107

8/15/2019 • 2 minutes to read • [Edit Online](#)

```
Version conflict detected for 'PackageA'. Install/reference 'PackageA' v4.0.0 directly to resolve this issue.
```

```
'PackageB' 3.5.0 -> 'PackageA' (= 3.5.0)
'PackageC' 4.0.0 -> 'PackageA' (= 4.0.0)
```

Issue

Unable to resolve dependency constraints between packages. Two different packages are asking for two different versions of 'PackageA'. The project needs to choose which version of 'PackageA' to use.

Solution

Install/reference 'PackageA' directly (in the project file) with the exact version that you choose. Generally, picking the higher version is the right choice.

To install a specific version, see the information for the tool that you're using:

- [Visual Studio](#)
- [dotnet CLI](#)
- [nuget.exe CLI](#)
- [Package Manager Console](#)

Note

Early versions of Visual Studio 2017 may have reported this as a warning (NU1607).

NuGet Error NU1108

9/4/2018 • 2 minutes to read • [Edit Online](#)

Cycle detected: A -> B -> A

Issue

A circular dependency was detected.

Solution

The package is authored incorrectly; contact the package owner to correct the bug.

Note

Early versions of Visual Studio 2017 may have reported this as a warning (NU1606).

NuGet Error NU1201

4/29/2019 • 2 minutes to read • [Edit Online](#)

Example 1

```
Project 'ProjectA' is not compatible with 'TargetFramework'. Project 'ProjectA' supports:  
- 'TargetFrameworkA'  
- 'TargetFrameworkB'
```

Issue

A dependency project doesn't contain a framework compatible with the current project. Typically, the project's target framework is a higher version than the consuming project.

Solution

Change the project's target framework to an equal or lower version than the consuming project.

Example 2 - NetStandard targetted projects cannot reference NetCoreApp targetted projects

```
Project 'ProjectB' is not compatible with netstandard2.0 (.NETStandard,Version=v2.0). Project  
'ProjectB' supports: netcoreapp2.0 (.NETCoreApp,Version=v2.0)
```

Issue

In this case:

- ProjectA targets NetStandard 2.0
- ProjectB targets NetCoreApp 2.0
- ProjectA has a project reference to ProjectB

NetStandard projects can never depend on a NetCoreApp project.

Solution

Either:

- change ProjectA to target NetCoreApp 2.0
- or
- change ProjectB to target NetStandard 2.0

NuGet Error NU1202

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
Package 'PackageName' 4.0.11 is not compatible with 'TargetFramework'. Package 'PackageName' 4.0.11
supports:
- 'TargetFrameworkA'
- 'TargetFrameworkB'
- 'TargetFrameworkC'
```

Issue

A dependency package doesn't contain any assets compatible with the project.

Solution

Change the project's target framework to one that the package supports.

NuGet Error NU1203

9/4/2018 • 2 minutes to read • [Edit Online](#)

'`PackageId`' 1.0.0 provides a compile-time reference assembly for 'Foo.dll' on 'TargetFramework', but there is no compatible run-time assembly.

Issue

The package doesn't support the project's `RuntimeIdentifier`.

Solution

Change the `RuntimeIdentifier` values used in the project as needed.

NuGet Error NU1401

9/4/2018 • 2 minutes to read • [Edit Online](#)

The 'PackageName' package requires NuGet client version '5.0.0' or above, but the current NuGet version is '4.3.0'.

Issue

The package requires features or frameworks not currently supported by the installed version of NuGet.

Solution

Install a newer version of NuGet. See [Installing NuGet client tools](#).

NuGet Warning NU1500

9/4/2018 • 2 minutes to read • [Edit Online](#)

Issue

The NuGet operation had a problem. NU1500 is used when we haven't yet assigned a unique warning code for that issue. So we can improve, please feel free to [file the issue](#) with details of your error.

Solution

Check the output window (in Visual Studio) or console output (with NuGet command line tools) for more information.

NuGet Warning NU1501

9/4/2018 • 2 minutes to read • [Edit Online](#)

The folder 'FolderPath' does not contain a project to restore.

Issue

The project restore is attempting to operate on was not found.

Solution

Run nuget restore in a folder that contains a project.

NuGet Warning NU1502

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
Unknown Compatibility Profile: 'aaa'
```

Issue

`RuntimeSupports` contains an invalid profile. Typically, the supports profile was not found in a `runtime.json` file from the current dependency packages.

Solution

Check the `RuntimeSupports` value in your project.

NuGet Warning NU1503

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
Skipping restore for project 'c:\a.csproj'. The project file may be invalid or missing targets required for restore.
```

Issue

A dependency project doesn't import NuGet's restore targets. This is similar to NU1105 but here the project is skipped and ignored instead of causing all of restore to fail. In complex solutions there are often other types of projects that may not support restore.

Solution

This can happen for projects that do not import common props/targets which automatically import restore. If the project doesn't need to be restored this can be ignored. Otherwise, edit the affected project to add targets for restore.

NuGet Warning NU1601

9/4/2018 • 2 minutes to read • [Edit Online](#)

Dependency specified was 'PackageName' (>= 3.5.0) but ended up with 'PackageName' 4.0.0.

Issue

A direct project dependency was bumped to a higher version than the project specified.

Solution

Update the dependency in the project to an appropriate version.

NuGet Warning NU1602

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
'PackageA' 4.0.0 does not provide an inclusive lower bound for dependency 'PackageB' (> 3.5.0). An approximate best match of 3.6.0 was resolved.
```

Issue

A package dependency is missing a lower bound. This doesn't allow restore to find the *best match*. Each restore will float downwards trying to find a lower version that can be used. This means that restore goes online to check all sources each time instead of using the packages that already exist in the user package folder.

Solution

This is usually a package authoring error. Contact the package author to resolve the issue.

NuGet Warning NU1603

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
'PackageA' 4.0.0 depends on 'PackageB' (>= 4.0.0) but 4.0.0 was not found. An approximate best match of 5.0.0 was resolved.
```

Issue

A package dependency specified a version that could not be found. Typically, the package sources do not contain the expected lower bound version. A higher version was used instead, which differs from what the package was authored against.

This means that restore did not find the *best match*. Each restore will float downwards trying to find a lower version that can be used. This means that restore goes online to check all sources each time instead of using the packages that already exist in the user package folder.

Solution

If the package expected has not been released then this may be a package authoring error. Contact the package author to resolve the issue. If the package has been released, then check that it's available on the package sources you're using. If using a private source, you may need to update the package on that feed.

NuGet Warning NU1604

9/4/2018 • 2 minutes to read • [Edit Online](#)

Project dependency 'PackageA' (`<= 9.0.0`) does not contain an inclusive lower bound. Include a lower bound in the dependency version to ensure consistent restore results.

Issue

A project dependency doesn't define a lower bound.

This means that restore did not find the *best match*. Each restore will float downwards trying to find a lower version that can be used. This means that restore goes online to check all sources each time instead of using the packages that already exist in the user package folder.

Solution

Update the project's `PackageReference` `Version` attribute to include a lower bound.

NuGet Warning NU1605

10/15/2019 • 2 minutes to read • [Edit Online](#)

Example 1

```
Detected package downgrade: 'PackageB' from 4.0.0 to 3.5.0. Reference the package directly from the project to select a different version.  
'PackageA' 3.5.0 -> 'PackageB' 3.5.0  
'PackageC' 4.0.0 -> 'PackageD' 4.0.0 -> 'PackageB' 4.0.0
```

Issue

A dependency package specified a version constraint on a higher version of a package than restore ultimately resolved. That is because of the [nearest-wins](#) rule - when resolving packages, the version of the nearer package in the graph will override that of the distant package with the same ID.

Solution

To the project exhibiting the restore error, add a package reference to the higher version of the package.

In the example above, you would add a package reference to Package B version 4.0.0:

```
'PackageA' 3.5.0 -> 'PackageB' 3.5.0  
  
'PackageC' 4.0.0 -> 'PackageD' 4.0.0 -> 'PackageB' 4.0.0  
  
'PackageB' 4.0.0
```

Since, [nearest-wins](#), the direct package reference to PackageB v4.0.0, will take precedence over the transitive reference to PackageB v3.5.0.

Example 2

```
Detected package downgrade: System.IO.FileSystem.Primitives from 4.3.0 to 4.0.1. Reference the package directly from the project to select a different version.  
  
Project -> System.IO.FileSystem 4.0.1 -> runtime.win.System.IO.FileSystem 4.3.0 -> System.IO.FileSystem.Primitives (>= 4.3.0)  
  
Project -> System.IO.FileSystem 4.0.1 -> System.IO.FileSystem.Primitives (>= 4.0.1)
```

Issue

Certain combinations of packages which shipped with .NET Core 1.0 and 1.1 are not compatible with each other when they are referenced together in a .NET Core 3.0 or higher project, and a `Runtimeldentifier` is specified. The problematic packages generally start with `System.` or `Microsoft.`, and have version numbers between 4.0.0 and 4.3.1. In this case, the downgrade message will have a package starting with `runtime.` in the dependency chain.

Solution

To work around this issue, add the following `PackageReference`:

```
<PackageReference Include="Microsoft.NETCore.Targets" Version="3.0.0" PrivateAssets="all" />
```

Example 3

```
Detected package downgrade: Microsoft.NETCore.App from 2.1.8 to 2.1.0. Reference the package directly from the project to select a different version.
```

```
test -> mvc -> Microsoft.NETCore.App (>= 2.1.8)
```

```
test -> Microsoft.NETCore.App (>= 2.1.0)
```

Issue

The mvc project specified a version constraint on a higher version of a package than restore ultimately resolved. That is because of the [nearest-wins](#) rule - when resolving packages, the version of the nearer package in the graph will override that of the distant package with the same ID.

Solution

This specific error (with Microsoft.NETCore.App package) is improved my moving your .NET Core SDK to 2.2.100 or later. Microsoft.NETCore.App is an auto-referenced package that the .NET Core SDK before version 3.0.100 chooses to bring in automatically. Also see [Self-contained deployment runtime roll forward](#).

NuGet Warning NU1608

9/4/2018 • 2 minutes to read • [Edit Online](#)

Detected package version outside of dependency constraint: 'PackageA' 1.0.0 requires 'PackageB' (= 1.0.0) but version 'PackageB' 2.0.0 was resolved.

Issue

A resolved package is higher than a dependency constraint allows. This means that a package referenced directly by a project overrides dependency constraints from other packages.

Solution

In some cases this is intentional and the warning can be suppressed. Otherwise, change the project's reference to the package to widen its version constraints.

NuGet Warning NU1701

9/4/2018 • 2 minutes to read • [Edit Online](#)

Package 'packageId' was restored using 'TargetFrameworkA' instead the project target framework 'TargetFrameworkB'. This package may not be fully compatible with your project.

Issue

`PackageTargetFallback` / `AssetTargetFallback` was used to select assets from a package. The warning let users know that the assets may not be 100% compatible.

Solution

Change the project's target framework to one that the package supports.

NuGet Warning NU1801

9/4/2018 • 2 minutes to read • [Edit Online](#)

Issue

An error occurred when reading the feed when `IgnoreFailedSources` is set to true, converting it to a non-fatal warning. This could contain any message and is generic.

Solution

Edit your configuration to specify valid sources.

NuGet Warning NU3000

7/24/2019 • 2 minutes to read • [Edit Online](#)

TIP

Use the `verify` command to view package signatures and timestamps.

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature does not have a timestamp.
```

Issue

The package has a primary signature which does not have a timestamp.

Solution

To enable long-term signature validity after the signature certificate has expired, please ensure that the package signature is timestamped.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': Multiple timestamps are not accepted.
```

Issue

The package has a signature with multiple timestamps.

Solution

Please ensure that each package signature contains no more than 1 timestamp.

Scenario 3

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The repository countersignature does not have a timestamp.
```

Issue

The package has a repository counter signature which does not have a timestamp.

Solution

For long term signature validity, please ensure that any package signature has a timestamp.

NuGet Error NU3001

9/4/2018 • 2 minutes to read • [Edit Online](#)

Scenario 1

Invalid password was provided for the certificate file 'certificate.pfx'. Please provide a valid password using the '-CertificatePassword' option.

Issue

A password protected certificate file was provided to the NuGet Sign operation. But an invalid or no password was provided.

Solution

If you are using a password protected certificate file to sign a NuGet package, then please use the `-CertificatePassword` option to pass the correct password.

Scenario 2

Certificate file 'certificate.pfx' not found. For a list of accepted ways to provide a certificate, please visit <https://docs.nuget.org/docs/reference/command-line-reference>.

Issue

A certificate file was provided to the NuGet Sign operation. But the file does not exist on disk.

Solution

Please ensure that any certificate file being used to sign a NuGet package exists on disk.

Scenario 3

Certificate file 'random_file.txt' is invalid. For a list of accepted ways to provide a certificate, please visit <https://docs.nuget.org/docs/reference/command-line-reference>.

Issue

A certificate file was provided to the NuGet Sign operation, but the file is not a valid certificate file.

Solution

Please ensure that any certificate file being used to sign a NuGet package is a valid certificate file.

Scenario 4

Multiple certificates were found that meet all the given criteria. Use the '`-CertificateFingerprint`' option with the hash of the desired certificate.

Issue

A certificate was suggested to the NuGet Sign command using the `-CertificateSubjectName` option. But multiple certificates were found to match the certificate subject name in the windows certificate store.

Solution

Please pass the '-CertificateFingerprint' option with the hash of the desired certificate to the NuGet Sign command to uniquely identify a certificate.

Scenario 5

No certificates were found that meet all the given criteria. For a list of accepted ways to provide a certificate, please visit <https://docs.nuget.org/docs/reference/command-line-reference>.

Issue

A certificate was suggested to the NuGet Sign command using the `-CertificateSubjectName` option. But no certificate was found to match the certificate subject name in the windows certificate store.

Solution

Please ensure that you passing the right subject name filter, otherwise pass the '-CertificateFingerprint' option with the hash of the desired certificate to the NuGet Sign command to uniquely identify a certificate.

Scenario 6

The following certificate cannot be used for package signing as the private key provider is unsupported:

Subject Name: CN=Certificate Subject Name
SHA1 hash: HASH
SHA256 hash: HASH
Issued by: Issuer Subject Name
Valid from: 4/9/2016 5:00:00 PM to 4/14/2020 5:00:00 AM

Issue

A certificate was passed to the NuGet Sign command which has an unsupported private key provider.

Solution

Currently, due to framework limitations, NuGet sign command does not support CNG key private key provider. Please use a certificate with a CAPI private key provider.

Scenario 7

The package already contains a signature. Please remove the existing signature before adding a new signature.

Issue

NuGet Sign command was used to sign a package which already has a package signature.

Solution

Please ensure that you are signing an unsigned package. If the package is already signed, then please use `-Overwrite` option to overwrite an existing signature.

NuGet Warning NU3002

4/11/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The '-Timestamper' option was not provided. The signed package will not be timestamped. To learn more about this option, please visit <https://docs.nuget.org/docs/reference/command-line-reference>.

Issue

The `-Timestamper` option was not provided to NuGet Sign command.

Solution

To enable long-term signature validity after the signature certificate has expired, please ensure that the package signature is timestamped.

NuGet Warning NU3003

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package is not signed. Unable to verify signature from an unsigned package.
```

Issue

NuGet client tried to verify a package which does not contain a package signature.

Solution

Please file an issue at [NuGet/Home](#) along with the package that generated this problem.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package signature is invalid or cannot be verified on this platform.
```

Issue

NuGet client tried to verify a package which contains an invalid package signature or a signature that cannot be verified on the current platform.

Solution

Please file an issue at [NuGet/Home](#) along with the package that generated this problem and the platform on which the issue was encountered.

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), NU3003 is raised as a warning. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3003 is elevated from a warning to an error.

NuGet Error NU3004

4/11/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package is not signed.

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': signatureValidationMode is set to require, so packages are allowed only if signed by trusted signers; however, this package is unsigned.

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': This repository indicated that all its packages are repository signed; however, this package is unsigned.

Issue

If from `nuget verify -signatures`

NuGet client tried to verify an unsigned package.

If from restore or install when specifying `signatureValidationMode` **to** `require`

The `require` validation mode does not support unsigned package and an unsigned package is trying to be installed.

Solution

Please ensure that any package intended to be installed or passed to `nuget verify -signatures` command contains a package signature.

NuGet Error NU3005

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package contains an invalid package signature file.
```

Issue

NuGet client tried to verify a package with a signature file which has an invalid Local File Header.

Solution

Please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#). If the problem persists, then please file an issue at [NuGet/Home](#) along with the package that generated this problem.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package contains multiple package signature files.
```

Issue

NuGet client tried to verify a package which contains multiple signature files.

Solution

Please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#). If the problem persists, then please file an issue at [NuGet/Home](#) along with the package that generated this problem.

Scenario 3

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package does not contain a valid package signature file.
```

Issue

NuGet client tried to verify a package which does not contain a package signature file.

Solution

Please file an issue at [NuGet/Home](#) along with the package that generated this problem.

NuGet Warning NU3006

4/11/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': Signed Zip64 packages are not supported.

Issue

NuGet client tried to verify a Zip64 package.

Solution

NuGet client does not support Zip64 signed packages. Please ensure that any package being verified is not a Zip64 package. You can read more about Zip64 in the [PKWARE Zip Specification](#).

NuGet Warning NU3007

4/11/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package signature format version is not supported. Updating your client may solve this problem.

Issue

NuGet client tried to verify a package which has a higher package signature version than the one supported by this client.

Solution

Please ensure that you are using the latest version of the NuGet client. You can read more about the releases on [NuGet.org](#).

NuGet Error NU3008

4/11/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package integrity check failed.

Issue

NuGet package being verified has changed since it was signed.

Solution

Please ensure that the package has not been tampered with since signing. If this was a temporary problem, then you can fix this by clearing your local cache(s) by running `nuget locals -Clear all` command. However, if the problem persists then please inform the package source and the package author.

If this issue happened on a package which came from [nuget.org](#) then please file an issue at [NuGet/Home](#) along with the package that caused this problem.

NuGet Warning NU3009

11/5/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package signature file does not contain exactly one primary signature.

Issue

NuGet client tried to verify a package signature which contained multiple `SignerInfo` fields.

Solution

Please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#). If the problem persists, then please file an issue at [NuGet/Home](#) along with the package that generated this problem.

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), NU3009 is raised as a warning. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3009 is elevated from a warning to an error.

NuGet Warning NU3010

11/5/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature does not have a signing certificate.

Issue

NuGet client tried to verify a package signature with a `SignerInfo` entry that does not contain a signing certificate.

Solution

Please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#). If the problem persists, then please file an issue at [NuGet/Home](#) along with the package that generated this problem.

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), NU3010 is raised as a warning. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3010 is elevated from a warning to an error.

NuGet Warning NU3011

11/5/2019 • 2 minutes to read • [Edit Online](#)

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature is invalid.
```

Issue

NuGet client was unable to read the certificate chain for the signing certificate used to sign the package.

Solution

Please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#). If the problem persists, then please file an issue at [NuGet/Home](#) along with the package that generated this problem.

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), NU3011 is raised as a warning. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3011 is elevated from a warning to an error.

NuGet Warning NU3012

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature validation failed.
```

Issue

NuGet client failed to verify the `SignedCms` signature present in the NuGet signature in the package.

Solution

You can get more details about the problem by looking at the debug logs. If the problem persists, then please file an issue at [NuGet/Home](#) along with the package that generated this problem.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature found a chain building issue: A certificate chain processed, but terminated in a root certificate which is not trusted by the trust provider.
```

Issue

NuGet client failed to verify the certificate chain for the signing certificate used to sign the package.

Solution

Please ensure that the package signature has a valid certificate chain. You can verify the package signature by running the `nuget verify -signatures` command on the package. If the problem persists, then please file an issue at [NuGet/Home](#) along with the package that generated this problem.

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), NU3012 is raised as a warning in most cases. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3012 is elevated from a warning to an error.

NuGet Warning NU3013

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The signing certificate has an unsupported signature algorithm.
```

Issue

The certificate used to sign the package has an unsupported signature algorithm.

Solution

Please ensure that the signing certificate has one of the following signature algorithms -

- `sha256WithRSAEncryption`
- `sha384WithRSAEncryption`
- `sha512WithRSAEncryption`

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature's certificate has an unsupported signature algorithm.
```

Issue

The certificate used to sign the package has an unsupported signature algorithm.

Solution

Please ensure that the package was signed using a certificate with one of the following signature algorithms -

- `sha256WithRSAEncryption`
- `sha384WithRSAEncryption`
- `sha512WithRSAEncryption`

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), NU3013 is raised as a warning. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3013 is elevated from a warning to an error.

NuGet Warning NU3014

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The signing certificate does not meet a minimum public key length requirement.
```

Issue

The certificate used to sign the package does not meet the minimum public key length requirement.

Solution

Please ensure that the signing certificate has an RSA public key of length \geq 2048 bits.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature's certificate does not meet a minimum public key length requirement.
```

Issue

The certificate used to sign the package does not meet the minimum public key length requirement.

Solution

Please ensure that the package was signed using a signing certificate with an RSA public key of length \geq 2048 bits.

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), NU3014 is raised as a warning. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3014 is elevated from a warning to an error.

NuGet Warning NU3015

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The lifetime signing EKU in the primary signature's certificate is not supported.
```

Issue

The certificate used to sign the package has an unsupported Extended Key Usage.

Solution

Please ensure that the signing certificate does not have lifetime signing Extended Key Usage.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The lifetime signing EKU in the signing certificate is not supported.
```

Issue

The certificate used to sign the package has an unsupported Extended Key Usage.

Solution

Please ensure that the package was signed using a signing certificate that does not have lifetime signing Extended Key Usage.

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), NU3015 is raised as a warning. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3015 is elevated from a warning to an error.

NuGet Warning NU3016

11/5/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package hash uses an unsupported hash algorithm.

Issue

NuGet client tried to verify a package which was signed using an unsupported hash algorithm.

Solution

Please ensure that the package was signed with one of the following hash algorithms -

- `sha256`
- `sha384`
- `sha512`

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), NU3016 is raised as a warning. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3016 is elevated from a warning to an error.

NuGet Warning NU3017

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The signing certificate is not yet valid.
```

Issue

The certificate used to sign the package has a validity in the future, but is not valid currently.

Solution

Please ensure that the signing certificate is currently valid.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature's certificate is not yet valid.
```

Issue

The certificate used to sign the package has a validity in the future, but is not valid currently.

Solution

Please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#) with a signing certificate which is currently valid.

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), NU3017 is raised as a warning. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3017 is elevated from a warning to an error.

NuGet Warning NU3018

11/5/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature found a chain building issue: A certificate chain processed, but terminated in a root certificate which is not trusted by the trust provider.

Issue

NuGet client failed to verify the certificate chain for the signing certificate used to sign the package.

Solution

Please ensure that the package signature has a valid certificate chain. You can verify the package signature by running the `nuget verify -signatures` command on the package. If the problem persists, then please file an issue at [NuGet/Home](#) along with the package that generated this problem.

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), NU3018 is raised as a warning. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3018 is elevated from a warning to an error in most cases.

NuGet Warning NU3019

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The timestamp integrity check failed.
```

Issue

The timestamp on the package signature has changed since it was generated by the Timestamp Authority.

Solution

Please try to re-sign and timestamp the package. If the problem persists, contact the Timestamp Authority to discover the source of the problem.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature's timestamp integrity check failed.
```

Issue

The timestamp on the package signature has changed since it was generated by the Timestamp Authority.

Solution

Please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#). If the problem persists, request the package author to contact the Timestamp Authority to discover the source of the problem.

NOTE

When running the `nuget verify -signatures` command, NU3019 is raised as an error. Otherwise, NU3019 is raised as a warning.

NuGet Warning NU3020

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The timestamp does not have a signing certificate.
```

Issue

The timestamp on the package signature does not contain a signing certificate.

Solution

Please try to re-sign and timestamp the package. If the problem persists, contact the Timestamp Authority to discover the source of the problem.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature's timestamp does not have a signing certificate.
```

Issue

The timestamp on the package signature does not contain a signing certificate.

Solution

Please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#). If the problem persists, request the package author to contact the Timestamp Authority to discover the source of the problem.

NOTE

When running the `nuget verify -signatures` command, NU3020 is raised as an error. Otherwise, NU3020 is raised as a warning.

NuGet Warning NU3021

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature's timestamp signature validation failed.
```

Issue

NuGet client failed to verify the `SignedCms` object inside the timestamp on the package signature.

Solution

Please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#). If the problem persists, request the package author to contact the Timestamp Authority to discover the source of the problem.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The timestamp signature validation failed.
```

Issue

The `SignedCms` object inside the timestamp on the package signature could not be verified.

Solution

Please try to re-sign and timestamp the package. If the problem persists, contact the Timestamp Authority to discover the source of the problem.

NOTE

When running the `nuget verify -signatures` command, NU3021 is raised as an error. Otherwise, NU3021 is raised as a warning.

NuGet Warning NU3022

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature's timestamp certificate has an unsupported signature algorithm.
```

Issue

The certificate used to timestamp the package signature has an unsupported signature algorithm.

Solution

Please ensure that the timestamp authority's signing certificate has one of the following signature algorithms -

- `sha256WithRSAEncryption`
- `sha384WithRSAEncryption`
- `sha512WithRSAEncryption`

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The timestamp certificate has an unsupported signature algorithm (SHA1). The following algorithms are supported: SHA256RSA, SHA384RSA, SHA512RSA.
```

Issue

The certificate used to timestamp the package signature has an unsupported signature algorithm.

Solution

Please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#) using the `-Timestamper` option such that the timestamp authority signing certificate has one of the following signature algorithms -

- `sha256WithRSAEncryption`
- `sha384WithRSAEncryption`
- `sha512WithRSAEncryption`

NOTE

When running the `nuget verify -signatures` command, NU3022 is raised as an error. Otherwise, NU3022 is raised as a warning.

NuGet Warning NU3023

4/11/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The timestamp certificate does not meet a minimum public key length requirement.
```

Issue

The certificate used to timestamp the package signature does not meet a minimum public key length requirement.

Solution

Please ensure that the Timestamp Authority's signing certificate has an RSA public key of length \geq 2048 bits.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature's timestamp certificate does not meet a minimum public key length requirement.
```

Issue

The certificate used to timestamp the package signature does not meet a minimum public key length requirement.

Solution

Please ensure that the package signature was timestamped using a signing certificate with an RSA public key of length \geq 2048 bits.

NOTE

When running the `nuget verify -signatures` command, NU3023 is raised as an error. Otherwise, NU3023 is raised as a warning.

NuGet Warning NU3024

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The timestamp signature has an unsupported digest algorithm. The following algorithms are supported: : SHA-2-256, SHA-2-384, SHA-2-512.
```

Issue

The timestamp's signature has an unsupported digest algorithm.

Solution

Ensure that the timestamp authority's signature has one of the following digest algorithms -

- `SHA-2-256`
- `SHA-2-384`
- `SHA-2-512`

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature's timestamp signature has an unsupported digest algorithm.
```

Issue

The timestamp's signature has an unsupported digest algorithm.

Solution

Request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#) using the `-Timestamper` option such that the timestamp authority signing certificate has one of the following digest algorithms -

- `SHA-2-256`
- `SHA-2-384`
- `SHA-2-512`

NOTE

When running the `nuget verify -signatures` command, NU3024 is raised as an error. Otherwise, NU3024 is raised as a warning.

NuGet Warning NU3025

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The timestamp signing certificate is not yet valid.
```

Issue

The certificate used to timestamp the package signature has a validity in the future, but is not valid currently.

Solution

Please ensure that the Timestamp Authority's signing certificate is currently valid. If the problem persists, contact the Timestamp Authority to discover the source of the problem.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature's timestamp signing certificate is not yet valid.
```

Issue

The certificate used to sign the package has a validity in the future, but is not valid currently.

Solution

Please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#). If the problem persists, request the package author to contact the Timestamp Authority to discover the source of the problem.

NOTE

When running the `nuget verify -signatures` command, NU3025 is raised as an error. Otherwise, NU3025 is raised as a warning.

NuGet Warning NU3026

4/11/2019 • 2 minutes to read • [Edit Online](#)

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The timestamp response is invalid. Nonces did not match.
```

Issue

The timestamp authority did not return an expected nonce in its response; therefore, its response is invalid.

Solution

Try to re-sign and timestamp the package. If the problem persists, contact the timestamp authority to discover the source of the problem.

NuGet Warning NU3027

11/5/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The signature should be timestamped to enable long-term signature validity after the certificate has expired.

Issue

The package signature does not contain a timestamp.

Solution

For long term signature validity, please request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#) using the `-Timestamper` option.

NuGet Warning NU3028

11/5/2019 • 2 minutes to read • [Edit Online](#)

NuGet 4.6.0+

The author primary signature's timestamp found a chain building issue: The revocation function was unable to check revocation because the revocation server could not be reached. For more information, visit <https://aka.ms/certificateRevocationMode>

Issue

Certificate chain building failed for the timestamp signature. The timestamp signing certificate is untrusted, revoked, or revocation information for the certificate is unavailable.

Solution

Use a trusted and valid certificate. Check internet connectivity.gits

Revocation check mode (4.8.1+)

If the machine has restricted internet access (such as a build machine in a CI/CD scenario), installing/restoring a signed nuget package will result in this warning since the revocation servers are not reachable. This is expected. However, in some cases, this may have unintended consequences such as the package install/restore taking longer than usual. If that happens, you can work around it by setting the `NUGET_CERT_REVOCATION_MODE` environment variable to `offline`. This will force NuGet to check the revocation status of the certificate only against the cached certificate revocation list, and NuGet will not attempt to reach revocation servers.

WARNING

It is not recommended to switch the revocation check mode to `offline` under normal circumstances. Doing so will cause NuGet to skip online revocation check and perform only an offline revocation check against the cached certificate revocation list which may be out of date. This means packages where the signing certificate may have been revoked, will continue to be installed/restored, which otherwise would have failed revocation check and would not have been installed.

When the revocation check mode is set to `offline`, the warning will be downgraded to an info.

The author primary signature's timestamp found a chain building issue: The revocation function was unable to check revocation because the certificate is not available in the cached certificate revocation list and `NUGET_CERT_REVOCATION_MODE` environment variable has been set to `offline`. For more information, visit <https://aka.ms/certificateRevocationMode>.

NOTE

NU3028 is raised as an error in most cases. When NuGet's [signature validation mode](#) is set to accept (default), NU3028 is raised as a warning in some cases.

NuGet Warning NU3029

4/11/2019 • 2 minutes to read • [Edit Online](#)

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The timestamp signature is invalid.
```

Issue

The timestamp signature is invalid.

Solution

Try to re-sign and timestamp the package with a valid timestamp. If the problem persists, contact the timestamp authority to discover the source of the problem.

NuGet Warning NU3030

11/5/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary signature's timestamp's message imprint uses an unsupported hash algorithm.

Issue

The primary signature's timestamp's message imprint uses an unsupported hash algorithm.

Solution

Request the package author to re-sign the package using the `nuget sign` command as described in [NuGet docs](#) using the `-Timestamper` option such that the timestamp's message imprint uses one of the following hash algorithms -

- `SHA-2-256`
- `SHA-2-384`
- `SHA-2-512`

NOTE

When running the `nuget verify -signatures` command, NU3030 is raised as an error. Otherwise, NU3030 is raised as a warning.

NuGet Warning NU3031

4/11/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The repository countersignature is invalid.

Issue

The repository countersignature is invalid.

Solution

Contact the repository that countersigned the package.

NuGet Warning NU3032

4/11/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package already contains a repository countersignature. Please remove the existing signature before adding a new repository countersignature.

Issue

A signed package must not contain more than 1 repository countersignature. The package already contained a repository countersignature when attempting to add a new repository countersignature.

Solution

Remove the existing signature before adding a new repository countersignature.

Scenario 2

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package signature contains multiple repository countersignatures.

Issue

The package signature contains multiple repository countersignatures.

Solution

Contact the repository that countersigned the package.

NuGet Warning NU3033

4/11/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': A repository primary signature must not have a repository countersignature.
```

Issue

The primary signature should be either an author signature or a repository signature. A repository primary signature cannot have a repository countersignature.

Solution

Try to re-sign the package with either an author signature or a repository signature as the primary signature, then re-countersign the package.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': A repository primary signature must not have a repository countersignature.
```

Issue

The primary signature should be either an author signature or a repository signature. A repository primary signature cannot have a repository countersignature.

Solution

Contact the repository that countersigned the package.

NuGet Error NU3034

4/11/2019 • 2 minutes to read • [Edit Online](#)

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json':
signatureValidationMode is set to require, so packages are allowed only if signed by trusted
signers; however, no trusted signers were specified.

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The package signature
certificate fingerprint does not match any certificate fingerprint in the allow list.

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': This repository
indicated that all its packages are repository signed; however, it listed no signing certificates.

Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': This package was not
repository signed with a certificate listed by this repository.

Issue

There is a missing allow list, or the package signer does not match any signer in the list. This list could either be sent by the repository or specified in the `trustedSigners` section of the `nuget.config`.

Solution

If in `require` mode, only packages signed by a trusted signer will pass validation. Otherwise, contact the repository where this was downloaded from to let them know they have a package that does not comply with the repository signing security guidelines.

NuGet Warning NU3035

4/11/2019 • 2 minutes to read • [Edit Online](#)

Issue

A complete certificate chain could not be built for the repository countersignature's signing certificate.

Solution

Contact the timestamp authority to discover the source of the problem.

NuGet Warning NU3036

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The timestamp's
generalized time is outside the timestamping certificate's validity period.
```

Issue

The timestamp's generalized time is outside the timestamping certificate's validity period.

Solution

Try to re-sign and timestamp the package. If the problem persists, contact the timestamp authority to discover the source of the problem.

Scenario 2

```
Package 'SamplePackage v1.0.0' from source 'https://contoso.com/index.json': The primary
signature's timestamp's generalized time is outside the timestamping certificate's validity period.
```

Issue

The certificate used to timestamp the package signature is invalid as the timestamp's generalized time is outside the timestamping certificate's validity period.

Solution

Request the package author to re-sign and timestamp the package using the `nuget sign` command as described in [NuGet docs](#). If the problem persists, request the package author to contact the timestamp authority to discover the source of the problem.

NOTE

When running the `nuget verify -signatures` command, NU3036 is raised as an error. Otherwise, NU3036 is raised as a warning.

NuGet Warning NU3037

11/5/2019 • 2 minutes to read • [Edit Online](#)

Issue

A NuGet package signature has expired. A package signature shares the same validity period as the certificate used to generate the signature. A package signature is invalid outside of that validity period. To ensure long-term validity --- even beyond the signing certificate's validity period --- a package signature should be timestamped with a trusted timestamp. Trusted timestamps must be added while a package signature is still valid and not expired.

Solution

- Resign the package with a non-expired certificate. Optionally, add a trusted timestamp at the time of signing to ensure long-term validity of the signature.
- For accept mode only, ignore the warning.

NOTE

When NuGet's [signature validation mode](#) is set to accept (default), a package with an expired package signature is treated as an unsigned package and installed anyway. NU3037 is raised as a warning. When NuGet's signature validation mode is set to require, or when running the `nuget verify -signatures` command, NU3037 is elevated from a warning to an error.

NuGet Warning NU3038

4/11/2019 • 2 minutes to read • [Edit Online](#)

Issue

Verification settings require a repository countersignature, but the package does not have a repository countersignature.

Solution

Use a package source that has the repository countersigned package.

NuGet Warning NU3040

11/26/2018 • 2 minutes to read • [Edit Online](#)

There are two certificates with conflicting `allowUntrustedRoot` attributes in the computed settings. The `allowUntrustedRoot` attribute is going to be set to `false`. Certificate: SHA256-3F9001EA83C560D712C24CF213C3D312CB3BFF51EE89435D3430BD06B5D0EECE

Issue

There are conflicting attributes in a certificate item in `nuget.config`. Two certificate items share the same `fingerprint` and `hashAlgorithm`, but have different `allowUntrustedRoot`.

Solution

NuGet will take the most restrictive of those settings (`allowUntrustedRoot=false`), to remove the warning make sure to either deduplicate the certificate items or set `allowUntrustedRoot` to the same value on both.

NuGet Error NU5000

9/4/2018 • 2 minutes to read • [Edit Online](#)

Issue

The NuGet pack operation had a problem. NU5000 is used when we haven't yet assigned a unique error code for that issue. So we can improve, please feel free to file the issue with details of your error.

Solution

Check the output window (in Visual Studio) or console output (with NuGet or dotnet command line tools) for more information.

NuGet Error NU5001

9/4/2018 • 2 minutes to read • [Edit Online](#)

Unable to output resolved nuspec file because it would overwrite the original at 'F:\project\project.nuspec'.

Issue

The NuGet pack operation was invoked with the `-InstallPackageToOutputPath` option, but the output path already contained a nuspec file.

Solution

Please ensure that the output path passed to the NuGet pack command does not already contain a nuspec file.

NuGet Error NU5002

9/4/2018 • 2 minutes to read • [Edit Online](#)

Please specify a nuspec, project.json, or project file to use.

Issue

No input file was provided for the NuGet pack operation.

Solution

Please use the path to a nuspec, project.json or project file to the pack command as an argument.

NuGet Error NU5003

9/4/2018 • 2 minutes to read • [Edit Online](#)

Failed to build package because of an unsupported targetFramework value on 'System.Net'.

Issue

A framework assembly specified in the nuspec file does not contain a valid framework.

Solution

Please fix the target framework specified in the referenced assembly.

NuGet Error NU5004

9/4/2018 • 2 minutes to read • [Edit Online](#)

Failed to build package. Ensure 'F:\project\project.nuspec' includes assembly files. For help on building symbols package, visit <http://docs.nuget.org/>.

Issue

The NuGet pack operation was performed using `-Symbol` option, but the folder containing the nuspec file contained no assembly files.

Solution

When building a symbols package, ensure that the folder containing the nuspec file contains assembly files for which the symbols need to be packaged.

NuGet Error NU5005

9/4/2018 • 2 minutes to read • [Edit Online](#)

Ensure 'F:\project\project.nuspec' includes source and symbol files. For help on building symbols package, visit <http://docs.nuget.org/>.

Issue

The NuGet pack operation was invoked with the `-Symbols` option, but the nuspec file does not contain any source or symbols files.

Solution

When building a symbols package, ensure that the folder containing the nuspec file contains assembly files for which the symbols need to be packaged.

NuGet Error NU5007

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
No build found in F:\project\bin\Debug\net461\project.dll. Use the -Build option or build the project.
```

Issue

The NuGet pack operation was invoked with the `-Symbols` option, but the project being packaged has not been built yet and hence cannot be packed.

Solution

Please build the project before running NuGet pack operation or use the `-Build` option to build the project before packaging.

NuGet Error NU5008

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
Manifest file not found at 'F:\project\project.nuspec'
```

Issue

The nuspec file passed to NuGet pack operation does not exist.

Solution

Please ensure that the nuspec file passed to the NuGet pack command exists on disk.

NuGet Error NU5009

9/4/2018 • 2 minutes to read • [Edit Online](#)

Cannot find version of msbuild.

Issue

NuGet pack operation cannot find msbuild.

Solution

Please pass the `-MsBuildPath` to NuGet pack command.

NuGet Error NU5010

9/4/2018 • 2 minutes to read • [Edit Online](#)

Version string specified for package reference '9.9.9.9.9' is invalid.

Issue

The version string given to NuGet pack operation is not a valid string.

Solution

Please ensure that the version string passed to NuGet pack operation is a valid SemVer2 string.

NuGet Error NU5011

9/4/2018 • 2 minutes to read • [Edit Online](#)

Unable to extract metadata from 'project.csproj'.

Issue

A runtime exception occurred while extracting metadata from an assembly or a project file.

Solution

Please create an issue at <https://github.com/NuGet/Home/issues>.

NuGet Error NU5012

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
Unable to find 'F:\project\bin\Debug\net461\project.dll'. Make sure the project has been built.
```

Issue

The project being packed has not been built yet and hence cannot be packaged.

Solution

Please build the project before running NuGet pack operation or use the `-Build` option to build the project before packaging.

NuGet Error NU5013

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
Failed to build 'project.csproj'
```

Issue

The project failed to build while running NuGet pack operation with the `-Build` option.

Solution

Please fix the build error in the project and try again. The reason for failure should be displayed in the console logs.

NuGet Error NU5014

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
Error occurred when processing file 'F:\project2\project2.csproj': The 'id' start tag on line 4
position 10 does not match the end tag of 'ids'. Line 4, position 20.
```

Issue

The NuGet pack operation was run with the `-IncludeReferencedProjects` option and an error occurred while reading the nuspec file with a referenced project.

Solution

Please fix the nuspec error as per the details listed in the error message.

NuGet Error NU5015

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
project.json cannot contain multiple Target Frameworks.
```

Issue

The project.json file passed to NuGet pack operation contains multiple target frameworks.

Solution

Please fix the project.json file to contain only a single target framework.

NuGet Error NU5016

9/4/2018 • 2 minutes to read • [Edit Online](#)

Package version constraints for 'NuGet.Versioning' return a version range that is empty.

Issue

The project has a package dependency which does not have a valid version.

Solution

Please fix the dependency version to be semver compliant.

NuGet Error NU5017

9/4/2018 • 2 minutes to read • [Edit Online](#)

Cannot create a package that has no dependencies nor content.

Issue

The package being saved does not contain any package references, assemblies or framework references.

Solution

Please ensure that the package being created contains either assemblies or package references.

NuGet Error NU5018

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
Invalid assembly reference 'xunit.dll'. Ensure that a file named 'xunit.dll' exists in the lib directory.
```

Issue

The nuspec file used to create the NuGet pack operation contained `references` that do not exist in the `.\lib` folder in the project directory.

Solution

Please ensure that any referenced assemblies are placed inside `.\lib` folder which is in the same folder as the nuspec file used to create the package.

NuGet Error NU5019

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
File not found: 'bad_file.path'
```

Issue

The nuspec file used to create the NuGet pack operation contained `files` that do not exist.

Solution

Please ensure that any `file` entries in the `files` element in the nuspec file is available at the path specified as the `src`.

NuGet Error NU5020

11/28/2018 • 2 minutes to read • [Edit Online](#)

A source file was added with an empty path.

Issue

A source file was passed to `msbuild -t:pack` operation with an empty path.

Solution

Please ensure that all source files passed to `msbuild -t:pack` operation have a valid path and exist on the disk.

NuGet Error NU5021

9/4/2018 • 2 minutes to read • [Edit Online](#)

The project directory for the source file 'src/Project/Code.cs' could not be found.

Issue

The NuGet pack operation was invoked with the `-Symbols` option, but the project directory for a source file was not found.

Solution

Please ensure that the source files are present on idsk. Otherwise please file an issue at [NuGet/Home](#)

NuGet Error NU5022

11/28/2018 • 2 minutes to read • [Edit Online](#)

```
MinClient Version string specified '9.9.9.9' is invalid.
```

Issue

The `MinClientVersion` property passed to msbuild -t:pack operation is not a valid version string.

Solution

Please fix the version string passed as `MinClientVersion` to be a valid SemVer version.

NuGet Error NU5023

11/28/2018 • 2 minutes to read • [Edit Online](#)

The assets file produced by restore does not exist. Try restoring the project again. The expected location of the assets file is F:\project\obj\project.assets.json.

Issue

The project passed to the `msbuild -t:pack` does not contain an assets file in the obj directory.

Solution

Please run `msbuild -t:restore` operation on the project being packaged before running the pack operation.

NuGet Error NU5024

11/28/2018 • 2 minutes to read • [Edit Online](#)

```
PackageVersion string specified '9.9.9.9.9' is invalid.
```

Issue

The `PackageVersion` property passed to msbuild -t:pack operation is not a valid version string.

Solution

Please fix the version string passed as `PackageVersion` to be a valid SemVer version.

NuGet Error NU5025

11/28/2018 • 2 minutes to read • [Edit Online](#)

The assets file found does not contain a valid package spec. Try restoring the project again. The location of the assets file is F:\project\obj\project.assets.json.

Issue

The project passed to the `msbuild -t:pack` does not contain a valid assets file in the obj directory.

Solution

Please run `msbuild -t:restore -p:restoreforce=true` operation on the project being packaged before running the pack operation.

NuGet Error NU5026

8/17/2019 • 2 minutes to read • [Edit Online](#)

```
The file ''F:\project\bin\Debug\net461\project.exe' to be packed was not found on disk.
```

Issue

The project being packed has not been built yet and hence cannot be packed.

Solution

Please build the project before running dotnet pack operation or do not use `--no-build` option to allow dotnet pack to build the project before packaging.

You may have written a project that does not output assemblies. If you intend to ship an assembly-free NuGet package, disable `dotnet pack`'s requirement for an output assembly. You can do this by setting the `IncludeBuildOutput` property to `false` in your project file.

Also see [related mbuild properties](#).

NuGet Error NU5027

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
Invalid target framework for the file 'F:\project\project.dll'.
```

Issue

An assembly being packaged does not contain a valid target framework metadata.

Solution

Please ensure that the assembly being packaged contains a valid `TargetFramework` property.

NuGet Error NU5028

11/28/2018 • 2 minutes to read • [Edit Online](#)

No project was provided to the PackTask.

Issue

No project file was specified to the `msbuild -t:pack` operation.

Solution

Please specify the project to pack operation. You can either specify it in the command (

`msbuild -t:pack project.csproj`) or run `msbuild -t:pack` operation in a folder containing a project file.

NuGet Error NU5029

11/28/2018 • 2 minutes to read • [Edit Online](#)

NuspecProperties should be in the form of 'key1=value1;key2=value2'.

Issue

Properties passed to `msbuild -t:pack -p:NuspecFile=project.nuspec` command were not in the right format to be parsed.

Solution

Please pass any nuspec properties using `-p:NuspecProperties` in the format of `key=value`. For instance

```
msbuild -t:pack -p:NuspecFile=project.nuspec -p:NuspecProperties="configuration=release"
```

NuGet Error NU5030

12/19/2018 • 2 minutes to read • [Edit Online](#)

```
The license file 'LICENSE.txt' does not exist in the package.
```

Issue

The license file is referenced in the metadata with either PackageLicenseFile in the csproj or the license element in the nuspec, but the file itself was not included in the expected location within the package.

Solution

Include the file in the package, for example:

If pack with the targets:

```
<PropertyGroup>
  <PackageLicenseFile>LICENSE.txt</PackageLicenseFile>
</PropertyGroup>

<ItemGroup>
  <None Include="licenses\LICENSE.txt" Pack="true" PackagePath="" />
</ItemGroup>
```

If packing with a nuspec:

```
<package>
  <metadata>
    <license type="file">LICENSE.txt</license>
  </metadata>
  <files>
    <file src="licenses\LICENSE.txt" target="" />
  </files>
</package>
```

NuGet Error NU5031

11/28/2018 • 2 minutes to read • [Edit Online](#)

```
The license file 'LICENSE.txt' has an invalid extension. Valid options are .txt, .md or none.
```

Issue

Only the listed file extensions are allowed.

Solution

Use a file extensions from the allowed list.

NuGet Error NU5032

11/28/2018 • 2 minutes to read • [Edit Online](#)

The license expression 'MIT OR OR Apache-2.0' cannot be parsed successfully. The license expression is invalid.

Issue

The license expression does not conform to the NuGet license expression grammar.

Solution

In this case there are 2 'OR' operators. The operators have to be surrounded with operands. For example:

```
MIT OR Apache-2.0
```

NuGet Error NU5033

11/28/2018 • 2 minutes to read • [Edit Online](#)

Invalid metadata. Cannot specify both a License Expression and a License File.

Issue

The `PackageLicenseExpression` and `PackageLicenseFile` properties are mutually exclusive.

Solution

Use one or the other. If you are using a well known license that's part of the standard license list, consider using an expression. Otherwise please use a license file.

NuGet Error NU5034

12/4/2018 • 2 minutes to read • [Edit Online](#)

The `PackageLicenseExpressionVersion` string '2.0.0-InvalidSystemVersion' is not a valid version.

Issue

The `PackageLicenseExpressionVersion` string is a `System.Version`. That means no labels are allowed.

Solution

Change the property value to `2.0.0`

NuGet Error NU5035

11/28/2018 • 2 minutes to read • [Edit Online](#)

The `PackageLicenseUrl` cannot be used in conjunction with the `PackageLicenseFile` and `PackageLicenseExpression`.

Issue

When using `PackageLicenseFile` or `PackageLicenseExpression`, you should not set the `PackageLicenseUrl` property. The `licenseUrl` metadata will be auto-generated by the client tools to a down-level friendly url.

Solution

Do not set the `PackageLicenseUrl`.

NuGet Error NU5036

1/15/2019 • 2 minutes to read • [Edit Online](#)

This package has an improperly escaped Url in LicenseUrl

Issue

LicenseUrl metadata value in the nuspec file should be properly escaped. Some versions of nuget pack functionality have a problem which will be updated asap.

Solution

Use a fixed version of nuget pack functionality:

- "NuGet.exe pack" - fixed in 4.9.1
- "dotnet.exe pack" - broken in 2.1.500. No fix available yet. (don't use new License expression or file feature yet, to avoid problem.)
- "msbuild /t:pack" - broken in Visual Studio 15.9.1. No fix available yet. (don't use new License expression or file feature yet, to avoid problem.)

NuGet Error NU5037

9/3/2019 • 2 minutes to read • [Edit Online](#)

The package is missing the required nuspec file.

Issue

Restore fails when the [nuspec](#) file is missing from the package on the feed, or from the global packages folder of the package.

Scenario 1

nuspec file is missing from the package (nupkg) file.

Solution

Contact the package author.

Scenario 2

Restore fails for a project managed with the [PackageReference](#) format. For example:

The package is missing the required nuspec file. Path: C:\..\..\nuget\packages\x\1.0.0.'x' represents package name and '1.0.0' represents package version.

Solution

Delete the package folder specified in the error message and run the restore again. For example:

Consider sample error message specified in the above section.
Delete the package folder '1.0.0' from 'C:\..\..\nuget\packages\x' directory and run the restore again.

Scenario 3

Restore fails for a project managed with the [packages.config](#) format.

Solution

Identifying the package with missing nuspec file requires manual probing of dependencies. Delete corrupted package folder or entire solution packages folder if the package with missing nuspec file is unknown and run the restore again.

NuGet Error NU5046

9/5/2019 • 2 minutes to read • [Edit Online](#)

The icon file 'icon.png' does not exist in the package.

Issue

NuGet is unable find the icon file in the package.

Solution

- Make sure that the file that is marked as the package icon exists at the source and it is readable, and the target matches the path expected by the `icon` property.
- Ensure that the file is referenced in the nuspec or in the project file.
 - When creating a package from a MSBuild project file, make sure to reference the icon file in the project, as follows:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    ...
    <PackageIcon>icon.png</PackageIcon>
    ...
  </PropertyGroup>

  <ItemGroup>
    ...
    <None Include="images\icon.png" Pack="true" PackagePath="" />
    ...
  </ItemGroup>
</Project>
```

- When you are creating a package from a nuspec file, make sure to include the icon file in the `<files/>` section:

```
<package>
  <metadata>
    ...
    <icon>images\icon.png</icon>
    ...
  </metadata>
  <files>
    ...
    <file src="..\icon.png" target="images\" />
    ...
  </files>
</package>
```

[Learn more about packaging an icon image file.](#)

NuGet Error NU5047

9/5/2019 • 2 minutes to read • [Edit Online](#)

The icon file size must not exceed 1 megabyte.

Issue

The file that is specified as the package icon is larger than 1 megabyte (MB). NuGet only allows icons whose file size is less than 1 MB.

Solution

Use an image editor program to reduce the size of the package icon file.

NuGet Warning NU5048

9/9/2019 • 2 minutes to read • [Edit Online](#)

The 'PackageIconUrl'/'iconUrl' element is deprecated. Consider using the 'PackageIcon'/'icon' element instead. Learn more at <https://aka.ms/deprecateIconUrl>

Issue

Icon URL is deprecated in favor of embedding the icon inside the NuGet package. Possible causes are:

- When creating a package from a nuspec file, it contains a `<iconUrl/>` entry.
- When creating a package from a MSBuild project file, it contains a `<PackageIconUrl>` property.

Solution

To stop seeing this warning, add an embedded icon to your package.

For MSBuild project files, add an `<PackageIcon/>` property, as follows:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    ...
    <PackageIcon>icon.png</PackageIcon>
    ...
  </PropertyGroup>

  <ItemGroup>
    ...
    <None Include="images\icon.png" Pack="true" PackagePath="" />
    ...
  </ItemGroup>
</Project>
```

For nuspec files, add an `<icon/>` entry that points to the file that will be the package icon:

```
<package>
  <metadata>
    ...
    <icon>images\icon.png</icon>
    ...
  </metadata>
  <files>
    ...
    <file src="..\icon.png" target="images\" />
    ...
  </files>
</package>
```

[Learn more about packaging an icon image file.](#)

NuGet Warning NU5100

8/14/2019 • 2 minutes to read • [Edit Online](#)

The assembly 'bin\Debug\net461\project.dll' is not inside the 'lib' folder and hence it won't be added as a reference when the package is installed into a project. Move it into the 'lib' folder if it needs to be referenced.

Issue

The folder being packaged contains an assembly file that is not in `lib` folder.

Solution

While packaging a folder please ensure that all assembly files are placed inside a framework-specific folder under a `lib` folder. For more information on the folder structure, see [Framework version folder structure](#).

NuGet Warning NU5101

11/5/2019 • 2 minutes to read • [Edit Online](#)

Scenario 1

The assembly 'lib\project.dll' is placed directly under 'lib' folder. It is recommended that assemblies be placed inside a framework-specific folder. Move it into a framework-specific folder.

Issue

The folder being packaged contains an assembly file that is directly under a `lib` folder.

Solution

While packaging a folder please ensure that all assembly files are placed inside a framework-specific folder under a `lib` folder.

Scenario 2

The assembly 'lib\project.dll' will be ignored when the package is installed after the migration.

Issue

The package contains an assembly file that is directly under a `lib` folder. The package will not be compatible with a package reference type of project.

Solution

Please request the package author to fix the package such that all assemblies are placed inside a framework-specific folder under a `lib` folder. You can read more about it in [NuGet docs](#).

NuGet Warning NU5102

9/4/2018 • 2 minutes to read • [Edit Online](#)

The value "http://project_url_here_or_delete_this_line/" for ProjectUrl is a sample value and should be removed. Replace it with an appropriate value or remove it and rebuild your package.

Issue

A default value was used for a Nuspec property.

Solution

Please change the value for the property specified.

NuGet Warning NU5103

9/4/2018 • 2 minutes to read • [Edit Online](#)

The folder 'lib\random_tfm\temp.dll' under 'lib' is not recognized as a valid framework name or a supported culture identifier. Rename it to a valid framework name or culture identifier.

Issue

An assembly was detected to be under an invalid target framework or culture identifier folder under the lib folder.

Solution

Please rename the folder to a valid framework name or culture identifier.

NuGet Warning NU5104

9/4/2018 • 2 minutes to read • [Edit Online](#)

A stable release of a package should not have a prerelease dependency. Either modify the version spec of dependency "NuGet.Versioning [4.7.0-preview4.5065,)" or update the version field in the nuspec.

Issue

The project or nuspec being packaged contains a dependency on a prerelease package.

Solution

If you intend to create a prerelease package then please refer to SemVer2 guidelines and add a prerelease tag to the version property i.e. `<version>1.0.0-pre</version>`. If you intend to create a stable package then please update the specified dependency version to a stable version.

NuGet Warning NU5105

9/24/2019 • 2 minutes to read • [Edit Online](#)

The package version '1.2.3+semver2.metadata' uses SemVer 2.0.0 or components of SemVer 1.0.0 that are not supported on legacy clients. Change the package version to a SemVer 1.0.0 string. If the version contains a release label it must start with a letter. This message can be ignored if the package is not intended for older clients.

Issue

The `version` property specified contains SemVer 2.0.0 components or SemVer 1.0.0 components that may not be supported on NuGet clients older than v4.3

Solution

If you intend for your package to be available to NuGet clients older than v4.3 then please adapt the `version` property to be compatible with the legacy clients. You can read more about the SemVer 2.0.0 support [here](#).

NuGet Warning NU5106

9/4/2018 • 2 minutes to read • [Edit Online](#)

The file at 'lib\WinRT\temp.dll' uses the obsolete 'WinRT' as the framework folder. Replace 'WinRT' or 'WinRT45' with 'NetCore45'.

Issue

A file was detected to be under an `WinRT` or `WinRT45` target framework folder under the lib folder. However, `WinRT` and `WinRT45` target frameworks are now obsolete.

Solution

Please rename the target framework folder to `NetCore45`.

NuGet Warning NU5107

9/4/2018 • 2 minutes to read • [Edit Online](#)

The file 'tools/subfolder/init.ps1' will be ignored by NuGet because it is not directly under 'tools' folder. Place the file directly under 'tools' folder.

Issue

An `init.ps1` file was detected in a folder under the `tools` folder.

Solution

Please place the `init.ps1` file directly under the `tools` folder for it to be compatible with NuGet tooling.

NuGet Warning NU5108

9/4/2018 • 2 minutes to read • [Edit Online](#)

The transform file 'other\code.pp' is outside the 'content' folder and hence will not be transformed during installation of this package. Move it into the 'content' folder.

Issue

An `.pp` or a `.transform` file was detected in a folder other than `content` folder.

Solution

Please place the `.pp` or a `.transform` file under the `content` folder for it to be compatible with NuGet tooling.

NuGet Warning NU5109

9/4/2018 • 2 minutes to read • [Edit Online](#)

The file at 'tools/._.' uses the symbol for empty directory '._.', but it is present in a directory that contains other files. Please remove this file from directories that contain other files.

Issue

A file with file name `._.` was detected in a folder that contains other files.

Solution

The file name '.' is used as a symbol of empty directories. Any file with that file name should be present in an empty directory. Please remove this file from directories that contain other files.

NuGet Warning NU5110

9/4/2018 • 2 minutes to read • [Edit Online](#)

The script file 'other\init.ps1' is outside the 'tools' folder and hence will not be executed during installation of this package. Move it into the 'tools' folder.

Issue

An `.ps1` file was detected in a folder other than `tools` folder.

Solution

Please place the `.ps1` file under the `tools` folder for it to be compatible with NuGet tooling.

NuGet Warning NU5111

9/4/2018 • 2 minutes to read • [Edit Online](#)

The script file 'tools\random.ps1' is not recognized by NuGet and hence will not be executed during installation of this package. Rename it to install.ps1, uninstall.ps1 or init.ps1 and place it directly under 'tools'.

Issue

An unrecognized `.ps1` file was detected being packaged.

Solution

Please rename the file to `init.ps1` and place it directly under `tools` folder for it to be compatible with NuGet tooling.

NuGet Warning NU5112

9/4/2018 • 2 minutes to read • [Edit Online](#)

The version of dependency 'NuGet.Versioning' is not specified. Specify the version of dependency and rebuild your package.

Issue

A dependency was specified in the nuspec file without a version i.e. - `<dependency id="NuGet.Versioning" />`

Solution

Please add a version to all the dependencies specified in the nuspec being packaged.

NuGet Warning NU5114

11/28/2018 • 2 minutes to read • [Edit Online](#)

```
'SolutionDir' key already exists in Properties collection. Overriding value.
```

Issue

A property was specified in the project file and also passed through pack command using `-Properties` or `-p:NuspecProperties` options.

Solution

Please ensure that is a property is defined in the project file then, you do not pass the same property through pack command using `-Properties` or `-p:NuspecProperties` options.

NuGet Warning NU5115

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
Description was not specified. Using 'Description'.
```

Issue

A property was not specified to the pack command, hence a default value was chosen instead.

Solution

Please specify the mentioned property in the csproj as an msbuild property if using project file or in the nuspec file.

NuGet Warning NU5116

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
'Content\sample.txt' was included in the project but doesn't exist. Skipping...
```

Issue

A content file specified in the project being packaged does not exist on the disk and is being skipped.

Solution

Please ensure that all the content file specified in the project exist on disk.

NuGet Warning NU5117

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
'$(MSBuildProjectDirectory)/tools/sample.txt' was included in the project but the path could not be resolved. Skipping...
```

Issue

A file was added in the project file but the file path could not be resolved during the pack operation.

Solution

The pack operation could not resolve a file that was added as an msbuild variable. Please update the path to be a fully resolved path.

NuGet Warning NU5118

9/4/2018 • 2 minutes to read • [Edit Online](#)

```
File 'F :\validation\test\proj\tools\readme.1.txt' is not added because the package already
contains file 'tools\readme.txt'
```

Issue

A file added as `Content` in the project file could not be added to the package being generated because another with the same `PackagePath` was already added to the package.

Solution

Please ensure that any `Content` files being added to the package using the project file have unique `PackagePath` metadata.

NuGet Warning NU5119

9/4/2018 • 2 minutes to read • [Edit Online](#)

File 'F:\project\binary\Libuv.1.10.0.nupkg' was not added to the package. Files and folders starting with '.' or ending with '.nupkg' are excluded by default. To include this file, use `-NoDefaultExcludes` from the commandline

Issue

The NuGet pack operation found a `.nupkg` file or a file/folder starting with `.` to be added to the package.

Solution

If you intent to package a `.nupkg` file or a file/folder starting with `.` then please use `-NoDefaultExcludes` property to allow the packaging of those files/folders.

NuGet Warning NU5120

11/5/2019 • 2 minutes to read • [Edit Online](#)

`install.ps1` script will be ignored when the package is installed after the migration.

Issue

The package contains an `install.ps1` file. The file will not be executed if the package is installed to a package reference type of project.

Solution

Please request the package author to fix the package such that it no longer contains an `install.ps1` file. You can read more about it in [NuGet docs](#).

NuGet Warning NU5121

11/5/2019 • 2 minutes to read • [Edit Online](#)

'content' assets will not be available when the package is installed after the migration.

Issue

The package contains an files under a `Content` folder. These assets will not be available if the package is installed to a package reference type of project.

Solution

Please request the package author to fix the package such that it no longer contains an `Content` folder. You can read more about it in [NuGet docs](#).

NuGet Warning NU5122

11/5/2019 • 2 minutes to read • [Edit Online](#)

XDT transform file 'tools/transform.xdt' will not be applied when the package is installed after the migration.

Issue

The package contains an assembly file that is not under a target framework specific folder. The package will not be compatible with a package reference type of project.

Solution

Please request the package author to fix the package such that it no longer contains an `install.ps1` file. You can read more about it in [NuGet docs](#).

NuGet Warning NU5123

9/4/2018 • 2 minutes to read • [Edit Online](#)

The file 'content//readme.txt' path, name, or both are too long. Your package might not work without long file path support. Please shorten the file path or file name.

Issue

A file was detected to have an installed path of longer than 200 characters. Installed path is defined as the `<package_id>/<version>/target_file_path`, where `target_file_path` is defined in the `target` property of the `<files>` section in the nuspec file.

Solution

Please ensure that the path `<package_id>/<version>/target_file_path` for all the files included in the package are lesser than 200 characters, where `target_file_path` is defined in the `target` property of the `<files>` section in the nuspec file.

NuGet Warning NU5124

11/28/2018 • 2 minutes to read • [Edit Online](#)

The license identifier 'Microsoft-SpecialLicense' is not recognized by the current toolset.

Issue

The NuGet Client tools have a known list of license identifiers. As this knowledge is contained within the product, sometimes version of the tools is not aware of the standardization of the license. Other times, the license simply is not a standardized license, with no clear timeline of ever becoming one.

Solution

Use an updated client that understands the license identifier you are trying to use. Use a license file instead of an expression if there is no timeline for the said license to become an SPDX standard license.

NuGet Warning NU5125

4/18/2019 • 2 minutes to read • [Edit Online](#)

The 'licenseUrl' element will be deprecated. Consider using the 'license' element instead.

Issue

The `licenseUrl` element is being replaced by the `license` element.

Solution

If you create your NuGet package using `dotnet pack` or `msbuild -t:pack`, follow the documentation on [packaging a license expression or a license file using MSBuild targets](#).

If you use a `.nuspec` file, use the `license` element.

NuGet Warning NU5127

9/26/2019 • 2 minutes to read • [Edit Online](#)

This package does not contain a `lib/` or `ref/` folder, and will therefore be treated as compatible for all frameworks. Since framework specific files were found under the `build/` directory for `net45`, `netstandard2.0`, consider creating the following empty files to correctly narrow the compatibility of the package:

```
-lib/net45/_._  
-lib/netstandard2.0/_._
```

Issue

Projects using packages with `PackageReference` only use `lib/` and `ref/` assemblies to determine package compatibility. Therefore, a package without any `lib/` or `ref/` files will be considered compatible with all projects. However, if that package contains build files specific to one or more [Target Framework Monikers \(TFMs\)](#), a package consumer may expect the package to fail if none of the build files are used.

Solution

As the warning message suggests, create an empty file named `_._` in the `lib` folder for the TFMs listed. This will allow NuGet to fail the restore for `PackageReference` projects when the project is incompatible with the package.

NuGet Warning NU5128

9/26/2019 • 4 minutes to read • [Edit Online](#)

Scenario 1

Some target frameworks declared in the dependencies group of the nuspec and the lib/ref folder do not have exact matches in the other location. Consult the list of actions below:

- Add a dependency group for .NETStandard2.0 to the nuspec

Issue

The `lib/<tfm>/` or `ref/<tfm>/` directory in the package contains at least one file for the [Target Framework Moniker \(TFM\)](#) specified in the warning message. However, no dependency group exists for this TFM in the `nuspec` file. This may cause package consumers to believe the package is not compatible with the TFM, even if the package does not have dependencies. If the package has undeclared dependencies, the project using the package will experience runtime errors.

Solution

- Run NuGet's pack target on the project

If possible, use [NuGet's MSBuild pack target](#), as it automatically matches assembly TFM's with dependency groups from the project's target frameworks. Note that your project must use `PackageReference` for its own NuGet dependencies. If your project uses `packages.config`, you need to use `nuget.exe pack` and a `nuspec` file.

- Manually edited `nuspec` file

If you are using a custom `nuspec` file, we recommend each TFM for which `lib/` or `ref/` assemblies exist should have a matching dependency group, even if the dependencies are the same as the next compatible TFM. For example, if a package contains `netstandard1.0` and `netstandard2.0` assemblies, and the dependencies are the same for both, we recommend both TFM's be listed as dependency groups with duplicate dependency items.

Note that the TFM identifier used in the assembly paths use a different format to the TFM identifier used in dependency groups. The warning message specifies the correct name to use in the dependency group. If your package does not have any dependencies for that target framework, use an empty group. For example:

```
<package>
  <metadata>
    ...
    <dependencies>
      <group targetFramework=".NETFramework4.7.2" />
    </dependencies>
  </metadata>
  ...
</package>
```

- Remove the `lib/` or `ref/` files

If you do not wish your package to be compatible with the reported TFM, modify your project such that no `lib/<tfm>/` or `ref/<tfm>/` files are in the package for that TFM. For example, if the warning says to add a dependency group for `.NETFramework4.7.2` to the `nuspec`, then remove any `lib/net472/*` and `ref/net472/*` files from your package.

Scenario 2

Some target frameworks declared in the dependencies group of the nuspec and the lib/ref folder do not have exact matches in the other location. Consult the list of actions below:

- Add lib or ref assemblies for the netstandard2.0 target framework

Issue

The `nuspec` file has a dependency group for the reported Target Framework Moniker (TFM), but no assemblies exist for this TFM in either `lib/` or `ref/`. If there are assemblies for a compatible TFM, the package will still install, but the dependencies might be incorrect for assemblies used at compile time and could cause the project to fail at runtime.

Solution

- Run NuGet's pack target on the project

If possible, use [NuGet's MSBuild pack target](#), as it automatically matches assembly TFM's with dependency groups from the project's target frameworks. Note that your project must use `PackageReference` for its own NuGet dependencies. If your project uses `packages.config`, you need to use `nuget.exe pack` and a `nuspec` file.

- Manually edit the `nuspec` file

Add the reported TFM as an additional Target Framework for which your project compiles for, and add the assemblies to the package. If you are using an SDK style project to multi-target multiple TFM's, NuGet's MSBuild pack targets can automatically add assemblies in the correct `lib/<tfm>/` folder and create dependency groups with the correct TFM's and dependencies. If you are using a non-SDK style project, you will likely need to create an additional project file for the additional TFM, and modify your `nuspec` file to copy the output assemblies in the correct location in the package.

- Add an empty `._.` file

If your package does not contain any assemblies, such as a meta-package, consider adding an empty `._.` file to the `lib/<tfm>/` directories for the TFM's listed in the warning message. For example, if the warning says to add assemblies for the `netstandard2.0` target framework, create an empty `lib/netstandard2.0/._.` file in your package.

- Remove the dependency group

If you use a custom `nuspec` file, remove the dependency group for the reported TFM, leaving only dependency groups for TFM's for which `lib/<tfm>/` or `ref/<tfm>/` files exist.

- Remove all dependencies for packages that are not related to assemblies

If your package does not contain any `lib/` or `ref/` files and is not a meta-package, it likely does not have any dependencies that the package consumer needs. If you are packing with NuGet's MSBuild Pack target, you can set `<SuppressDependenciesWhenPacking>true</SuppressDependenciesWhenPacking>` in any `PropertyGroup` in your project file.

If you are using a custom `nuspec` file, remove the `<dependencies>` element.

- Other scenarios

This warning was added during NuGet 5.3's development, and first was available in .NET Core SDK 3.0 Preview 9.

[NuGet/Home#8583](#) tracks an issue where the warning was being raised in too many scenarios. You can use the `NoWarn` MSBuild property (add `<NoWarn>$(NoWarn);NU5128</NoWarn>` to any `PropertyGroup` in your project file). If

you have multiple projects affected, you can use [Directory.Build.targets](#) to automatically add `NoWarn` to all projects.

NuGet Warning NU5129

9/26/2019 • 2 minutes to read • [Edit Online](#)

At least one `.file` was found in '///', but '///.' was not.

`<extension>` is one of: `targets`, `props`. `<build_folder>` is one of: `build`, `buildTransitive`, `buildCrossTargeting`, `buildMultiTargeting`. `<tfm>` is a [Target Framework Moniker](#), or may be absent. `<package_id>` is the [package identifier](#) of your package.

Examples:

```
At least one .targets file was found in 'build/netstandard2.0/', but 'build/netstandard2.0/MyPackage.targets' was not.  
At least one .props file was found in 'build/netstandard2.0/', but 'build/netstandard2.0/MyPackage.props' was not.  
At least one .props file was found in 'buildTransitive/net472/', but 'buildTransitive/net472/My.Package.Id.props' was not.  
At least one .targets file was found in 'buildMultitargeting/netcoreapp3.0/', but 'buildMultitargeting/netcoreapp3.0/Contoso.Utilities.targets' was not.  
At least one .props file was found in 'build/', but 'build/AdventureWorks.Tools.props' was not.
```

Issue

Packages that [include MSBuild props and targets](#) need to follow the naming convention of using the package id before the `.props` or `.targets` extension. Files that do not follow this convention will not be imported into projects that use the package.

Example: If the package id is `Contoso.Utilities` and contains the files `build/Contoso.Utilities.props` and `build/Utilities.targets`, only the `Contoso.Utilities.props` file will be imported into projects using the package. `Utilities.targets` will not be imported by NuGet.

Solution

Rename the file to meet the convention.

In the above example, `build/netstandard2.0/Utilities.targets` should be renamed to `build/netstandard2.0/Contoso.Utilities.targets` for NuGet to start importing it. If `Utilities.targets` is being imported in `Contoso.Utilities.props`, then rename the file to use the `.props` extension as well.

NuGet Warning NU5130

9/26/2019 • 2 minutes to read • [Edit Online](#)

Some target frameworks declared in the dependencies group of the nuspec and the lib/ref folder have compatible matches, but not exact matches in the other location. Unless intentional, consult the list of actions below:

- Add a dependency group for .NETFramework4.7.2 to the nuspec

Issue

The `lib/<tfm>/` or `ref/<tfm>/` directory in the package contains at least one file for the [Target Framework Moniker \(TFM\)](#) specified in the warning message. However, no dependency group exists for this TFM in the `nuspec` file. This may cause package consumers to believe the package is not compatible with the TFM. If the assemblies for the specified TFM have different dependencies to the compatible TFM defined in the dependencies group, the project using the package may experience runtime failures.

Solution

- Run NuGet's pack target on the project

If possible, use [NuGet's MSBuild pack target](#), as it automatically matches assembly TFM with dependency groups from the project's target frameworks. Note that your project must use `PackageReference` for its own NuGet dependencies. If your project uses `packages.config`, you need to use `nuget.exe pack` and a `nuspec` file.

- Manually edit the `nuspec` file

If you are using a custom `nuspec` file, we recommend each TFM for which `lib/` or `ref/` assemblies exist should have a matching dependency group, even if the dependencies are the same as the next compatible TFM. For example, if a package contains `netstandard1.0` and `netstandard2.0` assemblies, and the dependencies are the same for both, we recommend both TFM be listed as dependency groups with duplicate dependency items.

Note that the TFM identifier used in the assembly paths use a different format to the TFM identifier used in dependency groups. The warning message specifies the correct name to use in the dependency group. If your package does not have any dependencies for that target framework, use an empty group. For example:

```
<package>
  <metadata>
    ...
    <dependencies>
      <group targetFramework=".NETFramework4.7.2" />
    </dependencies>
  </metadata>
  ...
</package>
```

- Remove the `lib/` or `ref/` files

If you do not wish your package to be compatible with the reported TFM, modify your project such that no `lib/<tfm>/` or `ref/<tfm>/` files are in the package for that TFM. For example, if the warning says to add a dependency group for `.NETFramework4.7.2` to the `nuspec`, then remove any `lib/net472/*` and `ref/net472/*` files from your package.

NuGet Warning NU5131

10/15/2019 • 2 minutes to read • [Edit Online](#)

References were found in the nuspec, but some reference assemblies were not found in both the nuspec and ref folder. Add the following reference assemblies:

- Add AssemblyName.dll to the ref/net472/ directory

Issue

NuGet has a feature to allow package authors to [select which assemblies will be available at compile time](#) in projects that use the package.

If the required conventions are not followed, projects using the package with `PackageReference` may fail at runtime due to missing assemblies.

Solution

The list of assemblies in the nuspec file's `<references>` section should have matching assemblies in `ref/<tfm>/`.

For example, consider a package with the following files:

```
lib\net472\MyLib.dll
lib\net472\MyHelpers.dll
lib\net472\MyUtilities.dll
```

The package author wants to prevent package consumers from writing code that directly calls `MyUtilities.dll`, so they add the following to their nuspec file:

```
<references>
  <group targetFramework="net472">
    <reference file="MyLib.dll" />
    <reference file="MyHelpers.dll" />
  </group>
</references>
```

This package will not work as intended when using `PackageReference`. To fix this, the package must also contain the following files:

```
ref\net472\MyLib.dll
ref\net472\MyHelpers.dll
```

NuGet Warning NU5500

9/4/2018 • 2 minutes to read • [Edit Online](#)

Issue

The NuGet pack operation had a problem. NU5000 is used when we haven't yet assigned a unique warning code for that issue. So we can improve, please feel free to file the issue with details of your error.

Solution

Check the output window (in Visual Studio) or console output (with NuGet or dotnet command line tools) for more information.

project.json reference

8/15/2019 • 4 minutes to read • [Edit Online](#)

NuGet 3.x+

The `project.json` file maintains a list of packages used in a project, known as a package management format. It supersedes `packages.config` but is in turn superseded by [PackageReference](#) with NuGet 4.0+.

The `project.lock.json` file (described below) is also used in projects employing `project.json`.

`project.json` has the following basic structure, where each of the four top-level objects can have any number of child objects:

```
{  
  "dependencies": {  
    "PackageID" : "{version_constraint}"  
  },  
  "frameworks" : {  
    "TxM" : {}  
  },  
  "runtimes" : {  
    "RID": {}  
  },  
  "supports" : {  
    "CompatibilityProfile" : {}  
  }  
}
```

Dependencies

Lists the NuGet package dependencies of your project in the following form:

```
"PackageID" : "version_constraint"
```

For example:

```
"dependencies": {  
  "Microsoft.NETCore": "5.0.0",  
  "System.Runtime.Serialization.Primitives": "4.0.10"  
}
```

The `dependencies` section is where the NuGet Package Manager dialog adds package dependencies to your project.

The Package id corresponds to the id of the package on [nuget.org](#), the same as the id used in the package manager console: `Install-Package Microsoft.NETCore`.

When restoring packages, the version constraint of `"5.0.0"` implies `>= 5.0.0`. That is, if 5.0.0 is not available on the server but 5.0.1 is, NuGet installs 5.0.1 and warns you about the upgrade. NuGet otherwise picks the lowest possible version on the server matching the constraint.

See [Dependency resolution](#) for more details on resolution rules.

Managing dependency assets

Which assets from dependencies flow into the top-level project is controlled by specifying a comma-delimited set of tags in the `include` and `exclude` properties of the dependency reference. The tags are listed in the table below:

INCLUDE/EXCLUDE TAG	AFFECTED FOLDERS OF THE TARGET
contentFiles	Content
runtime	Runtime, Resources, and FrameworkAssemblies
compile	lib
build	build (MSBuild props and targets)
native	native
none	No folders
all	All folders

Tags specified with `exclude` take precedence over those specified with `include`. For example,

`include="runtime, compile" exclude="compile"` is the same as `include="runtime"`.

For example, to include the `build` and `native` folders of a dependency, use the following:

```
{
  "dependencies": {
    "packageA": {
      "version": "1.0.0",
      "include": "build, native"
    }
  }
}
```

To exclude the `content` and `build` folders of a dependency, use the following:

```
{
  "dependencies": {
    "packageA": {
      "version": "1.0.0",
      "exclude": "contentFiles, build"
    }
  }
}
```

Frameworks

Lists the frameworks that the project runs on, such as `net45`, `netcoreapp`, `netstandard`.

```
"frameworks": {
  "netcore50": {}
}
```

Only a single entry is allowed in the `frameworks` section. (An exception is `project.json` files for ASP.NET projects that are build with deprecated DNX tool chain, which allows for multiple targets.)

Runtimes

Lists the operating systems and architectures that your app runs on, such as `win10-arm`, `win8-x64`, `win8-x86`.

```
"runtimes": {
  "win10-arm": { },
  "win10-arm-aot": { },
  "win10-x86": { },
  "win10-x86-aot": { },
  "win10-x64": { },
  "win10-x64-aot": { }
}
```

A package containing a PCL that can run on any runtime doesn't need to specify a runtime. This must also be true of any dependencies, otherwise you must specify runtimes.

Supports

Defines a set of checks for package dependencies. You can define where you expect the PCL or app to run. The definitions are not restrictive, as your code may be able to run elsewhere. But specifying these checks makes NuGet check that all dependencies are satisfied on the listed TxMs. Examples of the values for this are: `net46.app`, `uwp.10.0.app`, etc.

This section should be populated automatically when you select an entry in the Portable Class Library targets dialog.

```
"supports": {
  "net46.app": {},
  "uwp.10.0.app": {}
}
```

Imports

Imports are designed to allow packages that use the `dotnet` TxM to operate with packages that don't declare a `dotnet` TxM. If your project is using the `dotnet` TxM then all the packages you depend on must also have a `dotnet` TxM, unless you add the following to your `project.json` to allow non `dotnet` platforms to be compatible with `dotnet`:

```
"frameworks": {
  "dotnet": { "imports" : "portable-net45+win81" }
}
```

If you are using the `dotnet` TxM then the PCL project system adds the appropriate `imports` statement based on the supported targets.

Differences from portable apps and web projects

The `project.json` file used by NuGet is a subset of that found in ASP.NET Core projects. In ASP.NET Core `project.json` is used for project metadata, compilation information, and dependencies. When used in other project systems, those three things are split into separate files and `project.json` contains less information. Notable differences include:

- There can only be one framework in the `frameworks` section.
- The file cannot contain dependencies, compilation options, etc. that you see in DNX `project.json` files.

Given that there can only be a single framework it doesn't make sense to enter framework-specific dependencies.

- Compilation is handled by MSBuild so compilation options, preprocessor defines, etc. are all part of the MSBuild project file and not `project.json`.

In NuGet 3+, developers are not expected to manually edit the `project.json`, as the Package Manager UI in Visual Studio manipulates the content. That said, you can certainly edit the file, but you must build the project to start a package restore or invoke restore in another way. See [Package restore](#).

project.lock.json

The `project.lock.json` file is generated in the process of restoring the NuGet packages in projects that use `project.json`. It holds a snapshot of all the information that is generated as NuGet walks the graph of packages and includes the version, contents, and dependencies of all the packages in your project. The build system uses this to choose packages from a global location that are relevant when building the project instead of depending on a local packages folder in the project itself. This results in faster build performance because it's necessary to read only `project.lock.json` instead of many separate `.nuspec` files.

`project.lock.json` is automatically generated on package restore, so it can be omitted from source control by adding it to `.gitignore` and `.tfignore` files (see [Packages and source control](#)). However, if you include it in source control, the change history shows changes in dependencies resolved over time.

project.json and UWP

9/4/2018 • 8 minutes to read • [Edit Online](#)

IMPORTANT

This content is deprecated. Projects should use either the `packages.config` or `PackageReference` formats.

This document describes the package structure that employs features in NuGet 3+ (Visual Studio 2015 and later). The `minClientVersion` property of your `.nuspec` can be used to state that you require the features described here by setting it to 3.1.

Adding UWP support to an existing package

If you have an existing package and you want to add support for UWP applications, then you don't need to adopt the packaging format described here. You only need to adopt this format if you require the features it describes and are willing to work only with clients that have updated to version 3+ of the NuGet client.

I already target netcore45

If you target `netcore45` already, and you don't need to take advantage of the features here, no action is needed. `netcore45` packages can be consumed by UWP applications.

I want to take advantage of Windows 10 specific APIs

In this case you need to add the `uap10.0` target framework moniker (TFM or TxM) to your package. Create a new folder in your package and add the assembly that has been compiled to work with Windows 10 to that folder.

I don't need Windows 10 specific APIs, but want new .NET features or don't have netcore45 already

In this case you would add the `dotnet` TxM to your package. Unlike other TxMs, `dotnet` doesn't imply a surface area or platform. It is stating that your package works on any platform that your dependencies work on. When building a package with the `dotnet` TxM, you are likely to have many more TxM-specific dependencies in your `.nuspec`, as you need to define the BCL packages you depend on, such `System.Text`, `System.Xml`, etc. The locations that those dependencies work on define where your package works.

How do I find out my dependencies

There are two ways to figure out which dependencies to list:

1. Use the [NuSpec Dependency Generator 3rd party](#) tool. The tool automates the process and updates your `.nuspec` file with the dependant packages on build. It is available via a NuGet package, [NuSpec.ReferenceGenerator](#).
2. (The hard way) Use `ILDasm` to look at your `.dll` to see what assemblies are actually needed at runtime. Then determine which NuGet package they each come from.

See the [project.json](#) topic for details on features that help in the creation of a package that supports the `dotnet` TxM.

IMPORTANT

If your package is intended to work with PCL projects, we highly recommend to create a `dotnet` folder, to avoid warnings and potential compatibility issues.

Directory structure

NuGet packages using this format have the following well-known folder and behaviors:

FOLDER	BEHAVIORS
Build	Contains MSBuild targets and props files in this folder are integrated differently into the project, but otherwise there is no change.
Tools	<code>install.ps1</code> and <code>uninstall.ps1</code> are not run. <code>init.ps1</code> works as it always has.
Content	Content is not copied automatically into a user's project. Support for content inclusion in the project is planned for a later release.
Lib	For many packages the <code>lib</code> works the same way it does in NuGet 2.x, but with expanded options for what names can be used inside it and better logic for picking the correct sub-folder when consuming packages. However, when used in conjunction with <code>ref</code> , the <code>lib</code> folder contains assemblies that implement the surface area defined by the assemblies in the <code>ref</code> folder.
Ref	<code>ref</code> is an optional folder that contains .NET assemblies defining the public surface (public types and methods) for an application to compile against. The assemblies in this folder may have no implementation, they are purely used to define surface area for the compiler. If the package has no <code>ref</code> folder, then the <code>lib</code> is both the reference assembly and the implementation assembly.
Runtimes	<code>runtimes</code> is an optional folder that contains OS specific code, such as CPU architecture and OS specific or otherwise platform-dependent binaries.

MSBuild targets and props files in packages

NuGet packages can contain `.targets` and `.props` files which are imported into any MSBuild project that the package is installed into. In NuGet 2.x, this was done by injecting `<Import>` statements into the `.csproj` file, in NuGet 3.0 there is no specific "installation to project" action. Instead the package restore process writes two files `[projectname].nuget.props` and `[projectname].NuGet.targets`.

MSBuild knows to look for these two files and automatically imports them near the beginning and near the end of the project build process. This provides very similar behavior to NuGet 2.x, but with one major difference: *there is no guaranteed order of targets/props files in this case*. However, MSBuild does provide ways to order targets through the `BeforeTargets` and `AfterTargets` attributes of the `<Target>` definition (see [Target Element \(MSBuild\)](#)).

Lib and Ref

The behavior of the `lib` folder hasn't changed significantly in NuGet v3. However, all assemblies must be within sub-folders named after a TxM, and can no longer be placed directly under the `lib` folder. A TxM is the name of a platform that a given asset in a package is supposed to work for. Logically these are an extension of the Target Framework Monikers (TFM) e.g. `net45`, `net46`, `netcore50`, and `dnxcore50` are all examples of TxMs (see [Target Frameworks](#)). TxM can refer to a framework (TFM) as well as other platform-specific surface areas. For example the UWP TxM (`uap10.0`) represents the .NET surface area as well as the Windows surface area for UWP applications.

An example lib structure:

```
lib
└── net40
    └── MyLibrary.dll
└── wp81
    └── MyLibrary.dll
```

The `lib` folder contains assemblies that are used at runtime. For most packages a folder under `lib` for each of the target TxMs is all that is required.

Ref

There are sometimes cases where a different assembly should be used during compilation (.NET Reference Assemblies do this today). For those cases, use a top-level folder called `ref` (short for "Reference Assemblies").

Most package authors don't require the `ref` folder. It is useful for packages that need to provide a consistent surface area for compilation and IntelliSense but then have different implementation for different TxMs. The biggest use case of this are the `System.*` packages that are being produced as part of shipping .NET Core on NuGet. These packages have various implementations that are being unified by a consistent set of ref assemblies.

Mechanically, the assemblies included in the `ref` folder are the reference assemblies being passed to the compiler. For those of you who have used `csc.exe` these are the assemblies we are passing to the [C# /reference option](#) switch.

The structure of the `ref` folder is the same as `lib`, for example:

```
└── MyImageProcessingLib
    └── lib
        ├── net40
        │   └── MyImageProcessingLibrary.dll
        ├── net451
        │   └── MyImageProcessingLibrary.dll
        └── win81
            └── MyImageProcessingLibrary.dll
    └── ref
        ├── net40
        │   └── MyImageProcessingLibrary.dll
        └── portable-net451-win81
            └── MyImageProcessingLibrary.dll
```

In this example the assemblies in the `ref` directories would all be identical.

Runtimes

The runtimes folder contains assemblies and native libraries required to run on specific "runtimes", which are

generally defined by Operating System and CPU architecture. These runtimes are identified using [Runtime Identifiers \(RIDs\)](#) such as `win`, `win-x86`, `win7-x86`, `win8-64`, etc.

Native helpers to use platform-specific APIs

The following example shows a package that has a purely managed implementation for several platforms, but uses native helpers on Windows 8 where it can call into Windows 8-specific native APIs.

```
└─MyLibrary
  ├─lib
  │  └─net40
  │    └─MyLibrary.dll
  └─runtimes
    ├─win8-x64
    │  ├─lib
    │  └─net40
    │    └─MyLibrary.dll
    └─native
      └─MyNativeLibrary.dll
    └─win8-x86
      ├─lib
      └─net40
        └─MyLibrary.dll
      └─native
        └─MyNativeLibrary.dll
```

Given the above package the following things happen:

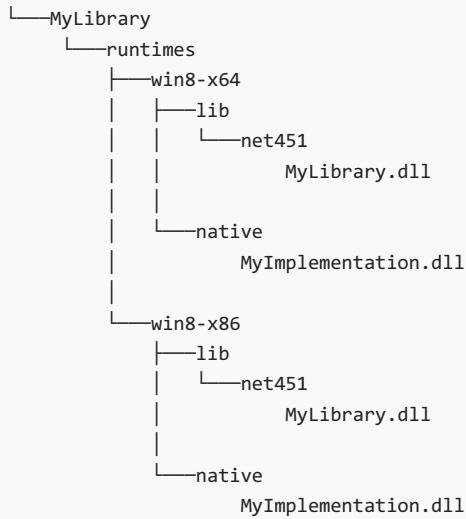
- When not on Windows 8 the `lib/net40/MyLibrary.dll` assembly is used.
- When on Windows 8 the `runtimes/win8-<architecture>/lib/MyLibrary.dll` is used and the `native/MyNativeHelper.dll` is copied to the output of your build.

In the example above the `lib/net40` assembly is purely managed code, whilst the assemblies in the runtimes folder will p/invoke into the native helper assembly to call APIs specific to Windows 8.

Only a single `lib` folder is ever be picked, so if there is a runtime specific folder it's chosen over non-runtime specific `lib`. The native folder is additive, if it exists it's copied to the output of the build.

Managed wrapper

Another way to use runtimes is to ship a package that is purely a managed wrapper over a native assembly. In this scenario you create a package like the following:



In this case there is no top-level `lib` folder as that folder as there is no implementation of this package that doesn't rely on the corresponding native assembly. If the managed assembly, `MyLibrary.dll`, was exactly the same in both of these cases then we could put it in a top level `lib` folder, but because the lack of a native assembly doesn't cause the package to fail installing if it was installed on a platform that wasn't `win-x86` or `win-x64` then the top level `lib` would be used but no native assembly would be copied.

Authoring packages for NuGet 2 and NuGet 3

If you want to create a package that can be consumed by projects using `packages.config` as well as packages using `project.json` then the following apply:

- Ref and runtimes only work on NuGet 3. They are both ignored by NuGet 2.
- You cannot rely on `install.ps1` or `uninstall.ps1` to function. These files execute when using `packages.config`, but are ignored with `project.json`. So your package needs to be usable without them running. `init.ps1` still runs on NuGet 3.
- Targets and Props installation is different, so make sure that your package works as expected on both clients.
- Subdirectories of `lib` must be a TxM in NuGet 3. You cannot place libraries at the root of the `lib` folder.
- Content is not copied automatically with NuGet 3. Consumers of your package could copy the files themselves or use a tool like a task runner to automate copying the files.
- Source and config file transforms are not run by NuGet 3.

If you are supporting NuGet 2 and 3 then your `minClientVersion` should be the lowest version of NuGet 2 client that your package works on. In the case of an existing package it shouldn't need to change.

Impact of project.json when creating packages

8/15/2019 • 3 minutes to read • [Edit Online](#)

IMPORTANT

This content is deprecated. Projects should use either the `packages.config` or `PackageReference` formats.

The `project.json` system used in NuGet 3+ affects package authors in several ways as described in the following sections.

Changes affecting existing packages usage

Traditional NuGet packages support a set of features that are not carried over to the transitive world.

Install and uninstall scripts are ignored

The transitive restore model, described in [Dependency resolution](#), does not have a concept of "package install time". A package is either present or not present, but there is no consistent process that occurs when a package is installed.

Also, install scripts were supported only in Visual Studio. Other IDEs had to mock the Visual Studio extensibility API to attempt to support such scripts, and no support was available in common editors and command-line tools.

Content transforms are not supported

Similar to install scripts, transforms run on package install and are typically not idempotent. Since there is no install time anymore, XDT Transform and similar features are not supported, and are ignored if such a package is used in a transitive scenario.

Content

Traditional NuGet packages are shipping content files such as source code and configuration files. There are used typically in two scenarios:

1. Initial files dropped into the project so the user can edit them at a later time. The common example is default configuration files.
2. Content files used as companions to the assemblies installed in the project. The example here would be a logo image used by an assembly.

Support for content is currently disabled for similar reasons for scripts and transforms, but we are in the process of designing support for content.

Content files can still be carried inside the packages, and are ignored currently, however the end user can still copy them into the right spot.

You can see one of the proposals for bringing back content files, and follow its progress, here:
<https://github.com/NuGet/Home/issues/627>.

Impact for package authors

Packages using the above features would have to use a different mechanism. The most commonly useful mechanism for this would be the MSBuild targets/props that continues to get fully supported. The build system can choose to pick up other conventions in the package. This is how MSBuild targets are supported as well as

Roslyn analyzers. It is possible to build packages that supports targets and analyzers for `packages.config` and `project.json` scenarios.

Packages that attempt to modify the project to ease startup typically work in a very limited set of scenarios, and should instead provide a readme, or guidance on how to use the package.

Most existing packages should not need to use the package format described below.

The format enables native content as a first class scenario. This means that managed assemblies depend on close to hardware implementations to ship binary implementations alongside the managed assemblies based on the target platform. For example `System.IO.Compression` package is utilizing this technology.

<https://www.nuget.org/packages/System.IO.Compression>

In summary if the functionality above is not absolutely necessary, we recommend sticking with the existing package format, as the format described here is supported only by NuGet 3.x+.

It would be possible to build packages to work for both `packages.config` and `project.json` scenarios through shimming, however it's often simpler to just structure the packages the traditional way, without the deprecated features mentioned above.

3.x package format

The 3.x package format allows for several additional features beyond NuGet 2.x:

1. Defining a reference assembly used for compilation and a set of implementation assemblies used for runtime on different platforms/devices. Which allows you to take advantage of platform specific APIs while providing a common surface area for your consumers. Specifically this makes writing intermediate portable libraries easier.
2. Allows packages to pivot on platforms e.g. operating systems or CPU architecture.
3. Allows for separation of platform specific implementations to companion packages.
4. Support Native dependencies as a first class citizen.

NuGet cross platform plugins

10/15/2019 • 9 minutes to read • [Edit Online](#)

In NuGet 4.8+ support for cross platform plugins has been added. This was achieved with by building a new plugin extensibility model, that has to conform to a strict set of rules of operation. The plugins are self-contained executables (runnables in the .NET Core world), that the NuGet Clients launch in a separate process. This is a true write once, run everywhere plugin. It will work with all NuGet client tools. The plugins can be either .NET Framework (NuGet.exe, MSBuild.exe and Visual Studio), or .NET Core (dotnet.exe). A versioned communication protocol between the NuGet Client and the plugin is defined. During the startup handshake, the 2 processes negotiate the protocol version.

In order to cover all NuGet client tools scenarios, one would need both a .NET Framework and a .NET Core plugin. The below describes the client/framework combinations of the plugins.

CLIENT TOOL	FRAMEWORK
Visual Studio	.NET Framework
dotnet.exe	.NET Core
NuGet.exe	.NET Framework
MSBuild.exe	.NET Framework
NuGet.exe on Mono	.NET Framework

How does it work

The high level workflow can be described as follows:

1. NuGet discovers available plugins.
2. When applicable, NuGet will iterate over the plugins in priority order and starts them one by one.
3. NuGet will use the first plugin that can service the request.
4. The plugins will be shut down when they are no longer needed.

General plugin requirements

The current protocol version is 2.0.0. Under this version, the requirements are as follows:

- Have a valid, trusted Authenticode signature assemblies that will run on Windows and Mono. There is no special trust requirement for assemblies run on Linux and Mac yet. [Relevant issue](#)
- Support stateless launching under the current security context of NuGet client tools. For example, NuGet client tools will not perform elevation or additional initialization outside of the plugin protocol described later.
- Be non interactive, unless explicitly specified.
- Adhere to the negotiated plugin protocol version.
- Respond to all requests within a reasonable time period.
- Honor cancellation requests for any in-progress operation.

The technical specification is described in more detail in the following specs:

- [NuGet Package Download Plugin](#)
- [NuGet cross plat authentication plugin](#)

Client - Plugin interaction

NuGet client tools and the plugins communicate with JSON over standard streams (stdin, stdout, stderr). All data must be UTF-8 encoded. The plugins are launched with the argument "-Plugin". In case a user directly launches a plugin executable without this argument, the plugin can give an informative message instead of waiting for a protocol handshake. The protocol handshake timeout is 5 seconds. The plugin should complete the setup in as short of an amount as possible. NuGet client tools will query a plugin's supported operations by passing in the service index for a NuGet source. A plugin may use the service index to check for the presence of supported service types.

The communication between the NuGet client tools and the plugin is bidirectional. Each request has a timeout of 5 seconds. If operations are supposed to take longer the respective process should send out a progress message to prevent the request from timing out. After 1 minute of inactivity a plugin is considered idle and is shut down.

Plugin installation and discovery

The plugins will be discovered via a convention based directory structure. CI/CD scenarios and power users can use environment variables to override the behavior. When using environment variables, only absolute paths are allowed. Note that `NUGET_NETFX_PLUGIN_PATHS` and `NUGET_NETCORE_PLUGIN_PATHS` are only available with 5.3+ version of the NuGet tooling and later.

- `NUGET_NETFX_PLUGIN_PATHS` - defines the plugins that will be used by the .NET Framework based tooling (NuGet.exe/MSBuild.exe/Visual Studio). Takes precedence over `NUGET_PLUGIN_PATHS`. (NuGet version 5.3+ only)
- `NUGET_NETCORE_PLUGIN_PATHS` - defines the plugins that will be used by the .NET Core based tooling (dotnet.exe). Takes precedence over `NUGET_PLUGIN_PATHS`. (NuGet version 5.3+ only)
- `NUGET_PLUGIN_PATHS` - defines the plugins that will be used for that NuGet process, priority preserved. If this environment variable is set, it overrides the convention based discovery. Ignored if either of the framework specific variables is specified.
- User-location, the NuGet Home location in `%UserProfile%/.nuget/plugins`. This location cannot be overriden. A different root directory will be used for .NET Core and .NET Framework plugins.

FRAMEWORK	ROOT DISCOVERY LOCATION
.NET Core	<code>%UserProfile%/.nuget/plugins/netcore</code>
.NET Framework	<code>%UserProfile%/.nuget/plugins/netfx</code>

Each plugin should be installed in its own folder. The plugin entry point will be the name of the installed folder, with the .dll extensions for .NET Core, and .exe extension for .NET Framework.

```

.nuget
  plugins
    netfx
      myPlugin
        myPlugin.exe
        nuget.protocol.dll
        ...
    netcore
      myPlugin
        myPlugin.dll
        nuget.protocol.dll
        ...
  ...

```

NOTE

There is currently no user story for the installation of the plugins. It's as simple as moving the required files into the predetermined location.

Supported operations

Two operations are supported under the new plugin protocol.

OPERATION NAME	MINIMUM PROTOCOL VERSION	MINIMUM NUGET CLIENT VERSION
Download Package	1.0.0	4.3.0
Authentication	2.0.0	4.8.0

Running plugins under the correct runtime

For the NuGet in dotnet.exe scenarios, plugins need to be able to execute under that specific runtime of the dotnet.exe. It's on the plugin provider and the consumer to make sure a compatible dotnet.exe/plugin combination is used. A potential issue could arise with the user-location plugins when for example, a dotnet.exe under the 2.0 runtime tries to use a plugin written for the 2.1 runtime.

Capabilities caching

The security verification and instantiation of the plugins is costly. The download operation happens way more frequently than the authentication operation, however the average NuGet user is only likely to have an authentication plugin. To improve the experience, NuGet will cache the operation claims for the given request. This cache is per plugin with the plugin key being the plugin path, and the expiration for this capabilities cache is 30 days.

The cache is located in `%LocalAppData%/NuGet/plugins-cache` and be overriden with the environment variable `NUGET_PLUGINS_CACHE_PATH`. To clear this [cache](#), one can run the `locals` command with the `plugins-cache` option. The `a11` `locals` option will now also delete the plugins cache.

Protocol messages index

Protocol Version 1.0.0 messages:

1. Close

- Request direction: NuGet -> plugin
- The request will contain no payload

- No response is expected. The proper response is for the plugin process to promptly exit.

2. Copy files in package

- Request direction: NuGet -> plugin
- The request will contain:
 - the package ID and version
 - the package source repository location
 - destination directory path
 - an enumerable of files in the package to be copied to the destination directory path
- A response will contain:
 - a response code indicating the outcome of the operation
 - an enumerable of full paths for copied files in the destination directory if the operation was successful

3. Copy package file (.nupkg)

- Request direction: NuGet -> plugin
- The request will contain:
 - the package ID and version
 - the package source repository location
 - the destination file path
- A response will contain:
 - a response code indicating the outcome of the operation

4. Get credentials

- Request direction: plugin -> NuGet
- The request will contain:
 - the package source repository location
 - the HTTP status code obtained from the package source repository using current credentials
- A response will contain:
 - a response code indicating the outcome of the operation
 - a username, if available
 - a password, if available

5. Get files in package

- Request direction: NuGet -> plugin
- The request will contain:
 - the package ID and version
 - the package source repository location
- A response will contain:
 - a response code indicating the outcome of the operation
 - an enumerable of file paths in the package if the operation was successful

6. Get operation claims

- Request direction: NuGet -> plugin
- The request will contain:
 - the service index.json for a package source
 - the package source repository location
- A response will contain:
 - a response code indicating the outcome of the operation

- o an enumerable of supported operations (e.g.: package download) if the operation was successful. If a plugin does not support the package source, the plugin must return an empty set of supported operations.

NOTE

This message has been updated in version 2.0.0. It is on the client to preserve backward compatibility.

7. Get package hash

- Request direction: NuGet -> plugin
- The request will contain:
 - o the package ID and version
 - o the package source repository location
 - o the hash algorithm
- A response will contain:
 - o a response code indicating the outcome of the operation
 - o a package file hash using the requested hash algorithm if the operation was successful

8. Get package versions

- Request direction: NuGet -> plugin
- The request will contain:
 - o the package ID
 - o the package source repository location
- A response will contain:
 - o a response code indicating the outcome of the operation
 - o an enumerable of package versions if the operation was successful

9. Get service index

- Request direction: plugin -> NuGet
- The request will contain:
 - o the package source repository location
- A response will contain:
 - o a response code indicating the outcome of the operation
 - o the service index if the operation was successful

10. Handshake

- Request direction: NuGet <-> plugin
- The request will contain:
 - o the current plugin protocol version
 - o the minimum supported plugin protocol version
- A response will contain:
 - o a response code indicating the outcome of the operation
 - o the negotiated protocol version if the operation was successful. A failure will result in termination of the plugin.

11. Initialize

- Request direction: NuGet -> plugin
- The request will contain:
 - o the NuGet client tool version

- the NuGet client tool effective language. This takes into consideration the ForceEnglishOutput setting, if used.
- the default request timeout, which supersedes the protocol default.
- A response will contain:
 - a response code indicating the outcome of the operation. A failure will result in termination of the plugin.

12. Log

- Request direction: plugin -> NuGet
- The request will contain:
 - the log level for the request
 - a message to log
- A response will contain:
 - a response code indicating the outcome of the operation.

13. Monitor NuGet process exit

- Request direction: NuGet -> plugin
- The request will contain:
 - the NuGet process ID
- A response will contain:
 - a response code indicating the outcome of the operation.

14. Prefetch package

- Request direction: NuGet -> plugin
- The request will contain:
 - the package ID and version
 - the package source repository location
- A response will contain:
 - a response code indicating the outcome of the operation

15. Set credentials

- Request direction: NuGet -> plugin
- The request will contain:
 - the package source repository location
 - the last known package source username, if available
 - the last known package source password, if available
 - the last known proxy username, if available
 - the last known proxy password, if available
- A response will contain:
 - a response code indicating the outcome of the operation

16. Set log level

- Request direction: NuGet -> plugin
- The request will contain:
 - the default log level
- A response will contain:
 - a response code indicating the outcome of the operation

Protocol Version 2.0.0 messages

17. Get Operation Claims

- Request direction: NuGet -> plugin
 - The request will contain:
 - the service index.json for a package source
 - the package source repository location
 - A response will contain:
 - a response code indicating the outcome of the operation
 - an enumerable of supported operations if the operation was successful. If a plugin does not support the package source, the plugin must return an empty set of supported operations.

If the service index and package source are null, then the plugin can answer with authentication.

18. Get Authentication Credentials

- Request direction: NuGet -> plugin
- The request will contain:
 - Uri
 - isRetry
 - NonInteractive
 - CanShowDialog
- A response will contain
 - Username
 - Password
 - Message
 - List of Auth Types
 - MessageResponseCode

NuGet cross platform authentication plugin

7/18/2019 • 2 minutes to read • [Edit Online](#)

In version 4.8+, all NuGet clients (NuGet.exe, Visual Studio, dotnet.exe and MSBuild.exe) can use an authentication plugin built on top of the [NuGet cross platform plugins](#) model.

Authentication in dotnet.exe

Visual Studio and NuGet.exe are by default interactive. NuGet.exe contains a switch to make it [non interactive](#). Additionally the NuGet.exe and Visual Studio plugins prompt the user for input. In dotnet.exe there is no prompting and the default is non interactive.

The authentication mechanism in dotnet.exe is device flow. When the restore or add package operation is run interactively, the operation blocks and instructions to the user how to complete the authentications will be provided on the command line. When the user completes the authentication the operation will continue.

To make the operation interactive, one should pass `--interactive`. Currently only the explicit `dotnet restore` and `dotnet add package` commands support an interactive switch. There is no interactive switch on `dotnet build` and `dotnet publish`.

Authentication in MSBuild

Similar to dotnet.exe, MSBuild.exe is by default non interactive the MSBuild.exe authentication mechanism is device flow. To allow the restore to pause and wait for authentication, call restore with

```
msbuild -t:restore -p:NuGetInteractive="true" .
```

Creating a cross platform authentication plugin

A sample implementation can be found in [Microsoft Credential Provider plugin](#).

It's very important that the plugins conforms to the security requirements set forth by the NuGet client tools. The minimum required version for a plugin to be an authentication plugin is 2.0.0. NuGet will perform the handshake with the plugin and query for the supported operation claims. Please refer to the NuGet cross platform plugin [protocol messages](#) for more details about the specific messages.

NuGet will set the log level and provide proxy information to the plugin when applicable. Logging to the NuGet console is only acceptable after NuGet has set the log level to the plugin.

- .NET Framework plugin authentication behavior

In .NET Framework, the plugins are allowed to prompt a user for input, in the form of a dialog.

- .NET Core plugin authentication behavior

In .NET Core, a dialog cannot be shown. The plugins should use device flow to authenticate. The plugin can send log messages to NuGet with instructions to the user. Note that logging is available after the log level has been set to the plugin. NuGet will not take any interactive input from the command line.

When the client calls the plugin with a Get Authentication Credentials, the plugins need to conform to the interactivity switch and respect the dialog switch.

The following table summarizes how the plugin should behave for all combinations.

ISNONINTERACTIVE	CANSHOWDIALOG	PLUGIN BEHAVIOR
true	true	The IsNonInteractive switch takes precedence over the dialog switch. The plugin is not allowed to pop a dialog. This combination is only valid for .NET Framework plugins
true	false	The IsNonInteractive switch takes precedence over the dialog switch. The plugin is not allowed to block. This combination is only valid for .NET Core plugins
false	true	The plugin should show a dialog. This combination is only valid for .NET Framework plugins
false	false	The plugin should/can not show a dialog. The plugin should use device flow to authenticate by logging an instruction message via the logger. This combination is only valid for .NET Core plugins

Please refer to the following specs before writing a plugin.

- [NuGet Package Download Plugin](#)
- [NuGet cross plat authentication plugin](#)

Authenticating feeds in Visual Studio with NuGet credential providers

9/5/2019 • 3 minutes to read • [Edit Online](#)

The NuGet Visual Studio Extension 3.6+ supports credential providers, which enable NuGet to work with authenticated feeds. After you install a NuGet credential provider for Visual Studio, the NuGet Visual Studio extension will automatically acquire and refresh credentials for authenticated feeds as necessary.

A sample implementation can be found in [the VsCredentialProvider sample](#).

Starting with 4.8+ NuGet in Visual Studio supports the new cross platform authentication plugins as well, but they are not the recommended approach for performance reasons.

NOTE

NuGet credential providers for Visual Studio must be installed as a regular Visual Studio extension and will require [Visual Studio 2017](#) or above.

NuGet credential providers for Visual Studio work only in Visual Studio (not in dotnet restore or nuget.exe). For credential providers with nuget.exe, see [nuget.exe Credential Providers](#). For credential providers in dotnet and msbuild see [NuGet cross platform plugins](#)

Available NuGet credential providers for Visual Studio

There is a credential provider built into the Visual Studio NuGet extension to support Visual Studio Team Services.

The NuGet Visual Studio Extension uses an internal `vsCredentialProviderImporter` which also scans for plug-in credential providers. These plug-in credential providers must be discoverable as a MEF Export of type `IVsCredentialProvider`.

Available plug-in credential providers include:

- [MyGet Credential Provider for Visual Studio](#)

Creating a NuGet credential provider for Visual Studio

The NuGet Visual Studio Extension 3.6+ implements an internal CredentialService that is used to acquire credentials. The CredentialService has a list of built-in and plug-in credential providers. Each provider is tried sequentially until credentials are acquired.

During credential acquisition, the credential service will try credential providers in the following order, stopping as soon as credentials are acquired:

1. Credentials will be fetched from NuGet configuration files (using the built-in `SettingsCredentialProvider`).
2. If the package source is on Visual Studio Team Services, the `VisualStudioAccountProvider` will be used.
3. All other plug-in Visual Studio credential providers will be tried sequentially.
4. Try to use all NuGet cross platform credential providers sequentially.
5. If no credentials have been acquired yet, the user will be prompted for credentials using a standard basic authentication dialog.

Implementing `IVsCredentialProvider.GetCredentialsAsync`

To create a NuGet credential provider for Visual Studio, create a Visual Studio Extension that exposes a public MEF Export implementing the `IVsCredentialProvider` type, and adheres to the principles outlined below.

```
public interface IVsCredentialProvider
{
    Task<ICredentials> GetCredentialsAsync(
        Uri uri,
        IWebProxy proxy,
        bool isProxyRequest,
        bool isRetry,
        bool nonInteractive,
        CancellationToken cancellationToken);
}
```

A sample implementation can be found in [the VsCredentialProvider sample](#).

Each NuGet credential provider for Visual Studio must:

1. Determine whether it can provide credentials for the targeted URI before initiating credential acquisition. If the provider cannot supply credentials for the targeted source, then it should return `null`.
2. If the provider does handle requests for the targeted URI, but cannot supply credentials, an exception should be thrown.

A custom NuGet credential provider for Visual Studio must implement the `IVsCredentialProvider` interface available in the [NuGet.VisualStudio package](#).

GetCredentialAsync

INPUT PARAMETER	DESCRIPTION
Uri uri	The package source Uri for which credentials are being requested.
IWebProxy proxy	Web proxy to use when communicating on the network. Null if there is no proxy authentication configured.
bool isProxyRequest	True if this request is to get proxy authentication credentials. If the implementation is not valid for acquiring proxy credentials, then null should be returned.
bool isRetry	True if credentials were previously requested for this Uri, but the supplied credentials did not allow authorized access.
bool nonInteractive	If true, the credential provider must suppress all user prompts and use default values instead.
CancellationToken cancellationToken	This cancellation token should be checked to determine if the operation requesting credentials has been cancelled.

Return value: A credentials object implementing the [System.Net.ICredentials interface](#).

Authenticating feeds with nuget.exe credential providers

9/4/2018 • 3 minutes to read • [Edit Online](#)

NuGet 3.3+

When `nuget.exe` needs credentials to authenticate with a feed, it looks for them in the following manner:

1. NuGet first looks for credentials in `Nuget.Config` files.
2. NuGet then uses plug-in credential providers, subject to the order given below. (An example is the [Visual Studio Team Services Credential Provider](#).)
3. NuGet then prompts the user for credentials on the command line.

Note that the credential providers described here work only in `nuget.exe` and not in 'dotnet restore' or Visual Studio. For credential providers with Visual Studio, see [nuget.exe Credential Providers for Visual Studio](#)

nuget.exe credential providers can be used in 3 ways:

- **Globally:** to make a credential provider available to all instances of `nuget.exe` run under the current user's profile, add it to `%LocalAppData%\NuGet\CredentialProviders`. You may need to create the `CredentialProviders` folder. Credential providers can be installed at the root of the `CredentialProviders` folder or within a subfolder. If a credential provider has multiple files/assemblies, you can use subfolders to keep the providers organized.
- **From an environment variable:** Credential providers can be stored anywhere and made accessible to `nuget.exe` by setting the `%NUGET_CREDENTIALPROVIDERS_PATH%` environment variable to the provider location. This variable can be a semicolon-separated list (for example, `path1;path2`) if you have multiple locations.
- **Alongside nuget.exe:** nuget.exe credential providers can be placed in the same folder as `nuget.exe`.

When loading credential providers, `nuget.exe` searches the above locations, in order, for any file named `credentialprovider*.exe`, then loads those files in the order they're found. If multiple credential providers exist in the same folder, they're loaded in alphabetical order.

Creating a nuget.exe credential provider

A credential provider is a command-line executable, named in the form `CredentialProvider*.exe`, that gathers inputs, acquires credentials as appropriate, and then returns the appropriate exit status code and standard output.

A provider must do the following:

- Determine whether it can provide credentials for the targeted URI before initiating credential acquisition. If not, it should return status code 1 with no credentials.
- Not modify `Nuget.Config` (such as setting credentials there).
- Handle HTTP proxy configuration on its own, as NuGet does not provide proxy information to the plugin.
- Return credentials or error details to `nuget.exe` by writing a JSON response object (see below) to stdout, using UTF-8 encoding.
- Optionally emit additional trace logging to stderr. No secrets should ever be written to stderr, since at verbosity levels "normal" or "detailed" such traces are echoed by NuGet to the console.
- Unexpected parameters should be ignored, providing forward compatibility with future versions of NuGet.

Input parameters

PARAMETER/SWITCH	DESCRIPTION
Uri {value}	The package source URI requiring credentials.
NonInteractive	If present, provider does not issue interactive prompts.
IsRetry	If present, indicates that this attempt is a retry of a previously failed attempt. Providers typically use this flag to ensure that they bypass any existing cache and prompt for new credentials if possible.
Verbosity {value}	If present, one of the following values: "normal", "quiet", or "detailed". If no value is supplied, defaults to "normal". Providers should use this as an indication of the level of optional logging to emit to the standard error stream.

Exit codes

CODE	RESULT	DESCRIPTION
0	Success	Credentials were successfully acquired and have been written to stdout.
1	ProviderNotApplicable	The current provider does not provide credentials for the given URI.
2	Failure	The provider is the correct provider for the given URI, but cannot provide credentials. In this case, nuget.exe will not retry authentication and will fail. A typical example is when a user cancels an interactive login.

Standard output

PROPERTY	NOTES
Username	Username for authenticated requests.
Password	Password for authenticated requests.
Message	Optional details about the response, used only to show additional details in failure cases.

Example stdout:

```
{ "Username" : "freddy@example.com",
  "Password" : "bwm3bcx6txhprzmxh12x63mdsul6grctazoomtdb6kfb0f7m3a3z",
  "Message"  : "" }
```

Troubleshooting a credential provider

At present, NuGet doesn't provide much direct support for debugging custom credential providers; [issue 4598](#) is tracking this work.

You can also do the following:

- Run nuget.exe with the `-verbosity` switch to inspect detailed output.
- Add debug messages to `stdout` in appropriate places.
- Be sure that you're using nuget.exe 3.3 or higher.
- Attach debugger on startup with this code snippet:

```
while (!Debugger.IsAttached)
{
    System.Threading.Thread.Sleep(100);
}
Debugger.Break();
```

For further help, [submit a support request a nuget.org](#).

NuGet API in Visual Studio

11/3/2018 • 13 minutes to read • [Edit Online](#)

In addition to the Package Manager UI and Console in Visual Studio, NuGet also exports some useful services through the [Managed Extensibility Framework \(MEF\)](#). This interface allows other components in Visual Studio to interact with NuGet, which can be used to install and uninstall packages, and to obtain information about installed packages.

As of NuGet 3.3+, NuGet exports the following services all of which reside in the `NuGet.VisualStudio` namespace in the `NuGet.VisualStudio.dll` assembly:

- `IRegistryKey` : Method to retrieve a value from a registry subkey.
- `IVsPackageInstaller` : Methods to install NuGet packages into projects.
- `IVsPackageInstallerEvents` : Events for package install/uninstall.
- `IVsPackageInstallerProjectEvents` : Batch events for package install/uninstall.
- `IVsPackageInstallerServices` : Methods to retrieve installed packages in the current solution and to check whether a given package is installed in a project.
- `IVsPackageManagerProvider` : Methods to provide alternative Package Manager suggestions for a NuGet package.
- `IVsPackageMetadata` : Methods to retrieve information about an installed package.
- `IVsPackageProjectMetadata` : Methods to retrieve information about a project where NuGet actions are being executed.
- `IVsPackageRestorer` : Methods to restore packages installed in a project.
- `IVsPackageSourceProvider` : Methods to retrieve a list of NuGet package sources.
- `IVsPackageUninstaller` : Methods to uninstall NuGet packages from projects.
- `IVsTemplateWizard` : Designed for project/item templates to include pre-installed packages; this interface is *not* meant to be invoked from code and has no public methods.

Using NuGet services

1. Install the `NuGet.VisualStudio` package into your project, which contains the `NuGet.VisualStudio.dll` assembly.

When installed, the package automatically sets the **Embed Interop Types** property of the assembly reference to **True**. This makes your code resilient against version changes when users update to newer versions of NuGet.

WARNING

Do not use any other types besides the public interfaces in your code, and do not reference any other NuGet assemblies, including `NuGet.Core.dll`.

1. To use a service, import it through the [MEF Import attribute](#), or through the `IComponentModel` service.

```

//Using the Import attribute
[Import(typeof(IVsPackageInstaller2))]
public IVsPackageInstaller2 packageInstaller;
packageInstaller.InstallLatestPackage(null, currentProject,
    "Newtonsoft.Json", false, false);

//Using the IComponentModel service
var componentModel = (IComponentModel)GetService(typeof(SComponentModel));
IVsPackageInstallerServices installerServices =
    componentModel.GetService<IVsPackageInstallerServices>();

var installedPackages = installerServices.GetInstalledPackages();

```

For reference, the source code for NuGet.VisualStudio is contained within the [NuGet.Clients repository](#).

IRRegistryKey interface

```

/// <summary>
/// Specifies methods for manipulating a key in the Windows registry.
/// </summary>
public interface IRegistryKey
{
    /// <summary>
    /// Retrieves the specified subkey for read or read/write access.
    /// </summary>
    /// <param name="name">The name or path of the subkey to create or open.</param>
    /// <returns>The subkey requested, or null if the operation failed.</returns>
    IRegistryKey OpenSubKey(string name);

    /// <summary>
    /// Retrieves the value associated with the specified name.
    /// </summary>
    /// <param name="name">The name of the value to retrieve. This string is not case-sensitive.</param>
    /// <returns>The value associated with name, or null if name is not found.</returns>
    object GetValue(string name);

    /// <summary>
    /// Closes the key and flushes it to disk if its contents have been modified.
    /// </summary>
    void Close();
}

```

IVsPackageInstaller interface

```

public interface IVsPackageInstaller
{
    /// <summary>
    /// Installs a single package from the specified package source.
    /// </summary>
    /// <param name="source">
    /// The package source to install the package from. This value can be <c>null</c>
    /// to indicate that the user's configured sources should be used. Otherwise,
    /// this should be the source path as a string. If the user has credentials
    /// configured for a source, this value must exactly match the configured source
    /// value.
    /// </param>
    /// <param name="project">The target project for package installation.</param>
    /// <param name="packageId">The package ID of the package to install.</param>
    /// <param name="version">
    /// The version of the package to install. <c>null</c> can be provided to

```

```
/// install the latest version of the package.
/// </param>
/// <param name="ignoreDependencies">
/// A boolean indicating whether or not to ignore the package's dependencies
/// during installation.
/// </param>
void InstallPackage(string source, Project project, string packageId, Version version, bool ignoreDependencies);

/// <summary>
/// Installs a single package from the specified package source.
/// </summary>
/// <param name="source">
/// The package source to install the package from. This value can be <c>null</c>
/// to indicate that the user's configured sources should be used. Otherwise,
/// this should be the source path as a string. If the user has credentials
/// configured for a source, this value must exactly match the configured source
/// value.
/// </param>
/// <param name="project">The target project for package installation.</param>
/// <param name="packageId">The package ID of the package to install.</param>
/// <param name="version">
/// The version of the package to install. <c>null</c> can be provided to
/// install the latest version of the package.
/// </param>
/// <param name="ignoreDependencies">
/// A boolean indicating whether or not to ignore the package's dependencies
/// during installation.
/// </param>
void InstallPackage(string source, Project project, string packageId, string version, bool ignoreDependencies);

/// <summary>
/// Installs a single package from the specified package source.
/// </summary>
/// <param name="repository">The package repository to install the package from.</param>
/// <param name="project">The target project for package installation.</param>
/// <param name="packageId">The package id of the package to install.</param>
/// <param name="version">
/// The version of the package to install. <c>null</c> can be provided to
/// install the latest version of the package.
/// </param>
/// <param name="ignoreDependencies">
/// A boolean indicating whether or not to ignore the package's dependencies
/// during installation.
/// </param>
/// <param name="skipAssemblyReferences">
/// A boolean indicating if assembly references from the package should be
/// skipped.
/// </param>
void InstallPackage(IPackageRepository repository, Project project, string packageId, string version, bool ignoreDependencies, bool skipAssemblyReferences);

/// <summary>
/// Installs one or more packages that exist on disk in a folder defined in the registry.
/// </summary>
/// <param name="keyName">
/// The registry key name (under NuGet's repository key) that defines the folder on disk
/// containing the packages.
/// </param>
/// <param name="isPreUnzipped">
/// A boolean indicating whether the folder contains packages that are
/// pre-unzipped.
/// </param>
/// <param name="skipAssemblyReferences">
/// A boolean indicating whether the assembly references from the packages
/// should be skipped.
/// </param>
/// <param name="project">The target project for package installation.</param>
```

```
/// <param name="packageVersions">
/// A dictionary of packages/versions to install where the key is the package id
/// and the value is the version.
/// </param>
/// <remarks>
/// If any version of the package is already installed, no action will be taken.
/// <para>
/// Dependencies are always ignored.
/// </para>
/// </remarks>
void InstallPackagesFromRegistryRepository(string keyName, bool isPreUnzipped, bool skipAssemblyReferences,
Project project, IDictionary<string, string> packageVersions);

/// <summary>
/// Installs one or more packages that exist on disk in a folder defined in the registry.
/// </summary>
/// <param name="keyName">
/// The registry key name (under NuGet's repository key) that defines the folder on disk
/// containing the packages.
/// </param>
/// <param name="isPreUnzipped">
/// A boolean indicating whether the folder contains packages that are
/// pre-unzipped.
/// </param>
/// <param name="skipAssemblyReferences">
/// A boolean indicating whether the assembly references from the packages
/// should be skipped.
/// </param>
/// <param name="ignoreDependencies">A boolean indicating whether the package's dependencies should be
ignored</param>
/// <param name="project">The target project for package installation.</param>
/// <param name="packageVersions">
/// A dictionary of packages/versions to install where the key is the package id
/// and the value is the version.
/// </param>
/// <remarks>
/// If any version of the package is already installed, no action will be taken.
/// </remarks>
void InstallPackagesFromRegistryRepository(string keyName, bool isPreUnzipped, bool skipAssemblyReferences,
bool ignoreDependencies, Project project, IDictionary<string, string> packageVersions);

/// <summary>
/// Installs one or more packages that are embedded in a Visual Studio Extension Package.
/// </summary>
/// <param name="extensionId">The Id of the Visual Studio Extension Package.</param>
/// <param name="isPreUnzipped">
/// A boolean indicating whether the folder contains packages that are
/// pre-unzipped.
/// </param>
/// <param name="skipAssemblyReferences">
/// A boolean indicating whether the assembly references from the packages
/// should be skipped.
/// </param>
/// <param name="project">The target project for package installation.</param>
/// <param name="packageVersions">
/// A dictionary of packages/versions to install where the key is the package id
/// and the value is the version.
/// </param>
/// <remarks>
/// If any version of the package is already installed, no action will be taken.
/// <para>
/// Dependencies are always ignored.
/// </para>
/// </remarks>
void InstallPackagesFromVSExtensionRepository(string extensionId, bool isPreUnzipped, bool
skipAssemblyReferences, Project project, IDictionary<string, string> packageVersions);

/// <summary>
/// Installs one or more packages that are embedded in a Visual Studio Extension Package.

```

```

...
/// </summary>
/// <param name="extensionId">The Id of the Visual Studio Extension Package.</param>
/// <param name="isPreUnzipped">
/// A boolean indicating whether the folder contains packages that are
/// pre-unzipped.
/// </param>
/// <param name="skipAssemblyReferences">
/// A boolean indicating whether the assembly references from the packages
/// should be skipped.
/// </param>
/// <param name="ignoreDependencies">A boolean indicating whether the package's dependencies should be
ignored</param>
/// <param name="project">The target project for package installation</param>
/// <param name="packageVersions">
/// A dictionary of packages/versions to install where the key is the package id
/// and the value is the version.
/// </param>
/// <remarks>
/// If any version of the package is already installed, no action will be taken.
/// </remarks>
void InstallPackagesFromVSExtensionRepository(string extensionId, bool isPreUnzipped, bool
skipAssemblyReferences, bool ignoreDependencies, Project project, IDictionary<string, string> packageVersions);
}

```

IVsPackageInstallerEvents interface

```

public interface IVsPackageInstallerEvents
{
    /// <summary>
    /// Raised when a package is about to be installed into the current solution.
    /// </summary>
    event VsPackageEventHandler PackageInstalling;

    /// <summary>
    /// Raised after a package has been installed into the current solution.
    /// </summary>
    event VsPackageEventHandler PackageInstalled;

    /// <summary>
    /// Raised when a package is about to be uninstalled from the current solution.
    /// </summary>
    event VsPackageEventHandler PackageUninstalling;

    /// <summary>
    /// Raised after a package has been uninstalled from the current solution.
    /// </summary>
    event VsPackageEventHandler PackageUninstalled;

    /// <summary>
    /// Raised after a package has been installed into a project within the current solution.
    /// </summary>
    event VsPackageEventHandler PackageReferenceAdded;

    /// <summary>
    /// Raised after a package has been uninstalled from a project within the current solution.
    /// </summary>
    event VsPackageEventHandler PackageReferenceRemoved;
}

```

IVsPackageInstallerProjectEvents interface

```

public interface IVsPackageInstallerProjectEvents
{
    /// <summary>
    /// Raised before any IVsPackageInstallerEvents events are raised for a project.
    /// </summary>
    event VsPackageProjectEventHandler BatchStart;

    /// <summary>
    /// Raised after all IVsPackageInstallerEvents events are raised for a project.
    /// </summary>
    event VsPackageProjectEventHandler BatchEnd;
}

```

IVsPackageInstallerServices interface

```

public interface IVsPackageInstallerServices
{
    // IMPORTANT: do NOT rearrange the methods here. The order is important to maintain
    // backwards compatibility with clients that were compiled against old versions of NuGet.

    /// <summary>
    /// Get the list of NuGet packages installed in the current solution.
    /// </summary>
    IEnumerable<IVsPackageMetadata> GetInstalledPackages();

    /// <summary>
    /// Checks if a NuGet package with the specified Id is installed in the specified project.
    /// </summary>
    /// <param name="project">The project to check for NuGet package.</param>
    /// <param name="id">The id of the package to check.</param>
    /// <returns><c>true</c> if the package is install. <c>false</c> otherwise.</returns>
    bool IsPackageInstalled(Project project, string id);

    /// <summary>
    /// Checks if a NuGet package with the specified Id and version is installed in the specified project.
    /// </summary>
    /// <param name="project">The project to check for NuGet package.</param>
    /// <param name="id">The id of the package to check.</param>
    /// <param name="version">The version of the package to check.</param>
    /// <returns><c>true</c> if the package is install. <c>false</c> otherwise.</returns>
    bool IsPackageInstalled(Project project, string id, SemanticVersion version);

    /// <summary>
    /// Checks if a NuGet package with the specified Id and version is installed in the specified project.
    /// </summary>
    /// <param name="project">The project to check for NuGet package.</param>
    /// <param name="id">The id of the package to check.</param>
    /// <param name="versionString">The version of the package to check.</param>
    /// <returns><c>true</c> if the package is install. <c>false</c> otherwise.</returns>
    /// <remarks>
    /// The reason this method is named IsPackageInstalledEx, instead of IsPackageInstalled, is that
    /// when client project compiles against this assembly, the compiler would attempt to bind against
    /// the other overload which accepts SemanticVersion and would require client project to reference
    NuGet.Core.
    /// </remarks>
    bool IsPackageInstalledEx(Project project, string id, string versionString);

    /// <summary>
    /// Get the list of NuGet packages installed in the specified project.
    /// </summary>
    /// <param name="project">The project to get NuGet packages from.</param>
    IEnumerable<IVsPackageMetadata> GetInstalledPackages(Project project);
}

```

IVsPackageManagerProvider interface

```
public interface IVsPackageManagerProvider
{
    /// <summary>
    /// Localized display package manager name.
    /// </summary>
    string PackageManagerName { get; }

    /// <summary>
    /// Package manager unique id.
    /// </summary>
    string PackageManagerId { get; }

    /// <summary>
    /// The tool tip description for the package
    /// </summary>
    string Description { get; }

    /// <summary>
    /// Check if a recommendation should be surfaced for an alternate package manager.
    /// This code should not rely on slow network calls, and should return rapidly.
    /// </summary>
    /// <param name="packageId">Current package id</param>
    /// <param name="projectName">Unique project name for finding the project through VS dte</param>
    /// <param name="token">Cancellation Token</param>
    /// <returns>return true if need to direct to integrated package manager for this package</returns>
    Task<bool> CheckForPackageAsync(string packageId, string projectName, CancellationToken token);

    /// <summary>
    /// This Action should take the user to the other package manager.
    /// </summary>
    /// <param name="packageId">Current package id</param>
    /// <param name="projectName">Unique project name for finding the project through VS dte</param>
    void GoToPackage(string packageId, string projectName);
}
```

IVsPackageMetadata interface

```

public interface IVsPackageMetadata
{
    /// <summary>
    /// Id of the package.
    /// </summary>
    string Id { get; }

    /// <summary>
    /// Version of the package.
    /// </summary>
    /// <remarks>
    /// Do not use this property because it will require referencing NuGet.Core.dll assembly. Use the
    VersionString
    /// property instead.
    /// </remarks>
    [Obsolete("Do not use this property because it will require referencing NuGet.Core.dll assembly. Use the
    VersionString property instead.")]
    NuGet.SemanticVersion Version { get; }

    /// <summary>
    /// Title of the package.
    /// </summary>
    string Title { get; }

    /// <summary>
    /// Description of the package.
    /// </summary>
    string Description { get; }

    /// <summary>
    /// The authors of the package.
    /// </summary>
    IEnumerable<string> Authors { get; }

    /// <summary>
    /// The location where the package is installed on disk.
    /// </summary>
    string InstallPath { get; }

    // IMPORTANT: This property must come LAST, because it was added in 2.5. Having it declared
    // LAST will avoid breaking components that compiled against earlier versions which doesn't
    // have this property.
    /// <summary>
    /// The version of the package.
    /// </summary>
    /// <remarks>
    /// Use this property instead of the Version property because with the type string,
    /// it doesn't require referencing NuGet.Core.dll assembly.
    /// </remarks>
    string VersionString { get; }
}

```

IVsPackageProjectMetadata interface

```

public interface IVsPackageProjectMetadata
{
    /// <summary>
    /// Unique batch id for batch start/end events of the project.
    /// </summary>
    string BatchId { get; }

    /// <summary>
    /// Name of the project.
    /// </summary>
    string ProjectName { get; }
}

```

IVsPackageRestorer interface

```

public interface IVsPackageRestorer
{
    /// <summary>
    /// Returns a value indicating whether the user consent to download NuGet packages
    /// has been granted.
    /// </summary>
    /// <returns>true if the user consent has been granted; otherwise, false.</returns>
    bool IsUserConsentGranted();

    /// <summary>
    /// Restores NuGet packages installed in the given project within the current solution.
    /// </summary>
    /// <param name="project">The project whose NuGet packages to restore.</param>
    void RestorePackages(Project project);
}

```

IVsPackageSourceProvider interface

```

public interface IVsPackageSourceProvider
{
    /// <summary>
    /// Provides the list of package sources.
    /// </summary>
    /// <param name="includeUnOfficial">Unofficial sources will be included in the results</param>
    /// <param name="includeDisabled">Disabled sources will be included in the results</param>
    /// <returns>Key: source name Value: source URI</returns>
    IEnumerable<KeyValuePair<string, string>> GetSources(bool includeUnOfficial, bool includeDisabled);

    /// <summary>
    /// Raised when sources are added, removed, disabled, or modified.
    /// </summary>
    event EventHandler SourcesChanged;
}

```

IVsPackageUninstaller interface

```
public interface IVsPackageUninstaller
{
    /// <summary>
    /// Uninstall the specified package from a project and specify whether to uninstall its dependency packages
    /// too.
    /// </summary>
    /// <param name="project">The project from which the package is uninstalled.</param>
    /// <param name="packageId">The package to be uninstalled</param>
    /// <param name="removeDependencies">
    /// A boolean to indicate whether the dependency packages should be
    /// uninstalled too.
    /// </param>
    void UninstallPackage(Project project, string packageId, bool removeDependencies);
}
```

IVsTemplateWizard interface

```
/// <summary>
/// Defines the logic for a template wizard extension.
/// </summary>

public interface IVsTemplateWizard : IWizard
{
}
```

NuGet support for the Visual Studio project system

9/4/2018 • 3 minutes to read • [Edit Online](#)

To support third-party project types in Visual Studio, NuGet 3.x+ supports the [Common Project System \(CPS\)](#), and NuGet 3.2+ supports non-CPS project systems as well.

To integrate with NuGet, a project system must advertise its own support for all the project capabilities described in this topic.

NOTE

Don't declare capabilities that your project does not actually have for the sake of enabling packages to install in your project. Many features of Visual Studio and other extensions depend on project capabilities besides the NuGet client. Falsely advertising capabilities of your project can lead these components to malfunction and your users' experience to degrade.

Advertise project capabilities

The NuGet client determines which packages are compatible with your project type based on the [project's capabilities](#), as described in the following table.

CAPABILITY	DESCRIPTION
AssemblyReferences	Indicates that the project supports assembly references (distinct from WinRTReferences).
DeclaredSourceItems	Indicates that the project is a typical MSBuild project (not DNX) in that it declares source items in the project itself.
UserSourceItems	Indicates that the user is allowed to add arbitrary files to their project.

For CPS-based project systems, the implementation details for project capabilities described in the rest of this section have been done for you. See [declaring project capabilities in CPS projects](#).

Implementing VsProjectCapabilitiesPresenceChecker

The `VsProjectCapabilitiesPresenceChecker` class must implement the `IVsBooleanSymbolPresenceChecker` interface, which is defined as follows:

```

public interface IVsBooleanSymbolPresenceChecker
{
    /// <summary>
    /// Checks whether the symbols defined may have changed since the last time
    /// this method was called.
    /// </summary>
    /// <param name="versionObject">
    /// The response version object assigned at the last call.
    /// May be null to get the initial version.
    /// At the conclusion of this method call, the reference may be changed so that on a subsequent call
    /// we know what version was last observed. The caller should treat this value as an opaque object,
    /// and should not assume any significance from whether the pointer changed or not.
    /// </param>
    /// <returns>
    /// <c>true</c> if the symbols defined may have changed since the last call to this method
    /// or if <paramref name="versionObject"/> was <c>null</c> upon entering this method.
    /// <c>false</c> if the responses would all be the same.
    /// </returns>
    bool HasChangedSince(ref object versionObject);

    /// <summary>
    /// Checks for the presence of a given symbol.
    /// </summary>
    /// <param name="symbol">The symbol to check for.</param>
    /// <returns><c>true</c> if the symbol is present; <c>false</c> otherwise.</returns>
    bool IsSymbolPresent(string symbol);
}

```

A sample implementation of this interface would then be:

```

class VsProjectCapabilitiesPresenceChecker : IVsBooleanSymbolPresenceChecker
{
    /// <summary>
    /// The set of capabilities this project system actually has.
    /// This should be kept current, and honestly reflect what the project can do.
    /// </summary>
    private static readonly HashSet<string> ActualProjectCapabilities = new HashSet<string>
    (StringComparer.OrdinalIgnoreCase)
    {
        "AssemblyReferences",
        "DeclaredSourceItems",
        "UserSourceItems",
    };

    public bool HasChangedSince(ref object versionObject)
    {
        // If your project capabilities do not change over time while the project is open,
        // you may simply `return false;` from your `HasChangedSince` method.
        return false;
    }

    public bool IsSymbolPresent(string symbol)
    {
        return ActualProjectCapabilities.Contains(symbol);
    }
}

```

Remember to add/remove capabilities from the `ActualProjectCapabilities` set based on what your project system actually supports. See the [project capabilities documentation](#) for full descriptions.

Responding to queries

A project declares this capability by supporting the `VSHPROPID_ProjectCapabilitiesChecker` property through the

`IVsHierarchy::GetProperty`. It should return an instance of `Microsoft.VisualStudio.Shell.Interop.IVsBooleanSymbolPresenceChecker`, which is defined in the `Microsoft.VisualStudio.Shell.Interop.14.0.DesignTime.dll` assembly. Reference this assembly by installing [its NuGet package](#).

For example, you might add the following `case` statement to your `IVsHierarchy::GetProperty` method's `switch` statement:

```
case __VSHPROPID8.VSHPROPID_ProjectCapabilitiesChecker:
    propVal = new VsProjectCapabilitiesPresenceChecker();
    return VSConstants.S_OK;
```

DTE Support

NuGet drives the project system to add references, content items, and MSBuild imports by calling into [DTE](#), which is the top-level Visual Studio automation interface. DTE is a set of COM interfaces that you may already implement.

If your project type is based on CPS, DTE is implemented for you.

Packages in Visual Studio templates

9/4/2018 • 5 minutes to read • [Edit Online](#)

Visual Studio project and item templates often need to ensure that certain packages are installed when a project or item is created. For example, the ASP.NET MVC 3 template installs jQuery, Modernizr, and other packages.

To support this, template authors can instruct NuGet to install the necessary packages, rather than individual libraries. Developers can then easily update those packages at any later time.

To learn more about authoring templates themselves, refer to [How to: Create Project Templates](#) or [Creating Custom Project and Item Templates](#).

The remainder of this section describes the specific steps to take when authoring a template to properly include NuGet packages.

- [Adding packages to a template](#)
- [Best practices](#)

For an example, see the [NuGetInVsTemplates sample](#).

Adding packages to a template

When a template is instantiated, a [template wizard](#) is invoked to load the list of packages to install along with information about where to find those packages. Packages can be embedded in the VSIX, embedded in the template, or located on the local hard drive in which case you use a registry key to reference the file path. Details on these locations are given later in this section.

Preinstalled packages work using [template wizards](#). A special wizard gets invoked when the template gets instantiated. The wizard loads the list of packages that need to be installed and passes that information to the appropriate NuGet APIs.

Steps to include packages in a template:

1. In your `vstemplate` file, add a reference to the NuGet template wizard by adding a `WizardExtension` element:

```
<WizardExtension>
  <Assembly>NuGet.VisualStudio.Interop, Version=1.0.0.0, Culture=neutral,
  PublicKeyToken=b03f5f7f11d50a3a</Assembly>
  <FullClassName>NuGet.VisualStudio.TemplateWizard</FullClassName>
</WizardExtension>
```

`NuGet.VisualStudio.Interop.dll` is an assembly that contains only the `TemplateWizard` class, which is a simple wrapper that calls into the actual implementation in `NuGet.VisualStudio.dll`. The assembly version will never change so that project/item templates continue to work with new versions of NuGet.

2. Add the list of packages to install in the project:

```
<WizardData>
  <packages>
    <package id="jQuery" version="1.6.2" />
  </packages>
</WizardData>
```

The wizard supports multiple `<package>` elements to support multiple package sources. Both the `id` and `version` attributes are required, meaning that specific version of a package will be installed even if a newer version is available. This prevents package updates from breaking the template, leaving the choice to update the package to the developer using the template.

3. Specify the repository where NuGet can find the packages as described in the following sections.

VSIX package repository

The recommended deployment approach for Visual Studio project/item templates is a [VSIX extension](#) because it allows you to package multiple project/item templates together and allows developers to easily discover your templates using the VS Extension Manager or the Visual Studio Gallery. Updates to the extension are also easy to deploy using the [Visual Studio Extension Manager automatic update mechanism](#).

The VSIX itself can serve as the source for packages required by the template:

1. Modify the `<packages>` element in the `.vstemplate` file as follows:

```
<packages repository="extension" repositoryId="MyTemplateContainerExtensionId">
  <!-- ... -->
</packages>
```

The `repository` attribute specifies the type of repository as `extension` while `repositoryId` is the unique identifier of the VSIX itself (This is the value of the `ID` attribute in the extension's `vsixmanifest` file, see [VSIX Extension Schema 2.0 Reference](#)).

2. Place your `nupkg` files in a folder called `Packages` within the VSIX.
3. Add the necessary package files as `<Asset>` in your `vsixmanifest` file (see [VSIX Extension Schema 2.0 Reference](#)):

```
<Asset Type="Moq.4.0.10827.nupkg" d:Source="File" Path="Packages\Moq.4.0.10827.nupkg"
d:VsixSubPath="Packages" />
```

4. Note that you can deliver packages in the same VSIX as your project templates or you can put them in a separate VSIX if that makes more sense for your scenario. However, do not reference any VSIX over which you do not have control, because changes to that extension could break your template.

Template package repository

If you are distributing only a single project/item template and do not need to package multiple templates together, you can use a simpler but more limited approach that includes packages directly in the project/item template ZIP file:

1. Modify the `<packages>` element in the `.vstemplate` file as follows:

```
<packages repository="template">
  <!-- ... -->
</packages>
```

The `repository` attribute has the value `template` and the `repositoryId` attribute is not required.

2. Place packages in the root folder of the project/item template ZIP file.

Note that using this approach in a VSIX that contains multiple templates leads to unnecessary bloat when one or more packages are common to the templates. In such cases, use the [VSIX as the repository](#) as described in the previous section.

Registry-specified folder path

SDKs that are installed using an MSI can install NuGet packages directly on the developer's machine. This makes them immediately available when a project or item template is used, rather than having to extract them during that time. ASP.NET templates use this approach.

1. Have the MSI install packages to the machine. You can install only the `.nupkg` files, or you can install those along with the expanded contents, which saves an additional step when the template is used. In this case, follow NuGet's standard folder structure wherein the `.nupkg` files are in the root folder, and then each package has a subfolder with the id/version pair as the subfolder name.
2. Write a registry key to identify the package location:
 - Key location: Either the machine-wide `HKEY_LOCAL_MACHINE\SOFTWARE[\Wow6432Node]\NuGet\Repository` or if it's per-user installed templates and packages, alternatively use `HKEY_CURRENT_USER\SOFTWARE\NuGet\Repository`
 - Key name: use a name that's unique to you. For example, the ASP.NET MVC 4 templates for VS 2012 use `AspNetMvc4VS11`.
 - Values: the full path to the packages folder.
3. In the `<packages>` element in the `.vstemplate` file, add the attribute `repository="registry"` and specify your registry key name in the `keyName` attribute.
 - If you have pre-unzipped your packages, use the `isPreunzipped="true"` attribute.
 - (*NuGet 3.2+*) If you want to force a design-time build at the end of package installation, add the `forceDesignTimeBuild="true"` attribute.
 - As an optimization, add `skipAssemblyReferences="true"` because the template itself already includes the necessary references.

```
<packages repository="registry" keyName="AspNetMvc4VS11" isPreunzipped="true">
  <package id="EntityFramework" version="5.0.0" skipAssemblyReferences="true" />
  <!-- ... -->
</packages>
```

Best Practices

1. Declare a dependency on the NuGet VSIX by adding a reference to it in your VSIX manifest:

```
<Reference Id="NuPackToolsVsix.Microsoft.67e54e40-0ae3-42c5-a949-fdd5739e7a5"
MinVersion="1.7.30402.9028">
  <Name>NuGet Package Manager</Name>
  <MoreInfoUrl>http://docs.microsoft.com/nuget/</MoreInfoUrl>
</Reference>
<!-- ... -->
```

2. Require project/item templates to be saved on creation by including `<PromptForSaveOnCreation>true</PromptForSaveOnCreation>` in the `.vstemplate` file.
3. Templates do not include a `packages.config` file, and do not include or any references or content that would be added when NuGet packages are installed.

NuGet governance

9/4/2018 • 5 minutes to read • [Edit Online](#)

This document is based upon the [Benevolent Dictator Governance Model](#) by the University of Oxford. It is licensed under a [Creative Commons Attribution-ShareAlike 2.0 UK: England & Wales License](#).

The NuGet project is led by a Benevolent Dictator and managed by the community. That is, the community actively contributes to the day-to-day maintenance of the project, but the general strategic line is drawn by the benevolent dictator. In case of disagreement, the benevolent dictator has the last word.

It is the benevolent dictator's job to resolve disputes within the community and to ensure that the project is able to progress in a coordinated way. In turn, it's the community's job to guide the decisions of the benevolent dictator through active engagement and contribution.

Roles and responsibilities

There are four roles described here: Benevolent Dictator, Committers, Contributors, and Users.

Benevolent dictator

The NuGet core team is self-appointed as Benevolent Dictator or project lead. However, because the community always has the ability to fork, the team is fully answerable to the community. The project lead is expected to understand the community as a whole and strive to satisfy as many conflicting needs as possible, while ensuring that the project survives in the long term.

In many ways, the role of the benevolent dictator is less about dictatorship and more about diplomacy. The key is to ensure that, as the project expands, the right people are given influence over it and the community rallies behind the vision of the project lead. The lead's job is then to ensure that the committers (see below) make the right decisions on behalf of the project. Generally speaking, as long as the committers are aligned with the project's strategy, the project lead will allow them to proceed as they desire.

Additionally, .NET Foundation staff consider the project lead the primary or first point of contact for NuGet for purposes of business operations including domain registrations, and technical services (e.g. code-signing).

Committers

Committers are contributors who have made sustained valuable contributions to NuGet and are appointed by the Benevolent Dictator. Once appointed, committers are relied upon to both write code directly to the repository and screen the contributions of others. Committers are often developers but can contribute in other ways.

Typically, a committer focuses on a specific aspect of the project, and brings a level of expertise and understanding that earns them the respect of the community and the project lead. The role of committer is not an official one, it's simply a position that influential members of the community assume as the project lead looks to them for guidance and support.

Committers have no authority where the overall direction of NuGet is concerned. However, they do have the ear of the project lead. It is a committer's job to ensure that the lead is aware of the community's needs and collective objectives, and to help develop or elicit appropriate contributions to the project. Often, committers are given informal control over their specific areas of responsibility, and are assigned rights to directly modify certain areas of the source code. That is, although committers do not have explicit decision-making authority, they will often find that their actions are synonymous with the decisions made by the lead.

Contributors

Contributors are community members who submit patches to NuGet. These patches may be a one-time occurrence or occur over time. Expectations are that contributors submit patches that are small at first and grow larger when the contributor, committers, and the project lead have built confidence in the quality of a contributor's patches. Contributors are recognized in the associated product release notes document.

Before a contributor's first patch is put into the repository, they must sign a [Contributor License Agreement](#) or an assignment agreement to the .NET Foundation. The patch can be submitted and discussed but it can't actually be committed to the repository without the appropriate paperwork in place. To obtain a contributor license agreement, please send a request in email to contributions@nuget.org.

To become a contributor, submit a pull request to one of the following repositories:

- [NuGet Client](#)
- [NuGet Gallery](#)
- [NuGet Docs](#)

The detailed process for submitting a pull request varies by repository:

- [Contribution instructions for NuGet Client and NuGet Gallery](#)
- [Contribution instructions for NuGet Docs](#)

Users

Users are community members who have a need for and use NuGet, as package consumers and/or authors. Users are the most important members of the community: without them, the project would have no purpose. Anyone can be a user; there are no specific requirements.

Users should be encouraged to participate in the life of NuGet and the community as much as possible. User contributions enable the project team to ensure that they are satisfying the needs of those users. Common user activities include but are not limited to the following:

- Advocating for use of the project
- Informing developers of project strengths and weaknesses from a new user's perspective
- Providing moral support (a thank you goes a long way)
- Writing documentation and tutorials
- Filing bug reports and feature requests
- Participating in community events, such as bug bashes
- Participating on discussion boards or forums

Users who continue to engage with the project and its community will often find themselves becoming more and more involved. Such users may then go on to become contributors, as described above.

Package succession under special circumstances

In the unfortunate situation where a NuGet account holder is incapacitated or deceased, we'll work with the community to add appropriate owner/s to the package where the said account has sole ownership and the package is published under an [OSI approved license](#). To request ownership you must send us the following documents:

1. A photocopy of your government-issued photo ID.
2. One of the following documents proving the previous account holder's status:
 - An official, government-issued death certificate if the previous account holder is deceased, or,
 - A certified document such as a certificate signed by a medical professional in charge of the care of an incapacitated account holder.
3. One of the following documents proving your right to ownership:
 - Marriage certificate showing that you are the surviving spouse of the account holder,

- Signed power of attorney,
- Copy of a will or trust document naming you as executor or beneficiary,
- Birth certificate for the account holder, if you are their parent, or,
- Guardianship paperwork if you are a legal guardian of the account holder.

If you find yourself in need of invoking this policy, please send us an email at support@nuget.org with the ID and version of the package.

Transparency

Building community trust in the governance of an open-source project is vital to its success. To that end, decision making must be done in a transparent, open fashion. Discussion about the project's direction must be done publicly. The community should never be caught off-guard by a decision by the Benevolent Dictator. Additionally, discussion about project decisions must be archived so that community members can understand the entire history of a decision and its context.

An overview of the NuGet ecosystem

9/4/2018 • 2 minutes to read • [Edit Online](#)

Since it's introduction in 2010, NuGet has presented a great opportunity to improve and automate different aspects of the development processes.

Because NuGet is open source under a permissive [Apache v2 license](#), other projects can leverage NuGet and companies can build support for it in their products. Whether for open-source projects or enterprise application development, NuGet and other applications built on and around NuGet provide a broad ecosystem of tools for improving your software development process.

All of these projects are able to innovate because of developer contributions. Just as you contribute to NuGet itself, also make contribution to these projects by reporting defects and new feature ideas, providing feedback, writing documentation, and contributing code where possible.

.NET Foundation projects

NuGet provides a free, open source package management system for the Microsoft development platform. It consists of a few client tools as well as the set of services that comprise the [official NuGet Gallery](#). Combined, these form the NuGet project which is governed by the [.NET Foundation](#).

The NuGet Organization contains various repositories on GitHub. <https://github.com/Nuget/Home> gives an overview of all the repositories and where to find the various NuGet components.

Microsoft projects

Microsoft has contributed extensively to the development of NuGet. All contributions made by Microsoft employees are also open source and are donated (including copyrights) to the .NET Foundation.

Non-Microsoft projects

Many other individuals and companies have made significant contributions to the NuGet ecosystem. Each project listed here may have a different license than the core NuGet components, so confirm that the license terms are acceptable prior to use:

- [AppVeyor CI](#)
- [Artifactory](#)
- [BoxStarter](#)
- [Chocolatey](#)
- [CoApp](#)
- [JetBrains ReSharper](#)
- [JetBrains TeamCity](#)
- [Klondike](#)
- [MinimalNugetServer](#)
- [MyGet \(or NuGet-as-a-service\)](#)
- [NuGet Package Explorer](#)
- [NuGet Server](#)
- [OctopusDeploy](#)
- [Paket](#)

- [ProGet \(Inedo\)](#)
- [scriptcs](#)
- [SharpDevelop](#)
- [Sonatype Nexus](#)
- [SymbolSource](#)
- [Xamarin and MonoDevelop](#)

Other NuGet-based utilities

These are tools and utilities built on NuGet:

- [Glimpse Extensions](#) (plug-ins are packages)
- [NuGetMustHaves.com](#)
- [Orchard](#) (CMS modules are fetched from a v1 NuGet feed hosted in the Orchard Gallery)
- [Java implementation of NuGet Server](#)
- [NuGetLatest](#) (Twitter bot tweeting new package publications)
- [DefinitelyTyped](#) ([Automatic](#) TypeScript Type Definitions published to NuGet)

Training materials and references

Using a new tool or technology usually comes with a learning curve. Luckily for you, NuGet has no steep learning curve at all! In fact, anyone can [get started consuming packages](#) quickly.

That said, authoring packages—and especially good packages—along with embracing NuGet in automated build and deployment processes, requires spending a little more time with the following resources:

- [NuGet Blog](#)
- [NuGet team on Twitter, @nuget](#)
- Books:
 - [Apress Pro NuGet](#)
 - [NuGet 2 Essentials](#)

Documentation for individual packages

[NuDoq](#) provides straightforward access and updates and documentation for NuGet packages.

NuDoq regularly polls the nuget.org gallery server for the latest package updates, unpacks and processes the library documentation files, and updates the site accordingly.

Adding your project

If you have a NuGet ecosystem project that would be a valuable addition to this page, please submit a pull request with an edit to this page.

User Data Requests

6/28/2019 • 2 minutes to read • [Edit Online](#)

nuget.org users can submit information delete requests and information export requests through [nuget.org](#). Both types are submitted in the form of support requests and are be executed by the nuget.org administrators within 30 days.

The following user data is directly accessible through nuget.org:

- Account related data such as email address, login account, profile picture, and email notification settings
- Owned API Keys
- List of owned packages

This data is not included in data exported through the support request.

Identifying customer data

Customer data can be identified as nuget.org user account names.

Deleting customer data

To request deleting user data from nuget.org:

1. The user must sign-in to [nuget.org](#)
2. The user must submit a request for their account to be deleted [nuget.org/account/delete](#)

Users that are sole owners of packages are encouraged to find new owners before asking to have their account deleted. If package ownership is not transferred, the NuGet package is unlisted and, as a result, it is no longer available in search queries in Visual Studio or on the nuget.org website. Before deleting the account, the nuget.org administrators work with the user to find new owners for the packages they own.

The account delete action is completed by the nuget.org administrator within 30 days from the date of the request.

Upon account deletion, all of the user's data is be removed from the nuget.org system and the following actions are taken:

- The deleted account becomes unregistered with nuget.org
- All owned API Keys are deleted
- All reserved namespaces are released
- Any package ownership are removed

The owned packages are *not* deleted. Though unlisted from search results, they remain available through package restore to projects that depend on them.

Exporting customer data

After sign-in to nuget.org, a user can submit an export request through [nuget.org/policies/Contact](#)

The data exported is made available for 48 hours to the user for download through an Azure Blob. After 48 hours, access expires and the user must submit a new export request as needed.

The exported data includes:

- The user's support requests
- The user's actions (publish package, create account) as persisted in the audit logs
- Any user information as persisted in the IIS logs

Known Issues with NuGet

7/18/2019 • 9 minutes to read • [Edit Online](#)

These are the most common known issues with NuGet that are repeatedly reported. If you are having trouble installing NuGet or managing packages, please take a look through these known issues and their resolutions.

NOTE

Starting with NuGet 4.0, known issues are a part of the respective release notes.

Authentication issues with NuGet feeds in VSTS with nuget.exe v3.4.3

Problem:

When we use the following command to store the credentials, we end up double encrypting the Personal Access Token.

```
$PAT = "Your personal access token" $Feed = "Your url" .\nuget.exe sources add -Name Test -Source $Feed -UserName $UserName -Password $PAT
```

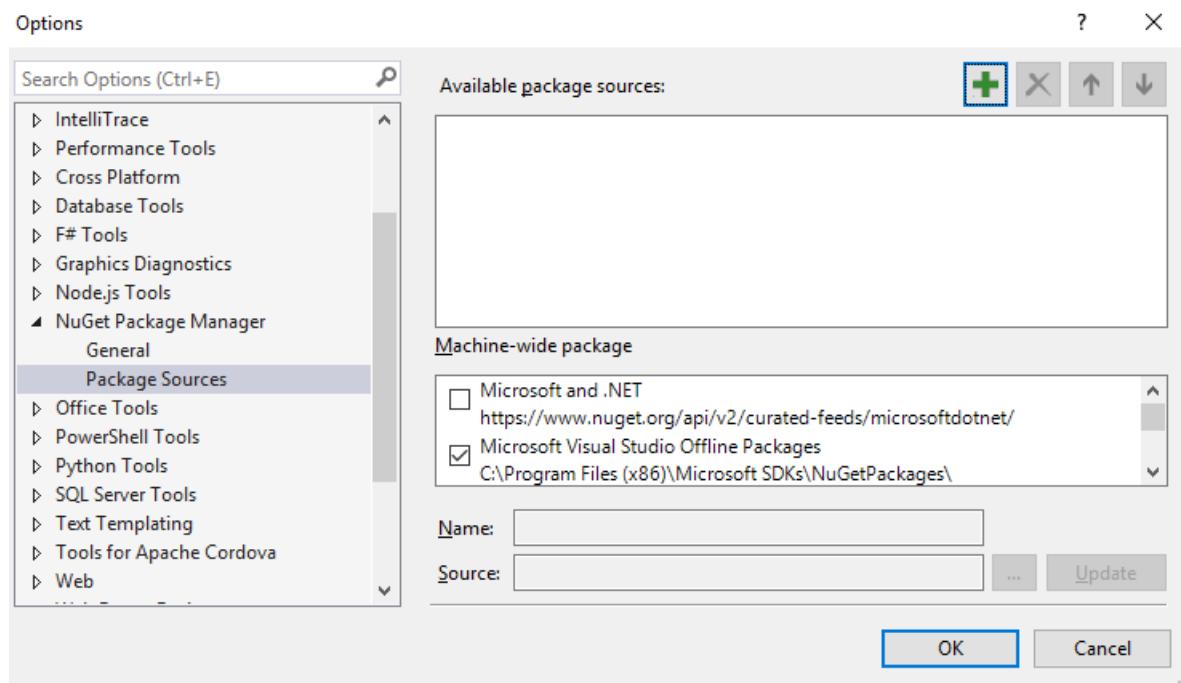
Workaround:

Store passwords in clear text using the [-StorePasswordInClearText](#) option.

Error installing packages with NuGet 3.4, 3.4.1

Problem:

In NuGet 3.4 and 3.4.1, when using the NuGet add-in, no sources are reported as available and you are unable to add new sources in the configuration window. The result is similar to the image below:



The `NuGet.Config` file in your `%AppData%\NuGet\` (Windows) or `~/.nuget/` (Mac/Linux) folder has accidentally been emptied. To fix this: close Visual Studio (on Windows, if applicable), delete the `NuGet.Config` file, and try the

operation again. NuGet generated a new `NuGet.Config` and you should be able to proceed.

Error installing packages with NuGet 2.7

Problem:

In NuGet 2.7 or above, when you attempt to install any package which contains assembly references, you may receive the error message "**Input string was not in a correct format.**", like below:

```
install-package log4net
Installing 'log4net 2.0.0'.
Successfully installed 'log4net 2.0.0'.
Adding 'log4net 2.0.0' to Tyson.OperatorUpload.
Install failed. Rolling back...
install-package : Input string was not in a correct format.
At line:1 char:1
    install-package log4net
    ~~~~~
CategoryInfo : NotSpecified: (:) [Install-Package], FormatException
FullyQualifiedErrorMessage : NuGetCmdletUnhandledException,NuGet.PowerShell.Commands.InstallPackageCommand
```

This is caused by the type library for the `VSLangProj.dll` COM component being unregistered on your system. This can happen, for example, when you have two versions of Visual Studio installed side-by-side and you then uninstall the older version. Doing so may inadvertently unregister the above COM library.

Solution:

Run this command from an **elevated prompt** to re-register the type library for `VSLangProj.dll`

```
regsvr32 "C:\Program Files (x86)\Common Files\microsoft shared\MSEnv\VsLangproj.olb"
```

If the command fails, check to see if the file exists in that location.

For more information about this error, see this [work item](#).

Build failure after package update in VS 2012

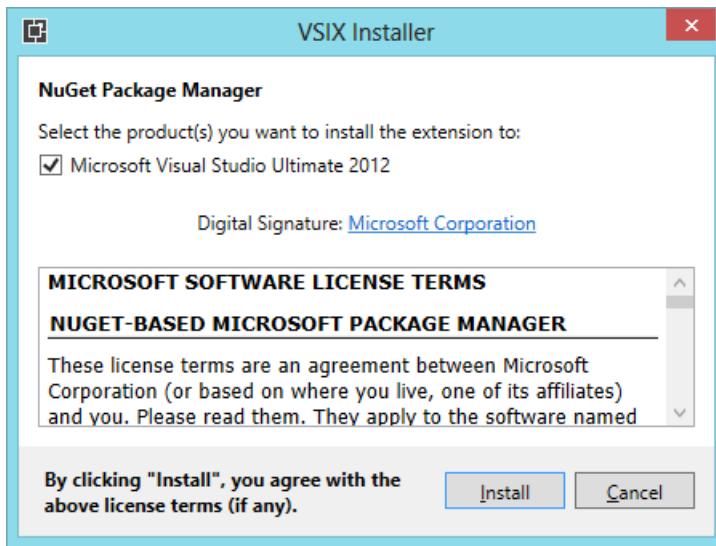
The problem: You are using VS 2012 RTM. When updating NuGet packages, you get this message: "One or more packages could not be completed uninstalled." and you are prompted to restart Visual Studio. After VS restart, you get weird build errors.

The cause is that certain files in the old packages are locked by a background MSBuild process. Even after VS restart, the background MSBuild process still uses the files in the old packages, causing the build failures.

The fix is to install VS 2012 Update, e.g. VS 2012 Update 2.

Upgrading to latest NuGet from an older version causes a signature verification error

If you are running VS 2010 SP1, you might run into the following error message when attempting to upgrade NuGet if you have an older version installed.



When viewing the logs, you might see a mention of a `SignatureMismatchException`.

To prevent this from occurring, there is a [Visual Studio 2010 SP1 hotfix](#) you can install. Alternatively, the workaround is to simply uninstall NuGet (while running Visual Studio as Administrator) and then install it from the VS Extension Gallery. See <http://support.microsoft.com/kb/2581019> for more information.

Package Manager Console throws an exception when the Reflector Visual Studio Add-In is also installed.

When running the Package Manager console, you may run into the following exception message if you have the Reflector VS Add-in installed.

```
The following error occurred while loading the extended type data file:  
Microsoft.PowerShell.Core, C:\Windows\SysWOW64\WindowsPowerShell\v1.0\types.ps1xml(2950) :  
Error in type "System.Security.AccessControl.ObjectSecurity":  
Exception: Cannot convert the "Microsoft.PowerShell.Commands.SecurityDescriptorCommandsBase"  
value of type "System.String" to type "System.Type".  
System.Management.Automation.ActionPreferenceStopException:  
Command execution stopped because the preference variable "ErrorActionPreference" or common parameter  
is set to Stop: Unable to find type
```

or

```
System.Management.Automation.CmdletInvocationException: Could not load file or assembly 'Scripts\NuGet.psm1' or
one of its dependencies. <br />The parameter is incorrect. (Exception from HRESULT: 0x80070057 (E_INVALIDARG))
---&gt; System.IO.FileLoadException: Could not load file or <br />assembly 'Scripts\NuGet.psm1' or one of its
dependencies. The parameter is incorrect. (Exception from HRESULT: 0x80070057 (E_INVALIDARG)) <br />---&gt;
System.ArgumentException: Illegal characters in path.
   at System.IO.Path.CheckInvalidPathChars(String path)
   at System.IO.Path.Combine(String path1, String path2)
   at Microsoft.VisualStudio.Platform.VsAppDomainManager.<AssemblyPaths>d__1.MoveNext()
   at Microsoft.VisualStudio.Platform.VsAppDomainManager.InnerResolveHandler(String name)
   at Microsoft.VisualStudio.Platform.VsAppDomainManager.ResolveHandler(Object sender, ResolveEventArgs args)
   at System.AppDomain.OnAssemblyResolveEvent(RuntimeAssembly assembly, String assemblyFullName)
   --- End of inner exception stack trace ---
   at Microsoft.PowerShell.Commands.ModuleCmdletBase.LoadBinaryModule(Boolean trySnapInName, String moduleName,
String fileName, <br />Assembly assemblyToLoad, String moduleBase, SessionState ss, String prefix, Boolean
loadTypes, Boolean loadFormats, Boolean& found)
   at Microsoft.PowerShell.Commands.ModuleCmdletBase.LoadModuleNamedInManifest(String moduleName, String
moduleBase, <br />Boolean searchModulePath, <br />String prefix, SessionState ss, Boolean loadTypesFiles,
Boolean loadFormatFiles, Boolean& found)
   at Microsoft.PowerShell.Commands.ModuleCmdletBase.LoadModuleManifest(ExternalScriptInfo scriptInfo,
ManifestProcessingFlags <br />manifestProcessingFlags, Version version)
   at Microsoft.PowerShell.Commands.ModuleCmdletBase.LoadModule(String fileName, String moduleBase, String
prefix, SessionState ss, <br />Boolean& found)
   at Microsoft.PowerShell.Commands.ImportModuleCommand.ProcessRecord()
   at System.Management.Automation.Cmdlet.DoProcessRecord()
   at System.Management.Automation.CommandProcessor.ProcessRecord()
   --- End of inner exception stack trace ---
   at System.Management.Automation.Runspaces.PipelineBase.Invoke(IEnumerable input)
   at System.Management.Automation.Runspaces.Pipeline.Invoke()
   at NuGetConsole.Host.PowerShell.Implementation.PowerShellHost.Invoke(String command, Object input, Boolean
outputResults)
   at NuGetConsole.Host.PowerShell.Implementation.PowerShellHostExtensions.ImportModule(PowerShellHost host,
String modulePath)
   at NuGetConsole.Host.PowerShell.Implementation.PowerShellHost.LoadStartupScriptScripts()
   at NuGetConsole.Host.PowerShell.Implementation.PowerShellHost.Initialize()
   at NuGetConsole.Implementation.Console.Dispatcher.Start()
   at NuGetConsole.Implementation.PowerConsoleToolWindow.MoveFocus(FrameworkElement consolePane)
```

We've contacted the author of the add-in in the hopes of working out a resolution.

Update: We have verified that the latest version of Reflector, 6.5, no longer causes this exception in the console.

Opening Package Manager Console fails with ObjectSecurity exception

You might see these errors when trying to open the Package Manager Console:

```
The following error occurred while loading the extended type data file: Microsoft.PowerShell.Core,
C:\Windows\SysWOW64\WindowsPowerShell\v1.0\types.ps1xml(2977) : Error in type
"System.Security.AccessControl.ObjectSecurity": Exception: The getter method should be public, non void,
static, and have one parameter of type PSObject.
The following error occurred while loading the extended type data file: Microsoft.PowerShell.Core,
C:\Windows\SysWOW64\WindowsPowerShell\v1.0\types.ps1xml(2984) : Error in type
"System.Security.AccessControl.ObjectSecurity": Exception: The getter method should be public, non void,
static, and have one parameter of type PSObject.
The following error occurred while loading the extended type data file: Microsoft.PowerShell.Core,
C:\Windows\SysWOW64\WindowsPowerShell\v1.0\types.ps1xml(2991) : Error in type
"System.Security.AccessControl.ObjectSecurity": Exception: The getter method should be public, non void,
static, and have one parameter of type PSObject.
The following error occurred while loading the extended type data file: Microsoft.PowerShell.Core,
C:\Windows\SysWOW64\WindowsPowerShell\v1.0\types.ps1xml(2998) : Error in type
"System.Security.AccessControl.ObjectSecurity": Exception: The getter method should be public, non void,
static, and have one parameter of type PSObject.
The following error occurred while loading the extended type data file: Microsoft.PowerShell.Core,
C:\Windows\SysWOW64\WindowsPowerShell\v1.0\types.ps1xml(3005) : Error in type
"System.Security.AccessControl.ObjectSecurity": Exception: The getter method should be public, non void,
static, and have one parameter of type PSObject.
The term 'Get-ExecutionPolicy' is not recognized as the name of a cmdlet, function, script file, or operable
program. Check the spelling of the name, or if a path was included, verify that the path is correct and try
again.
```

If so, follow the solution [discussed on StackOverflow](#) to fix them.

The Add Package Library Reference dialog throws an exception if the solution contains InstallShield Limited Edition Project

We have identified that if your solution contains one or more InstallShield Limited Edition project, the **Add Package Library Reference** dialog will throw an exception when opened. There is currently no workaround yet except either removing InstallShield projects or unloading them.

Uninstall Button Greyed out? NuGet Requires Admin Privileges to Install/Uninstall

If you try to uninstall NuGet via the Visual Studio Extension Manager, you may notice that the Uninstall button is disabled. NuGet requires admin access to install and uninstall. Relaunch Visual Studio as an administrator to uninstall the extension. NuGet does not require admin access to use it.

The Package Manager Console crashes when I open it in Windows XP. What's wrong?

NuGet requires Powershell 2.0 runtime. Windows XP, by default, doesn't have Powershell 2.0. You can download the Powershell 2.0 runtime from <http://support.microsoft.com/kb/968929>. After you install it, restart Visual Studio and you should be able to open Package Manager Console.

Visual Studio 2010 SP1 Beta crashes on exit if the Package Manager Console is open.

If you have installed Visual Studio 2010 SP1 Beta, you may notice that if you leave the Package Manager Console open and close Visual Studio, it will crash. This is a known issue of Visual Studio and will be fixed in SP1 RTM release. For now, just ignore the crash or uninstall SP1 Beta if you can.

The element 'metadata' ... has invalid child element exception occurs

If you installed packages built with a pre-release version of NuGet, you might encounter an error message stating "The element 'metadata' in namespace 'schemas.microsoft.com/packaging/2010/07/nuspec.xsd' has invalid child element" when running the RTM version of NuGet with that project. You need to uninstall and then re-install each package using the RTM version of NuGet.

Attempting to install or uninstall results in the error "Cannot create a file when that file already exists."

For some reason, Visual Studio extensions can get in a weird state where you've uninstalled the VSIX extension, but some files were left behind. To work around this issue:

1. Exit Visual Studio
2. Open the following folder (it might be on a different drive on your machine)
C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\Extensions\Microsoft Corporation\NuGet Package Manager<version>\
3. Delete all the files with the *.deleteme* extensions.
4. Re-open Visual Studio

After following these steps, you should be able to continue.

In rare cases, compiling with Code Analysis turned on causes error.

You might get the following error if you installs FluentNHibernate with the Package Manager console and then compile your project with "Code Analysis" turned on.

```
Error 3 CA0058 : The referenced assembly
'NHibernate, Version=3.0.0.2001, Culture=neutral, PublicKeyToken=aa95f207798dfdb4'
could not be found. This assembly is required for analysis and was referenced by:
C:\temp\Scratch\src\MyProject.UnitTests\bin\Debug\MyProject.UnitTests.dll.
MyProject.UnitTests
```

By default, FluentNHibernate requires NHibernate 3.0.0.2001. However, by design NuGet will install NHibernate 3.0.0.4000 in your project and add the appropriate binding redirects so that it will work. Your project will compile just fine if code analysis is not turned on. In contrast to the compiler, code analysis tool doesn't properly follow the binding redirects to use 3.0.0.4000 instead of 3.0.0.2001. You can work around the issue by either installing NHibernate 3.0.0.2001 or tell the code analysis tool to behave the same as the compiler by doing the following:

1. Go to %PROGRAMFILES%\Microsoft Visual Studio 10.0\Team Tools\Static Analysis Tools\FxCop
2. Open FxCopCmd.exe.config and change `AssemblyReferenceResolveMode` from `StrongName` to `StrongNameIgnoringVersion`.
3. Save the change and rebuild your project.

Write-Error command doesn't work inside install.ps1/uninstall.ps1/init.ps1

This is a known issue. Instead of calling Write-Error, try calling throw.

```
throw "My error message"
```

Installing NuGet with restricted access on Windows 2003 can crash

Visual Studio

When attempting to install NuGet using the Visual Studio Extension Manager and not running as an administrator, the “Run As” dialog is displayed with the checkbox labeled “Run this program with restricted access” checked by default.



Clicking OK with that checked crashes Visual Studio. Make sure to uncheck that option before installing NuGet.

Cannot uninstall NuGet for Windows Phone Tools

Windows Phone Tools does not have support for the Visual Studio Extension Manager. In order to uninstall NuGet, run the following command.

```
vsixinstaller.exe /uninstall:NuPackToolsVsix.Microsoft.67e54e40-0ae3-42c5-a949-fdd5739e7a5
```

Changing the capitalization of NuGet package IDs breaks package restore

As discussed at length on [this GitHub issue](#), changing the capitalization of NuGet packages can be done by NuGet support, but causes complications during package restore for users who have existing, differently-cased, packages in their *global-packages* folder. We recommend only requesting a case change when you have a way to communicate with existing users of your package about the break that may occur to their build-time package restore.

Reporting issues

To report NuGet issues, visit <https://github.com/nuget/home/issues>.

NuGet 5.3 Release Notes

11/5/2019 • 2 minutes to read • [Edit Online](#)

NuGet distribution vehicles:

NUGET VERSION	AVAILABLE IN VISUAL STUDIO VERSION	AVAILABLE IN .NET SDK(S)
5.3.0	Visual Studio 2019 version 16.3	3.0.100 ¹
5.3.1	Visual Studio 2019 version 16.3.6	Future version: 3.0.101

¹Installed with Visual Studio 2019 with .NET Core workload

Summary: What's New in 5.3

- [Package Icon can be embedded in the package](#), instead of needing an external URL. - #352
- Improved security with SHA tracking and enforcement for Packages.Config - #7281
- Enable deprecation of obsolete/legacy NuGet Packages [#2867](#) | [Blog post](#) | [Docs](#)

Issues fixed in this release

Bugs

- NuGet packages produced with 3.0.100-preview9 SDK cannot be used by 2.2 SDK users...depending on your timezone [#8603](#)
- Quote " characters in PATH cause "Illegal characters in path" failure in `nuget restore` [#8168](#)
- VS: assemblies are fully ngen-ed not partially ngen-ed - [#8513](#)
- Reduce memory usage (unsubscribe from events) - [#8471](#)
- "Error_UnableToFindProjectInfo" message is not grammatically correct - [#8441](#)
- NU1403 improvements - validate all packages, include the expected/actual sha values - [#8424](#)
- Multiple enumeration in `NuGetPackageManager.PreviewUpdatePackagesAsync` - [#8401](#)
- Revert "public -> internal" change in PluginProcess - [#8390](#)
- IVsPackageSourceProvider.GetSources(...) has ill-defined exception behavior - [#8383](#)
- Make PluginManager constructor public again - [#8379](#)
- Metrics to track the refresh rate of the PM UI - [#8369](#)
- Decrease the number of UI refreshes when installing through the Package Manager UI - [#8358](#)
- Telemetry: datetime values use culture-specific formats - [#8351](#)
- Reduce UI refreshes in browse tab of Package Manager UI #6570 - [#8339](#)
- [Test Failure] "Unable to parse config file" will prompt twice - [#8320](#)
- Raise NU5037 error with good doc page that explains customer fixes (The package is missing the required nuspec file) - [#8291](#)

- Locked-mode restore fails when a project's RuntimeIdentifier is changed - [#8260](#)
- Make the Settings reading in VS lazy - [#8156](#)
- Regression in `Nuget sources add` causes "The ':' character, hexadecimal value 0x3A, cannot be included in a name" errors - [#7948](#)
- NuGet plugin credential providers - hide the process window - [#7511](#)
- Enforce PackagePathResolver is an absolute path - [#7349](#)
- Reduce UI refreshes in install and update tabs of Package Manager UI - [#6570](#)

DCR:

- Update Xamarin frameworks to map to NetStandard 2.1 - [#8368](#)
- Enable copying the contents of package manager "preview window" for install/update - [#8324](#)
- Enable restore on .proj files - [#8212](#)
- Introduce `NUGET_NETFX_PLUGIN_PATHS` and `NUGET_NETCORE_PLUGIN_PATHS` to support configuration of both at same time - [#8151](#)
- Enable multiple versions for a PackageDownload via Version attribute - [#8074](#)
- Add -SolutionDirectory and -PackageDirectory options to nuget.exe pack - [#7163](#)

List of all issues fixed in this release - 5.3

Summary: What's New in 5.3.1

- Plugin: A task was canceled - don't let cancellations affect plugin instantiation - [#8648](#)
- Restore Task cannot be safely run twice in one process (when Credential providers are used) - [#8688](#)

NuGet 5.2 Release Notes

7/24/2019 • 2 minutes to read • [Edit Online](#)

NuGet distribution vehicles:

NUGET VERSION	AVAILABLE IN VISUAL STUDIO VERSION	AVAILABLE IN .NET SDK(S)
5.2.0	Visual Studio 2019 version 16.2	2.1.80X¹ , 2.2.40X²

¹Installed with Visual Studio 2019 with .NET Core workload

²Available as an optional install with Visual Studio 2019 with .NET Core workload

Summary: What's New in 5.2

- Fixed a critical bug that caused occasional NuGet operation failures due to path issues on Linux & Mac - [#7341](#)
- Improved UI responsiveness when browsing packages using the NuGet package manager UI in Visual Studio especially noticeable for slow sources - [#8039](#)
- Tons of reliability fixes for lock file ([#8187](#),[#8160](#),[#8114](#),[#7840](#)) and authentication plugin ([#8300](#),[#8271](#),[#8269](#),[#8210](#),[#8198](#),[#7845](#))

Issues fixed in this release

Bugs

- Perf: Package Manager Console: UI delay updating "Default project" combobox selected value - [#8235](#)
- Perf: Performance improvements in the PM UI - [#8039](#)
- Perf: UI Delay when reading Default Project in PMC - [#6824](#)
- Perf: [vsfeedback] NuGet Update tab freezes for a local package source - [#6470](#)
- Plugins: NuGet waits full handshake timeout if plugin fails to launch or terminates early - [#8300](#)
- Plugins: improve diagnosability of plugin launch failure - [#8271](#)
- Plugins: Issue with nuget.exe discovery of built in plugins - [#8269](#)
- Plugins: cache file is never read - [#8210](#)
- Plugins: "A task was canceled." errors with authentication plugin during restore - [#8198](#)
- Plugins cache not discoverable intermittently on linux platforms - [#7845](#)
- LockFile: with ATF, it has false NU1004 due to a bad target framework equality check - [#8187](#)
- LockFile: '--locked-mode' restore flag not respected if lock file is empty or malformed - [#8160](#)
- LockFile: Don't lowercase projects with custom assembly names in packages lock file - [#8114](#)
- LockFile: Make project reference lower case in lock file - [#7840](#)
- Restore: installing a tampered signed package results in multiple failed install attempts (with repeated output) - [#8175](#)

- VS: solution user options fail to deserialize after NuGet update - [#8166](#)
- dotnet-list-package in a UnitTest project returns an error - [#8154](#)
- Create NuGet package group for VS installer - fixing some VSIX setup problems - [#8033](#)
- GeneratePackageOnBuild should not set NoBuild. - [#7801](#)
- The new option "-SymbolPackageFormat snupkg" generates an error when the .nuspec file contains an explicit assembly reference element - [#7638](#)
- NuGet.targets(498,5): error : Could not find a part of the path '/tmp/NuGetScratch' - [#7341](#)

DCR:

- Add an msbuild property that indicates that PackageDownload is supported - [#8106](#)
- FrameworkReference suppress dependency flow via FrameworkReference.PrivateAssets - [#7988](#)
- Mechanism for supplying runtime.json outside of a package - [#7351](#)

[List of all issues fixed in this release - 5.2 RTM](#)

NuGet 5.1 Release Notes

7/12/2019 • 2 minutes to read • [Edit Online](#)

NuGet distribution vehicles:

NUGET VERSION	AVAILABLE IN VISUAL STUDIO VERSION	AVAILABLE IN .NET SDK(S)
5.1.0	Visual Studio 2019 version 16.1	2.1.70X ¹ , 2.2.30X ²

¹Installed with Visual Studio 2019 with .NET Core workload

²Available as an optional install with Visual Studio 2019 with .NET Core workload

Summary: What's New in 5.1

- Support to skip a package push if it already exists to allow for better integration with CI/CD workflows - [#1630](#)
- Visual Studio now provides a convenient link to the the package's nuget.org gallery page - [#5299](#)
- Support for new .NET Core 3.0 assets such as [Targeting Packs](#) and [Runtime Packs](#)
 - NuGet pack and restore support for FrameworkReferences to enable targeting and runtime package references - [#7342](#)
 - Support "download only" package scenario with PackageDownload - [#7339](#)
 - Exclude runtime and targeting packs from search results & restore graph using PackageType - [#7337](#)

Issues fixed in this release

Bugs

- Plugins: exception details lost during plugin creation - [#8057](#)
- PackageReference range with exclusive lower bound does not work if the lower bound is present on one of the sources. - [#8054](#)
- Improve IsPackableFalseError message - [#8021](#)
- Packages Lock File - regenerate lock file when project graph changes - [#8019](#)
- ProjectSystem bug: Nuget Packages getting auto removed - [#8017](#)
- Add a target for returning the FrameworkReference similar to CollectPackageDownloads and CollectPackageReferences - [#8005](#)
- HTTP cache: RepositoryResources resource is not cached in a versioned way - [#7997](#)
- Logging: exception callstacks are not reported with detailed verbosity - [#7955](#)
- Change all NuGet Docs URLs to use HTTPS - [#7950](#)
- Improve NU3024 warning message - [#7933](#)
- lock file not updating when packagereference removed - [#7930](#)
- Improve the error case handling when validating licenseurl and license element in nuspec - [#7915](#)

- PM UI - right click on tab header and clicking "Open file location" results in error - [#7913](#)
- Plugins: log when plugin process exits - [#7907](#)
- Plugins: high collision rate in logging datetime values - [#7899](#)
- Manifest.ReadFrom fails on any nuspec with LicenseExpression - [#7894](#)
- RestoreLockedMode: Unexpected NU1004 when ProjectReference refers to a project with custom AssemblyName - [#7889](#)
- Better error message when the plugin startup fails with an exception - [#7857](#)
- When doing a NoOp restore, avoid *.dgspec.json write in obj directory - [#7854](#)
- GeneratePathProperty=true fails to generate property on case mismatch - [#7843](#)
- Settings: illegal character in package source path can crash VS - [#7820](#)
- If lock file is deleted, restore does not generate lock file on NoOp - [#7807](#)
- License URL and license causes read error with Metadata - [#7547](#)
- Unhandled exceptions in V2FeedParser - [#7523](#)
- nuget.exe returns exit code zero for invalid arguments - [#7178](#)
- Update Errors and warning docs to reflect signing related scenarios - [#6498](#)
- Assets file should use relative paths to enable moving projects more easily - [#4582](#)

DCRs

- Plugins: enable diagnostic logging - [#7859](#)
- Make Tizen 6 map to NetStandard 2.1 - [#7773](#)

[List of all issues fixed in this release - 5.1 RTM](#)

NuGet 5.0 Release Notes

11/5/2019 • 4 minutes to read • [Edit Online](#)

NuGet distribution vehicles:

NUGET VERSION	AVAILABLE IN VISUAL STUDIO VERSION	AVAILABLE IN .NET SDK(S)
5.0.0	Visual Studio 2019 version 16.0	2.1.602¹, 2.2.202²
5.0.2	Visual Studio 2019 version 16.0.4	2.1.60X¹, 2.2.20X²

¹Installed with Visual Studio 2019 with .NET Core workload

²Available as an optional install with Visual Studio 2019 with .NET Core workload

Summary: What's New in 5.0

- Support for restoring [filtered solutions](#) in Visual Studio 2019 - [#5820](#)
- `BuildTransitive` folder enables packages to transitively contribute targets/props to the host project - [#6091](#)
- Better support for PackageReference scenarios in NuGet IVs APIs - [#7005](#), [#7493](#)
- `nuget.exe pack project.json` has been deprecated - [#7928](#)
- Gen 1 Credential Provider plugin has been superseded by [Gen 2](#) and will soon be deprecated - [#7819](#)

Issues fixed in this release

Bugs

- When doing a NoOp restore, avoid *.dgspec.json write in obj directory - [#7854](#)
- Permissions on files created inside ~/.nuget are too open - [#7673](#)
- `dotnet list package --outdated` doesn't work with sources that need auth - [#7605](#)
- NuGet.VisualStudio.IVsPackageInstaller - calling on a project with no package references always uses packages.config, even if the default is set to PackageReference - [#7005](#)
- PMC: Update-Package reinstall fails ("Unable to find package") on delisted packages. - [#7268](#)
- Add third party notice in our repo and VSIX - [#7409](#)
- NuGet.VisualStudio.IVsPackageInstaller.InstallPackage should install latest version when no version given - [#7493](#)
- --interactive support for dotnet nuget push - [#7519](#)
- When restoring with lock file, NU1603 warning shouldn't be raised. - [#7529](#)
- NuGet should not print project path during restore with minimal logging - [#7647](#)
- --interactive support for dotnet remove package - [#7727](#)
- Add back NuGet.Packaging.Core with TypeForwardedTo attrs - [#7768](#)
- plugins_cache needs shorter path to work well - [#7770](#)

- Prefer path for msbuild discovery if user didn't ask for specific msbuild version - [#7786](#)
- `nuget.exe /?` should list correct msbuild versions - [#7794](#)
- NuGet.targets(498,5): error : Could not find a part of the path '/tmp/NuGetScratch - on mono - [#7793](#)
- restore unnecessarily enumerates the contents of all versions of referenced package in the machine cache - [#7639](#)
- MSBuild auto-detection always selects 16.0 after installing VS 2019 Preview - [#7621](#)
- dotnet list package on a solution outputs duplicate entries for framework - [#7607](#)
- Exception "Empty path name is not legal" when calling IVsPackageInstaller.InstallPackage on old projects and packages folder does not exist. - [#5936](#)
- msbuild /t:restore minimal verbosity should be more minimal - [#4695](#)
- VS 16.0's NuGet UI has unreadable tabs due to color problems - [#7735](#)
- NuGet.Core & NuGet.Clients License.txt clarification - [#7629](#)
- Restore unnecessarily enumerates global package folder in attempt to determine type - [#7596](#)
- Errors from lock file enforcement should show up in Error List Window - [#7429](#)
- Fix NuGet.Configuration issues - [#7326](#)
- Adapt to MSBuild updating its install location - [#7325](#)
- NuGet.Build.Tasks.Pack should be a development dependency - [#7249](#)
- Add pack extension point for including debug symbols - [#7234](#)
- `dotnet pack` should preserve dependency version range in the created nupkg (even if floating version is used) - [#7232](#)
- `dotnet restore` fails on authenticated source when user-level config also has source - [#7209](#)
- Pack should not restrict the set of BuildActions for content files - [#7155](#)
- Using a ProjectReference which requires AssetTargetFallback to succeed, should warn. - [#7137](#)
- Deadlock due to threading issues when calling into CPS (CommonProjectSystem) - [#7103](#)
- `dotnet add package` doesn't use credentials from global config for a source specified in local config - [#6935](#)
- Threading issues with MEF being called on async code paths - [#6771](#)
- Signing: error reported twice and without call stack - [#6455](#)
- Installing a signed package with untrusted signing certificate should show error - [#6318](#)
- NuGet restore improperly NoOps when 2 projects are sharing obj directory - [#6114](#)
- Cannot use PAT with `dotnet restore` on Linux with packages from authenticated feed - [#5651](#)
- dotnet restore fails due to disabled machine wide feed - [#5410](#)

DCRs

- Warn of future removal of "dotnet pack project.json" - [#7928](#)
- Add a deprecation warning for Gen1 credential plugin - [#7819](#)

- Signing: Enabled Repo to require client verification of every package as repo signed -- via RepositorySignatures/5.0.0 resource - [#7759](#)
- limit http request number per source through NuGet.Config - [#4538](#)
- NuGet should target Net472 (to help Cleanup the 16.0 build of the VSIX) - [#7143](#)
- PMC: Remove OpenPackagePage command - [#7384](#)
- Make NetCoreApp 3.0 map to NetStandard 2.1 - [#7762](#)
- Add netstandard2.0 support to NuGet.* packages - [#6516](#)
- Allow package authors to define build assets transitive behavior - [#6091](#)
- Support VS 2019 Solution Filter feature. Also supports project not in solution, or unloaded projects. Need to restore complete solution (via CLI or VS) first - [#5820](#)
- NuGet 5.0 assemblies to require .NET 4.7.2 (via TFM change) - [#7510](#)
- NuGetLicenseData from NuGet.Packaging should be a public type. Update license metadata ingested from spdx. - [#7471](#)
- Remove obsolete Settings APIs - [#7294](#)
- Workaround restore timeouts on systems with 1 cpu - [#6742](#)
- NuGet prefers NTLM auth even if there are credentials in NuGet.config - add config option to filter auth types for credentials - [#5286](#)
- Enable EmbedInteropTypes for PackageReference (matching Packages.Config capability) - [#2365](#)

List of all issues fixed in this release - 5.0 RTM

Summary: What's New in 5.0.2

- Security (when run via dotnet.exe or mono.exe) - The obj folder should be created with correct permissions [#7908](#)
- nuget.exe restore on mono/MacOS fails with custom nuget.config and `PackageSignatureValidity: False` [#8011](#)

Known issues

Packages in FallbackFolders installed by .NET Core SDK are custom installed, and fail signature validation. - [#7414](#)

Issue

When using dotnet.exe 2.x to restore a project that multi-targets netcoreapp 1.x and netcoreapp 2.x, the fallback folder is treated as a file feed. This means, when restoring, NuGet will pick the package from the fallback folder and try to install it into the global packages folder and do the usual signing validation which fails.

Workaround

Disable the usage of the fallback folder by setting the `RestoreAdditionalProjectSources` to nothing:

```
<RestoreAdditionalProjectSources></RestoreAdditionalProjectSources>
```

Use this with caution as packages that would be restored from the fallback folder will now be downloaded from NuGet.org.

NuGet 4.9 Release Notes

3/25/2019 • 4 minutes to read • [Edit Online](#)

NuGet distribution vehicles:

NUGET VERSION	AVAILABLE IN VISUAL STUDIO VERSION	AVAILABLE IN .NET SDK(S)
4.9.0	Visual Studio 2017 version 15.9.0	2.1.500, 2.2.100
4.9.1	n/a	n/a
4.9.2	Visual Studio 2017 version 15.9.4	2.1.502, 2.2.101
4.9.3	Visual Studio 2017 version 15.9.6	2.1.504, 2.2.104

Summary: What's New in 4.9.0

- Signing: Enable ClientPolicies to require use of a set of Trusted Authors and Repositories listed in NuGet.Config - [#6961, blog post](#)
- create ".snupkg" files to contain symbols in pack -- enhance push to understand nuget protocol to accept snupkg files for symbol server - [#6878, blog post](#)
- NuGet credential plugin V2 - [#6642](#)
- Self-Contained NuGet Packages - License - [#4628, announcement](#)
- Enable opt-in "GeneratePathProperty" metadata on a PackageReference to generate a per package MSBuild property to "Foo.Bar\1.0" directory - [#6949](#)
- Improve customer success with NuGet operations - [#7108](#)
- Enable repeatable package restores using a lock file - [#5602, announcement, blog post](#)

Issues fixed in this release

- Warnings elevated to errors (via WarnAsErrors) raised by PackageExtraction should never leave extracted package around - [#7445](#)
- Badly signed packages should not end up in the global packages folder - [#7423](#)
- binding redirect generation should not skip facade assemblies - [#7393](#)
- VersionRange Equals doesn't compare floating ranges - [#7324](#)
- Restore: performance regression using new .NET Core 2.1 HTTP stack - [#7314](#)
- Update of a Package should not modify PrivateAssets of a PackageReference - [#7285](#)
- Signing: signing should fail if a package has too many package entries (>65534) - [#7248](#)
- "dotnet nuget push" codepath should support the new credential provider - [#7233](#)
- Support executing plugins with invariant culture (as happens in docker) - [#7223](#)
- nuget sources add should not delete credentials from NuGet.config - [#7200](#)

- installing a devDependency PackageReference should default to excludeassets=compile - [#7084](#)
- fix migrator option to be displayed for all projects and show error if project is incompatible - [#6958](#)
- "dotnet add package" should commit the restore it performs to the assets file - [#6928](#)
- Signing: improve signing related error messages - [#6906](#)
- [Test Failure][zh-TW]String "Package Manager Console" doesn't localize on Package Manager Console - [#6381](#)
- Error message around "Unable to find project information" should be a little more specific inside VS - [#5350](#)
- Unhelpful error message when incorrectly using nuspec version tag of nuget pack - [#2714](#)
- DCR - Signing: support NuGet protocol: RepositorySignatures/4.9.0 resource - [#7421](#)
- DCR - .nupkg.metadata file will now be created during package extraction - contains "content-hash" - [#7283](#)
- DCR - Skip authenticode verification for plugins while executing on Mono - [#7222](#)

[List of all issues fixed in this release 4.9.0](#)

Summary: What's New in 4.9.1

- Add support for reading a writing to the nuget.config via a new command trusted-signers - [#7480](#)

Issues fixed in this release

- Fix license link generation - [#7515](#)
- Error codes regression for validating signatures - [#7492](#)
- NuGet.Build.Tasks.Pack package does not have license information - [#7379](#)

[List of all issues fixed in this release 4.9.1](#)

Summary: What's New in 4.9.2

Issues fixed in this release

- VS/dotnet.exe/nuget.exe/msbuild.exe restore doesn't use credentials when source name contains a whitespace - [#7517](#)
- LicenseAcceptanceWindow and LicenseFileDialog Accessibility issues - [#7452](#)
- Fix FormatException in DateTime.Parse from DateTimeConverter - [#7539](#)

[List of all issues fixed in this release 4.9.2](#)

Summary: What's New in 4.9.3

Issues fixed in this release

"Repeatable Package Restores Using a Lock File" Issues

- Locked mode not working as hash is calculated incorrectly for previously cached packages - [#7682](#)
- Restore resolves to a different version than defined in `packages.lock.json` file - [#7667](#)
- '--locked-mode / RestoreLockedMode' causes spurious Restore failures when ProjectReferences are involved - [#7646](#)
- MSBuild SDK resolver tries to validate SHA for a SDK package which fails restore when using

packages.lock.json - [#7599](#)

"Lock Down Your Dependencies Using Configurable Trust Policies" Issues

- dotnet.exe should not evaluate trusted-signers while signed packages are not supported - [#7574](#)
- Order of trustedSigners in config file affects trust evaluation - [#7572](#)
- Can't implement ISettings [Caused by refactoring of settings APIs to support Trust Policies feature] - [#7614](#)

"Improved Debugging Experience" Issues

- Cannot publish symbol package for .NET Core Global Tool - [#7632](#)

"Self-Contained NuGet Packages - License" Issues

- Error building symbol .snupkg package when using embedded license file - [#7591](#)

[List of all issues fixed in this release 4.9.3](#)

Summary: What's New in 4.9.4

- Security Fix: Permissions on files created inside ~/nuget are too open [#7673 CVE-2019-0757](#)

Known issues

dotnet nuget push --interactive gives an error on Mac. - [#7519](#)

Issue

The `--interactive` argument is not being forwarded by the dotnet cli and results in the error

```
error: Missing value for option 'interactive'
```

Workaround

Run any other dotnet command with the interactive option such as `dotnet restore --interactive` and authenticate. The authentication then might be cached by the credential provider. Then run `dotnet nuget push`.

Packages in FallbackFolders installed by .NET Core SDK are custom installed, and fail signature validation. - [#7414](#)

Issue

When using dotnet.exe 2.x to restore a project that multi-targets netcoreapp 1.x and netcoreapp 2.x, the fallback folder is treated as a file feed. This means, when restoring, NuGet will pick the package from the fallback folder and try to install it into the global packages folder and do the usual signing validation which fails.

Workaround

Disable the usage of the fallback folder by setting the `RestoreAdditionalProjectSources` to nothing.

```
<RestoreAdditionalProjectSources/> Use this with caution as it will cause a lot of packages to be downloaded from NuGet.org which otherwise would have been restored from the fallback folder.
```

NuGet 4.8 Release Notes

11/5/2019 • 4 minutes to read • [Edit Online](#)

Visual Studio 2017 15.8 RTW comes with NuGet 4.8 functionality.

Command line versions of the same functionality are also available:

- NuGet.exe 4.8 - nuget.org/downloads
- DotNet.exe - [.NET Core SDK 2.1.400](https://dotnet.microsoft.com/download/dotnet-core/2.1.400)

Summary: What's New in 4.8.0

- NuGet.exe now supports longfilenames on Windows 10 - [#6937](#)
- Authentication plugins now work across MsBuild, DotNet.exe, NuGet.exe and Visual Studio, including cross platform. The first generation of authentication plugins were not supported in MsBuild, DotNet.exe. Note: VS 2017 15.9 Preview builds have a VSTS authentication plugin included. [#6486](#)
- MsBuild's SDK Resolver now builds as part of NuGet and installs with NuGet tools for VS. This will avoid versions getting out sync. [#6799](#)
- PackageReference now supports DevelopmentDependency metadata - [#4125](#)

Summary: What's New in 4.8.2

- Security Fix: Permissions on files created inside `~/.nuget` are too open [#7673 CVE-2019-0757](#)

Known issues

Installing signed packages on a CI machine or in an offline environment takes longer than usual

Issue

If the machine has restricted internet access (such as a build machine in a CI/CD scenario), installing/restoring a signed nuget package will result a warning ([NU3028](#)) since the revocation servers are not reachable. This is expected. However, in some cases, this may have unintended consequences such as the package install/restore taking longer than usual.

Workaround

Update to Visual Studio 15.8.4 and NuGet.exe 4.8.1 where we introduced an environment variable to switch the revocation check mode. Setting the `NUGET_CERT_REVOCATION_MODE` environment variable to `offline` will force NuGet to check the revocation status of the certificate only against the cached certificate revocation list, and NuGet will not attempt to reach revocation servers. When the revocation check mode is set to `offline`, the warning will be downgraded to an info.

WARNING

It is not recommended to switch the revocation check mode to offline under normal circumstances. Doing so will cause NuGet to skip online revocation check and perform only an offline revocation check against the cached certificate revocation list which may be out of date. This means packages where the signing certificate may have been revoked, will continue to be installed/restored, which otherwise would have failed revocation check and would not have been installed.

The `Migrate packages.config to PackageReference...` option is not available in the right-click context menu

Issue

When a project is first opened, NuGet may not have initialized until a NuGet operation is performed. This causes

the migration option to not show up in the right-click context menu on `packages.config` or `References`.

Workaround

Perform any one of the following NuGet actions:

- Open the Package Manager UI - Right-click on `References` and select `Manage NuGet Packages...`
- Open the Package Manager Console - From `Tools > NuGet Package Manager`, select `Package Manager Console`
- Run NuGet restore - Right-click on the solution node in the Solution Explorer and select `Restore NuGet Packages`
- Build the project which also triggers NuGet restore

You should now be able to see the migration option. Note that this option is not supported and will not show up for ASP.NET and C++ project types. Note: This has been fixed in VS 2017 15.9 Preview 3

Issues fixed in this release

Bugs

Signing

- Signing: Installing signed package in offline environment [#7008](#) -- Fixed in 4.8.1
- Signing: incorrect URL check - [#7174](#)
- Signing: check package integrity in RepositorySignatureVerifier when package is repository countersigned - [#6926](#)
- "Package Integrity check failed." should have package ID in message (and error code) - [#6944](#)
- Repository signed package verification allows packages signed by different certificate - [#6884](#)
- NuGet - Signing - Timestamp URL can not be `https:// ?` - [#6871](#)
- Don't NullRef in NuSpec packing scenario, also improve options - [#6866](#)
- Memory is invalid while updating signer info when adding timestamp to countersignature - [#6840](#)
- Signing: remove CTL exceptions - [#6794](#)
- Signing: contentUrl MUST be HTTPS - [#6777](#)
- Signing: `SignedPackageVerifierSettings.VSClientDefaultPolicy` is unused - [#6601](#)

Pack

- restore and build should not be needed when using `dotnet.exe` to pack `nuspec` - [#6866](#)
- Allow empty replacement tokens in `NuspecProperties` - [#6722](#)
- `PackTask` throws `NullReferenceException` when `NuspecProperties` is specified - [#4649](#)

Accessibility

- [Accessibility] String 'Prerelease' under package button is covered by its package description in PM UI - [#4504](#)
- [Accessibility] Package source drop down and settings button truncated when selecting 'Microsoft Visual Studio Offline Packages' in PM UI - [#4502](#)

Powershell Management Console (PMC)

- `Update-Package` ignores `PackageReference` version range - [#6775](#)
- `Update-Package -reinstall` solution wide issue - [#3127](#)
- `Update-Package [packagename] -reinstall` reinstalls all packages instead of just the named one - [#737](#)
- Can update to unlisted NuGet package from the Package Manager Console - [#4553](#)

Misc

- To fix `NuGet update self` `NuGet.Commandline.nupkg` should not be semver2.0 - [#7116](#)
- Improve experiences with NU1107 install failures - [#7107](#)
- The serialization of `GetAuthenticationCredentialRequest` is incorrect - [#6983](#)
- NuGet Visual Studio AsyncPackage fails to load when initialized off the UI thread - [#6976](#)

- Restore is reporting misleading errors stating project.json is needed - [#6959](#)
- Package manager UI : preview changes, ok button not automatically useable by keyboard - [#6893](#)
- RestoreSources are ignored for project with p2p references - [#6776](#)
- Creating unit test project using .NET Framework template will open older project model with packages.config - [#6736](#)
- allow project reference to override package dependency - [#6536](#)
- Expose NoDefaultExcludes in MSBuild task - [#6450](#)
- Status message for "Clear All NuGet Cache(s)" can be hidden on window resize - [#5938](#)

[List of all issues fixed in this release](#)

NuGet 4.7 Release Notes

11/5/2019 • 2 minutes to read • [Edit Online](#)

Visual Studio 2017 15.7 RTW comes with [NuGet 4.7.0](#).

Summary: What's New in 4.7.0

- We have augmented package signing to enable [Repository Signed packages](#)
- With Visual Studio Version 15.7, we have introduced the capability to [migrate existing projects that use the packages.config format to use PackageReference](#) instead.

Summary: What's New in 4.7.2

- Security Fix: Permissions on files created inside ~/.nuget are too open [#7673 CVE-2019-0757](#)

Summary: What's New in 4.7.3

- Security Fix: Files inside of NUPKGS can have a relative path above the NUPKG directory [#7906](#)

Known issues

The [Migrate packages.config to PackageReference...](#) option is not available in the right-click context menu

Issue

When a project is first opened, NuGet may not have initialized until a NuGet operation is performed. This causes the migration option to not show up in the right-click context menu on [packages.config](#) or [References](#).

Workaround

Perform any one of the following NuGet actions:

- Open the Package Manager UI - Right-click on [References](#) and select [Manage NuGet Packages...](#)
- Open the Package Manager Console - From [Tools > NuGet Package Manager](#), select [Package Manager Console](#)
- Run NuGet restore - Right-click on the solution node in the Solution Explorer and select [Restore NuGet Packages](#)
- Build the project which also triggers NuGet restore

You should now be able to see the migration option. Note that this option is not supported and will not show up for ASP.NET and C++ project types.

Issues with .NET Standard 2.0 with .NET Framework & NuGet

.NET Standard & its tooling was designed such that projects targeting .NET Framework 4.6.1 can consume NuGet packages & projects targeting .NET Standard 2.0 or earlier. [This document](#) summarizes the issues around that scenario, the plan for addressing them, and workarounds you can deploy with today's state of the tooling.

Top issues fixed in this release

Bugs

- NuGet runs into a deadlock in .Net Core project system (new regression). - [#6733](#)
- Pack: PackagePath is constructed incorrectly if TfmSpecificPackageFile is used with globbing paths - [#6726](#)
- Pack: web api project cannot create package unless ispackable is explicitly set. - [#6156](#)

- VS UI and PMC take 30min to see new package (nuget.exe sees it right away) - [#6657](#)
- Signing: SignatureUtility.GetCertificateChain(...) does not check all chain statuses - [#6565](#)
- Signing: improve DER GeneralizedTime handling - [#6564](#)
- Signing: VS does not show a NU3002 error when installing a tampered package - [#6337](#)
- lockFile.GetLibrary is case sensitive - [#6500](#)
- Install/update restore code and Restore code paths are not consistent - [#3471](#)
- Solution PackageManager Version ComboBox can select separator via keyboard - [#2606](#)
- Unable to load the service index for source `https://www.myget.org/F/<id>` --->
System.Net.Http.HttpRequestException: Response status code does not indicate success: 403 (Forbidden) - [#2530](#)

DCRs

- Emit X-NuGet-Session-Id header to correlate across requests [feature proposal] - [#5330](#)
- Expose a way to wait on running restore operation running in Visual Studio via IVs apis. - [#6029](#)
- NuGet.exe -NoServiceEndpoint will avoid appending service url suffix - [#6586](#)
- add commit hash to informational version - [#6492](#)
- Signing: enable removal of repository signature/countersignature - [#6646](#)
- Signing: API for stripping repository signature/countersignature - [#6589](#)
- Log source information in VS - [#6527](#)
- Filter /tools on only TFM and RID, so the settings XML can be put in /tools folder - [#6197](#)
- Warn when Pack command excludes a file that starts with . - [#3308](#)

[List of all issues fixed in this release](#)

NuGet 4.6 Release Notes

3/25/2019 • 2 minutes to read • [Edit Online](#)

Visual Studio 2017 15.6 RTW comes with [NuGet 4.6.0](#).

Summary: What's New in 4.6.0

- We have added support for [signing packages](#).
- Visual Studio 2017 and nuget.exe now verifies package integrity before installing, restoring packages for [signed packages](#).
- We have improved performance of successive restores.

Summary: What's New in 4.6.3

- Security Fix: Permissions on files created inside ~/nuget are too open [#7673 CVE-2019-0757](#)

Summary: What's New in 4.6.4

- Security Fix: Files inside of NUPKGs can have a relative path above the NUPKG directory [#7906](#)

Known issues

Issues with .NET Standard 2.0 with .NET Framework & NuGet

.NET Standard & its tooling was designed such that projects targeting .NET Framework 4.6.1 can consume NuGet packages & projects targeting .NET Standard 2.0 or earlier. [This document](#) summarizes the issues around that scenario, the plan for addressing them, and workarounds you can deploy with today's state of the tooling.

Top issues fixed in this release

Performance fixes

- Don't write asset files when there is no change - [#6491](#)
- Restore causes extra MSBuild evaluations when child projects' TFM do not match with the parent project's - [#6311](#)
- Improve NoOp restore perf by optimizing dependency graph spec creation - [#6252](#)

Bugs

- Push to local folder leaves nupkg locked - [#6325](#)
- NuGet Plugin implementation: multiple issues - [#6149](#)
- UIHang - Remove query service call from MEF initialization in VS SolutionManager - [#6110](#)
- Error reporting exception for cancelled package download task - [#6096](#)
- NuGet.exe replaces '+' with '%2B' in assembly name - [#5956](#)
- Fn+F1 does not take to the right help page for PM UI and Console - [#5912](#)
- VS NuGet writes absolute paths into project files under specific circumstances - [#5888](#)
- Fix 4.3 regression - Placeholders \$product\$ and \$AssemblyGuid\$ not replaced in contentfile through transformation - [#5880](#)
- dotnet restore with multiple sources crashes - [#5817](#)
- Pack should re-evaluate project versions to allow git versioning - [#4790](#)

- Improve hard to understand errors when you install an incompatible package - [#4555](#)
- TemplateWizard needs option to install packages as PackageReferences - [#4549](#)
- Package-delivered props files ignored when MSBuild.exe is run from outside a Developer Command Prompt - [#4530](#)
- Fix poor error message when referencing .NET Standard Library that is not applicable to project - [#4423](#)
- dotnet add package fails for package targeting portable profile with little guidance - [#4349](#)
- dotnet pack - version suffix missing from ProjectReference - [#4337](#)
- Build errors and VS crash with .NET Core template - [#3973](#)
- Unable to load the service index for source https:/* - [#3681](#)
- nuget.exe list -allversions doesn't work - [#3441](#)
- Misleading dependency resolution error message - [#2984](#)
- nuget.exe restore doesn't produce .props and .targets files for .msbuildproj (regression in v3.3.0-3.4.4 upgrade) - [#2921](#)
- UI delay when updating a NuGet package with XAML file open - [#2878](#)
- WebSite project from IIS fails with illegal characters in path - [#2798](#)
- Nuget add hangs on CentOS - [#2708](#)
- Restore with packagesavemode -nupkg fails for json.net - [#2706](#)
- Package Manager filter not available in vs output window for restore command - [#2704](#)

[List of all issues fixed in this release](#)

NuGet 4.5 Release Notes

3/25/2019 • 2 minutes to read • [Edit Online](#)

Visual Studio 2017 15.5 RTW comes with [NuGet 4.5 RTM](#).

Summary: What's New in 4.5.0

Summary: What's New in 4.5.2

- Security Fix: Permissions on files created inside `~/.nuget` are too open [#7673 CVE-2019-0757](#)

Summary: What's New in 4.5.3

- Security Fix: Files inside of NUPKGs can have a relative path above the NUPKG directory [#7906](#)

Known issues

Issues with .NET Standard 2.0 with .NET Framework & NuGet

.NET Standard & its tooling was designed such that projects targeting .NET Framework 4.6.1 can consume NuGet packages & projects targeting .NET Standard 2.0 or earlier. [This document](#) summarizes the issues around that scenario, the plan for addressing them, and workarounds you can deploy with today's state of the tooling.

You are unable to view, add, or update DotNetCLITools, using Nuget Package Manager

Issue

NuGet Package Manager does not display and does not allow add/update of DotNetCLITools. [NuGet#4256](#)

Workaround

DotNetCLIToolReferences must be manually edited in your project file.

Retargeting target framework version may lead to incomplete Intellisense

Issue

Retargeting target framework version may lead to incomplete Intellisense, in Visual Studio. This happens when you are using PackageReferences as the package manager format. [NuGet#4216](#)

Workaround

Do a manual restore.

A package in a .NET Core project that contains an assembly with an invalid signature, can trigger an infinite restore loop

Issue

Occasionally, when you use a package that contains an assembly with an invalid signature or when the package version is set with 'DateTime' ticker, it causes the package auto-restore to run in an infinite loop [dotnet/project-system#1457](#).

Workaround

There is no workaround at this time.

Issues fixed in NuGet 4.5 RTM timeframe

For issues fixed in NuGet 4.4 RTM, please refer to [NuGet 4.4 RTM Release Notes](#)

Features

- Disable auto-push of symbols package - [#6113](#)

Bugs

- [Regression] in 15.5p1: Portable0.0 is skipped - [#6105](#)
- Assets from packages are missing after restore - [#5995](#)
- Plugin credential providers do not work with URIs containing spaces - [#5982](#)
- If package failed to restore, error should be printed in the output even with Minimal verbosity ON - [#5658](#)
- dotnet
 - dotnetcore restore at solution-level doesn't follow ProjectReference with ReferenceOutputAssembly of false leading to random build failures - [#5490](#)
- Auto-complete in PMC works incorrectly with object methods - [#4800](#)
- nuget.exe restore fails with Visual Studio 2015 toolset - [#4713](#)
- perf - pmc is expensive to instantiate in vs2017 - [#4205](#)
- Slow to get dependency information on slow connection - [#4089](#)
- uninstall-package w/ -RemoveDependencies will fail if multiple packages share a common dependency - [#4026](#)
- Finalize NuGet.Core.nupkg for publishing - [#3581](#)
- NuGet pack resolves dependency ID from directory name when -IncludeProjectReferences is used for csproj + project.json - [#3566](#)
- The type initializer for 'NuGet.ProxyCache' threw an exception - [#3144](#)
- nuget restore performance issue with kudu - [#3087](#)
- UI Client fails to show any error or warning when search is ahead of registration blobs - [#2149](#)
- Get-Packages -Updates generates an incorrect query - [#2135](#)

Links to GitHub issues fixed in 4.5 RTM

[Issues list](#)

NuGet 4.4 Release Notes

3/25/2019 • 4 minutes to read • [Edit Online](#)

Visual Studio 2017 15.4 RTW comes with NuGet 4.4 RTM.

Summary: What's New in 4.4.0

Summary: What's New in 4.4.2

- Security Fix: Permissions on files created inside `~/.nuget` are too open [#7673 CVE-2019-0757](#)

Summary: What's New in 4.4.3

- Security Fix: Files inside of NUPKGs can have a relative path above the NUPKG directory [#7906](#)

Known issues

Issues with .NET Standard 2.0 with .NET Framework & NuGet

.NET Standard & its tooling was designed such that projects targeting .NET Framework 4.6.1 can consume NuGet packages & projects targeting .NET Standard 2.0 or earlier. [This document](#) summarizes the issues around that scenario, the plan for addressing them, and workarounds you can deploy with today's state of the tooling.

While using Package Manager Console, 'Enter' key may not work

Issue

Occasionally, the enter key does not work in the Package Manager Console. If you see this, please check out the progress on the fix, and provide any additional helpful information about your repro steps. [NuGet#4204](#)

[NuGet#4570](#)

Workaround

Restart Visual Studio and open the PMC before opening the solution. Alternatively, try deleting the `project.lock.json` and restoring again.

You are unable to view, add, or update DotNetCLITools, using Nuget Package Manager

Issue

NuGet Package Manager does not display and does not allow add/update of DotNetCLITools. [NuGet#4256](#)

Workaround

DotNetCLIToolReferences must be manually edited in your project file.

Retargeting target framework version may lead to incomplete Intellisense

Issue

Retargeting target framework version may lead to incomplete Intellisense, in Visual Studio. This happens when you are using `PackageReferences` as the package manager format. [NuGet#4216](#)

Workaround

Do a manual restore.

A package in a .NET Core project that contains an assembly with an invalid signature, can trigger an infinite restore loop

Issue

Occasionally, when you use a package that contains an assembly with an invalid signature or when the package version is set with 'DateTime' ticker, it causes the package auto-restore to run in an infinite loop (dotnet/project-

system#1457).

Workaround

There is no workaround at this time.

Issues fixed in NuGet 4.4 RTM timeframe

[NuGet 4.3 RTM Release Notes](#) - Lists all the issues fixed for NuGet 4.3 RTM

Features

- Support for Lightweight Solution Load in PMC and NuGet PM UI scenarios - [#5180](#)
- The msbuild pack target should have a public hook for running user targets before itself - [#5143](#)
- Feature: Add dependencyVersion switch to nuget install - [#1806](#)
- uap10.0.TODO.0 should map to .NET Standard 2.0 for NuGet - [#5684](#)
- Support Visual Studio Build Tools SKU with msbuild /t:restore - [#5562](#)
- During restore, generate an error if .NET 4.6.1 support for .NET Standard 2.0 is required but not installed - [#5325](#)
- Package ID prefix reservation client UI - [#5572](#)
- deliver localized nuget components to support dotnet.exe localization - [#4336](#)

Bugs

- Different project path casings can cause restore to lose PackageReferences - [#5855](#)
- Move error codes with warning numbers to error range - [#5824](#)
- Misleading error when .NET Standard version is not known to be compatible with target framework - [#5818](#)
- Test files with confusing licenses - [#5776](#)
- Missing license headers in EndToEnd test templates - [#5774](#)
- packages.config restore shows errors as NU1000 - [#5743](#)
- nuget.exe install should have DisableParallelProcessing on mono - [#5741](#)
- nuget.exe install incorrectly disables caching - [#5737](#)
- VS Running the restore command for packages.config when Restore is disabled displays incorrect message - [#5718](#)
- VS; Running the restore command when Restore is disabled displays a confusing message - [#5659](#)
- GetRestoreDotnetCliToolsTask fails when missing version metadata - [#5716](#)
- dotnet
 - dotnetcore add package can clear empty lines from a csproj - [#5697](#)
- Source names of credential settings in NuGet.Config are case sensitive - [#5695](#)
- Enabling GeneratePackageOnBuild deleted my entire history of packages - [#5676](#)
- Restore will not restore mono.cecil or semver packages, but all other packages get restored. - [#5649](#)
- Errors and Warnings - bad error when a source is unavailable. - [#5644](#)
- [DesignConsistency] NuGet Installation status text doesn't look correct on dark theme currently. - [#5642](#)

- Update packages at solution updates/installs for all the projects - [#5508](#)
- dotnet
 - dotnetcore pack behaves differently depending on TargetFramework vs TargetFrameworks - [#5281](#)
- Included DLLs inside Tools folder throw warnings - [#5020](#)
- NuGet.ContentModel consumes too much memory for string operations - [#4714](#)
- RuntimeEnvironmentHelper.IsLinux returns true for OSX - [#4648](#)
- 'dotnet pack' puts nuspec under obj instead of obj\Debug - [#4644](#)
- Nuget extremely slow package upgrade - [#4534](#)
- CPS out of sync with Restore with larger solutions that haven't turned on LSL (lightweight solution restore) - [#4307](#)
- SemVer 2.0 - nuget pack with provided version ignores metadata (3.5.0-rtm-1938) - [#3643](#)
- Nuget.exe (3.+) install package with Version number and ExcludeVersion flag doesn't update package to newer version - [#2405](#)
- Project.json restore should warn when top-level packages violate constraints - [#2358](#)
- -ConfigFile is not setting custom config on install command - [#1646](#)
- nuget.exe install does not honor '-DisableParallelProcessing' switch - [#1556](#)
- Disabled sources still used by DotNet.exe or msbuild.exe - [#5704](#)
- Fix hangs in LSL scenario - [#5685](#)

DCRs

- nuget.exe install TargetFramework support - [#5736](#)
- Add different msbuild task UserAgent strings (netcore vs desktop msbuild) - [#5709](#)
- PackagePathResolver.GetPackageDirectoryName should be virtual - [#5700](#)
- [DesignConsistency] Confusing message when adding a NuGet package - [#5641](#)
- [Warnings and errors] NoWarn does not flow transitively through P2P references - [#5501](#)
- Lightweight Solution Load: Common Core for PM UI, PMC, and IVs- - [#5057](#)
- Lightweight Solution Load: Support - PMC - [#5053](#)
- Add support for pre-restore MSBuild target that Visual Studio triggers - [#4781](#)
- Add a public target to NuGet.targets that can be referenced using BeforeTargets - [#4634](#)
- Pack target can't create contentFiles with build actions correctly - [#4166](#)
- RestoreOperationLogger.Do blocks thread pool threads - [#5663](#)

Docs

- Docs for Install command DependencyVersion and Framework flags - [#5858](#)
- Update to docs on NuGet warnings and errors - [#5857](#)

Links to GitHub issues fixed in 4.4 RTM

[Issues List 1](#)

[Issues List 2](#)

[Issues List 3](#)

NuGet 4.3 Release Notes

3/25/2019 • 3 minutes to read • [Edit Online](#)

Visual Studio 2017 15.3 RTW comes with NuGet 4.3 RTM which adds support for new scenarios such as .NET Standard 2.0/.NET Core 2.0, contains many quality fixes, and improves performance. This release also brings several improvements like support for Semantic Versioning 2.0.0, MSBuild integration of NuGet warnings and errors, and more.

Summary: What's New in 4.3.0

Summary: What's New in 4.3.1

- Security Fix: Permissions on files created inside `~/.nuget` are too open [#7673 CVE-2019-0757](#)
- Security Fix: Files inside of NUPKGs can have a relative path above the NUPKG directory [#7906](#)

Known issues

NuGet restore may treat disabled package sources as enabled in some cases

Issue

The following restore command-line techniques treat disabled packages sources as enabled. [NuGet#5704](#)

- `msbuild /t:restore`
- `dotnet restore` (either with dotnet.exe that ships with VS, or the one that comes with NetCore SDK 2.0.0)

Workaround

1. Use Visual Studio (2017 15.3 or later) or NuGet.exe (v4.3.0 or later)
2. Delete your disabled source and continue to use msbuild or dotnet.exe.
3. For your solution, you could use "Clear" in NuGet.config and then define the sources necessary for that solution.

While using Package Manager Console, 'Enter' key may not work

Issue

Occasionally, the enter key does not work in the Package Manager Console. If you see this, please check out the progress on the fix, and provide any additional helpful information about your repro steps. [NuGet#4204](#)

[NuGet#4570](#)

Workaround

Restart Visual Studio and open the PMC before opening the solution. Alternatively, try deleting the `project.lock.json` and restoring again.

You are unable to view, add, or update DotNetCLITools, using Nuget Package Manager

Issue

NuGet Package Manager does not display and does not allow add/update of DotNetCLITools. [NuGet#4256](#)

Workaround

DotNetCLIToolReferences must be manually edited in your project file.

Retargeting target framework version may lead to incomplete Intellisense

Issue

Retargeting target framework version may lead to incomplete Intellisense, in Visual Studio. This happens when you are using PackageReferences as the package manager format. [NuGet#4216](#)

Workaround

Do a manual restore.

Issues fixed in NuGet 4.3 RTM timeframe

[NuGet 4.0 RTM Release Notes](#) - Lists all the issues fixed for NuGet 4.0 RTM

Features

- Improve NuGet Restore Perf - Implement smarter NoOp for command line restores and VS - [#5080](#)
- NET Core 2.0: VS/Dotnet CLI should start using existing NuGet functionality: FallBack folders - [#4939](#)
- NET Core 2.0: Enable users to ignore specific restore warnings (or elevate to error) - [#4898](#)
- NET Core 2.0: CLI localized assemblies - [#4896](#)
- NET Core 2.0: register all warnings/errors to assets file (including PackageTargetFallback) - [#4895](#)
- Enable TFM support: NetStandard2.0, Tizen - [#4892](#)
- Reduce the number of NuGet.Core and NuGet.Client projects (and thus DLLs) - [#2446](#)
- Add ability to mark nuget warnings as errors - [#2395](#)

Bugs

- msbuild /t:pack fails with The "DevelopmentDependency" parameter is not supported by the "PackTask" task - [#5584](#)
- Directory structure for content files flattened if not adding Windows directory separator at the end of PackagePath - [#4795](#)
- netcore projects don't support setting as developmentDependency - [#4694](#)
- RestoreManagerPackage being loaded synchronously which blocked UI thread and deadlocked VS - [#4679](#)
- dotnet
 - dotnetcore Restore (& therefore msbuild /t:restore) skips projects with an explicit solution project dependency [#4578](#)
- If your solution has projectreferences that refer to the same project, with different casing, restore may not work. This also affects different relative paths, without a difference in casing - [#4574](#)
- Executables restored from NuGet packages are no longer executable with .NET Core 2.0 - [#4424](#)
- NuGet.exe swallows details of exception when parsing solution file - [#4411](#)
- Pack puts content files in wrong location if ContentTargetFolders contains a path that ends with '/' on Windows - [#4407](#)
- Can't restore a DotNetCliToolReference for a tools package that targets netcoreapp1.1 - [#4396](#)
- Nuget update CLI leaves the old package version condition in project file (C++) - [#2449](#)

DCRs

- Read DotnetCliToolTargetFramework from CPS nomination - [#5397](#)
- TPMinV check should work for pj style UWP - [#4763](#)
- Improve UI description for AutoReferenced packages - [#4471](#)
- NuGet restore is selecting compile assets from runtime section. - [#4207](#)
- Put dependency diagnostics in the lock file - [#1599](#)

Links to GitHub issues fixed in 4.3 RTM

[Issues List](#)

NuGet 4.0 RTM Release Notes

9/4/2018 • 10 minutes to read • [Edit Online](#)

Visual Studio 2017 comes with NuGet 4.0 which adds support for .NET Core, has a bunch of quality fixes and improves performance. This release also brings several improvements like support for PackageReference, NuGet commands as MSBuild targets, background package restores, and more.

Known issues

NuGet restore may fail when you have multiple projects referencing another project in a solution

Issue

NuGet restore may not work if, in a solution, you have project references to the same project with different casing or with different relative paths. [NuGet#4574](#)

Workaround

Fix the casings or relative paths to be the same for all project references.

While using Package Manager Console, 'Enter' key may not work

Issue

Occasionally, the enter key does not work in the Package Manager Console. If you see this, please check out the progress on the fix, and provide any additional helpful information about your repro steps. [NuGet#4204](#)

[NuGet#4570](#)

Workaround

Restart Visual Studio and open the PMC before opening the solution. Alternatively, try deleting the `project.lock.json` and restoring again.

In .NET Core projects, you may end up in infinite restore loop when you use a package containing an assembly with an invalid signature

Issue

Occasionally, when you use a package containing an assembly with an invalid signature or when the package version is set with 'DateTime' ticker, it causes package auto-restore to run in infinite loop. [NuGet#4542](#)

Workaround

There is no workaround at this time.

You are unable to view, add, or update DotNetCLITools, using Nuget Package Manager

Issue

NuGet Package Manager does not display and does not allow add/update of DotNetCLITools. [NuGet#4256](#)

Workaround

DotNetCLIToolReferences must be manually edited in your project file.

NuGet restore will fail when you set PackageId property for projects

Issue

For .NET Core projects, NuGet restore in Visual Studio does not respect PackageId property of projects.

[NuGet#4586](#)

Workaround

Run restore using the command-line.

When your project does not have 'obj' folder, package restore may fail

Issue

Visual Studio fails to restore PackageReferences when 'obj' folder has been deleted. [NuGet#4528](#)

Workaround

Create 'obj' folder manually and the restore should work.

Manually updating packages using Update-Package in console may fail

Issue

Using Update-Package manually in the console only works once for PackageReferences projects that were just converted. [NuGet#4431](#)

Workaround

There is no workaround at this time.

Retargeting target framework version may lead to incomplete Intellisense

Issue

Retargeting target framework version may lead to incomplete Intellisense, in Visual Studio. This happens when you are using PackageReferences as the package manager format. [NuGet#4216](#)

Workaround

Do a manual restore.

msbuild /t:restore fails when a project targeting .NET461 references another project targeting .NETStandard

Issue

msbuild /t:restore fails when a PackageReference based project targeting .NET461 references another PackageReference based project targeting .NETStandard. [NuGet#4532](#)

Workaround

There is no workaround at this time.

Issues fixed in NuGet 4.0 RTM timeframe

[NuGet 4.0 RC Release Notes](#) - Lists all the issues fixed for NuGet 4.0 RC

Features

- Localize strings in NuGet.Core.sln - [#2041](#)
- Nuget forces to load web application projects in LSL mode - [#4258](#)
- AutoReferenced PackageReference support to block version changes in UI for "sdk installed" packages - [#4044](#)
- Correctly communicate PackageSpec.Version for any project dependencies (PackageRef) - [#3902](#)
- support for removing references into `.csproj` from commandline(s) - [#4101](#)
- Support restore for PackageReference projects (normal and xplat) and Lightweight Solution Load - [#4003](#)
- support for adding references into `.csproj` from commandline(s) - [#3751](#)
- Support NuGet restore for Lightweight Solution Load for `packages.config` or `project.json` - [#3711](#)
- contentFiles support in nuget generated targets file - [#3683](#)
- Establish a Mono CI for nuget.exe validation on Mac using MSBuild - [#3646](#)
- Move NuGet off of v2 NuGet.Core dependencies - [#3645](#)

Bugs

- NuGet restore in Visual Studio does not respect PackageId property of projects - [#4586](#)
- NuGet ProjectSystemCache error when adding package in vsix package - [#4545](#)

- Pack throws exception if IncludeSource is used in a project with multiple TFM - [#4536](#)
- VS 2017 RC3 crashes on using update from Solution-wide package management - [#4474](#)
- Cannot uninstall newly installed package - [#4435](#)
- When migrating to PackageRef, hybrid solutions have strange restore behavior - [#4433](#)
- Building soon after starting NuGet operation (install, update, restore), can cause VS to Hang - [#4420](#)
- UI Hang - Deadlock initializing NuGet.SolutionRestoreManager.RestoreManagerPackage [#4371](#)
- add package command should add version as attribute instead of element - [#4325](#)
- dotnet
 - dotnetcore Restore foo.sln -- fails when configurations in SLN cause duplicate (but diff config) projects in restore graph - [#4316](#)
- Content only packages - [#3668](#)
- By default opt out of package format selector option - [#4468](#)
- Perf: CreateUAP_CSharp_VS.01.1.Create project regressed Duration_TotalElapsedTime by 3,153.570 ms (149.1%). Baseline 26129.02 - [#4452](#)
- Perf: ManagedLangs_CS_DDRIT.0300.Rebuild Solution regressed Duration_TotalElapsedTime by 1.5sec. Baseline 26105 - [#4441](#)
- Nomination fails in multi-TFM projects - [#4419](#)
- Perf: WebForms_DDRIT.1200.Close Solution regressed VM_ImagesInMemory_Total_devenv by 3.000 Count (0.5%). Baseline 26123.04 - [#4408](#)
- vsfeedback - Pack warnings when targeting netcoreapp1.1 - [#4397](#)
- PathTooLongException when trying to add a NuGet package to empty ASP.NET Core web application - [#4391](#)
- Pack runs too often -- dotnet
 - dotnetcore pack fails with There is a circular dependency in the target dependency graph involving target "Pack" - [#4381](#)
- Pack runs too often -- Generate NuGet package doesn't include all the configurations - [#4380](#)
- NullReferenceException adding nuget with packageref in C++ project - [#4378](#)
- Accessibility : Narrator does not narrate the checkbox to select the projects to install the package to - [#4366](#)
- NuGet VS17 sporadically fails connecting to VSO/VSTS feeds - VS Bug 365798 - [#4365](#)
- contentFiles get output to wrong location if PackagePath specifies path as "contentFiles" - [#4348](#)
- Pack target appends PackageVersion property with VersionSuffix - [#4324](#)
- Specifying package path doesn't work with dotnet pack - [#4321](#)
- NuGet outputs a bunch of warnings about duplicate imports during restore - [#4304](#)
- Choose "NuGet Package Manager Format" dialog looks bad under dark theme - [#4300](#)
- VS crash on build restore - [#4298](#)
- Visual Studio deadlocks if you add TFM in targetframeworks, save, then build. 10% of time - [#4295](#)

- nuget pack does not output success message on packing a project successfully - [#4294](#)
- PackTask fails due to System.IO.Compression 4.1 not being found - [#4290](#)
- Pack runs too often -- PackTask frequently fails with file access conflict - [#4289](#)
- NuGet opens the output window during background restore - [#4274](#)
- Eliminate ServiceProvider as dangerous coding pattern (which can cause hangs) - [#4268](#)
- Perf/UIHang - Improve DownloadTimeoutStream reads - [#4266](#)
- Visual Studio deadlocks if you attempt to close a project before NuGet restore has finished - [#4257](#)
- Issues with PackTask and packing `.nuspec` - [#4250](#)
- [vsfeedback] Cannot resolve nuget packages on new project (needs to restart visual studio) - [#4217](#)
- [vsfeedback] The "Version" drop down that shows available package versions, struggles to stay in-sync with the selected nuGet package... - [#4198](#)
- NuGet.Client should use CPS JoinableTaskFactory when interacting with CPS to prevent deadlocks - [#4185](#)
- NuGet 3.5.0 not unpacking `.targets` from package - [#4171](#)
- dotnet
 - dotnetcore pack does not support title in `.csproj` - [#4150](#)
- Install-Package results in error dialog in VS2017 RC - [#4127](#)
- Updating a package for .net core project appears to not work, as the UI doesn't get the CPS update from the nominate. - [#4035](#)
- Improve unresolved reference warning - [#3955](#)
- dotnet
 - dotnetcore pack - ProjectReference loses version information - [#3953](#)
- Create UWP app create project & rebuild total elapsed time regressions - [#3873](#)
- Successful restore message is displayed even after error during restore. - [#3799](#)
- re-Publish NugetCommandLine 3.4.4 to Nuget.org - [#2931](#)
- On Migrate, projects change from `project.json` to `.csproj` --- restore fails - [#4297](#)
- Restore failing on newly created xunit Test project - [#4296](#)
- Core projects can hang, lock up UI on open - [#4269](#)
- fix targets file for build tasks - [#4267](#)
- Error list has error after build solution which unload the referenced project - [#4208](#)
- MSB4057: The target "_GenerateRestoreGraphProjectEntry" does not exist in the project. - [#4194](#)
- vsfeedback: nuget manager ui for solution crashes when you select all projects - [#4191](#)
- nuget.exe msbuildpath fails when there is a trailing slash - [#4180](#)
- vsfeedback: NuGet restore give several project reference warnings for LinqToTwitter project - [#4156](#)
- Pack from `.csproj` does not include the minClientVersion attribute - [#4135](#)
- NuGet.Build.Tasks.Pack.dll shipped delay signed in VS2017 (d15rel 26014.00) - [#4122](#)

- VSFeedback: Restore fails for a VS 2015 project generated with CMake 3.7.1 - [#4114](#)
- VSFeedback: Restore errors can obscure more complete error messages that build could give - [#4113](#)
- [VSFeedback] Error occurred while restoring NuGet packages for website project: Value cannot be null. - [#4092](#)
- Migration Throws "Object reference Exception" in
NuGet.PackageManagement.VisualStudio.SolutionRestoreWorker - [#4067](#)
- dotnet
 - dotnetcore pack should pack tools with the versions that the package was built against - [#4063](#)
- New background restore writes milliseconds to status bar when it takes seconds to restore - [#4036](#)
- Typo on failed to resolve all project references - [#4018](#)
- Enable PCM workflows in package reference scenarios - [#4016](#)
- Can not find installed packages in package manager UI - [#4015](#)
- dotnet
 - dotnetcore pack fails when PackagePath is empty - [#3993](#)
- Restore task fails in an multi user scenario - [#3897](#)
- Cannot change Content type when packing using NuGet Pack Task - [#3895](#)
- Default Copy of ContentFiles are incorrect for MsBuild /t:pack - [#3894](#)
- Install package restore double logs the restoring packages message - [#3785](#)
- Remove Guardrails - Restore of "runtimes" section should only apply to the current project - [#3768](#)
- Pack task puts content files in both 'content/' and 'contentFiles/' - [#3718](#)
- dotnet
 - dotnetcore pack3 does extra tag splitting - [#3701](#)
- dotnet
 - dotnetcore pack: packing projects with package references results in duplicate import warning - [#3665](#)
- Restore logging in VS doesn't always show - [#3633](#)
- nuget locals help text still mentioned packages cache - [#3592](#)
- Restore3 couples PackageReferences with TargetFrameworks. - [#3504](#)
- Nuget picks unexpected version of MSBuild in VS "15" Preview 4 dev. command prompt - [#3408](#)
- Write out targets/props files on failed restore - [#3399](#)
- NuGet during restore doesn't respect the same compat shims as MSBuild when running in VS 15 command prompt - [#3387](#)
- Re-enable PackFromProjectWithDevelopmentDependencySet for VS15 - [#3272](#)
- Blend problems with NuGet - [#4043](#)
- Integrate 4.0.0.2067 into CLI and SDK repos to ship with RC2 - [#4029](#)
- VS Hangs when you Create new Core Console App, Close Solution, Open Solution and Close Solution - [#4008](#)

- Hitting hang opening project against d15prerel.25916.01 - [#3982](#)
- Fix dotnet/nuget.exe locals doc/help message - [#3919](#)
- Inspect PackTask for issues with trailing or leading whitespace - [#3906](#)
- dotnet
 - dotnetcore pack is packing from obj not bin - [#3880](#)
- dotnet
 - dotnetcore pack always seems to set ProjectReference version to 1.0.0 - [#3874](#)
- dotnet
 - dotnetcore pack fails with project references and - [#3865](#)
- LockRecursionException in ProjectSystemCache.TryGetProjectNameByShortName - [#3861](#)
- Trim whitespace from MSBuild properties - [#3819](#)
- Consolidate the two project events raised on project load - [#3759](#)
- P2P libraries in `project.assets.json` file have incorrect Version - [#3748](#)
- Restore crash due to unresponsive feed and unavailable package - [#3672](#)
- nuget.exe could hang on a large amount of MSBuild error output - [#3572](#)
- Restore-on-build for Blend fails first time, succeeds second time (VS scenario fixed) - [#2121](#)

DCRs

- migrate vsix from v2 vsix to v3 vsix - [#4196](#)
- NuGet should have a mechanism for getting the path to the lock file in MSBuild - [#3351](#)
- Add build assets to the TFM compatibility check and assets file - [#3296](#)
- Define a new ProjectCapability "Pack" in Pack targets for enabling Package related capabilities - [#4146](#)
- Run Pack as a post build target conditioned on "GeneratePackageOnBuild" MSBuild property - [#4145](#)
- Use NuGet property RestoreProjectStyle to create specific NuGet project - [#4134](#)
- Adapt Restore for Transitive Project References change - [#4076](#)
- Add NuGet properties in target file for non-UWP projects - [#4030](#)
- UWP TargetPlatformVersion support - [#3923](#)
- Communicate project reference metadata to NuGet project system - [#3922](#)
- Add UI for packaging mode - [#3921](#)
- Legacy `.csproj` needs NugetTargetMoniker and RuntimeIdentifiers set in proj/targets - [#3854](#)
- Install package may overlap with auto-restore - [#3836](#)
- Context menu QueryStatus doesn't happen when VSPackage is not loaded - [#3835](#)
- Solution Restore and Build Restore still show dialogs - [#3789](#)
- Isolate VSSDK version in NuGet.Clients solution build - [#3890](#)

Links to GitHub issues fixed in RTM

[Issues list 1](#)

[Issues list 2](#)

[Issues list 3](#)

[Issues list 4](#)

[Issues list 5](#)

NuGet 4.0 RC Release Notes

9/4/2018 • 4 minutes to read • [Edit Online](#)

NuGet 3.5 RTM Release Notes

NuGet 4.0 RC for Visual Studio 2017 is focused on adding support for .NET Core scenarios, addressing key customer feedback and improving performance in a variety of scenarios. This release brings several improvements like support for PackageReference, NuGet commands as MSBuild targets, background package restore, and more.

Bug Fixes

- Behavioral changes in `dotnet pack --version-suffix foo` - [#3838](#)
- nuget.exe restore on vs "15" machine only fails - [#3834](#)
- .NETCore file new project should block the build during restore - [#3780](#)
- ASP.NET Core web app, migrated from VS2015 to VS "15", unable to restore. - [#3773](#)
- [Test Failure]Package 'jQuery Validation' can't be uninstalled by PM UI - [#3755](#)
- When a package is installed to UWP `project.json`, parent projects should also be restored - [#3731](#)
- Modify the NuGet targets to log the package sources as High verbosity instead of Normal - [#3719](#)
- dotnet
 - dotnetcore pack3 should include XML documentation by default - [#3698](#)
- Batch update fails from UI when source without the package is first and All source is selected - [#3696](#)
- Nuget pack command does not include all files - [#3678](#)
- OOM issue - [#3661](#)
- ProjectFileDependencyGroups section of the assets file should use library names for projects - [#3611](#)
- "dotnet restore" and recursing directories - [#3517](#)
- Restore3 failures are logged as warnings instead of errors - [#3503](#)
- TFS issue: "[file]not be found in your workspace, or you do not have permission to access it" - [#2805](#)
- Typing "nuget " in vs quicklaunch search box keeps "nuget " prefix - [#2719](#)
- System.Xml.XmlException: Unrecognized root element in Core Properties part. Line 2, position 2. - [#2718](#)
- `.nuspec` with escaped < or > in text fields no longer builds - [#2651](#)
- nuget.exe delete won't prompt for credentials (it's in non-interactive mode) - [#2626](#)
- nuget.exe delete warns about API Key for local sources, even though it makes no sense - [#2625](#)
- Error experience poor when installing EF -pre package - [#2566](#)
- Visual Studio crashed attempting after changing selection in Package Manager - [#2551](#)
- dotnet
 - dotnetcore restore performs case sensitive Id lookups on flat-list local repositories when floating versions

are used - [#2516](#)

- nuget.exe delete is broken for V2 feed - [#2509](#)
- nuget.exe push timeout needs a better error message - [#2503](#)
- Tool restore without proper imports silently fails. - [#2462](#)
- NuGet prompts to enter credentials when there is a private feed even when installing from nuget.org - [#2346](#)
- ApplicationInsights 2.0 package is listed but doesn't exist yet - [#2317](#)
- UIDelay in VS "15" preview 5 branch - [#3500](#)
- First OnBuild event is missed for Restore during Build for UWP - [#3489](#)
- PowerShell5 breaks EntityFramework install? - [#3312](#)
- Add source to detailed logging (consider for 3.5) - [#3294](#)
- NoCache parameter not honored in nuget client version 3.4+ - [#3074](#)
- When a credential provider fails to load in VS, don't break NuGet - [#2422](#)

Features

- Set up CI to run x86 - [#3868](#)
- Auto Restore 3/3: non blocking UI - [#3658](#)
- Auto Restore 2/3: background restore on nomination - [#3657](#)
- Restore project refs to match build behavior (recurse) - [#3615](#)
- DPL support in VS "15" - minbar - [#3614](#)
- Move settings file to Program Files - [#3613](#)
- Generated restore props and targets need cross-targeting participation support - [#3496](#)
- NuGet Restore support for PackageTargetFallback (f.k.a Imports) - [#3494](#)
- ToolsRef implementation - [#3472](#)
- Restore3 for a RID - [#3465](#)
- NuGet UI to support Add/Remove/Update of PackageRefs - [#3457](#)
- Auto Restore 1/3: Implementation of Nomination API via Caching Project Restore Info - [#3456](#)
- [0] NuGet Restore Task & Targets - [#2994](#)
- [1] Enable Solution level restore in MSBuild - [#2993](#)
- Support credential provider public extensibility in Visual Studio - [#2909](#)
- Recursive nuget restore - [#2533](#)
- Can't load Microsoft.TeamFoundation.Client on dev15, need to update Microsoft.TeamFoundation.Client version to 15.0 for VS "15" Preview - [#2392](#)
- Unable to install C++ package to C++ UWP project in VS "15" Preview - [#2369](#)
- Nupkg needs to support \buildCrossTargeting\ folder - and import `.targets` / `.props` for "crosstargeting"

MSBuild scope. - [#3499](#)

- ToolsReference Design - [#3462](#)
- Fix NuGet UI to support restore w/ PackageReferences in `.csproj` - [#3455](#)
- Adding clear cache button to VS package manager settings - [#3289](#)

DCRs

- Solution Restore should be blocked while auto restore is happening. - [#3797](#)
- NetCore install from NuGet Package Manager UI installs to every TFM , instead of ones that the package supports - [#3721](#)
- Restore nominator API needs to support DotNetCliToolsReferences too. - [#3702](#)
- Mark our VS "15" vsix as a systemcomponent - [#3700](#)
- Migrate from referencing MS.VS.Services.Client to MS.VS.Services.Client.Interactive - [#3670](#)
- `$(RestoreLegacyPackagesDirectory)` should be respected at a project level by restore - [#3618](#)
- Restore to project with single TargetFramework must not condition props - [#3588](#)
- dotnet
 - `dotnetcore restore3` `foo.csproj` should follow projectref dependencies, and restore those too. Like build. - [#3577](#)
- "type": "platform" Dependencies represented as "type":"package" in lock file - [#2695](#)
- nuget.exe Verbose mode should show the download url - [#2629](#)
- Move NuGet xplat to Microsoft.NetCore.App and netcoreapp1.0 - [#2483](#)
- Push - It should be possible to override the symbol server when pushing from the command line - [#2348](#)
- Consolidate code for finding the global packages path - [#2296](#)
- Need a better name than suppressParent - [#2196](#)
- Determine `project.json` dependency name to use for MSBuild projects - [#1914](#)
- Add SemVer 2.0.0 support to NuGet.Core - [#3383](#)
- Allow transitive dependency NuPkgs with `.targets` to be available in MSBuild - [#3342](#)
- NuGet restore from the commandline is significantly slower than VS - [#3330](#)
- Make package ID and version comparison case insensitive - [#2522](#)
- NoCache option does not work for `packages.config` based restore/install (GlobalPackagesFolder) - [#1406](#)
- FindPackageByIdResource resources needs a default cache context and logger - [#1357](#)

NuGet 3.5 Release Notes

9/4/2018 • 8 minutes to read • [Edit Online](#)

[NuGet 3.5-RC Release Notes](#) | [NuGet 4.0 RC Release Notes](#)

Bug Fixes

- Pack doesn't use MSBuild 14.1 on mono - [#3550](#)
- Update tab doesn't select the latest available version to update instead select current installed version - [#3498](#)
- Fix Crash after authenticating a private v2 MyGet feed and clicking "Show x more results" - [#3469](#)
- Log messages seem to be in reverse order for UI - [#3446](#)
- v3.4.4 - Nuget restore throws "The given path's format is not supported" - [#3442](#)
- NuGet cmdLine 3.6 beta does not honor -Prop Configuration = Release - [#3432](#)
- Nuget IKVM slow install on large project - [#3428](#)
- nuget.exe Update -Self keeps on updating itself - [#3395](#)
- 3.5 install/restore from UNC share has performance Regression from 3.4.4 - [#3355](#)
- Error when installing Moq from the Package management UI for a net451 project - [#3349](#)
- Install tab at solution level doesn't show package's version - [#3339](#)
- xproj `project.json` update from Installed tab loses state - [#3303](#)
- NuGet pack on `.csproj` ignores empty files element in `.nuspec` file - [#3257](#)
- Website projects hosted in IIS should not cause restore to fail - [#3235](#)
- Credentials not retrieved from Nuget.Config when v3 endpoint redirects to v2 - [#3179](#)
- NuGet pack fails to resolve assembly when retrieving portable assembly metadata - [#3128](#)
- Nuget can't find `msbuild.exe` on Mono - [#3085](#)
- nuget.exe pack doesn't allow a pre-release tag which begins with numbers - [#1743](#)
- nuget package install fails on VS2015E - [#1298](#)
- allowedVersions filter not working at solution level - [#333](#)
- Restore randomly fails with An item with the same key has already been added. - [#2646](#)
- Cannot install Nuget.Common in `.csproj` - [#2635](#)
- When using the UI to search a V2 source, FindPackagesById is called twice for each ID - [#2517](#)
- Packages cannot depend on projects - [#2490](#)
- nuget.exe pack -Exclude is documented but not supported - [#2284](#)
- Issues with error messages when 'contentFiles' section of `.nuspec` is invalid - [#1686](#)

- Push always sends entire package twice with authenticated package sources - [#1501](#)
- No information was given when calling nuget.exe update *.csproj while the project does not have a `packages.config` - [#1496](#)
- `packages.config` restore does not retry on 5xx status codes from V2 sources - [#1217](#)
- Double dot in file src in `.nuspec` doesn't work - [#2947](#)
- CoreCLR restore needs to ignore feeds with encryption - [#2942](#)
- nuget.exe push 403 handling - Incorrectly prompting for credentials - [#2910](#)
- NuGet update through package manager removes properties from the `project.json` - [#2888](#)
- NuGet.PackageManagement.VisualStudio try to load "NuGet.TeamFoundationServer14", but that DLL name changed to "NuGet.TeamFoundationServer" - [#2857](#)
- Package manager UI doesn't show newly updated version - [#2828](#)
- update-package trying to use packageid,version instead of package.version - [#2771](#)
- nuget restore csproj should error if the project isn't using nuget (`packages.config` or `project.json`) - [#2766](#)
- TFS Error "[file]not be found in your workspace, or you do not have permission to access it" during upgrade or uninstall when solution/project is bound to TFS source control - [#2739](#)
- update package doesn't get dependencies for non-target packages - [#2724](#)
- There is no way to set logs verbosity level for Nuget package manager UI actions - [#2705](#)
- nuget configuration is invalid - VS 2015 VSIX (v3.4.3) - [#2667](#)
- DefaultPushSource in `NuGetDefaults.Config` (`ProgramData\NuGet`) doesn't work - [#2653](#)
- nuget 3.4.3 release - getting Value cannot be null on package build - [#2648](#)
- restore is not using stored credentials from Nuget.Config for VSTS feeds - [#2647](#)
- [dotnet restore] --configfile is relative to project dir instead of the cmd dir - [#2639](#)
- Excessive allocations in version comparsion code - [#2632](#)
- Multiple instances of nuget.exe trying to install the same package in parallel causes a double write - [#2628](#)
- Dependency information is not cached for multi-project operations - [#2619](#)
- Install and update download packages without checking the packages folder first - [#2618](#)
- If package source list is empty, cannot add package source via UI (NuGet 3.4.x) - [#2617](#)
- Misleading error when attempting to install package that depends on design-time facades - [#2594](#)
- Installing a package from PackageManager console with setting "All" tries only first source - [#2557](#)
- Latest beta not unzipping ModernHttpClient - [#2518](#)
- VS2015 crash at startup with self-built NuGet 3.4.1 - [#2419](#)
- Update command might be a bit more verbose if i ask it to be so... - [#2418](#)
- VSIX built locally should have the same DLLs and files as the CI build. - [#2401](#)
- Fix NuGet downgrade warnings in the build - [#2396](#)

- Failing to authenticate package source (3 times) is blocked forever - [#2362](#)
- Package content is not restored correctly when installing a package from a nuget v3.3+ feed with the argument -NoCache when the package contains `.nupkg` files - [#2354](#)
- Nuget Install with All Package Sources, but package missing from 1 source, fails - [#2322](#)
- [PerfWatson] UIDelay:
`nuget.packagemanagement.visualstudio.dll!NuGet.PackageManagement.VisualStudio.VSMSBuildNuGetProjectSystem+*It>c__DisplayClass_0+<<AddReference>b__>d.MoveNext` - [#2285](#)
- Install blocks if a single source fails authorization - [#2034](#)
- `.nuspec` version range should override `-IncludeReferencedProjects` version - [#1983](#)
- Update-Package super slow - "Attempting to gather dependencies information" - [#1909](#)
- NuGet stealth downgrades package when batch updating its dependencies - [#1903](#)
- nuget.exe update drops the assembly strong name and Private attribute. - [#1778](#)
- Relative file path for "DefaultPushSource" - [#1746](#)
- Improve resolver failure messages - [#1373](#)
- update-package in v3 fails with packages not in the specified source - [#1013](#)
- Using relative paths for package sources is problematic to use - [#865](#)
- Missing dependency in NUPKG-file generated from project if indirect dependency already exists with a lower version requirement - [#759](#)
- Deleting a project closes corresponding UI window, but, renaming a project does not rename the UI window. Note that PMC listens to project rename and project remove events - [#670](#)
- [Willow Web Workload] Creating Razor v3 WSP hangs - [#3241](#)
- Install/restore of a particular package fails with "Package contains multiple nuspec files." - [#3231](#)
- Lowercase IDs & `packages.config` scenarios - [#3209](#)
- [3.5-beta2] Package restore fails to restore "legacy" packages - [#3208](#)
- nuget pack forcefully adds .tt files to content folder no matter what - [#3203](#)
- update-package of ASP.NET web app generates warning related to file: source - [#3194](#)
- nuget pack csproj (with `project.json`) crashes if there are no packOptions and owner in JSON file - [#3180](#)
- nuget pack for `project.json` ignores packOptions tags like summary, authors, owners etc - [#3161](#)
- NullReferenceException via `NuGet.Packaging.PhysicalPackageFile.GetStream` - [#3160](#)
- NuGet pack ignores dependencies in output `.nuspec` for `project.json` - [#3145](#)
- Updating multiple packages with rollback leaves the project in a broken state - [#3139](#)
- ContentFiles under any are not added for netstandard projects - [#3118](#)
- Cannot package library targeting .Net Standard correctly - [#3108](#)
- File -> New Project -> Class Library (Portable) project fails in VS2015 and Dev15 - [#3094](#)
- nuGet error - 1.0.0-* is not a valid version string - [#3070](#)

- Find-Package fails to display but Install-Package works - [#3068](#)
- Error when "Install-Package jquery.validation" on dev15 - [#3061](#)
- nuget pack of xproj is defaulting to invalid target path - [#3060](#)
- When installed VS 2015 update 3 on a VS that uses NuGet version 3.5.0 error occurs - [#3053](#)
- "Blocked by packages.config" in `project.json` (UWP, a.k.a build integrated) project - [#3046](#)
- update dotnet cli installed by build script to preview2-003121, which is the official preview2 build. - [#3045](#)
- Package manager UI: Doesn't display new version after updating a package - [#3041](#)
- -ApiKey on delete command line is not read/sent in 3.5.0-beta - [#3037](#)
- Incorrect string: A stable release of a package should not have on a prerelease dependency. - [#3030](#)
- OptimizedZipPackage cache leaves empty folders - [#3029](#)
- Creating PCL (net46 and windows 10) project get NullRef exception. - [#3014](#)
- Nuget update should provide informative message when a higher version is restricted by allowedVersions constraint - [#3013](#)
- Nuget v3 restore issues - [#2891](#)
- Credential plugin exited with error -1 / error downloading package when using credential providers with multiple sources - [#2885](#)
- `project.json` nuget restore causes recompilation when nothing changed - [#2817](#)
- Symbols packages should not ever be used in install or update - [#2807](#)
- VS doesn't support environment variables in repositoryPath (nuget.exe does) - [#2763](#)
- Label the unlabeled UIElements in Package Manager UI for accessibility - [#2745](#)
- Portable frameworks with hyphenated profiles are rejected. - [#2734](#)
- NuGet package manager should make it clear that options list in packages detail does not apply to `project.json` - [#2665](#)
- nuget.exe push/delete won't use API Key - [#2627](#)
- Remove the locked property from the lock file - [#2379](#)
- NuGet 3.3.0 update fails with 'An additional constraint ... defined in packages.config prevents this operation.' - [#1816](#)
- Installing package from a local source that doesn't exist throws a bogus message - [#1674](#)
- "Upgrade available" filter shows upgrades that violate the version constraint - [#1094](#)
- Unable to update native packages - [#1291](#)

Features

- Support setting CopyLocal to false on references added by NuGet - [#329](#)
- nuget.exe support for MSBuild 15 - [#1937](#)
- Pack support for `.csproj` + `project.json` - [#1689](#)

- Disable user action when there are user actions being executed - [#1440](#)
- NuGet should add support for `runtimes/{rid}/nativeassets/{txm}/` - [#2782](#)
- Add framework compatibilities missing in NuGet 2.x (which are already in 3.x) - [#2720](#)
- Support for fallback package folders - [#2899](#)
- Design and implement a notion of package type to support tool packages - [#2476](#)
- Add an API to get the path to the global packages folder - [#2403](#)
- Enable SemVer 2.0.0 in pack - [#3356](#)

DCRs

- nuget.exe push - timeout parameter doesn't work - [#2785](#)
- Package Description text should be selectable - [#1769](#)
- Enable nuget.exe to produce `.props` and `.targets` files for `.nuproj` projects [#2711](#)
- Add extensibility API to compare frameworks with imports - [#2633](#)
- Hide dependency options when using `project.json` - [#2486](#)
- Print out nuget.exe version header in detailed output - [#1887](#)
- NuGet needs to let users know that upgrading/installing in a dotnet tfm based PCL could cause issues - [#3138](#)
- Warn bad install/upgrade for project w/ `tfm="dotnet"` - [#3137](#)
- Fix performance issues with ReShaper and NuGet for Update - [#3044](#)
- Add `netcoreapp11` and `netstandard17` support - [#2998](#)
- Leverage AssemblyMetadata attribute for `.nuspec` token replacements - [#2851](#)

NuGet 3.5 RC Release Notes

9/4/2018 • 3 minutes to read • [Edit Online](#)

[NuGet 3.5-Beta2 Release Notes](#) | [NuGet 3.5-RTM Release Notes](#)

3.5 release is focused on improving quality and performance of NuGet clients. In addition, we have shipped a few features like support for [Fallback folders](#), [PackageType](#) support in `.nuspec` and more.

[Issues List](#)

Bug Fixes

- Install/restore of a package fails with "Package contains multiple `.nuspec` files." - [#3231](#)
- nuget pack forcefully adds `.tt` files to content folder no matter what - [#3203](#)
- nuget pack csproj (with `project.json`) crashes if there are no packOptions and owner in JSON file - [#3180](#)
- nuget pack for `project.json` ignores packOptions tags like summary, authors , owners etc - [#3161](#)
- nuget pack ignores dependencies in output `.nuspec` for `project.json` - [#3145](#)
- Updating multiple packages with rollback leaves the project in a broken state - [#3139](#)
- ContentFiles under any are not added for netstandard projects - [#3118](#)
- Cannot package library targeting .Net Standard correctly - [#3108](#)
- File -> New Project -> Class Library (Portable) project fails in VS2015 and Dev15 - [#3094](#)
- NuGet error - 1.0.0-* is not a valid version string - [#3070](#)
- Find-Package fails to display but Install-Package works - [#3068](#)
- Error when "Install-Package jquery.validation" on dev15 - [#3061](#)
- When installed VS 2015 update 3 on a VS that uses NuGet version 3.5.0 error occurs - [#3053](#)
- Package manager UI: Doesn't display new version after updating a package - [#3041](#)
- -ApiKey on delete command line is not read/sent in 3.5.0-beta - [#3037](#)
- Incorrect string: A stable release of a package should not have on a prerelease dependency. - [#3030](#)
- Creating PCL (net46 and windows 10) project get NullRef exception. - [#3014](#)
- Nuget update should provide informative message when a higher version is restricted by allowedVersions constraint - [#3013](#)
- Credential plugin exited with error -1 / error downloading package when using credential providers with multiple sources - [#2885](#)
- nuget pack - Missing Newtonsoft.Json package dependency - [#2876](#)
- Bug in ExecuteSynchronizedCore on Linux/MacOS + Mono - [#2860](#)
- VS doesn't support environment variables in repositoryPath (nuget.exe does) - [#2763](#)
- Fix Accessibility Issues - [#2745](#)

- Portable frameworks with hyphenated profiles are rejected. - [#2734](#)
- NuGet package manager should make it clear that options list in packages detail does not apply to `project.json` - [#2665](#)
- NuGet 3.3.0 update fails with 'An additional constraint ... defined in packages.config prevents this operation.' - [#1816](#)
- Installing package from a local source that doesn't exist throws a bogus message - [#1674](#)
- "Upgrade available" filter shows upgrades that violate the version constraint - [#1094](#)

Performance Improvements

- Performance: Improve ContentModel target framework parsing - [#3162](#)
- Performance: Avoid reading `runtime.json` files for restores that do not have RIDs [#3150](#). On CI machines, restore of a sample ASP.NET Web Application reduced from over 15 secs to 3 secs.
- Performance: Package Manager Console init.ps1 load time [#2956](#). Time to open PackageManagerConsole improved in some cases from 132s to 10s.
- Solve ReSharper performance issues in NuGet Update - [#3044](#): On a sample project, time taken to install packages reduced from 140s to 68s.

DCRs

- NuGet needs to let users know that upgrading/installing in a dotnet tfm based PCL could cause issues - [#3138](#)
- Warn bad install/upgrade for project w/ tfm="dotnet" - [#3137](#)
- Add netcoreapp11 and netstandard17 support - [#2998](#)
- Print NuGet-Warning header contents to console in nuget.exe - [#2934](#)
- Leverage AssemblyMetadata attribute for `.nuspec` token replacements - [#2851](#)
- Remove the locked property from the lock file - [#2379](#)
- Symbol packages should not ever be used in install or update #2807

Features

- Support for fallback package folders - [#2899](#)
- Design and implement a notion of package type to support tool packages - [#2476](#)
- API to get the path to the global packages folder - [#2403](#)
- Native packages update support - [#1291](#)

NuGet 3.5 Beta2 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.5-Beta Release Notes](#) | [NuGet 3.5-RC Release Notes](#)

NuGet 3.5 Beta 2 RTM was released June 27, 2016 for Visual Studio 2013 and nuget.exe

[Full Changelog](#)

[Issues List](#)

Bug Fixes

- Updated error message to lack of support for password decryption in .NET Core for authenticated feeds - [#2942](#)
- Package Manager Console Get-Package fails if .NET Core project is open - [#2932](#)
- Fix incorrect handling of 403 in NuGet push command [#2910](#)
- Fix issues in uninstalling packages in a solution bound to TFS source control when disableSourceControlIntegration is set to true - [#2739](#)
- Fix package update to take into account non-target packages - [#2724](#)
- Use MSBuild verbosity level to set logger level for Nuget package manager UI actions - [#2705](#)
- Fix NuGet configuration is invalid error in WebSite projects - VS 2015 VSIX (v3.4.3) - [#2667](#)
- Fix pack issues from `.csproj` when content files are included - [#2658](#)
- DefaultPushSource in `NuGetDefaults.Config` (`ProgramData\NuGet`) doesn't work - [#2653](#)
- Fix issue in Nuget 3.4.3 release - Value cannot be null on package creation - [#2648](#)
- Restore uses stored credentials from Nuget.Config for VSTS feeds - [#2647](#)
- Performance - Fix excessive allocations in version comparison code - [#2632](#)
- Fix issues when multiple instances of nuget.exe tries to install the same package in parallel - [#2628](#)
- Performance - Cache dependency information for multi-project operations - [#2619](#)
- Fix issue where package sources cannot be added from settings when source list is empty - [#2617](#)
- Fix Misleading error when attempting to install package that depends on design-time facades - [#2594](#)
- Installing a package from PackageManager console with setting "All" tries only first source - [#2557](#)
- Fix issues with packages that have files with write times in the future (Mono) - [#2518](#)
- Display exception when there is a failure finding projects in update command - [#2418](#)
- Package content is not restored correctly when installing a package from a nuget v3.3+ feed with the argument `-NoCache` when the package contains `.nupkg` files - [#2354](#)
- Fix issue with package install (All Sources) when package is missing from 1 source - [#2322](#)
- Install blocks if a single source fails authorization - [#2034](#)

- `.nuspec` version range should override `-IncludeReferencedProjects` version - [#1983](#)
- NuGet 3.3.0 update fails with 'An additional constraint ... defined in packages.config prevents this operation.' - [#1816](#)
- nuget.exe update drops the assembly strong name and Private attribute. - [#1778](#)
- Fix issues with relative file path for "DefaultPushSource" - [#1746](#)
- Improve Update resolver failure messages - [#1373](#)

Features and Behavior Changes

- nuget.exe push - timeout parameter doesn't work - [#2785](#)
- nuget.exe restore doesn't produce `.props` and `.targets` files for `.nuproj` projects (regression in v3.4.3.855) - [#2711](#)
- Need extensibility API to compare frameworks with imports - [#2633](#)
- Hide dependency options when using `project.json` - [#2486](#)
- Print out nuget.exe version header in detailed output - [#1887](#)
- NuGet should add support for `/runtimes/{rid}/nativeassets/{txm}/` - [#2782](#)

NuGet 3.5 Beta Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.4 Release Notes](#) | [NuGet 3.5-Beta2 Release Notes](#)

NuGet 3.5 Beta was released on May 16, 2016 as part of the ASP.NET Core Preview Tooling wave. This release adds support for .NET Core RC2 and ASP.NET Core RC2. For more information about this release please refer to <http://dot.net>.

You can download both the VSIX and nuget.exe [here](#).

NuGet 3.5 Beta is a superset of the changes introduced in the 3.4.3 release. For a complete set of changes, please refer to the 3.4.3 release notes [here](#) and the notes for the 3.5 Beta below.

Updates and Improvements

- Adds support for ASP.NET Core RC2 and .NET Core RC2

Fixes

- The list of fixes and improvements in this release, is given [here](#).

NuGet 3.4.4 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.4.3 Release Notes](#) | [NuGet 3.5-Beta Release Notes](#)

The primary focus of this release was improvements to the quality of 3.4.3 version of nuget.exe with a few fixes to the Visual Studio extension as well.

You can download both the VSIX and nuget.exe [here](#).

3.4.4-rtm (2016-05-19)

[Full Changelog](#)

[List of Issues](#)

Changes

- Pack Improvements: Improvements to packing symbols, packing with `project.json` and more [#606](#)
- Display exception when there is a failure finding projects in update command [[#605](#)](<https://github.com/NuGet/NuGet.Client/pull/605>)
- Read package type from input `.nuspec` and `project.json` when packing [#603](#)
- Make NuGet.Shared not a project. [#602](#)
- Use the push timeout as the HTTP response timeout [#599](#)
- Package files with future times will not have their times used [#597](#)
- Updating `NuGet.Core.dll` version to 2.12.0 to fix XML issue [#594](#)
- Support `./NuGet.CommandLine.XPlat -v <verbosity> <mode>` [#593](#)
- Display error restoring without `project.json` or `packages.config` [#590](#)
- Fixing dependency versions when required versions differ [#559](#)

NuGet 3.4.3 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.4.2 Release Notes](#) | [NuGet 3.4.4 Release Notes](#)

NuGet 3.4.3 was released on April 22, 2016 to address several issues that were identified in the 3.4 and subsequent releases.

You can download both the VSIX and nuget.exe [here](#).

Updates and Improvements

- Improved Visual Studio reliability. We have fixed some issues in NuGet that caused crashes in Visual Studio.

Fixes

- Fixed some authorization issues with password protected private nuget feeds.
- Fixed an issue around being unable to restore PCL's from `project.json` with runtimes specified.
- Some customers were running into intermittent failures when installing packages. This has now been fixed in this release.
- Fixed an issue that caused restore failures in C++/CLI projects with `project.json`.
- Some packages (E.g ModernHttpClient) where not being unzipped correctly when you use nuget in mono. This has now been fixed in this release.

For the complete list of fixes and improvements in this release, check out the list of issues [here](#).

NuGet 3.4.2 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.4.1 Release Notes](#) | [NuGet 3.4.3 Release Notes](#)

NuGet 3.4.2 was released on April 8, 2016 to address several issues that were identified in the 3.4 and 3.4.1 release.

nuget.exe 3.4.2 RC is now available

You can download the release candidate of nuget.exe 3.4.2 [here](#).

Updates and Improvements

- We have significantly improved the performance of updates in a specific scenario, where updates on packages with deep dependency graphs took a really long time and hung Visual Studio.
- nuget restore without network traffic is 2.5x – 3x faster within Visual Studio.
- In addition to this change, we have fixed an issue where we were hitting the network twice when fetching the update count in the VS UI. This was partially responsible for some timeout issues customers experienced in 3.4/3.4.1.
- Added support for no_proxy setting

Fixes

- Fixed an issue where nuget.org source was missing in NuGet settings or config after updating to 3.4.1.
- Fixed an issue where a casing change to FindPackagesById in 3.4.1 breaks Artifactory.
- Corrected an issue with FIPS that caused failures with NuGet restore with nuget.exe.
- Fixed a crash when browsing sources with invalid icon URL.
- Fixed issues with merging versions and entries from 'All Sources'.

Known Issues in 3.4.2 Windows x86 Commandline (RC)

These issues will be fixed early next week before we hit RTM.

- Running nuget restore on a solution will fail if the solution file is placed in a lower folder hierarchy than the project files.
- Running nuget delete command on a package using the V2 feed will fail. Use V3 feed instead.

For the complete list of fixes and improvements in this release, check out the list of issues [here](#).

NuGet 3.4.1 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.4 Release Notes](#) | [NuGet 3.4.2 Release Notes](#)

NuGet 3.4.1 was released March 30, 2016 at the same time as the Visual Studio 2015 Update 2 and Visual Studio 15 Preview Release to address several issues that were identified in the 3.4 release.

Updates and Improvements

- Corrected an issue that prevented browsing packages from the Visual Studio UI with a minimum Visual Studio install
- Corrected an issue with Visual Studio locating `lucene.net.dll`
- All sources should not be the default repository source after a NuGet extension install or update. You can opt-in to this feature from the configuration settings.

We continue to track issues on our GitHub issues list which can be found at: <http://github.com/nuget/home/issues>

NuGet 3.4 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.4-RC Release Notes](#) | [NuGet 3.4.1 Release Notes](#)

NuGet 3.4 was released March 30, 2016 as part of the Visual Studio 2015 Update 2 and Visual Studio 15 Preview Release and was built with a few tenets in mind:

- Cross-Platform support
- Performance improvements
- Minor UI improvements

The following features were previously added in the RC and have been updated or completed for the 3.4 release:

New Features

- NuGet clients now support gzip content-encoding from repositories
- Support for PDBs from packages in xproj projects
- Support for iOS and Android build actions in the contentFiles element
- Support for the netstandard and netstandardapp framework monikers

New User Interface Features

- Significant performance improvements especially on the Installed, Updates, and Consolidate tabs
- Aggregate 'All Package Sources' Source is available with proper search result merging
- Installed and Updates tabs are now sorted alphabetically
- Added a Refresh button that allows a search to be refreshed
- Latest Build options at the top of the Version list

Updates and Improvements

- Packages referenced in `project.json` that have a floating version will not update on every build. Instead, they will update only when forced to restore, clean, rebuild, or modify `project.json`.
- nuget.org repository sources are no longer forced into a project configuration when you use the NuGet configuration UI.
- NuGet no longer restores packages in shared projects nor writes a lock file.
- We've improved network failure and retry handling for unreachable or slow-to-respond servers.
- Keyboard and mouse behaviors are improved in the Visual Studio Package Manager UI.
- We now support the latest `project.json` schema in DNX.

Breaking Changes

- Package version numbers are now normalized to the format `major.minor.patch-prerelease` Each of major, minor, and patch are treated as integers and drop any leading zeroes. The prerelease information is treated as a string and no changes are applied to it. These numbers are used in queries by the NuGet clients and the search provided by the nuget.org service. More details can be found in the NuGet Docs under [Prerelease Versions](#).

Known Issues

- **Issue:** Windows 10 v1511 users may experience issues or even a Visual Studio crash with Powershell in Visual Studio in the following scenarios:
 - Installing / Uninstalling packages that have install.ps1 / uninstall.ps1 scripts
 - Loading projects that have an init.ps1 script (like EntityFramework)
 - Publishing web content
- **Workaround:** Ensure that your Windows 10 install has the latest patches applied, especially the January 2016 (KB 3124263) or a later update. More details are available on [GitHub issue #1638](#)
- **Issue:** NuGet v2 protocol redirects are broken. Custom NuGet repositories that redirect requests to an alternative host do not honor the redirect request.
- **Workaround:** To work around this issue, configure the package repository URI in settings to point to the redirected server location. For more information, see [GitHub pull request #387](#).

We continue to track issues on our GitHub issues list which can be found at: <http://github.com/nuget/home/issues>

NuGet 3.4-RC Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.3 Release Notes](#) | [NuGet 3.4 Release Notes](#)

NuGet 3.4-RC was released March 3, 2016 alongside the Visual Studio 2015 Update 2 RC and was built with a few tenets in mind:

- Cross-Platform support
- Performance improvements
- Minor UI improvements

The following features are available in this RC, with more planned for the 3.4 final release.

New Features

- NuGet clients now support gzip content-encoding from repositories
- Support for PDBs from packages in xproj projects
- Support for iOS and Android build actions in the contentFiles element
- Support for the netstandard and netstandardapp framework monikers

New User Interface Features

- Significant performance improvements especially on the Installed, Updates, and Consolidate tabs
- Installed and Updates tabs are now sorted alphabetically
- Added a Refresh button that allows a search to be refreshed

Updates and Improvements

- Packages referenced in `project.json` that have a floating version will not update on every build. Instead, they will update only when forced to restore, clean, rebuild, or modify `project.json`.
- nuget.org repository sources are no longer forced into a project configuration when you use the NuGet configuration UI.
- NuGet no longer restores packages in shared projects nor writes a lock file.
- We've improved network failure and retry handling for unreachable or slow-to-respond servers.
- Keyboard and mouse behaviors are improved in the Visual Studio Package Manager UI.
- We now support the latest `project.json` schema in DNX.

Known Issues

We continue to track issues on our GitHub issues list which can be found at: <http://github.com/nuget/home/Issues>

NuGet 3.3 Release Notes

7/18/2019 • 2 minutes to read • [Edit Online](#)

[NuGet 3.2.1 Release Notes](#) | [NuGet 3.4-RC Release Notes](#)

NuGet 3.3 was released November 30, 2015 with a significant number of user interface updates and command-line features as well as a collection of useful fixes to the NuGet clients.

New Features

- Credential Providers have been introduced that allow NuGet command-line clients to be able to work seamlessly with an authenticated feed. [Instructions on how to install the Visual Studio Team Services credential provider](#) and configure the NuGet clients to use it are available on NuGet Docs.

New User Interface Features

- Separate Browse, Installed, and Updates Available tabs
- Updates Available badge indicating the number of packages with available updates
- Package badges in the package list to indicate if the package is installed or has an update available
- Download count and author added to the package list
- Highest available version number and currently installed version number on the package list
- Action buttons to allow quick install, update, and uninstall from the package list
- Clearer action buttons on the package detail panel
- Package update date on the package detail panel
- Consolidate panel in Solution view
- Sortable grid of projects and installed version numbers on the solution view

New Command-line Features

In this version we introduced the `add` and `init` commands to initialize folder-based repositories as described in the [nuget.exe reference](#). Repositories that are constructed and maintained with this folder structure will [deliver significant performance benefits](#) as outlined on our blog.

ContentFiles

Content is now supported in `project.json` managed projects through the new `contentFiles` folder and `.nuspec` `contentFiles` element notation. This content can be more directly specified by the package author for interactions with project systems. More information about how to configure `contentFiles` in a `.nuspec` document can be found in the [.nuspec Reference](#).

NuGet Locals Cache Management

The NuGet command-line has been updated to include information about how to manage the local caches on a workstation. More information about the `locals` command is available in the [NuGet command-line reference](#).

Fixed Issues

Notable Issues

- NuGet command-line restored support for restoring packages with a solution file on Mono - [1543](#)

The complete list of issues that were addressed in the 3.3 release can be found on GitHub under the [3.3 milestone](#).

The list of issues fixed in the 3.3 command-line release are recorded in the [3.3 Command-Line milestone](#).

Known Issues

We continue to track issues on our GitHub issues list which can be found at: <http://github.com/nuget/home/issues>

NuGet 3.2.1 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.2 Release Notes](#) | [NuGet 3.3 Release Notes](#)

NuGet 3.2.1 for the command-line was released October 12, 2015 with a handful of optimizations and fixes for the 3.2 release and is available from [dist.nuget.org](#).

Improvements

- NuGet now uses the configuration file with the original casing of `NuGet.Config`. This is important on case-sensitive operating systems [1427](#)
- NuGet restore will now ignore dnx projects (`*.xproj`) that should be processed with `dnu` [1227](#)
- Optimized network utilization when working with `index.json` and package registration data [1426](#)
- Improved resource download handling to be more robust with v2 services [1448](#)

Fixes

- NuGet update correctly updates `.csproj` / `.vcxproj` references [1483](#)
- Now preventing a local `.nuget` folder from being created when a `SpecialFolders.UserProfile` cannot be located [1531](#)
- Improved handling of packages in local cache that are corrupted during download [1405](#) [1157](#)

A complete list of issues addressed for the command-line and Visual Studio extension can be found in the NuGet GitHub [3.2.1 milestone](#)

Known Issues

We continue to track issues on our GitHub issues list which can be found at: <http://github.com/nuget/home/Issues>

NuGet 3.2 Release Notes

9/4/2018 • 5 minutes to read • [Edit Online](#)

[NuGet 3.2-RC Release Notes](#) | [NuGet 3.2.1 Release Notes](#)

NuGet 3.2 was released September 16, 2015 as a collection of improvements and fixes for the 3.1.1 release and is available from both [dist.nuget.org](#) and the [Visual Studio Gallery](#).

New Features

- Projects that live in the same folder can now have different `project.json` files in that folder specific to each project. For each project, name the `project.json` file `{ProjectName}.project.json` and NuGet will give preference to that configuration for each project appropriately. This is only supported with Windows 10 Tools v1.1 installed - [1102](#)
- NuGet clients support specifying a global `NUGET_PACKAGES` environment variable to specify the location of the shared global packages folder used in `project.json` managed projects with Windows 10 tools v1.1.

Command-line updates

This is the first version of the nuget.exe client that supports the NuGet v3 servers and restoring packages for projects managed with a `project.json` file.

There were a number of authenticated feed issues that were addressed in this release to improve interactions with the client.

- Install / restore interactions only submit credentials for the initial request to the authenticated feed - [1300](#), [456](#)
- Push command does not resolve credentials from configuration - [1248](#)
- User agent and headers are now submitted to NuGet repositories to help with statistics tracking - [929](#)

We made a number of improvements to better handle network failures while attempting to work with a remote NuGet repository:

- Improved error messages when unable to connect to remote feeds - [1238](#)
- Corrected NuGet restore command to properly return a 1 when an error condition occurs - [1186](#)
- Now retrying network connections every 200ms for a maximum of 5 attempts in the case of HTTP 5xx failures - [1120](#)
- Improved handling of server redirect responses during a push command - [1051](#)
- `nuget install -source` now supports either URL or repository name from Nuget.Config as an argument - [1046](#)
- Missing packages that were not located on a repository during a restore are now reported as errors instead of warnings [1038](#)
- Corrected multipartwebrequest handling of \r\n for Unix/Linux scenarios - [776](#)

There are a number of fixes to issues with various commands:

- Push command no longer does a GET before a PUT against a package source - [1237](#)
- List command no longer repeats version numbers - [1185](#)
- Pack with the -build argument now properly supports C# 6.0 - [1107](#)
- Corrected issues attempting to pack an F# project built with Visual Studio 2015 - [1048](#)
- Restore now no-ops when packages already exist - [1040](#)
- Improved error messages when `packages.config` file is malformed - [1034](#)

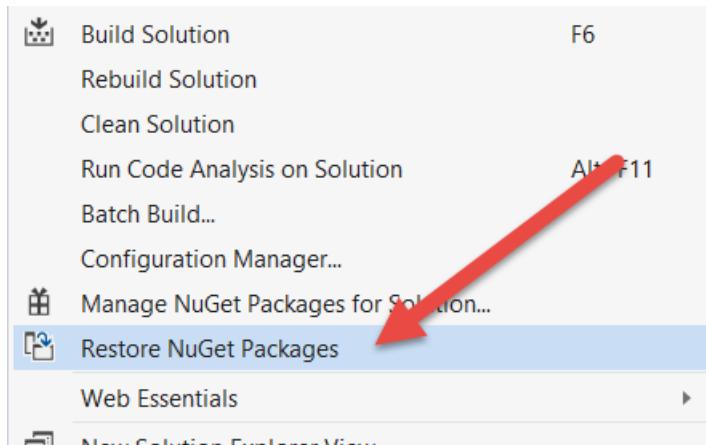
- Corrected restore command with `-SolutionDirectory` switch to work with relative paths - [992](#)
- Improved `Update` command to support solution-wide update - [924](#)

A complete list of issues addressed in this release can be found in the NuGet GitHub [Command-Line milestone](#).

Visual Studio extension updates

New Features in Visual Studio

- A new context menu item was added to the Solution Explorer on the solution node that allows packages to be restored without building the solution ([1274](#)).



Updates and Fixes in Visual Studio

The fixes for authenticated feeds were rolled up and addressed in the extension as well. The following authentication items were also addressed in the extension:

- Now correctly treating NuGet v3 authenticated feeds properly, instead of as v2 authenticated feeds - [1216](#)
- Corrected request for authentication credentials in projects using `project.json` and communicating with v2 feeds - [1082](#)

Network connectivity had affected the user interface in Visual Studio, and we addressed this with the following fixes:

- Improved the maintenance of the local cache of package versions - [1096](#)
- Changed the failure behavior when connecting to a v3 feed to no longer attempt to treat it as a v2 feed - [1253](#)
- Now preventing install failures when installing a package with multiple package sources - [1183](#)

We improved handling of interactions with build operations:

- Now continuing to build projects if restoring packages for a single project fails - [1169](#)
- Installing a package into a project that is depended on by another project in the solution forces a solution rebuild - [981](#)
- Corrected failed package installs to properly rollback changes to a project - [1265](#)
- Corrected inadvertent removal of the `developmentDependency` attribute on a package in `packages.config` - [1263](#)
- Calls to `install.ps1` now have a proper `$package.AssemblyReferences` object passed - [1245](#)
- No longer preventing uninstalls of packages in UWP projects while the project is in a bad state - [1128](#)
- Solutions containing a mix of `packages.config` and `project.json` projects are now properly built without requiring a second build operation - [1122](#)
- Properly locating `app.config` files if they are linked or located in a different folder - [1111, 894](#)
- UWP projects can now install unlisted packages - [1109](#)
- Package restore is now allowed while a solution is not in a saved state - [1081](#)

Handling updates to configuration files were corrected:

- No longer removing a targets file delivered from a package on subsequent builds of a `project.json` managed project - [1288](#)
- No longer modifying Nuget.Config files during ASP.NET 5 solution build - [1201](#)
- No longer changing allowed versions constraint during package update - [1130](#)
- Lock files now remain locked during build - [1127](#)
- Now modifying `packages.config` and not rewriting it during updates - [585](#)

Interactions with TFS source control are improved:

- No longer failing installs for packages that are bound to TFS - [1164](#), [980](#)
- Corrected NuGet user interface to allow TFS 2013 integration - [1071](#)
- Corrected references to packages restored to properly come from a packages folder - [1004](#)

Finally, we also improved these items:

- Verbosity of log messages reduced for `project.json` managed projects - [1163](#)
- Now properly displaying the installed version of a package in the user interface - [1061](#)
- Packages with dependency ranges specified in their nuspec now have pre-release versions of those dependencies installed for a stable package version - [1304](#)

A complete list of issues addressed for the Visual Studio extension can be found in the NuGet GitHub [3.2 milestone](#)

Known Issues

We continue to track issues on our GitHub issues list which can be found at: <http://github.com/nuget/home/Issues>

NuGet 3.2 RC Release Notes

9/4/2018 • 4 minutes to read • [Edit Online](#)

[NuGet 3.1.1 Release Notes](#) | [NuGet 3.2 Release Notes](#)

NuGet 3.2 release candidate was released September 2, 2015 as a collection of improvements and fixes for the 3.1.1 release. Also, these are the first releases that are published first to the new dist.nuget.org repository.

New Features

- Projects that live in the same folder can now have different `project.json` files in that folder specific to each project. For each project, name the `project.json` file `{ProjectName}.project.json` and NuGet will properly reference and use that content for each project appropriately. This supports a new feature [1102](#)
- `NuGet.Config` now supports a `globalPackagesFolder` as a relative path - [1062](#)

Command-line updates

This is the first version of the nuget.exe client that supports the NuGet v3 servers and restoring packages for projects managed with a `project.json` file.

There were a number of authenticated feed issues that were addressed in this release to improve interactions with the client.

- Install / restore interactions only submit credentials for the initial request to the authenticated feed - [1300](#), [456](#)
- Push command does not resolve credentials from configuration - [1248](#)
- User agent and headers are now submitted to NuGet repositories to help with statistics tracking - [929](#)

We made a number of improvements to better handle network failures while attempting to work with a remote NuGet repository:

- Improved error messages when unable to connect to remote feeds - [1238](#)
- Corrected NuGet restore command to properly return a 1 when an error condition occurs - [1186](#)
- Now retrying network connections every 200ms for a maximum of 5 attempts in the case of HTTP 5xx failures - [1120](#)
- Improved handling of server redirect responses during a push command - [1051](#)
- `nuget install -source` now supports either URL or repository name from Nuget.Config as an argument - [1046](#)
- Missing packages that were not located on a repository during a restore are now reported as errors instead of warnings [1038](#)
- Corrected multipartwebrequest handling of \r\n for Unix/Linux scenarios - [776](#)

There are a number of fixes to issues with various commands:

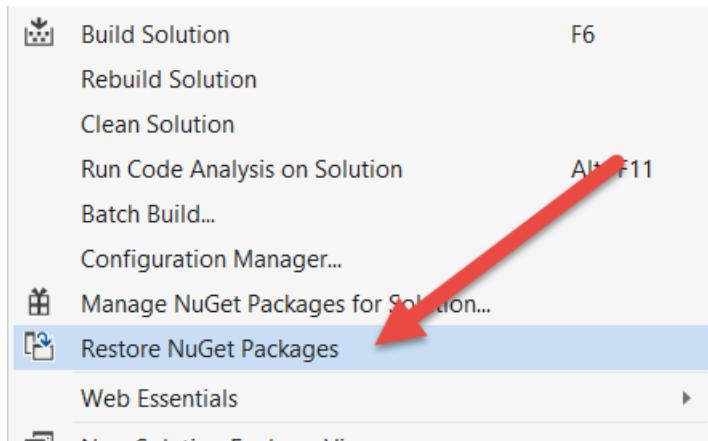
- Push command no longer does a GET before a PUT against a package source - [1237](#)
- List command no longer repeats version numbers - [1185](#)
- Pack with the -build argument now properly supports C# 6.0 - [1107](#)
- Corrected issues attempting to pack an F# project built with Visual Studio 2015 - [1048](#)
- Restore now no-ops when packages already exist - [1040](#)
- Improved error messages when `packages.config` file is malformed - [1034](#)
- Corrected restore command with `-SolutionDirectory` switch to work with relative paths - [992](#)
- Improved Updated command to support solution-wide update - [924](#)

A complete list of issues addressed in this release can be found in the NuGet GitHub [Command-Line milestone](#).

Visual Studio extension updates

New Features in Visual Studio

- A new context menu item was added to the Solution Explorer on the solution node that allows packages to be restored without building the solution ([1274](#)).



Updates and Fixes in Visual Studio

The fixes for authenticated feeds were rolled up and addressed in the extension as well. The following authentication items were also addressed in the extension:

- Now correctly treating NuGet v3 authenticated feeds properly, instead of as v2 authenticated feeds - [1216](#)
- Corrected request for authentication credentials in projects using `project.json` and communicating with v2 feeds - [1082](#)

Network connectivity had affected the user interface in Visual Studio, and we addressed this with the following fixes:

- Improved the maintenance of the local cache of package versions - [1096](#)
- Changed the failure behavior when connecting to a v3 feed to no longer attempt to treat it as a v2 feed - [1253](#)
- Now preventing install failures when installing a package with multiple package sources - [1183](#)

We improved handling of interactions with build operations:

- Now continuing to build projects if restoring packages for a single project fails - [1169](#)
- Installing a package into a project that is depended on by another project in the solution forces a solution rebuild - [981](#)
- Corrected failed package installs to properly rollback changes to a project - [1265](#)
- Corrected inadvertent removal of the `developmentDependency` attribute on a package in `packages.config` - [1263](#)
- Calls to `install.ps1` now have a proper `$package.AssemblyReferences` object passed - [1245](#)
- No longer preventing uninstalls of packages in UWP projects while the project is in a bad state - [1128](#)
- Solutions containing a mix of `packages.config` and `project.json` projects are now properly built without requiring a second build operation - [1122](#)
- Properly locating app.config files if they are linked or located in a different folder - [1111, 894](#)
- UWP projects can now install unlisted packages - [1109](#)
- Package restore is now allowed while a solution is not in a saved state - [1081](#)

Handling updates to configuration files were corrected:

- No longer removing a targets file delivered from a package on subsequent builds of a `project.json` managed project - [1288](#)
- No longer modifying Nuget.Config files during ASP.NET 5 solution build - [1201](#)
- No longer changing allowed versions constraint during package update - [1130](#)
- Lock files now remain locked during build - [1127](#)

- Now modifying `packages.config` and not rewriting it during updates - [585](#)

Interactions with TFS source control are improved:

- No longer failing installs for packages that are bound to TFS - [1164, 980](#)
- Corrected NuGet user interface to allow TFS 2013 integration - [1071](#)
- Corrected references to packages restored to properly come from a packages folder - [1004](#)

Finally, we also improved these items:

- Verbosity of log messages reduced for `project.json` managed projects - [1163](#)
- Now properly displaying the installed version of a package in the user interface - [1061](#)

A complete list of issues addressed for the Visual Studio extension can be found in the NuGet GitHub [3.2 milestone](#)

Known Issues

We continue to track issues on our GitHub issues list which can be found at: <http://github.com/nuget/home/issues>

NuGet 3.1.1 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.1 Release Notes](#) | [NuGet 3.2-RC Release Notes](#)

NuGet 3.1.1 was released July 27, 2015 as a patch update to the 3.1 VSIX with fix specific to a bug that effected Powershell policy implementation. <https://github.com/NuGet/Home/issues/974>

NuGet 3.1 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.0 Release Notes](#) | [NuGet 3.1.1 Release Notes](#)

NuGet 3.1 was released on July 27, 2015 as a bundled extension to the Universal Windows Platform SDK for Visual Studio 2015. We delivered this release with the Windows Platform SDK so that the Windows development experience could take advantage of the NuGet cross-platform work that had been previously started. This NuGet extension version is only available for Visual Studio 2015.

We recommend those developers that have access to the Visual Studio gallery update to the latest version that is available, as we are always publishing updates with bug fixes and new features.

NuGet Visual Studio Extension

Issues and features in this release are tagged on GitHub with the ["3.1 RTM UWP transitive support" milestone](#) In total, we closed 67 issues in the 3.1 release.

New Features

- `project.json` support for Windows UWP and ASP.NET 5 support
- Transitive package installation

Description and definition of these features can be found elsewhere in the documentation.

Deprecated

The following features are no longer available for Visual Studio 2015:

- Solution level packages can no longer be installed

The following features are no longer available for Visual Studio 2015 and projects that use the `project.json` specification

- `install.ps1` and `uninstall.ps1` - These scripts will be ignored during package install, restore, update, and uninstall
- Configuration transforms will be ignored
- Content will be carried, but not copied into a project.
 - The team is working to re-implement this feature, follow the discussion and progress at:
<https://github.com/NuGet/Home/issues/627>

Known Issues

There were a number of known issues delivered with this release.

- Installation of the 3.1 release with the Windows 10 SDK will downgrade any version of NuGet extension that was previously installed

NuGet Command-line

The NuGet command-line executable was updated and moved to a new distributable location so that historical versions of nuget.exe can continue to be made available. You can download the 3.1 beta version of nuget.exe for Windows at: <http://dist.nuget.org/win-x86-commandline/v3.1.0-beta/nuget.exe>

The new distributable location resides on the dist.nuget.org host, with a folder structure that follows this template:

```
{platform supported}/{version}/nuget.exe
```

New Features

- nuget.exe can restore and install packages into projects that use a `project.json` file.
- nuget.exe can connect to and consume the NuGet v3 protocol at: <https://api.nuget.org/v3/index.json>

Known Issues

1. Cannot execute pack against a `project.json` file - [928](#)
2. Is not supported on Mono - [1059](#)
3. Is not localized - [1058](#), [1057](#)
4. Is not signed, just like the existing <http://nuget.org/nuget.exe> - [1073](#)

NuGet 3.0 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.0 RC2 Release Notes](#) | [NuGet 3.1 Release Notes](#)

NuGet 3.0 was released on July 20, 2015 as a bundle extension to Visual Studio 2015. We pushed to deliver this release with Visual Studio so that the complete updated NuGet 3.0 experience would be available for new Visual Studio users. This NuGet extension version is only available for Visual Studio 2015.

We recommend those developers that have access to the Visual Studio gallery update to the latest version that is available, as we are publishing an update shortly after the release of Visual Studio 2015 that contains support for Windows 10 development.

In total, we closed 240 issues in the 3.0 release, and you can review the [complete list of issues on GitHub](#).

Known Issues

There were a number of known issues delivered with this release, and all of these items are fixed in our scheduled 3.1 release to coincide with the release of Windows 10 on July 29th. You are able to update your Visual Studio extension from the gallery on or after that date to fix these known issues.

- Translation is not provided for the "Do not show this again" label on the preview window and the "Authors" label in the package description window.
- When you working on a project by using TFS source control, NuGet cannot present the package manager user interface if the Nuget.Config file is marked as read-only.
 - **Workaround** Check out the file from TFS.
- Text in the yellow "restart bar" in the NuGet Powershell window is not visible when you use the Visual Studio dark theme.
 - **Workaround** Use the Visual Studio light theme.

Summary of top issues resolved

- [Frequent network update calls when package manager window refreshes](#)
- [Delayed scroll when changing to installed view in package manager](#)
- [Network calls should be run on a background thread](#)
- [Added 'Do not show preview window' checkbox](#)
- [Added process throttling to reduce processor usage](#)
- Improved portable-class-library reference handling
 - <https://github.com/NuGet/Home/issues/562>
 - <https://github.com/NuGet/Home/issues/454>
 - <https://github.com/NuGet/Home/issues/440>
- [Autocomplete service was case sensitive](#)
- [Update to reintroduce basic auth credentials](#)
- [Improved error logging](#)
- [Improved powershell error messages when calling Update-Package](#)
- [Fixed the 'Learn about Options' link to prevent crashing on Windows 10](#)
- [Remember pre-release checkbox setting](#)
- [Improved gather performance by caching results across projects in a solution](#)

- [Multiple Packages can be gathered in parallel](#)
- [Removed install-package -force command](#)

Please keep an eye on [our blog](#) for more progress and announcements as we get ready to deliver support for Windows 10 development.

NuGet 3.0 RC2 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.0 RC Release Notes](#) | [NuGet 3.0 Release Notes](#)

NuGet 3.0 RC2 was released on June 3, 2015 as an interim release available from the Visual Studio 2015 Extension Gallery and [Codeplex](#). This release has a number of important bug fixes and performance improvements that we felt were important to release before the completed Visual Studio 2015 release. This NuGet extension version is only available for Visual Studio 2015.

In total, we closed 158 issues in this release, and you can review the [complete list of issues on GitHub](#).

Summary of top issues resolved

- [Frequent network update calls when package manager window refreshes](#)
- [Delayed scroll when changing to installed view in package manager](#)
- [Network calls should be run on a background thread](#)
- [Added 'Do not show preview window' checkbox](#)
- [Added process throttling to reduce processor usage](#)
- Improved portable-class-library reference handling
 - <https://github.com/NuGet/Home/issues/562>
 - <https://github.com/NuGet/Home/issues/454>
 - <https://github.com/NuGet/Home/issues/440>
- [Autocomplete service was case sensitive](#)
- [Update to reintroduce basic auth credentials](#)
- [Improved error logging](#)
- [Improved powershell error messages when calling Update-Package](#)

Download this [update to the NuGet extension](#) from Codeplex and please keep an eye on [our blog](#) for more progress and announcements for NuGet 3.0!

NuGet 3.0 RC Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 3.0 Beta Release Notes](#) | [NuGet 3.0 RC2 Release Notes](#)

NuGet 3.0 RC was released on April 29, 2015 with the Visual Studio 2015 RC release. This release has a number of important bug fixes, performance improvements and updates to support the new frameworks. It is only available for Visual Studio 2015.

Continued Focus on Performance

Stability and performance of NuGet queries continue to be a hot topic that we are focusing on. With this release, you should start to see very quick search operations in the NuGet UI and website. We're monitoring the service and how you use the service so that we can continue to tune these operations.

Significant Issues Resolved

In order to stabilize the NuGet clients, we resolved many issues as part of this release. Here is just a brief list of some of the more important issues resolved:

- As part of the rename of the K framework for ASP.NET 5, framework monikers have been updated to handle dnx and dnxcore [link](#)
- Added help documentation from links in the Visual Studio UI [link](#)
- Better handling of complex references in `.nuspec` with comma-delimited framework references [link](#)
- Fixed support for Japanese cultures [link](#)
- Updated client to allow ASP.NET 5 projects to use new v3 endpoints [link](#)
- Updated to better handle packages folder with source control [link](#)
- Fixed support for satellite packages [link](#)
- Corrected support for framework-specific content files [link](#)

GitHub presence overhaul

We've made some changes to our [source code repositories on GitHub](#). If you have any issues with the NuGet Visual Studio client, the Powershell commands, or the command-line executable you can log those issues and monitor their progress on our [GitHub Home repository issues list](#). We are tracking issues for the gallery in our [GitHub NuGetGallery repository](#).

Stay Tuned

Please keep an eye on [our blog](#) for more progress and announcements for NuGet 3.0!

NuGet 3.0 Beta Release Notes

9/4/2018 • 3 minutes to read • [Edit Online](#)

[NuGet 3.0 Preview Release Notes](#) | [NuGet 3.0 RC Release Notes](#)

NuGet 3.0 Beta was released on February 23, 2015 for the Visual Studio 2015 CTP 6 release. This release means a lot to our team, as we have a number of architecture and performance improvements to share, and we're excited to start tuning the performance settings on our nuget.org service.

We strongly recommend that you uninstall any prior version of the NuGet Visual Studio 2015 extension before installing this new version. If you have any problems with this version of the extension, we recommend you revert to the [prior version](#) for use with Visual Studio 2015 preview.

Visual Studio 2012+

This NuGet 3.0 Beta is available to install in the Visual Studio 2015 CTP 6 Extension Gallery. We are working to get preview drops out for Visual Studio 2012 and Visual Studio 2013 very soon. We previously shared our intent to [discontinue updates for Visual Studio 2010](#), and we did make that difficult decision.

New Client/Server API

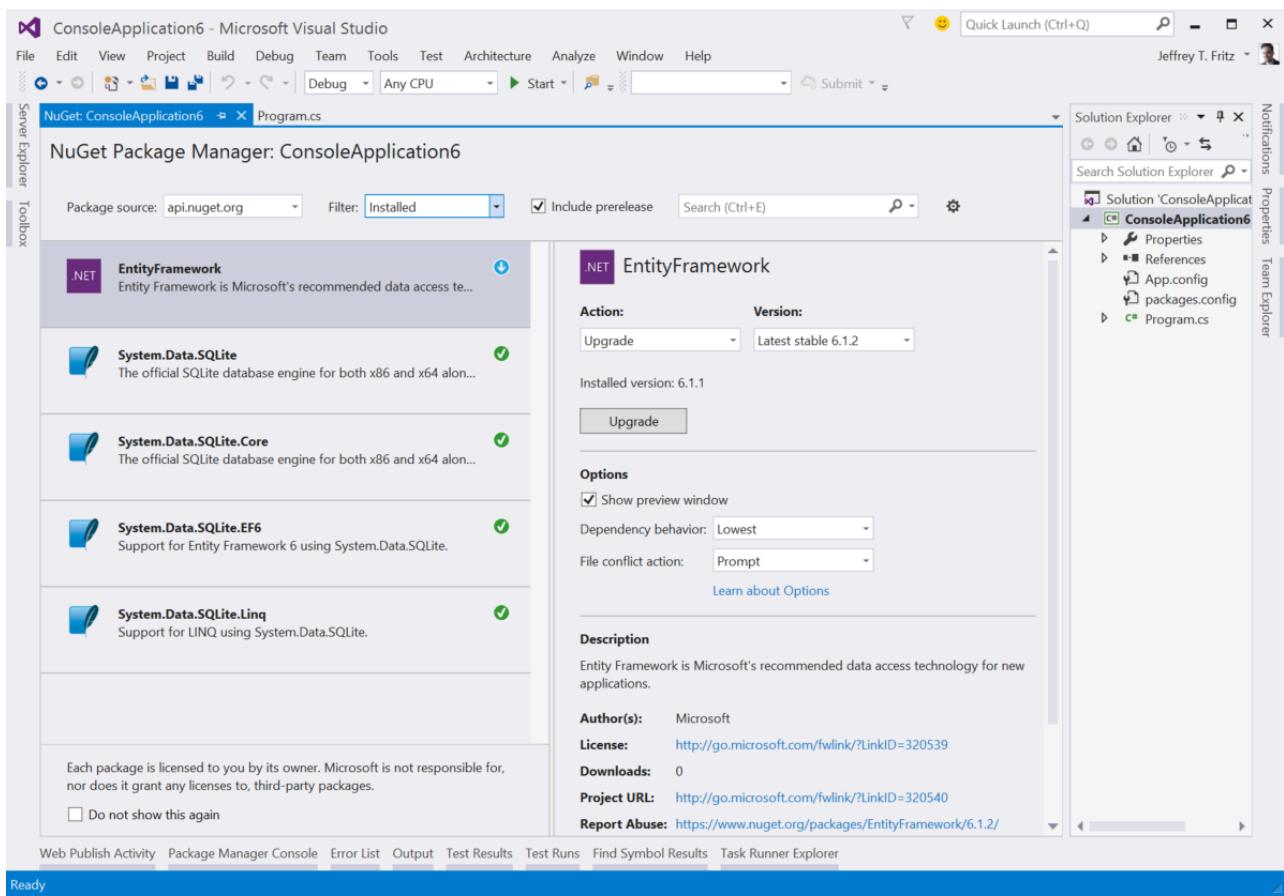
We've been working on some implementation details for NuGet's client/server protocol. The work we've done is to create "API v3" for NuGet, which is designed around high availability for critical scenarios such as package restore and installing packages. The new API is based on REST and Hypermedia and we've selected [JSON-LD](#) as our resource format.

In the NuGet 3.0 Beta bits, you see a new package source called "api.nuget.org" in the package source dropdown. If you select that package source, we'll use our new API rather to connect to nuget.org. In NuGet 3.0 RC, this new API v3-based package source will replace the v2-based "nuget.org" package source. We recommend disabling all of the other public package sources and leave only api.nuget.org as your only public package repository.

We've put a lot of time into building our v3 API and will continue to maintain the standard v2 API for old clients seeking to access the public repository.

Updated UI

We've enhanced the user-interface in this release to include a combobox that will allow you to choose an action to take with the package and transitioned the preview button into a checkbox in the options area of the screen. The options area is no longer collapsible and now provides a help link describing the options available.



Operation Logging

We removed the modal window with logging information that would quickly appear and hide while installing or uninstalling. This window added no value when you would really want to see the information or be able to copy and paste from it. Instead, we are now redirecting all of the output logging to the Package Manager pane of the Output window. We think this is more comfortable and similar to a typical build report that you would want to inspect.

Focus on Performance

We made a lot of changes in the name of improving performance of NuGet searches, and fetches. This was our number one concern from our customers, and we wanted to be sure we addressed it in this release. We've tuned our servers, built out a new CDN, and improved the query matching logic to hopefully deliver to you more relevant and faster package search results.

As we proceed through this phase of the development of NuGet 3.0, we will be tuning and monitoring the nuget.org service to ensure that we deliver an improved experience. We do not plan to engage in any downtime, but will be adding and changing resources in the service. Keep an eye on our [twitter feed](#) for details on when we change the service configuration.

Building NuGet with NuGet

We have now rearchitected our NuGet clients into several components that are themselves being built into NuGet packages. This re-use of our own libraries forces us to build components that are re-usable and that can be packaged properly. We have been able to eliminate duplicated code and have learned how to better configure our development process to support the need to build packages throughout our solutions. Look for a blog post soon where we will talk about how the code projects are structured and how our build process works.

Stay Tuned

Please keep an eye on [our blog](#) for more progress and announcements for NuGet 3.0!

NuGet 3.0 Preview Release Notes

9/4/2018 • 7 minutes to read • [Edit Online](#)

[NuGet 2.9 RC Release Notes](#) | [NuGet 3.0 Beta Release Notes](#)

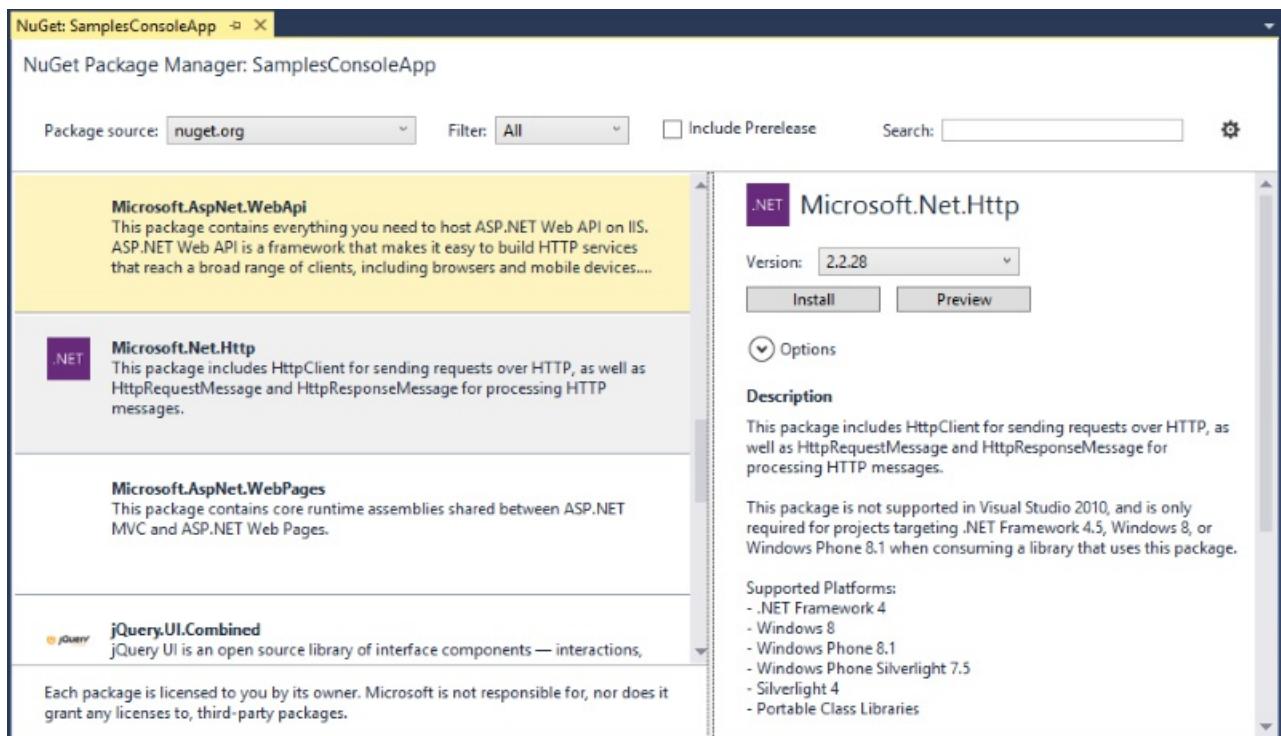
NuGet 3.0 Preview was released on November 12, 2014 as part of the Visual Studio 2015 Preview release. We released NuGet 3.0 Preview. This is a big release for us (albeit a preview), and we're excited to start getting feedback on our changes.

Visual Studio 2012+

This NuGet 3.0 Preview is included in Visual Studio 2015 Preview. We are working to get preview drops out for Visual Studio 2012 and Visual Studio 2013 very soon. We previously shared our intent to [discontinue updates for Visual Studio 2010](#), and we did make that difficult decision.

Brand New UI

The first thing you notice about NuGet 3.0 Preview is our brand new UI. It's no longer a modal dialog; it's now a full Visual Studio document window. This allows you to open the UI for multiple projects (and/or the solution) at once, tear the window off to another monitor, dock it however you'd like, etc.



Beyond the usability differences because of abandoning the modal dialog, we also have lots of new features in the new UI.

Version Selection

Perhaps the most requested UI feature is to allow version selection for package installation and update--this is now available.

Version: 2.2.28

Latest stable 2.2.28

2.2.28

2.2.27-beta

2.2.23-beta

2.2.22

This package enables sending requests over HTTP, as well as HttpResponseMessage and HTTP messages.

2.2.20

2.2.19

2.2.18

This package is required for Visual Studio 2010, and is only required for projects targeting .NET Framework when consuming a library that uses this package.

4.5, Windows 8

2.2.15

2.2.13

Supported

- .NET Framework
- Windows 8
- Windows 7
- Windows Vista
- Silverlight
- Portable

2.2.10-rc

2.2.7-beta

2.2.3-beta

2.1.10

2.1.6-rc

2.1.3-beta

Author(s):

- 2.0.20710
- 2.0.20505

Whether you are installing or updating a package, the version dropdown allows you to see all of the versions available for the package, with some notable versions promoted to the top of the list for easy selection. You no longer need to use the PowerShell Console to get specific versions that are not the latest.

Combined Installed/Online/Updates Workflows

Our previous UI had 3 tabs for Installed, Online, and Updates. The packages listed were specific to those workflows and the actions available were specific to the workflows as well. While that seemed logical, we heard that many of you would often get tripped up by this separation.

We now have a combined experience, where you can install, update, or uninstall a package regardless of how you got the package selected. To assist with the specific workflows, we now have a Filter dropdown that lets you filter the packages visible, but then the actions available for the package are consistent.

Package source: nuget.org Filter: Installed Include Prerelease Search:

Antlr

ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, interpreters, compilers...

Newtonsoft.Json

Json.NET is a popular high-performance JSON framework for .NET

Ninject

Stop writing monolithic applications that make you feel like you have to move mountains to make the simplest of changes....

Newtonsoft.Json

Version: 5.0.4

Uninstall Preview

Description

Json.NET is a popular high-performance JSON framework for .NET

Author(s): James Newton-King

License: <http://json.codeplex.com/license>

Downloads: 8808784

Date Published: 4/25/2013 4:48 AM -07:00

Project URL: <http://james.newtonking.com/projects/json-net.aspx>

Tags: json

By using the "Installed" filter, you can then easily see your installed packages, which ones have updates available, and then you can either uninstall or update the package by changing the version selection to see change the action available.

Package source: [nuget.org](#) Filter: [Installed](#) Include Prerelease Search: [⚙](#)

Newtonsoft.Json
Version: [Latest stable 6.0.6](#) [Update](#) [Preview](#)

[Options](#)

Description
Json.NET is a popular high-performance JSON framework for .NET

Author(s): James Newton-King
License: <https://raw.github.com/JamesNK/Newtonsoft.Json/master/LICENSE.md>
Downloads: 8808784
Date Published: 10/24/2014 10:57 AM -07:00
Project URL: <http://james.newtonking.com/json>
Tags: json

No more items

Version Consolidation

It's common to have the same package installed into multiple projects within your solution. Sometimes the versions installed into each project can drift apart and it's necessary to consolidate the versions in use. NuGet 3.0 Preview introduces a new feature for just this scenario.

The solution-level package management window can be accessed by right-clicking on the solution and choosing Manage NuGet Packages for Solution. From there, if you select a package that is installed into multiple projects, but with different versions in use, a new "Consolidate" action becomes available. In the screen shot below,

Newtonsoft.Json was installed into the [SamplesClassLibrary](#) with version [6.0.4](#) and installed into [SamplesConsoleApp](#) with version [5.0.4](#).

NuGet Package Manager: Solution 'Samples'

Package source: [nuget.org](#) Filter: [Installed](#) Include Prerelease Search: [⚙](#)

Newtonsoft.Json
Json.NET is a popular high-performance JSON framework for .NET

Antlr
ANOther Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators...

Ninject
Stop writing monolithic applications that make you feel like you have to move mountains to make the simplest of changes. Ninject helps you use the technique...

Newtonsoft.Json
Action: [Consolidate](#) Version: [6.0.4](#)
Select which projects to apply:
 SamplesClassLibrary (6.0.4)
 SamplesConsoleApp (5.0.4)

[Consolidate](#) [Preview](#)

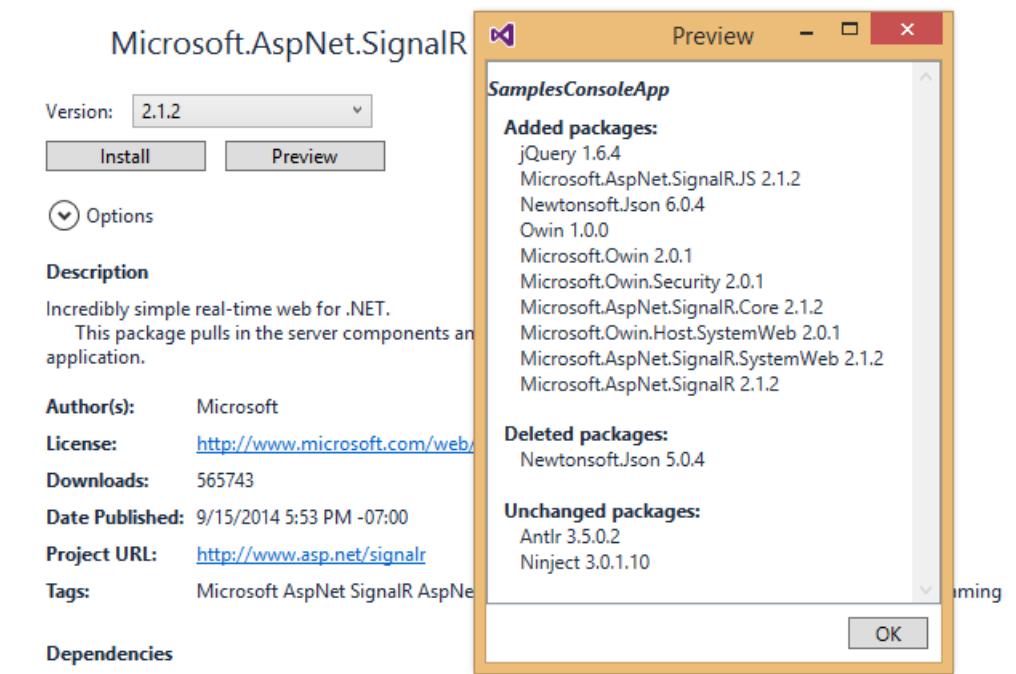
Here's the workflow for consolidating onto a single version.

1. Select the [Newtonsoft.Json](#) package in the list
2. Choose [Consolidate](#) from the [Action](#) dropdown
3. Use the [Version](#) dropdown to select the version to be consolidated onto
4. Check the boxes for the projects that should be consolidated onto that version (note that projects already on the selected version will be greyed out)
5. Click the [Consolidate](#) button to perform the consolidation

Operation Previews

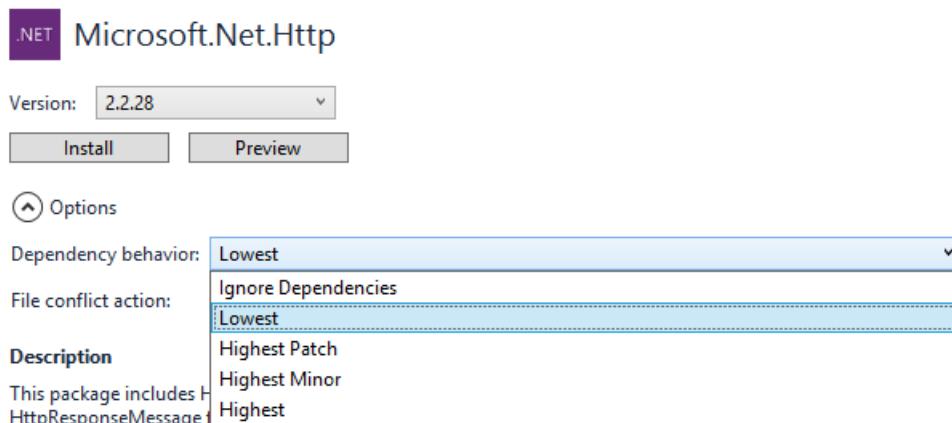
Regardless of which operation you're performing--install/update/uninstall--the new UI now offers a way to preview the changes that will be made to your project. This preview will show any new packages that will be installed, packages that will be updated, and packages that will be uninstalled, along with packages that will be unchanged during the operation.

In the example below, we can see that installing Microsoft.AspNet.SignalR will result in quite a few changes to the project.



Installation Options

Using the PowerShell Console, you've had control over a couple of notable installation options. We've now brought those features into the UI as well. You can now control the dependency resolution behavior for how versions of the dependencies are selected.



This package is not supported in Visual Studio 2010, and is only required for projects targeting .NET Framework 4.5, Windows 8, or Windows Phone 8.1 when consuming a library that uses this package.

Supported Platforms:

- .NET Framework 4
- Windows 8
- Windows Phone 8.1
- Windows Phone Silverlight 7.5
- Silverlight 4

You can also specify the action to take when content files from packages conflict with files already in your project.

.NET Microsoft.Net.Http

Version: 2.2.28

[Install](#)

[Preview](#)

[Options](#)

Dependency behavior: Lowest

File conflict action: Prompt

Prompt

Ignore All

This package includes [HttpMessageHandler](#) for processing HTTP messages.

This package is not supported in Visual Studio 2010, and is only required for projects targeting .NET Framework 4.5, Windows 8, or Windows Phone 8.1 when consuming a library that uses this package.

Supported Platforms:

- .NET Framework 4
- Windows 8
- Windows Phone 8.1
- Windows Phone Silverlight 7.5
- Silverlight 4

Infinite Scrolling

We used to get quite a bit of feedback on our UI having both the scrolling and paging paradigms when listing packages. It was pretty common to have to scroll to the bottom of the short list, click the next page number, and then scroll again. With the new UI, we've implemented infinite scrolling in the package list so that you only need to scroll--no more paging.

Package source: [nuget.org](#)

Filter: All

Include Prerelease

[Microsoft.Owin.Security.Cookies](#)

Middleware that enables an application to use cookie based authentication, similar to ASP.NET's forms authentication.



[Unity](#)

The Unity Application Block (Unity) is a lightweight extensible dependency injection container with support for constructor, property, and method call injection. It facilitates loosely-coupled design. Declarative configuration and...



[WindowsAzure.Storage](#)

This client library enables working with the Microsoft Azure storage services which include the blob and file service for storing binary and text data, the table service for storing structured non-relational data, and the queue service for st...

 Loading...

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

.NET M

Version:

[Install](#)

[Options](#)

Dependency:

File conflict:

Description:

This package includes [HttpMessageHandler](#) for processing HTTP messages.

This package is not supported in Visual Studio 2010, and is only required for projects targeting .NET Framework 4.5, Windows 8, or Windows Phone 8.1 when consuming a library that uses this package.

Supported Platforms:

- .NET Framework 4.5
- Windows 8
- Windows Phone 8.1
- Windows Phone Silverlight 7.5
- Silverlight 4

Make it Work, Make it Fast, Make it Pretty

We are excited to get this new UI out for you to try out. During this Preview milestone, we've been following the good old adage of "Make it work, make it fast, make it pretty." In this preview, we've accomplished most of that first goal--it works. We know it's not quite fast yet, and we know it's not quite pretty yet. Trust that we'll be working on those goals between now and the RC release. In the meantime, we would love to hear your feedback about the *usability* of the new UI--the workflows, operations, and how it *feels* to use the new UI.

There are a couple of functions that we've removed when compared to the old UI. One of these was intentional, and the other one just didn't get done in time.

Searching "All" Package Sources

The old UI allowed you to perform a package search against all of your package sources. We've removed that feature in the UI and we won't be bringing it back. This feature used to perform search operations against all of your package sources, weave the results together, and attempt to order the results based on your sorting selection.

We found that search relevance is really hard to weave together. Could you imagine performing a search against Google and Bing and weaving the results together? Additionally, this feature was slow, easy to *accidentally* use, and we believe it was rarely actually useful. Because of the problems the feature introduced, we received a number of bug reports on it that could never have been fixed.

Update All

We used to have an "Update All" button in the old UI that isn't there in the new UI yet. We will resurrect this feature for our RC release.

New Client/Server API

In addition to all of the new features in our new package management UI, we've also been working on some implementation details for NuGet's client/server protocol. The work we've done is to create "API v3" for NuGet, which is designed around high availability for critical scenarios such as package restore and installing packages. The new API is based on REST and Hypermedia and we've selected [JSON-LD](#) as our resource format.

In the NuGet 3.0 Preview bits, you see a new package source called "preview.nuget.org" in the package source dropdown. If you select that package source, we'll use our new API rather to connect to nuget.org. We've made the preview source available in the UI while we continue to test, revise, and improve the new API. In NuGet 3.0 RC, this new API v3-based package source will replace the v2-based "nuget.org" package source.

Despite the investment we're putting into API v3, we've made all of these new features also work with our existing API v2 protocol, which means they will work with existing package sources other than nuget.org as well.

New Features Coming

Between now and 3.0 RTM, we are also working on some fundamental new NuGet features, beyond what you see in the UI. Here's a short list of salient investment areas:

1. We're partnering with the Visual Studio and MSBuild teams to get [NuGet deeper into the platform](#).
2. We're working to abandon installation-time package conventions and instead apply those conventions at packaging time by introducing a new "authoritative" [package manifest](#).
3. We're working to refactor the NuGet codebase to make the client and server components reusable in different domains beyond package management in Visual Studio.
4. We're investigating the notion of "private dependencies" where a package can indicate that it has dependencies on other packages for implementation details only, and those dependencies shouldn't be surfaced as top-level dependencies.

Stay Tuned

Please keep an eye on [our blog](#) for more progress and announcements for NuGet 3.0!

NuGet 2.12 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

NuGet 2.12 RTM was released June 27, 2016 for Visual Studio 2013

Updates in this release

- Full NetStandard and NetCoreApp support for VS2013.
- Adding include/exclude to `.nuspec` dependency entries.
- Add support for "no_proxy" to specify proxy exceptions.
- TFS related fixes.

A list of fixes in this release can be found on GitHub in the [2.12 milestone](#)

Download the extension from Tools -> Extensions and Updates in Visual Studio

NuGet 2.12-RC Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

NuGet 2.12-RC was released June 22, 2016 as an update to the 2.12.0-rc VSIX for Visual Studio 2013.

Updates in this release

- Full NetStandard and NetCoreApp support for VS2013.
- Adding include/exclude to `.nuspec` dependency entries.
- Add support for "no_proxy" to specify proxy exceptions.
- TFS related fixes.

A list of fixes in this release can be found on GitHub in the [2.12 milestone](#)

Download the extension for:

- [Visual Studio 2013](#)

NuGet 2.9-RC Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.8.7 Release Notes](#) | [NuGet 3.0 Preview Release Notes](#)

NuGet 2.9 was released September 10, 2015 as an update to the 2.8.7 VSIX for Visual Studio 2012 and 2013.

Updates in this release

- Now skipping processing packages if their contained `.nuspec` document is malformed - [PR8](#)
- Corrected multipartwebrequest handling of `\r\n` for Unix/Linux scenarios - [776](#)
- Corrected integration with build events in Visual Studio 2013 Community edition - [1180](#)

The complete list of fixes in this release can be found on GitHub in the [2.8.8 milestone](#)

NuGet 2.8.7 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.8.6 Release Notes](#) | [NuGet 2.9-RC Release Notes](#)

NuGet 2.8.7 was released July 27, 2015 as a patch update to the 2.8.6 VSIX with fix specific to a bug that effected Powershell policy implementation. <https://github.com/NuGet/Home/issues/974>

NuGet 2.8.6 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.8.5 Release Notes](#) | [NuGet 2.8.7 Release Notes](#)

NuGet 2.8.6 was released July 20, 2015 as a minor update to our 2.8.5 VSIX with some targeted fixes and improvements to support packages that may be delivered with support for the Windows 10 UWP development model.

This version of the NuGet package manager extension provides support for Visual Studio 2013 only.

In this release, the NuGet Package Manager dialog had support added for:

- Introduced the UAP Target Framework Moniker to support Windows 10 Application Development.
- NuGet protocol version 3 endpoints
- Support for [Nuget.Config](#) protocolVersion attribute on repository sources. Default value is "2"
- Falling back to remote repository if a required package version is not available in the local cache

NuGet 2.8.5 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.8.3 Release Notes](#) | [NuGet 2.8.6 Release Notes](#)

NuGet 2.8.5 was released March 30, 2015. It is a minor update to our 2.8.3 VSIX with some targeted fixes.

In this release, the support for NuGet Package Manager dialog was added for [DNX Target Framework Monikers](#). These new framework monikers that are supported include:

- **core50** - A 'base' target framework moniker (TFM) that is compatible with the Core CLR.
- **dnx452** - A TFM specific to DNX-based apps using the full 4.5.2 version of the framework
- **dnx46** - A TFM specific to DNX-based apps using the full 4.6 version of the framework
- **dnxcore50** - A TFM specific to DNX-based apps using the Core 5.0 version of the framework

One bug was fixed that prevented packages from installing into FSharp projects properly:

<https://nuget.codeplex.com/workitem/4400>

NuGet 2.8.3 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.8.2 Release Notes](#) | [NuGet 2.8.5 Release Notes](#)

NuGet 2.8.3 was released October 17, 2014. It is a minor update to our 2.8.1 VSIX with some targeted fixes.

In this release, the support for NuGet Package Manager dialog was added for [ASP.NET vNext](#), [DevExtreme](#) and [BizTalk \(.btproj\)](#) project types. It also includes reliability bug fixes related to the scenarios of enabling package restore and saving package manager options.

NuGet 2.8.2 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.8.1 Release Notes](#) | [NuGet 2.8.3 Release Notes](#)

NuGet 2.8.2 was released on May 22, 2014. This release only included changes to the nuget.exe command-line, the NuGet.Server package and other NuGet packages. The release did not include an updated Visual Studio extension or WebMatrix extension.

Notable Updates

The most notable updates were in the nuget.exe command-line and the NuGet.Server package (for self-hosted NuGet feeds).

Important nuget.exe Bug Fixes

1. [nuget.exe Push fails and keeps retrying](#)
2. [nuget.exe Push does not send Basic Auth credentials correctly](#)
3. [nuget.exe Push won't follow temporary redirect](#)

Important NuGet.Server Bug Fix

1. [Wrong value of IsAbsoluteLatestVersion returned by NuGet.Server](#)

Packages Updated

The nuget.exe command-line and NuGet.Server fixes are shipped as NuGet package updates. There were other packages updated with 2.8.2 as well.

Here's the list of updated packages:

1. [NuGet.Core](#)
2. [NuGetCommandLine](#)
3. [NuGet.Server](#)
4. [NuGet.Build](#)
5. [NuGet.VisualStudio](#) (the package, not the extension)

All Changes

There were 10 issues addressed in the release. For a full list of the work items fixed in NuGet 2.8.2, please view the [NuGet Issue Tracker for this release](#).

NuGet 2.8.1 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.8 Release Notes](#) | [NuGet 2.8.2 Release Notes](#)

NuGet 2.8.1 was released on April 2, 2014.

Notable features in the release

Support for Windows Phone 8.1 Projects

This release now supports the following new target framework monikers which can be used to target Windows Phone 8.1 projects:

- WindowsPhone81 / WP81 (for Silverlight-based Windows Phone projects)
- WindowsPhoneApp81 / WPA81 (for WinRT-based Windows Phone App projects)

Update of the NuGet WebMatrix Extension

This release updates the NuGet client found in WebMatrix to [NuGet.Core](#) 2.6.1 and brings with it features such as XDT transformations. More importantly, the 2.6.1 core update enables the WebMatrix client to install NuGet packages which contain more recent versions of the `.nuspec` schema, which includes the ASP.NET NuGet packages.

For more information about the WebMatrix Extension update, see those specific [release notes](#).

Bug Fixes

In addition to these features, this release of NuGet includes other bug fixes. There were 16 total issues addressed in the release. For a full list of the work items fixed in NuGet 2.8.1, please view the [NuGet Issue Tracker for this release](#).

Reshipping with Visual Studio "14" CTP

In Visual Studio "14" CTP released on June 3rd 2014, NuGet 2.8.1 is shipped in the box. The features it supports remain in-par with other 2.8.1 VSIXes such as the one for Visual Studio 2013.

NuGet 2.8 Release Notes

9/4/2018 • 6 minutes to read • [Edit Online](#)

[NuGet 2.7.2 Release Notes](#) | [NuGet 2.8.1 Release Notes](#)

NuGet 2.8 was released on January 29, 2014.

Acknowledgements

1. [Llewellyn Pritchard \(@leppie\)](#)
 - [#3466](#) - When packing packages, verifying Id of dependency packages.
2. [Maarten Balliauw \(@maartenballiauw\)](#)
 - [#2379](#) - Remove the \$metadata suffix when persistening feed credentials.
3. [Filip De Vos \(@foxtricks\)](#)
 - [#3538](#) - Support specifying project file for the nuget.exe update command.
4. [Juan Gonzalez](#)
 - [#3536](#) - Replacement tokens not passed with -IncludeReferencedProjects.
5. [David Poole \(@Sarkie_Dave\)](#)
 - [#3677](#) - Fix nuget.push throwing OutOfMemoryException when pushing large package.
6. [Wouter Ouwens](#)
 - [#3666](#) - Fix incorrect target path when project references another CLI/C++ project.
7. [Adam Ralph \(@adamralph\)](#)
 - [#3639](#) - Allow packages to be installed as development dependencies by default
8. [David Fowler \(@davidfowl\)](#)
 - [#3717](#) - Remove implicit upgrades to the latest patch version
9. [Gregory Vandenbrouck](#)
 - Several bug fixes and improvements for NuGet.Server, the nuget.exe mirror command, and others.
 - This work was done over several months, with Gregory working with us on the right timing to integrate into master for 2.8.

Patch Resolution for Dependencies

When resolving package dependencies, NuGet has historically implemented a strategy of selecting the lowest major and minor package version which satisfies the dependencies on the package. Unlike the major and minor version, however, the patch version was always resolved to the highest version. Though the behavior was well-intentioned, it created a lack of determinism for installing packages with dependencies. Consider the following example:

```
PackageA@1.0.0 -[ >=1.0.0 ]-> PackageB@1.0.0

Developer1 installs PackageA@1.0.0: installed PackageA@1.0.0 and PackageB@1.0.0

PackageB@1.0.1 is published

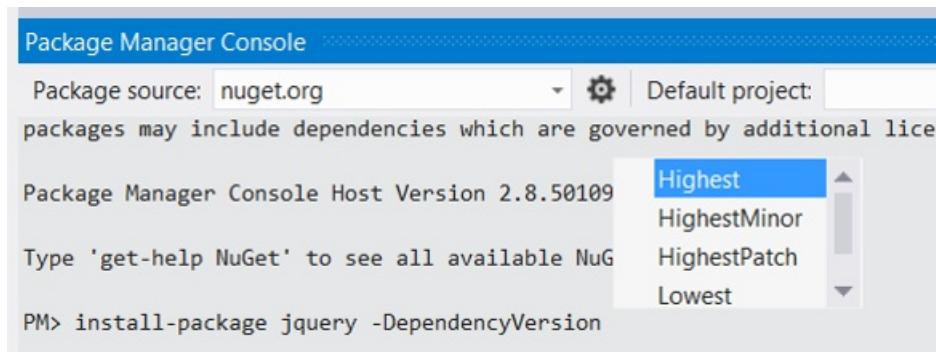
Developer2 installs PackageA@1.0.0: installed PackageA@1.0.0 and PackageB@1.0.1
```

In this example, even though Developer1 and Developer2 installed PackageA@1.0.0, each ended up with a different version of PackageB. NuGet 2.8 changes this default behavior such that the dependency resolution behavior for

patch versions is consistent with the behavior for major and minor versions. In the above example, then, PackageB@1.0.0 would be installed as a result of installing PackageA@1.0.0, regardless of the newer patch version.

-DependencyVersion Switch

Though NuGet 2.8 changes the *default* behavior for resolving dependencies, it also adds more precise control over dependency resolution process via the -DependencyVersion switch in the package manager console. The switch enables resolving dependencies to the lowest possible version (default behavior), the highest possible version, or the highest minor or patch version. This switch only works for install-package in the powershell command.



DependencyVersion Attribute

In addition to the -DependencyVersion switch detailed above, NuGet has also allowed for the ability to set a new attribute in the Nuget.Config file defining what the default value is, if the -DependencyVersion switch is not specified in an invocation of install-package. This value will also be respected by the NuGet Package Manager Dialog for any install package operations. To set this value, add the attribute below to your Nuget.Config file:

```
<config>
  <add key="dependencyversion" value="Highest" />
</config>
```

Preview NuGet Operations With -whatif

Some NuGet packages can have deep dependency graphs, and as such, it can be helpful during an install, uninstall, or update operation to first see what will happen. NuGet 2.8 adds the standard PowerShell -whatif switch to the install-package, uninstall-package, and update-package commands to enable visualizing the entire closure of packages to which the command will be applied. For example, running

```
install-package Microsoft.AspNet.WebApi -whatif
```

 in an empty ASP.NET Web application yields the following.

```
PM> install-package Microsoft.AspNet.WebApi -whatif
Attempting to resolve dependency 'Microsoft.AspNet.WebApi.WebHost (≥ 5.0.0)'.
Attempting to resolve dependency 'Microsoft.AspNet.WebApi.Core (≥ 5.0.0)'.
Attempting to resolve dependency 'Microsoft.AspNet.WebApi.Client (≥ 5.0.0)'.
Attempting to resolve dependency 'Newtonsoft.Json (≥ 4.5.11)'.
Install Newtonsoft.Json 4.5.11
Install Microsoft.AspNet.WebApi.Client 5.0.0
Install Microsoft.AspNet.WebApi.Core 5.0.0
Install Microsoft.AspNet.WebApi.WebHost 5.0.0
Install Microsoft.AspNet.WebApi 5.0.0
```

Downgrade Package

It is not uncommon to install a prerelease version of a package in order to investigate new features and then decide to roll back to the last stable version. Prior to NuGet 2.8, this was a multi-step process of uninstalling the prerelease

package and its dependencies, and then installing the earlier version. With NuGet 2.8, however, the update-package will now roll back the entire package closure (e.g. the package's dependency tree) to the previous version.

Development Dependencies

Many different types of capabilities can be delivered as NuGet packages - including tools that are used for optimizing the development process. These components, while they can be instrumental in developing a new package, should not be considered a dependency of the new package when it's later published. NuGet 2.8 enables a package to identify itself in the `.nuspec` file as a `developmentDependency`. When installed, this metadata will also be added to the `packages.config` file of the project into which the package was installed. When that `packages.config` file is later analyzed for NuGet dependencies during `nuget.exe pack`, it will exclude those dependencies marked as development dependencies.

Individual packages.config Files for Different Platforms

When developing applications for multiple target platforms, it's common to have different project files for each of the respective build environments. It is also common to consume different NuGet packages in different project files, as packages have varying levels of support for different platforms. NuGet 2.8 provides improved support for this scenario by creating different `packages.config` files for different platform-specific project files.

 <code>packages.reactiveUI.config</code>	1/12/2014 10:55 PM	CONFIG File	1 KB
 <code>packages.reactiveUI_Monoandroid.config</code>	1/12/2014 10:55 PM	CONFIG File	1 KB
 <code>packages.ReactiveUI_MonoMac.config</code>	1/12/2014 10:55 PM	CONFIG File	1 KB
 <code>packages.ReactiveUI_Monotouch.config</code>	1/12/2014 10:55 PM	CONFIG File	1 KB
 <code>packages.ReactiveUI_WP8.config</code>	1/12/2014 10:55 PM	CONFIG File	1 KB

Fallback to Local Cache

Though NuGet packages are typically consumed from a remote gallery such as [the NuGet gallery](#) using a network connection, there are many scenarios where the client is not connected. Without a network connection, the NuGet client was not able to successfully install packages - even when those packages were already on the client's machine in the local NuGet cache. NuGet 2.8 adds automatic cache fallback to the package manager console. For example, when disconnecting the network adapter and installing jQuery, the console shows the following:

```
PM> Install-Package jquery
The source at nuget.org [https://www.nuget.org/api/v2/] is unreachable. Falling back to NuGet Local Cache at
C:\Users\me\AppData\Local\NuGet\Cache
Installing 'jQuery 2.0.3'.
Successfully installed 'jQuery 2.0.3'.
Adding 'jQuery 2.0.3' to WebApplication18.
Successfully added 'jQuery 2.0.3' to WebApplication18.
```

The cache fallback feature does not require any specific command arguments. Additionally, cache fallback currently works only in the package manager console - the behavior does not currently work in the package manager dialog.

WebMatrix NuGet Client Updates

Along with NuGet 2.8, the NuGet extension for WebMatrix was also updated to include many of the major features delivered with [NuGet 2.5](#). New capabilities include those such as 'Update All', 'Minimum NuGet Version', and allowing for overwriting of content files.

To update your NuGet Package Manager extension in WebMatrix 3:

1. Open WebMatrix 3

2. Click the Extensions icon in the ribbon
3. Select the Updates tab
4. Click to update NuGet Package Manager to 2.5.0
5. Close and restart WebMatrix 3

This is the NuGet team's first release of the NuGet Package Manager extension for WebMatrix. The code was recently contributed by Microsoft into the open-source NuGet project. Previously, the NuGet integration was built into WebMatrix, and it could not be updated out of band from WebMatrix. We now have the capability to further update it alongside the rest of NuGet's client tools.

Bug Fixes

One of the major bug fixes made was performance improvement in the update-package -reinstall command.

In addition to these features and the aforementioned performance fix, this release of NuGet also includes many other bug fixes. There were 181 total issues addressed in the release. For a full list of the work items fixed in NuGet 2.8, please view the [NuGet Issue Tracker for this release](#).

NuGet 2.7.2 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.7.1 Release Notes](#) | [NuGet 2.8 Release Notes](#)

NuGet 2.7.2 was released on November 11, 2013.

Noteworthy Bug Fixes and Features

License Text

For quite some time, Microsoft has included the NuGet packages for several popular open-source libraries as a part of the default templates for Web application projects in Visual Studio. jQuery is probably the most well-known example of this type of library. Because of the support agreement associated with components that are delivered along with a product, the package's script file contains different license text than the script file found in the same package on the public nuget.org gallery. This difference in text can prevent package updates from proceeding as a result of the different license text blocks causing the script files to have different content hash values (and therefore to be treated as modified within the project).

To mitigate this issue, NuGet 2.7.2 allows the script author to include the license text block within a specially marked section which looks as follows.

```
***** NUGET: BEGIN LICENSE TEXT *****
* The following code is licensed under the MIT license
* Additional license information below is informational
* only.
***** NUGET: END LICENSE TEXT *****
```

When updating packages with content files containing this block, NuGet does not factor the contents of the block into the comparison with the version on the NuGet gallery, and can therefore delete and update the content file as though it matches the original copy.

This block is identified by the text "NUGET: BEGIN LICENSE TEXT" and "NUGET: END LICENSE TEXT" occurring anywhere on the beginning and ending lines. No other formatting requirements exist, allowing this feature to be used in any type of text file regardless of language.

Add Binding Redirects for non-Framework Assemblies

For assemblies that are part of the .NET Framework, NuGet skips adding binding redirects into the application's configuration file when updating the package. This fix addresses a regression in NuGet 2.7 whereby binding redirects were not being added for some assemblies, even though those assemblies are not considered a part of the .NET Framework. NuGet 2.7.2 restores the previous NuGet 2.5 and 2.6 behavior and adds the binding redirects.

Installing portable libraries with Xamarin Tools installed

When Xamarin's development tools are installed on a machine, they modify the supported frameworks configuration data to specify compatibility between existing target framework combinations and Xamarin frameworks. With version 2.7.2, NuGet is now aware of these implicit compatibility rules, and therefore makes it easy for developers targeting Xamarin platforms to install portable libraries that are Xamarin-compatible but not explicitly marked as such in the package metadata itself.

Machine-wide configuration settings honored

When using hierarchical Nuget.Config files, the repositoryPath key was not being honored for Nuget.Config files closest to the solution root. In Visual Studio 2013, NuGet installs a custom Nuget.Config file at

%ProgramData%\NuGet\Config\VisualStudio\12.0\Microsoft.VisualStudio.config in order to add the "Microsoft and .NET" package source. As a result, the work-around for using a custom repositoryPath in a solution was to delete the machine-level Nuget.Config - which also meant removing the "Microsoft and .NET" package source. NuGet 2.7.2 now honors the precedence rules for repositoryPath when using hierarchical Nuget.Config files.

All Changes

For a full list of work items fixed in NuGet 2.7.2, please view the [NuGet Issue Tracker for this release](#).

NuGet 2.7.1 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.7 Release Notes](#) | [NuGet 2.7.2 Release Notes](#)

NuGet 2.7.1 was released on October 7, 2013. This is a minor update to our recent 2.7 release with some targeted fixes to improve the experience of new 2.7 features. For a list of work items fixed in NuGet 2.7.1, please view the [NuGet Issue Tracker for this release](#).

The complete set of features in 2.7 can be found in the [release notes here](#).

NuGet 2.7 Release Notes

7/18/2019 • 10 minutes to read • [Edit Online](#)

[NuGet 2.6.1 for WebMatrix Release Notes](#) | [NuGet 2.7.1 Release Notes](#)

NuGet 2.7 was released on August 22, 2013.

Acknowledgements

We would like to thank the following external contributors for their significant contributions to NuGet 2.7:

1. [Mike Roth \(@mrxss\)](#)
 - Show License url when listing packages and verbosity is detailed.
2. [Adam Ralph \(@adamralph\)](#)
 - [#1956](#) - Add developmentDependency attribute to `packages.config` and use it in pack command to only include runtime packages
3. [Rafael Nicoletti \(@tkrafael\)](#)
 - Avoid duplicate Properties key in nuget.exe pack command.
4. [Ben Phegan \(@BenPhegan\)](#)
 - [#2610](#) - Increase machine cache size to 200.
5. [Slava Trenogin \(@derigel\)](#)
 - [#3217](#) - Fix NuGet dialog showing updates in the wrong tab
 - Fix Project.TargetFramework can be null in ProjectManager
 - [#3248](#) - Fix SharedPackageRepository FindPackage/FindPackagesById will fail on non-existent packageId
6. [Kevin Boyle \(@kevfromireland\)](#)
 - [#3234](#) - Enable support for Nomad project
7. [Corin Blaikie \(@corinblaikie\)](#)
 - [#3252](#) - Fix push command fails with exit code 0 when file doesn't exist.
8. [Martin Vesely](#)
 - [#3226](#) - Fix bug with Add-BindingRedirect command when a project references a database project.
9. [Miroslav Bajtos \(@bajtos\)](#)
 - [#2891](#) - Fix bug of nuget.pack parsing wildcard in the 'exclude' attribute incorrectly.
10. [Justin Dearing \(@zippy1981\)](#)
 - [#3307](#) - Fix bug `NuGet.targets` does not pass `$(Platform)` to nuget.exe when restoring packages.
11. [Brian Federici](#)
 - [#3294](#) - Fix bug in nuget.exe package command which would allow adding files with the same name but different casing, eventually causing "Item already exists" exception.
12. [Daniel Cazzulino \(@kzu\)](#)
 - [#2990](#) - Add Version property to NetPortableProfile class.
13. [David Simner](#)
 - [#3460](#) - Fix bug NullReferenceException if requireApiKey = true, but the header X-NUGET-APIKEY isn't present
14. [Michael Friis \(@friism\)](#)
 - [#3278](#) - Fixes NuGet.Build targets file to so that it works correctly on MonoDevelop
15. [Pranav Krishnamoorthy \(@pranav_km\)](#)
 - Improve Restore command performance by increasing parallelization

Notable features in the release

Package Restore by Default (with implicit consent)

NuGet 2.7 introduces a new approach to package restore, and also overcomes a major hurdle: Package restore consent is now on by default! The combination of the new approach and the implicit consent will drastically simplify package restore scenarios.

Implicit Consent

With NuGet versions 2.0, 2.1, 2.2, 2.5, and 2.6, users needed to explicitly allow NuGet to download missing packages during build. If this consent had not been explicitly given, then solutions that had enabled package restore would fail to build until the user had granted consent.

Starting with NuGet 2.7, package restore consent is ON by default while allowing users to explicitly *opt out* of package restore if desired, using the checkbox in NuGet's settings in Visual Studio. This change for implicit consent affects NuGet in the following environments:

- Visual Studio 2013 Preview
- Visual Studio 2012
- Visual Studio 2010
- nuget.exe Command-Line Utility

Automatic Package Restore in Visual Studio

Starting with NuGet 2.7, NuGet will automatically download missing packages during build in Visual Studio, even if package restore hasn't been explicitly enabled for the solution. This Automatic Package Restore happens in Visual Studio when you build a project or the solution, but before MSBuild is invoked. This yields a few significant benefits:

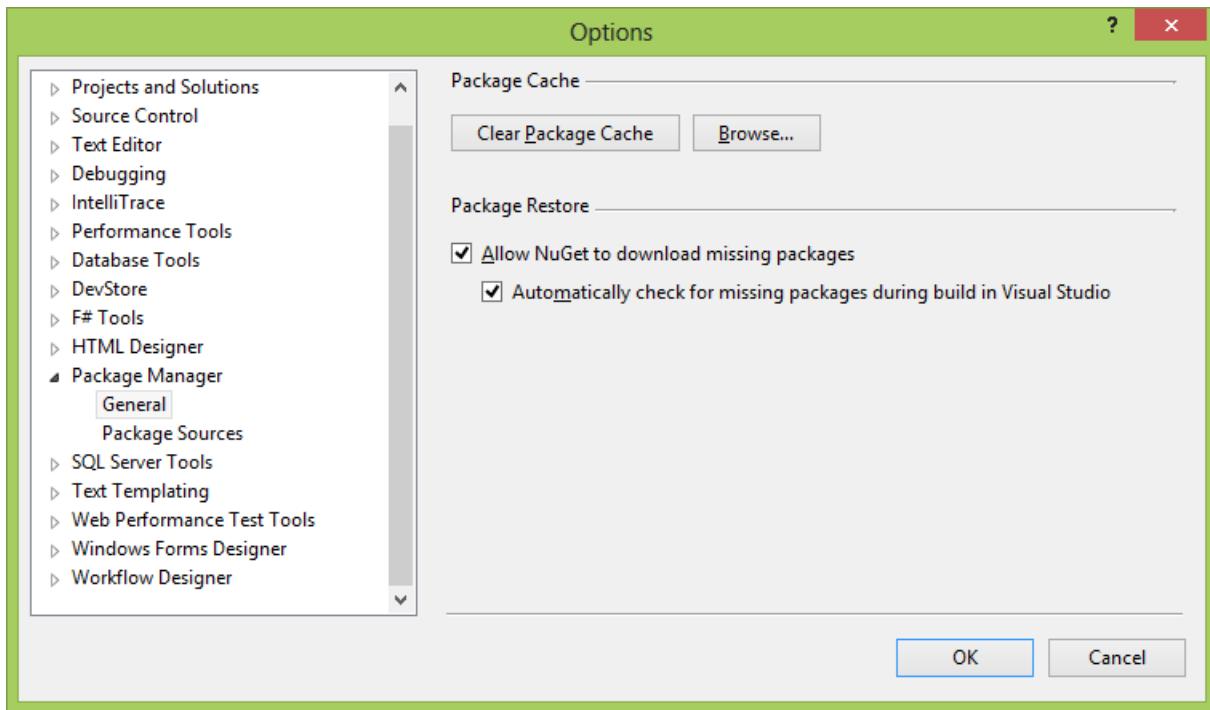
1. No further need to use the "Enable NuGet Package Restore" gesture on your solution
2. Projects don't need to be modified, and NuGet won't make changes to your project to ensure package restore is enabled
3. All NuGet packages, including those that included MSBuild imports for props/targets files, will be restored *before* MSBuild is invoked, ensuring those props/targets are properly recognized during the build

In order to use Automatic Package Restore in Visual Studio, you only need to take one (in)action:

1. Don't check in your `packages` folder

There are several ways to omit your `packages` folder from source control. For more information, see the [Packages and Source Control](#) topic.

While all users are implicitly opted into Automatic Package Restore consent, you can easily opt out through the Package Manager settings in Visual Studio.



Simplified Package Restore from the Command-Line

NuGet 2.7 introduces a new feature for `nuget.exe`: `nuget.exe restore`

This new Restore command allows you to easily restore all packages for a solution with a single command, by accepting a solution file or folder as an argument. Furthermore, that argument is implied when there's only a single solution in the current folder. That means the following all work from a folder that contains a single solution file (`MySolution.sln`):

1. `nuget.exe restore MySolution.sln`
2. `nuget.exe restore .`
3. `nuget.exe restore`

The Restore command will open the solution file and find all projects within the solution. From there, it will find the `packages.config` files for each of the projects and restore all of the packages found. It also restores solution-level packages found in the `.nuget\packages.config` file. More information about the new Restore command can be found in the [Command-Line Reference](#).

The New Package Restore Workflow

We are excited about these changes to Package Restore, as it introduces a new workflow. If you want to omit your packages from source control you simply don't commit the `packages` folder. Visual Studio users who open and build the solution will see the packages automatically restored. For command-line builds, simply invoke `nuget.exe restore` before invoking `msbuild`. You no longer need to remember to use the "Enable NuGet Package Restore" gesture on your solution, and we'll no longer need to modify your projects to alter the build. And this also yields a much improved experience for packages that include MSBuild imports, especially for imports added through NuGet's recent feature for [automatically importing props/targets files](#) from the `\build` folder.

In addition to the work we've done ourselves, we're also working with some important partners to round this new approach out. We don't have concrete timelines for any of these yet, but each partner is as excited as we are about the new approach.

- Team Foundation Service - They are working to integrate the call to `nuget.exe restore` into the default build scenarios.
- Windows Azure Web Sites - They are working to allow you to push your project to Azure and have `nuget.exe restore` called before your web site is built.
- TeamCity - They are updating their NuGet Installer plugin for TeamCity 8.x

- AppHarbor - They are working to allow you to push your repo to AppHarbor and have `nuget.exe restore` called before your solution is build.

With each of the partners above, they would use their own copy of nuget.exe and you would not need to carry nuget.exe in your solution.

Known Issues

There were two known issues with nuget.exe restore with the initial 2.7 release, but they were fixed on 9/6/2013 with an update to the [NuGet.CommandLine package](#). This update is also available on the [NuGet 2.7 download page](#) on CodePlex. Running `nuget.exe update -self` will update to the latest release.

The fixed were:

1. [New package restore doesn't work on Mono when using SLN file](#)
2. [New package restore doesn't work with Wix projects](#)

There is also a known issue with the new package restore workflow whereby [Automatic Package Restore does not work for projects under a solution folder](#). This issue was fixed in NuGet 2.7.1.

Project Retargeting and Upgrade Build Errors/Warnings

Many times after retargeting or upgrading your project, you find that some NuGet packages aren't functioning properly. Unfortunately, there is no indication of this and then there's no guidance on how to address it. With NuGet 2.7, we now use some Visual Studio events to recognize when you've retargeted or upgraded your project in a way that affects your installed NuGet packages.

If we detect that any of your packages were affected by the retargeting or upgrade, we'll produce immediate build errors to let you know. In addition to the immediate build error, we also persist a `requireReinstallation="true"` flag in your `packages.config` file for all packages that were affected by the retargeting, and each subsequent build in Visual Studio will raise build warnings for those packages.

Although NuGet cannot take automatic action to reinstall affected packages, we hope this indication and warning will guide help you discover when you need to reinstall packages. We are also working on [package reinstallation guidance documentation](#) that these error messages direct you to.

NuGet Configuration Defaults

Many companies are using NuGet internally, but have had a hard time guiding their developers to use internal package sources instead of nuget.org. NuGet 2.7 introduces a Configuration Defaults feature that allows machine-wide defaults to be specified for:

1. Enabled package sources
2. Registered, but disabled package sources
3. The default nuget.exe push source

Each of these can now be configured within a file located at `%ProgramData%\NuGet\NuGetDefaults.Config`. If this config file specifies package sources, then the default nuget.org package source will not be registered automatically, and the ones in `NuGetDefaults.Config` will be registered instead.

While not required to use this feature, we expect companies to deploy `NuGetDefaults.Config` files using Group Policy.

Note that this feature will never cause a package source to be removed from a developer's NuGet settings. That means if the developer has already used NuGet and therefore has the nuget.org package source registered, it won't be removed after the creation of a `NuGetDefaults.Config` file.

See [NuGet Configuration Defaults](#) for more information about this feature.

Renaming the Default Package Source

NuGet has always registered a default package source called "NuGet official package source" that points to nuget.org. That name was verbose and it also didn't specify where it was actually pointing. To address those two issues, we've renamed this package source to simply "nuget.org" in the UI. The URL for the package source was also changed to include the "www." prefix. After using NuGet 2.7, your existing "NuGet official package source" will automatically be updated to "nuget.org" as its name and "<https://www.nuget.org/api/v2/>" as its URL.

Performance Improvements

We made some performance improvement in 2.7 which will yield smaller memory footprint, less disk usage and faster package installation. We also made smarter queries to OData-based feeds which will reduce the overall payload.

New Extensibility APIs

We added some new APIs to our extensibility services to fill the gap of missing functionalities in previous releases.

IVsPackageInstallerServices

```
```cs
// Checks if a NuGet package with the specified Id and version is installed in the specified project.
bool IsPackageInstalledEx(Project project, string id, string versionString);

// Get the list of NuGet packages installed in the specified project.
IEnumerable<IVsPackageMetadata> GetInstalledPackages(Project project);
```
```

IVsPackageInstaller

```
```cs
// Installs one or more packages that exist on disk in a folder defined in the registry.
void InstallPackagesFromRegistryRepository(string keyName, bool isPreUnzipped, bool skipAssemblyReferences,
Project project, IDictionary<string, string> packageVersions);

// Installs one or more packages that are embedded in a Visual Studio Extension Package.
void InstallPackagesFromVSExtensionRepository(string extensionId, bool isPreUnzipped, bool
skipAssemblyReferences, Project project, IDictionary<string, string> packageVersions);
```
```

Development-Only Dependencies

This feature was contributed by [Adam Ralph](#) and it allows package authors to declare dependencies that were only used at development time and don't require package dependencies. By adding a `developmentDependency="true"` attribute to a package in `packages.config`, `nuget.exe pack` will no longer include that package as a dependency.

Removed Support for Visual Studio 2010 Express for Windows Phone

The new package restore model in 2.7 is implemented by a new VS Package which is different from the main NuGet VS Package. Due to a technical issue, this new VS Package doesn't work correctly in the *Visual Studio 2010 Express for Windows Phone* SKU as we share the same code base with other supported Visual Studio SKUs. Therefore, starting with NuGet 2.7, we are dropping support for *Visual Studio 2010 Express for Windows Phone* from the published extension. Support for *Visual Studio 2010 Express for Web* is still included in the primary extension published to the Visual Studio Extension Gallery.

Since we are unsure how many developers are still using NuGet in that version/edition of Visual Studio, we are publishing a separate Visual Studio extension specifically for those users and publishing it on CodePlex (rather than the Visual Studio Extension Gallery). We don't plan to continue to maintain that extension, but if this affects you please let us know by filing an issue on CodePlex.

To download the NuGet Package Manager (for Visual Studio 2010 Express for Windows Phone), visit the [NuGet 2.7 Downloads](#) page.

Bug Fixes

In addition to these features, this release of NuGet also includes many other bug fixes. There were 97 total issues addressed in the release. For a full list of work items fixed in NuGet 2.7, please view the [NuGet Issue Tracker for this release](#).

NuGet 2.6.1 for WebMatrix Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.6 Release Notes](#) | [NuGet 2.7 Release Notes](#)

The NuGet team released an updated NuGet Package Manager extension for WebMatrix on March 26, 2014. This update can be installed from the [WebMatrix Extension Gallery](#) using the following steps:

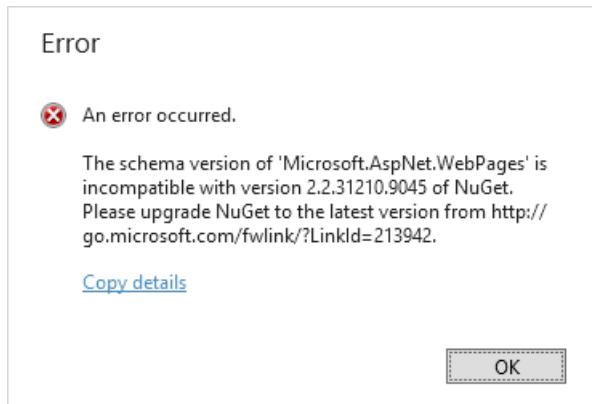
1. Open WebMatrix 3
2. Click the Extensions icon in the Home ribbon
3. Select the Updates tab
4. Click to update NuGet Package Manager to 2.6.1
5. Close and restart WebMatrix 3

Notable Changes

This extension update addresses two of the biggest issues users have faced consuming NuGet packages within WebMatrix. The first was a NuGet schema version error and the second was a bug leading to zero-byte DLLs in the `bin` folder.

NuGet Schema Version Error

Since WebMatrix 3 was released, new features have been introduced into NuGet that require a new schema version for the NuGet packages. When trying to manage your NuGet packages in your web site, these new packages can lead to errors that you see in WebMatrix.



This latest release provides compatibility with the newest NuGet packages, preventing this error from occurring. New versions of packages including `Microsoft.AspNet.WebPages` can now be installed in WebMatrix. Some of these packages were using NuGet features such as [XDT config transforms](#), which wasn't supported in WebMatrix until now.

Zero-Byte DLLs in `bin` Folder

Some users have reported that after installing NuGet packages in WebMatrix that include DLLs that get copied to `bin`, that the DLLs show up in the `bin` folder as 0-byte files. This breaks the application at runtime.

[This issue](#) has now been fixed.

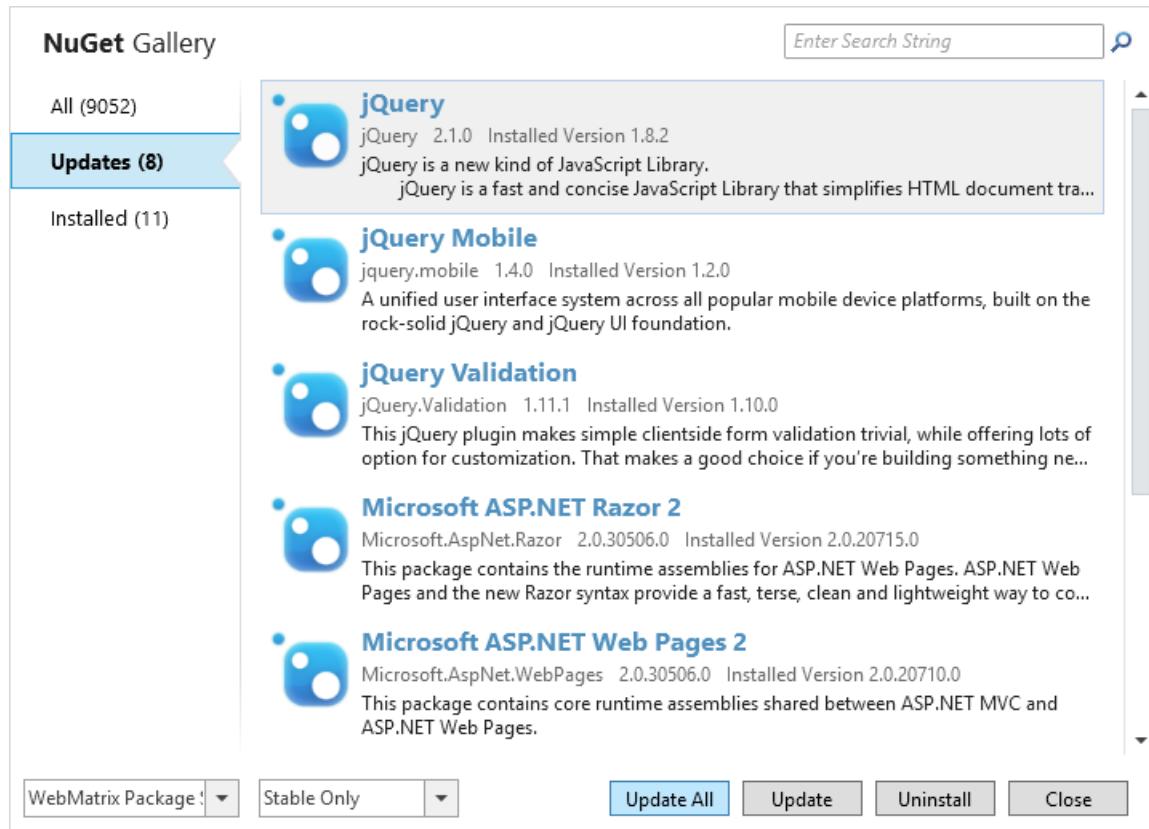
Other Recent Improvements

When NuGet Package Manager 2.8 was released for Visual Studio, we also released NuGet Package Manager 2.5.0 for WebMatrix. While this was mentioned in the [NuGet 2.8 Release Notes](#), we didn't mention the specific new

features that update introduced.

Update All

You can now update all of your web site's packages in one step! When you open the NuGet extension in WebMatrix, you see the list of all packages on the gallery, those installed, and the ones with updates available. Previously, every package would have to be updated individually but now there is a useful "Update All" button that shows up on the Updates tab.



NuGet Gallery

Enter Search String

All (9052)

Updates (8)

Installed (11)

jQuery
jQuery 2.1.0 Installed Version 1.8.2
jQuery is a new kind of JavaScript Library.
jQuery is a fast and concise JavaScript Library that simplifies HTML document tra...

jQuery Mobile
jquery.mobile 1.4.0 Installed Version 1.2.0
A unified user interface system across all popular mobile device platforms, built on the rock-solid jQuery and jQuery UI foundation.

jQuery Validation
jQuery.Validation 1.11.1 Installed Version 1.10.0
This jQuery plugin makes simple clientside form validation trivial, while offering lots of option for customization. That makes a good choice if you're building something ne...

Microsoft ASP.NET Razor 2
Microsoft.AspNet.Razor 2.0.30506.0 Installed Version 2.0.20715.0
This package contains the runtime assemblies for ASP.NET Web Pages. ASP.NET Web Pages and the new Razor syntax provide a fast, terse, clean and lightweight way to co...

Microsoft ASP.NET Web Pages 2
Microsoft.AspNet.WebPages 2.0.30506.0 Installed Version 2.0.20710.0
This package contains core runtime assemblies shared between ASP.NET MVC and ASP.NET Web Pages.

WebMatrix Package ▾

Stable Only ▾

Update All

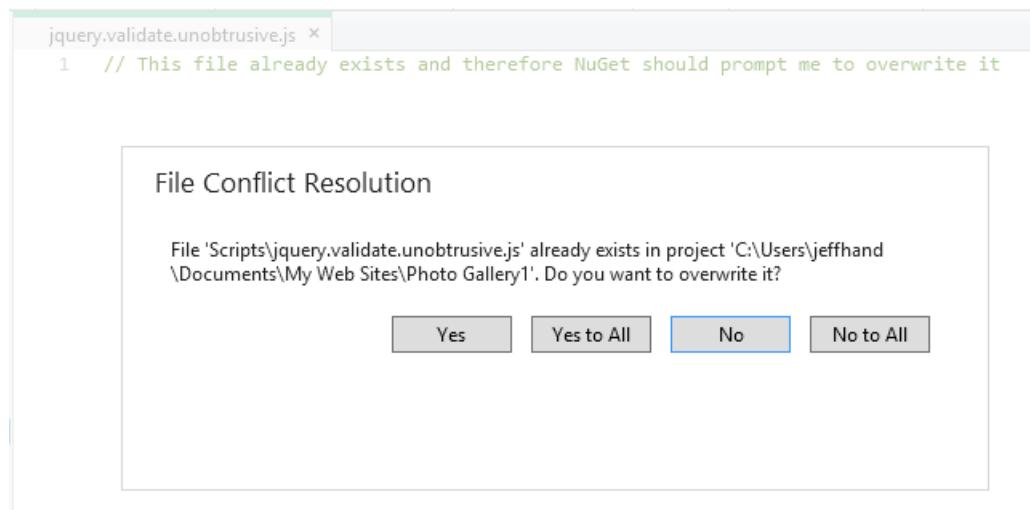
Update

Uninstall

Close

Overwrite Existing Files

When installing packages that contain files that already exist in your web site, NuGet has always just silently ignored those files (leaving your existing files alone). This could lead to the impression that a package was installed or updated correctly when in fact it wasn't. NuGet will now prompt for files to be overwritten.



jquery.validate.unobtrusive.js x

1 // This file already exists and therefore NuGet should prompt me to overwrite it

File Conflict Resolution

File 'Scripts\jquery.validate.unobtrusive.js' already exists in project 'C:\Users\jeffhand\Documents\My Web Sites\Photo Gallery1'. Do you want to overwrite it?

Yes

Yes to All

No

No to All

NuGet 2.6 Release Notes

9/4/2018 • 5 minutes to read • [Edit Online](#)

[NuGet 2.5 Release Notes](#) | [NuGet 2.6.1 for WebMatrix Release Notes](#)

NuGet 2.6 was released on June 26, 2013.

Notable features in the release

Support for Visual Studio 2013

NuGet 2.6 is the first release that provides support for Visual Studio 2013. And like Visual Studio 2012, the NuGet Package Manager extension is included in every edition of Visual Studio.

In order to provide the best possible support for Visual Studio 2013 while still supporting both Visual Studio 2010 and Visual Studio 2012, and keeping the extension sizes as small as possible, we are producing a separate extension for Visual Studio 2013 while the original extension continues to target both Visual Studio 2010 and 2012.

Starting with NuGet 2.6, we will publish two extensions as below:

1. [NuGet Package Manager](#) (applies to Visual Studio 2010 and 2012)
2. [NuGet Package Manager for Visual Studio 2013](#)

With this split, the [nuget.org](#) home page's "Install NuGet" button takes you to the [installing NuGet](#) page, where you can find more information about installing the different NuGet clients.

XDT Web.config transformation support

One of the most highly-requested features for the NuGet client has been to support more powerful XML transformations using the XDT transformation engine which is used in Visual Studio build configuration transformations.

In April 2013, we made two big announcements regarding NuGet support for XDT. The first was that the XDT library itself was being itself [released as a NuGet package](#) and [open sourced on CodePlex](#). This step enabled the XDT engine to be used freely by other open-source software, including the NuGet client. The second announcement was the plan to support use of the XDT engine for transformations in the NuGet client. NuGet 2.6 includes this integration.

How it works

To take advantage of NuGet's XDT support, the mechanics look similar to those of the [current config transformation feature](#). Transformation files are added to the package's content folder. However, while config transformations use a single file for both installation and uninstallation, XDT transformations enable fine-grained control over both of these processes using the following files:

- `Web.config.install.xdt`
- `Web.config.uninstall.xdt`

Additionally, NuGet uses the file suffix to determine which engine to run for transformations, so packages using the existing `web.config.transforms` will continue to work. XDT transformations can also be applied to any XML file (not just `web.config`), so you can leverage this for other applications in your project.

What you can do with XDT

One of XDT's greatest strengths is its [simple but powerful syntax](#) for manipulating the structure of an XML DOM. Rather than simply overlaying one fixed document structure onto another structure, XDT provides controls for matching elements in a variety of ways, from simple attribute name matching to full XPath support. Once a

matching element or set of elements is found, XDT provides a rich set of functions for manipulating the elements, whether that means adding, updating, or removing attributes, placing a new element at a specific location, or replacing or removing the entire element and its children.

Machine-Wide Configuration

One of the great strengths of NuGet is that it breaks down an otherwise large executable or library into a set of modular components which can be integrated, and most importantly maintained and versioned independently. One side effect of this, however, is that the conventional idea of a product or product family becomes potentially more fragmented. NuGet's custom package source feature provides one way of organizing packages; however, custom package sources are not discoverable on their own.

NuGet 2.6 extends the logic for configuring NuGet by searching the folder hierarchy under the path `%ProgramData%\NuGet\Config`. Product installers can add custom NuGet configuration files under this folder to register a custom package source for their products. Additionally, the folder structure supports semantics for product, version, and even SKU of the IDE. Settings from these directories are applied in the following order with a "last in wins" precedence strategy.

1. `%ProgramData%\NuGet\Config*.config`
2. `%ProgramData%\NuGet\Config{IDE}*.config`
3. `%ProgramData%\NuGet\Config{IDE}{Version}*.config`
4. `%ProgramData%\NuGet\Config{IDE}{Version}{SKU}*.config`

In this list, the `{IDE}` placeholder is specific to the IDE in which NuGet is running, so in the case of Visual Studio, it will be "VisualStudio". The `{Version}` and `{SKU}` placeholders are provided by the IDE (e.g. "11.0" and "WDEExpress", "VWDEExpress" and "Pro", respectively). The folder can then contain many different *.config files. Therefore, the ACME component company can, as a part of their product installer, add a custom package source which will be visible only in the Professional and Ultimate versions of Visual Studio 2012 by creating the following file path:

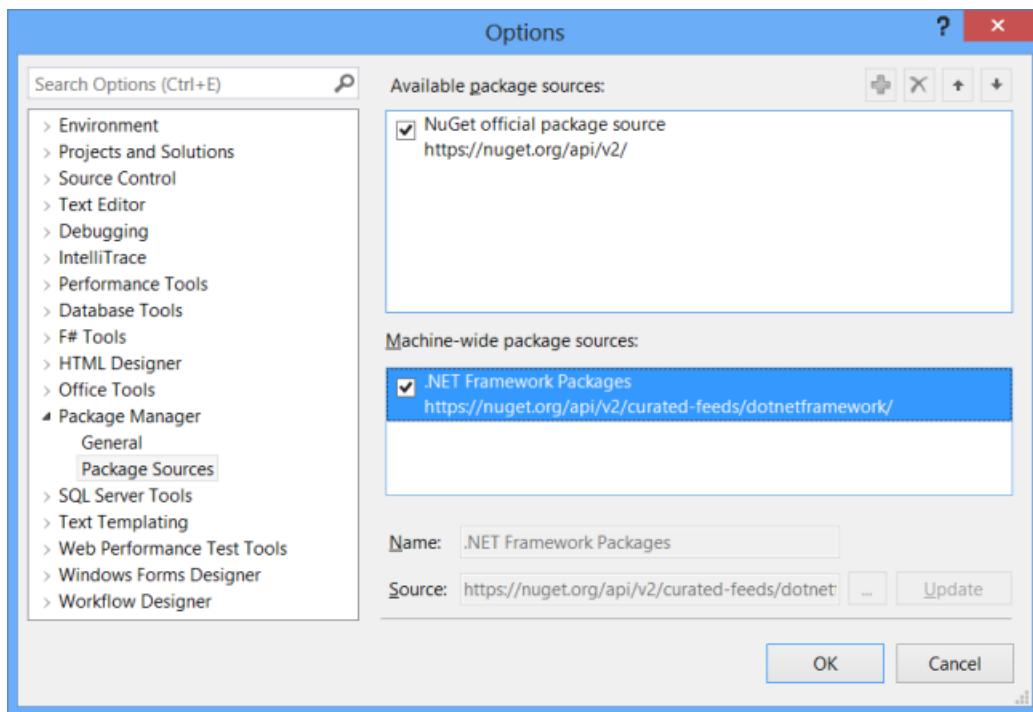
```
%ProgramData%\NuGet\Config\VisualStudio\11.0\Pro\acme.config
```

While the folder structure makes it straightforward for programs like software installers to add machine-wide package sources to NuGet's configuration, the NuGet configuration dialog has also been updated to allow for the registration of package sources as either user-specific (e.g. registered in `%AppData%\NuGet\NuGet.Config`) or machine-wide.

This feature is utilized by Visual Studio 2013, where a file is installed at:

```
%ProgramData%\NuGet\Config\VisualStudio\12.0\Microsoft.VisualStudio.config
```

Within this file, a new package source called ".NET Framework Packages" is configured.



Contextualizing Search

As the number of packages served by the NuGet gallery continues to grow at an exponential pace, improving search remains ever at the top of the NuGet priority list. One of the planned features for NuGet is contextual search, meaning that NuGet will use information about the version and SKU of Visual Studio that you are using and the type of project that you are building as criteria for determining the relevance of potential search results.

Starting with NuGet 2.6, each time a package is installed, the context for the installation is recorded as part of the installation operation data. Searches also send the same context information, which will allow the NuGet Gallery to boost search results by contextual installation trends. A future update to the NuGet Gallery will enable this context-sensitive relevance boosting.

Tracking Direct Installs vs. Dependency Installs

Package authors are relying more and more on the [Package Statistics](#) provided on the NuGet Gallery. One significant missing data point that authors have asked for is a differentiation between direct package installs and dependency installs. Until now, the NuGet client did not send any context around the installation operation for whether the developer directly installed the package or if it was installed to satisfy a dependency. Starting with NuGet 2.6, that data will now be sent for the installation operation. Package Statistics on the NuGet Gallery will expose that data as separate install operations, with a "-Dependency" suffix.

- Install
- Install-Dependency
- Update
- Update-Dependency
- Reinstall
- Reinstall-Dependency

In addition to the different operation name, the dependent package id is also recorded for the installation. A future update to the NuGet Gallery will expose that data within reports, allowing package authors to fully understand how developers are installing their packages.

Bug Fixes

NuGet 2.6 also includes several bug fixes. For a full list of work items fixed in NuGet 2.6, please view the [NuGet Issue Tracker for this release](#).

NuGet 2.5 Release Notes

9/4/2018 • 7 minutes to read • [Edit Online](#)

[NuGet 2.2.1 Release Notes](#) | [NuGet 2.6 Release Notes](#)

NuGet 2.5 was released on April 25, 2013. This release was so big, we felt compelled to skip versions 2.3 and 2.4! To date, this is the largest release we've had for NuGet, with over [160 work items](#) in the release.

Acknowledgements

We would like to thank the following external contributors for their significant contributions to NuGet 2.5:

1. [Daniel Plaisted \(@dsplaisted\)](#)
 - [#2847](#) - Add MonoAndroid, MonoTouch, and MonoMac to the list of known target framework identifiers.
2. [Andres G. Aragoneses \(@knocte\)](#)
 - [#2865](#) - Fix spelling of `NuGet.targets` for a case-sensitive OS
3. [David Fowler \(@davidfowl\)](#)
 - Make the solution build on Mono.
4. [Andrew Theken \(@atheken\)](#)
 - Fix unit tests failing on Mono.
5. [Olivier Dagenais \(@OlilIsCool\)](#)
 - [#2920](#) - nuget.exe pack command does not propagate Properties to MSBuild
6. [Miroslav Bajtos \(@bajtos\)](#)
 - [#1511](#) - Modified XML handling code to preserve formatting.
7. [Adam Ralph \(@adamralph\)](#)
 - Added recognized words to custom dictionary to allow build.cmd to succeed.
8. [Bruno Roggeri](#)
 - Fix unit tests when running in localized VS.
9. [Gareth Evans](#)
 - Extracted interface from PackageService
10. [Maxime Brugidou \(@brugidou\)](#)
 - [#936](#) - Handle project dependencies when packing
11. [Xavier Decoster \(@XavierDecoster\)](#)
 - [#2991, #3164](#) - Support Clear Text Password when storing package source credentials in nuget.config files
12. [James Manning \(@manningj\)](#)
 - [#3190, #3191](#) - Fix Get-Package help description

We also appreciate the following individuals for finding bugs with NuGet 2.5 Beta/RC that were approved and fixed before the final release:

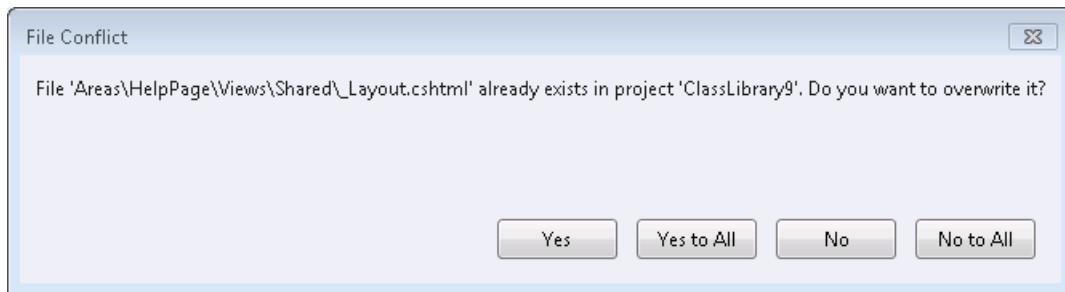
1. [Tony Wall \(@CodeChief\)](#)
 - [#3200](#) - MSTest broken with lastest NuGet 2.4 and 2.5 builds

Notable features in the release

Allow users to overwrite content files that already exist

One of the most requested features of all time has been the ability to overwrite content files that already exist on

disk when included in a NuGet package. Starting with NuGet 2.5, these conflicts are identified and you are prompted to overwrite the files, whereas previously these files were always skipped.



'nuget.exe update' and 'Install-Package' now both have a new option '-FileConflictAction' to set some default for command-line scenarios.

Set a default action when a file from a package already exists in the target project. Set to 'Overwrite' to always overwrite files. Set to 'Ignore' to skip files. If not specified, it will prompt for each conflicting file.

Automatic import of MSBuild targets and props files

A new conventional folder has been created at the top level of the NuGet package. 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 `<Import>` 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.

Specify different references per platform using `<References/>` element

Before 2.5, in `.nuspec` file, user can only specify the reference files, to be added for all framework. Now with this new feature in 2.5, user can author the `<reference/>` element for each of the supported platform, for example:

```
<references>
  <group targetFramework="net45">
    <reference file="a.dll" />
  </group>
  <group targetFramework="netcore45">
    <reference file="b.dll" />
  </group>
  <group>
    <reference file="c.dll" />
  </group>
</references>
```

Here is the flow for how NuGet adds references to projects based on the `.nuspec` file:

1. Find the `lib` folder that is appropriate for the target framework and get the list of assemblies from that folder
2. Separately find the references group that is appropriate for the target framework and get the list of assemblies from that group. Reference group without target framework specified is the fallback group.
3. Find the intersection of the two lists, and use that as the references to add

This new feature will allow package authors to use the References feature to apply subsets of assemblies to different frameworks when they would otherwise need to carry duplicate assemblies in multiple `lib` folders.

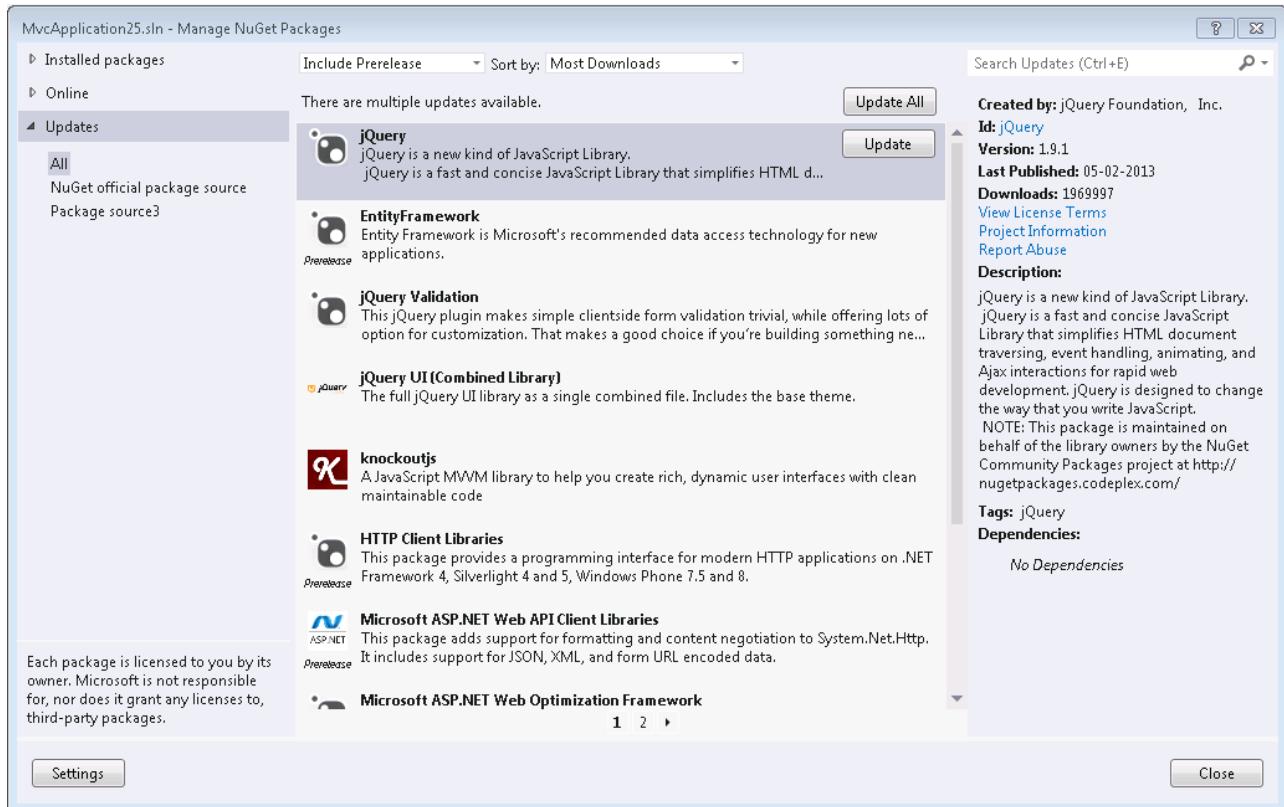
Note: you must presently use `nuget.exe pack` to use this feature; NuGet Package Explorer does not yet support it.

Update All button to allow updating all packages at once

Many of you know about the "Update-Package" PowerShell cmdlet to update all of your packages; now there's an easy way to do this through the UI as well.

To try this feature out:

1. Create a new ASP.NET MVC application
2. Launch the 'Manage NuGet Packages' dialog
3. Select 'Updates'
4. Click the 'Update All' button



Improved project reference support for nuget.exe Pack

Now nuget.exe pack command processes referenced projects with the following rules:

1. If the referenced project has corresponding `.nuspec` file, e.g. there is a file called `proj1.nuspec` in the same folder as `proj1.csproj`, then this project is added as a dependency to the package, using the id and version read from the `.nuspec` file.
2. Otherwise, the files of the referenced project are bundled into the package. Then projects referenced by this project will be processed using the same rules recursively.
3. All DLL, `.pdb`, and `.exe` files are added.
4. All other content files are added.
5. All dependencies are merged.

This allows a referenced project to be treated as a dependency if there is a `.nuspec` file, otherwise, it becomes part of the package.

More details here: <http://nuget.codeplex.com/workitem/936>

Add a 'Minimum NuGet Version' property to packages

A new metadata attribute called 'minClientVersion' can now indicate the minimum NuGet client version required to consume a package.

This feature helps package author to specify that a package will work only after a particular version of NuGet. As new `.nuspec` features are added after NuGet 2.5, packages will be able to claim a minimum NuGet version.

```
<metadata minClientVersion="2.6">
```

If the user has NuGet 2.5 installed and a package is identified as requiring 2.6, visual cues will be given to the user indicating the package will not be installable. The user will then be guided to update their version of NuGet.

This will improve upon the existing experience where packages begin to install but then fail indicating an unrecognized schema version was identified.

Dependencies are no longer unnecessarily updated during package installation

Before NuGet 2.5, when a package was installed that depended on a package already installed in the project, the dependency would be updated as part of the new installation, even if the existing version satisfied the dependency.

Starting with NuGet 2.5, if a dependency version is already satisfied, the dependency will not be updated during other package installations.

The scenario:

1. The source repository contains package B with version 1.0.0 and 1.0.2. It also contains package A which has a dependency on B ($\geq 1.0.0$).
2. Assume that the current project already has package B version 1.0.0 installed. Now you want to install package A.

In NuGet 2.2 and older:

- When installing package A, NuGet will auto-update B to 1.0.2, even though the existing version 1.0.0 already satisfies the dependency version constraint, which is $\geq 1.0.0$.

In NuGet 2.5 and newer:

- NuGet will no longer update B, because it detects that the existing version 1.0.0 satisfies the dependency version constraint.

For more background on this change, read the detailed [work item](#) as well as the related [discussion thread](#).

nuget.exe outputs http requests with detailed verbosity

If you are troubleshooting nuget.exe or just curious what HTTP requests are made during operations, the '-verbosity detailed' switch will now output all HTTP requests made.

```
C:\>nuget install jquery -verbosity detailed
GET https://nuget.org/api/v2/FindPackagesById()?id='jquery'
GET https://nuget.org/api/v2/package/jquery/1.9.1
Installing 'jQuery 1.9.1'.
Successfully installed 'jQuery 1.9.1'.
```

nuget.exe push now supports UNC and folder sources

Before NuGet 2.5, if you attempted to run 'nuget.exe push' to a package source based on a UNC path or local folder, the push would fail. With the recently added hierarchical configuration feature, it had become common for nuget.exe to need to target either a UNC/folder source, or an HTTP-based NuGet Gallery.

Starting with NuGet 2.5, if nuget.exe identifies a UNC/folder source, it will perform the file copy to the source.

The following command will now work:

```
nuget push -source \\mycompany\repo\ mypackage.1.0.0.nupkg
```

nuget.exe supports explicitly-specified Config files

nuget.exe commands that access configuration (all except 'spec' and 'pack') now support a new '-ConfigFile' option,

which forces a specific config file to be used in place of the default config file at %AppData%\nuget\Nuget.Config.

Example:

```
nuget sources add -name test -source http://test -ConfigFile C:\test\.nuget\Nuget.Config
```

Support for Native projects

With NuGet 2.5, the NuGet tooling is now available for Native projects in Visual Studio. We expect most native packages will utilize the MSBuild imports feature above, using a tool created by the [CoApp project](#). For more information, read [the details about the tool](#) on the coapp.org website.

The target framework name of "native" is introduced for packages to include files in \build, \content, and \tools when the package is installed into a native project. The 'lib' folder is not used for native projects.

NuGet 2.2.1 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.2 Release Notes](#) | [NuGet 2.5 Release Notes](#)

NuGet 2.2.1 was released on February 15, 2013. The VS Extension version number is 2.2.40116.9051.

Localization Refresh

When NuGet shipped as part of Visual Studio 2012, it was fully localized into English + 13 other languages. Since then, NuGet 2.1 and 2.2 have shipped but the localization had not been refreshed. The NuGet 2.2.1 release refreshes our localization.

NuGet's UI and PowerShell Console are localized into the following languages:

1. Chinese (Simplified)
2. Chinese (Traditional)
3. Czech
4. English
5. French
6. German
7. Italian
8. Japanese
9. Korean
10. Polish
11. Portuguese (Brazil)
12. Russian
13. Spanish
14. Turkish

Visual Studio Templates Support Multiple Preinstalled Package Repositories

If you produce Visual Studio templates, you can use NuGet to [preinstall packages](#) as part of the template. Until now, this feature had a limitation that all of the packages needed to come from the same source. With NuGet 2.2.1 though, you can have packages installed from multiple repositories (within the template, a VSIX, or a folder on disk defined in the registry).

The main scenario for this feature is custom ASP.NET project templates. The built-in ASP.NET templates use preinstalled packages, pulling packages from local disk. You can now create a custom ASP.NET project template that uses the existing packages installed by ASP.NET but add extra NuGet packages into your template.

Bug Fixes

NuGet 2.2.1 includes a few targeted bug fixes. For a list of work items fixed in NuGet 2.2.1, please view the [NuGet Issue Tracker for this release](#).

Known Issues

If you are extending ASP.NET project templates, all preinstalled package repositories must use the same value for the `isPreunzipped` attribute.

NuGet 2.2 Release Notes

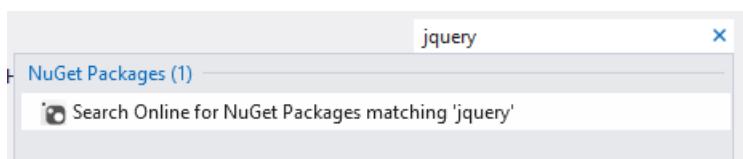
9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 2.1 Release Notes](#) | [NuGet 2.2.1 Release Notes](#)

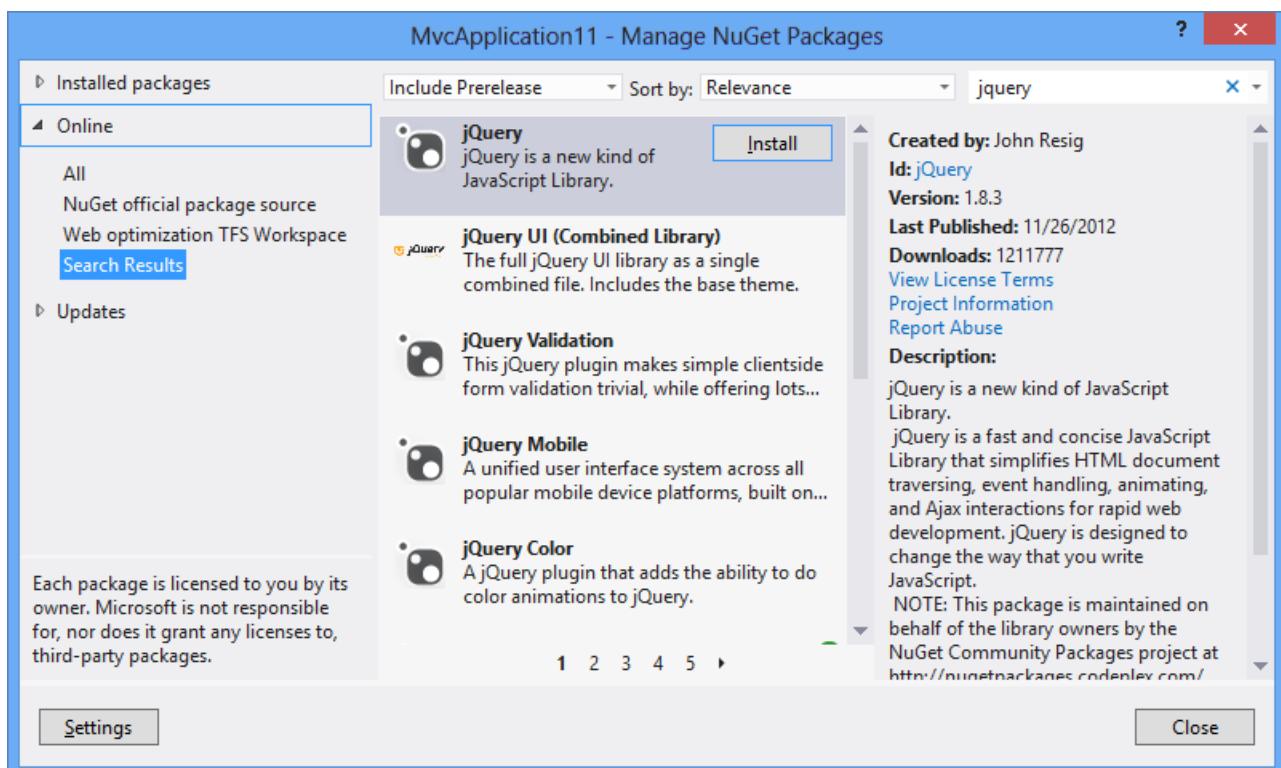
NuGet 2.2 was released on December 12, 2012.

Visual Studio Quick Launch

One of the new features that was added in Visual Studio 2012 was the [quick launch dialog](#). NuGet 2.2 extends this dialog, allowing it to initialize the package manager dialog with the search terms entered in the quick launch. For example, entering 'jquery' in quick launch now includes an option in the results to search NuGet packages matching 'jquery'.



Selecting this option will launch the standard NuGet package manager search experience for the term 'jquery'.



Specify Entire Folder for Package Contents

NuGet 2.2 now allows you to specify an entire folder in the `<file>` element of the `.nuspec` file to include all of the contents of that folder. For example, the following will cause all scripts in the package's scripts folder to be added to the contents\scripts folder when the package is installed into a project.

```
<file src="scripts\" target="content\scripts"/>
```

Update 6/24/16: Empty folders in the "content" folder are ignored when installing the package.

Known Issues

Package installation fails for F# projects when using the package manager console

When attempting to install a NuGet package into an F# project using the package manager console, an `InvalidOperationException` is thrown. We are actively working with the F# team to resolve the issue, but in the meantime, the workaround is to install NuGet packages into F# projects via NuGet's package manager dialog rather than the console. [More information is available on CodePlex](#).

Bug Fixes

NuGet 2.2 includes many bug fixes. For a full list of work items fixed in NuGet 2.2, please view the [NuGet Issue Tracker for this release](#).

NuGet 2.1 Release Notes

9/4/2018 • 6 minutes to read • [Edit Online](#)

[NuGet 2.0 Release Notes](#) | [NuGet 2.2 Release Notes](#)

NuGet 2.1 was released on October 4, 2012.

Hierarchical Nuget.Config

NuGet 2.1 gives you greater flexibility in controlling NuGet settings by way of recursively walking up the folder structure looking for `NuGet.Config` files and then building the configuration from the set of all found files. As an example, consider the scenario where a team has an internal package repository for CI builds of other internal dependencies. The folder structure for an individual project might look like the following:

```
C:\  
C:\myteam\  
C:\myteam\solution1  
C:\myteam\solution1\project1
```

Additionally, if package restore is enabled for the solution, the following folder will also exist:

```
C:\myteam\solution1\.nuget
```

In order to have the team's internal package repository available for all projects that the team works on, while not making it available for every project on the machine, we can create a new `Nuget.Config` file and place it in the `c:\myteam` folder. There is no way to specify a packages folder per project.

```
<configuration>  
  <packageSources>  
    <add key="Official project team source" value="http://teamserver/api/v2/" />  
  </packageSources>  
  <disabledPackageSources />  
  <activePackageSource>  
    <add key="Official project team source" value="http://teamserver/api/v2/" />  
  </activePackageSource>  
</configuration>
```

We can now see that the source was added by running the 'nuget.exe sources' command from any folder beneath `c:\myteam` as shown below:

```
1. NuGet official package source [Enabled]  
   https://nuget.org/api/v2/  
2. Official project team source [Enabled]  
   http://teamserver/api/v2/
```

`NuGet.Config` files are searched for in the following order:

1. `.nuget\Nuget.Config`
2. Recursive walk from project folder to root
3. Global `Nuget.Config` (`%appdata%\NuGet\Nuget.Config`)

The configurations are then applied in the *reverse order*, meaning that based on the above ordering, the global `Nuget.Config` would be applied first, followed by the discovered `Nuget.Config` files from root to project folder,

followed by `.nuget\Nuget.Config`. This is particularly important if you're using the `<clear/>` element to remove a set of items from config.

Specify 'packages' Folder Location

In the past, NuGet has managed a solution's packages from a known 'packages' folder found beneath the solution root folder. For development teams that have many different solutions which have NuGet packages installed, this can result in the same package being installed in many different places on the file system.

NuGet 2.1 provides more granular control over the location of the packages folder via the `repositoryPath` element in the `NuGet.Config` file. Building on the previous example of hierarchical Nuget.Config support, assume that we wish to have all projects under `C:\myteam\` share the same packages folder. To accomplish this, simply add the following entry to `c:\myteam\Nuget.Config`.

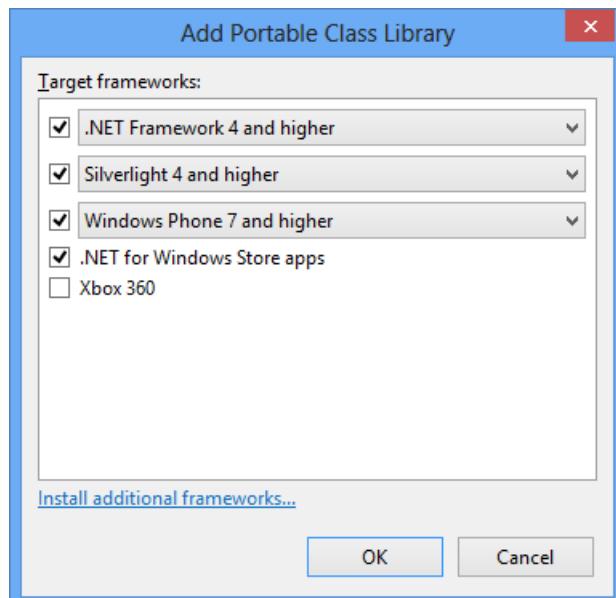
```
<configuration>
  <config>
    <add key="repositoryPath" value="C:\myteam\teampackages" />
  </config>
  ...
</configuration>
```

In this example, the shared `Nuget.Config` file specifies a shared packages folder for every project that is created beneath `C:\myteam`, regardless of depth. Note that if you have an existing packages folder underneath your solution root, you need to delete it before NuGet will place packages in the new location.

Support for Portable Libraries

[Portable libraries](#) is a feature first introduced with .NET 4 that enables you to build assemblies that can work without modification on different Microsoft platforms, from versions of the .NET Framework to Silverlight to Windows Phone and even Xbox 360 (though at this time, NuGet does not support the Xbox portable library target). By extending the [package conventions](#) for framework versions and profiles, NuGet 2.1 now supports portable libraries by enabling you to create packages that have compound framework and profile target `lib` folders.

As an example, consider the following portable class library's available target platforms.



After the library is built and the command `nuget.exe pack MyPortableProject.csproj` is run, the new portable library package folder structure can be seen by examining the contents of the generated NuGet package.

```
▲ lib
  ▲ portable-win+net40+sl40+wp
    PortableClassLibrary1.dll
```

As you can see, the portable library folder name convention follows the pattern 'portable-{framework 1}+{framework n}' where the framework identifiers follow the existing [framework name and version conventions](#). One exception to the name and version conventions is found in the framework identifier used for Windows Phone. This moniker should use the framework name 'wp' (wp7, wp71 or wp8). Using 'silverlight-wp7', for example, will result in an error.

When installing the package that is created from this folder structure, NuGet can now apply its framework and profile rules to multiple targets, as specified in the folder name. Behind NuGet's matching rules is the principle that "more specific" targets will take precedence over "less specific" ones. This means that monikers targeting a specific platform will always be preferred over portable ones if they are both compatible with a project. Additionally, if multiple portable targets are compatible with a project, NuGet will prefer the one where the set of platforms supported is "closest" to the project referencing the package.

Targeting Windows 8 and Windows Phone 8 Projects

In addition to adding support for targeting portable library projects, NuGet 2.1 provides new framework monikers for both Windows 8 Store and Windows Phone 8 projects, as well as some new general monikers for Windows Store and Windows Phone projects that will be easier to manage across future versions of the respective platforms.

For Windows 8 Store applications, the identifiers look as follows:

NUGET 2.0 AND EARLIER	NUGET 2.1
winRT45, .NETCore45	Windows, Windows8, win, win8

For Windows Phone projects, the identifiers look as follows:

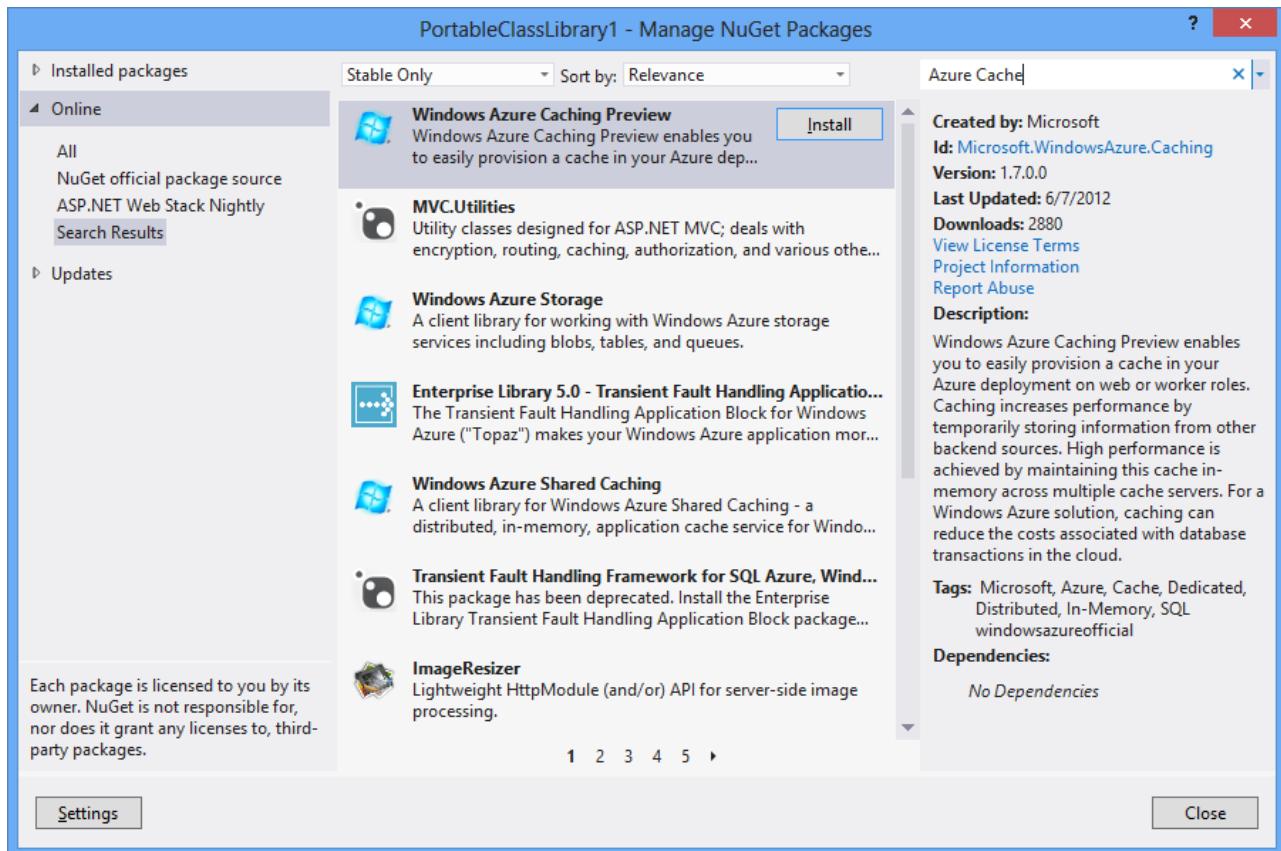
PHONE OS	NUGET 2.0 AND EARLIER	NUGET 2.1
Windows Phone 7	silverlight3-wp	wp, wp7, WindowsPhone, WindowsPhone7
Windows Phone 7.5 (Mango)	silverlight4-wp71	wp71, WindowsPhone71
Windows Phone 8	(not supported)	wp8, WindowsPhone8

In all of the above changes, the old framework names will continue to be fully supported by NuGet 2.1. Moving forward, the new names should be used as they will be more stable across future versions of the respective platforms. The new names will **not** be supported in versions of NuGet prior to 2.1, however, so plan accordingly for when to make the switch.

Improved Search in Package Manager Dialog

Over the past several iterations, changes have been introduced to the NuGet gallery that greatly improved the speed and relevance of package searches. However, these improvements were limited to the [nuget.org](#) Web site. NuGet 2.1 makes the improved search experience available through the NuGet package manager dialog. As an example, imagine that you wanted to find the Windows Azure Caching Preview package. A reasonable search query for this package may be "Azure Cache". In previous versions of the package manager dialog, the desired package would not even be listed on the first page of results. However, in NuGet 2.1, the desired package now

shows up at the top of the search results.



Force Package Update

Prior to NuGet 2.1, NuGet would skip updating a package when there was not a high version number. This introduced friction for certain scenarios – particularly in the case of build or CI scenarios where the team did not want to increment the package version number with each build. The desired behavior was to force an update regardless. NuGet 2.1 addresses this with the 'reinstall' flag. For example, previous versions of NuGet would result in the following when attempting to update a package that did not have a more recent package version:

```
PM> Update-Package Moq
No updates available for 'Moq' in project 'MySolution.MyConsole'.
```

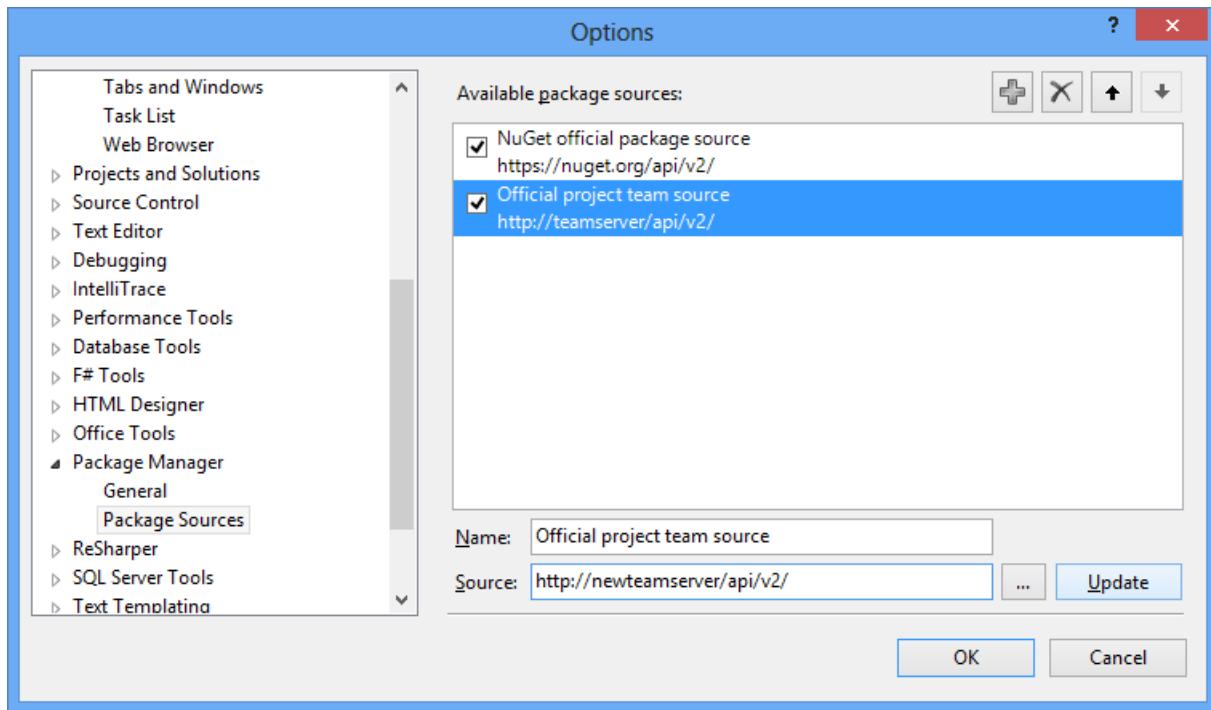
With the reinstall flag, the package will be updated regardless of whether there is a newer version.

```
PM> Update-Package Moq -Reinstall
Successfully removed 'Moq 4.0.10827' from MySolution.MyConsole.
Successfully uninstalled 'Moq 4.0.10827'.
Successfully installed 'Moq 4.0.10827'.
Successfully added 'Moq 4.0.10827' to MySolution.MyConsole.
```

Another scenario where the reinstall flag proves beneficial is that of framework re-targeting. When changing the target framework of a project (for example, from .NET 4 to .NET 4.5), Update-Package -Reinstall can update references to the correct assemblies for all NuGet packages installed in the project.

Edit Package Sources Within Visual Studio

In previous versions of NuGet, updating a package source from within the Visual Studio options dialog required deleting and re-adding the package source. NuGet 2.1 improves this workflow by supporting update as a first class function of the configuration user interface.



Bug Fixes

NuGet 2.1 includes many bug fixes. For a full list of work items fixed in NuGet 2.0, please view the [NuGet Issue Tracker for this release](#).

NuGet 2.0 Release Notes

9/4/2018 • 3 minutes to read • [Edit Online](#)

[NuGet 1.8 Release Notes](#) | [NuGet 2.1 Release Notes](#)

NuGet 2.0 was released on June 19, 2012.

Known Installation Issue

If you are running VS 2010 SP1, you might run into an installation error when attempting to upgrade NuGet if you have an older version installed.

The workaround is to simply uninstall NuGet and then install it from the VS Extension Gallery. See <http://support.microsoft.com/kb/2581019> for more information, or [go directly to the VS hotfix](#).

Note: If Visual Studio won't allow you to uninstall the extension (the Uninstall button is disabled), then you likely need to restart Visual Studio using "Run as Administrator."

Package restore consent is now active

As described in this [post on package restore consent](#), NuGet 2.0 will now require that consent be given to enable package restore to go online and download packages. Please ensure that you have provided consent via either the package manager configuration dialog or the `EnableNuGetPackageRestore` environment variable.

Group dependencies by target frameworks

Starting with version 2.0, package dependencies can vary based on the framework profile of the target project. This is accomplished using an updated `.nuspec` schema. The `<dependencies>` element can now contain a set of `<group>` elements. Each group contains zero or more `<dependency>` elements and a `targetFramework` attribute. All dependencies inside a group are installed together if the target framework is compatible with the target project framework profile. For example:

```
<dependencies>
  <group>
    <dependency id="RouteMagic" version="1.1.0" />
  </group>

  <group targetFramework="net40">
    <dependency id="jQuery" />
    <dependency id="WebActivator" />
  </group>

  <group targetFramework="sl30">
  </group>
</dependencies>
```

Note that a group can contain **zero** dependencies. In the example above, if the package is installed into a project that targets Silverlight 3.0 or later, no dependencies will be installed. If the package is installed into a project that targets .NET 4.0 or later, two dependencies, jQuery and WebActivator, will be installed. If the package is installed into a project that targets an early version of these 2 frameworks, or any other framework, RouteMagic 1.1.0 will be installed. There is no inheritance between groups. If a project's target framework matches the `targetFramework` attribute of a group, only the dependencies within that group will be installed.

A package can specify package dependencies in either of two formats: the old format of a flat list of `<dependency>` elements, or groups. If the `<group>` format is used, the package cannot be installed into versions of NuGet earlier than 2.0.

Note that mixing the two formats is not allowed. For example, the following snippet is **invalid** and will be rejected by NuGet.

```
<dependencies>
  <dependency id="jQuery" />
  <dependency id="WebActivator" />

  <group>
    <dependency id="RouteMagic" version="1.1.0" />
  </group>
</dependencies>
```

Grouping content files and PowerShell scripts by target framework

In addition to assembly references, content files and PowerShell scripts can also be grouped by target framework. The same folder structure found in the `lib` folder for specifying target framework can now be applied in the same way to the `content` and `tools` folders. For example:

```
\content
  \net11
    \MyContent.txt
  \net20
    \MyContent20.txt
  \net40
  \sl40
    \MySilverlightContent.html

\tools
  \init.ps1
  \net40
    \install.ps1
    \uninstall.ps1
  \sl40
    \install.ps1
    \uninstall.ps1
```

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

Also, a new feature in NuGet 2.0 is that a framework folder can be *empty*, in which case, NuGet will not add assembly references, add content files or run PowerShell scripts for the particular framework version. In the example above, the folder `content\net40` is empty.

Improved tab completion performance

The tab completion feature in the NuGet Package Manager Console has been updated to significantly improve performance. There will be much less delay from the time the tab key is pressed until the suggestion dropdown appears.

Bug Fixes

NuGet 2.0 includes many bug fixes with an emphasis on package restore consent and performance. For a full list of work items fixed in NuGet 2.0, please view the [NuGet Issue Tracker for this release](#).

NuGet 1.8 Release Notes

9/4/2018 • 3 minutes to read • [Edit Online](#)

[NuGet 1.7 Release Notes](#) | [NuGet 2.0 Release Notes](#)

NuGet 1.8 was released on May 23, 2012.

Known Installation Issue

If you are running VS 2010 SP1, you might run into an installation error when attempting to upgrade NuGet if you have an older version installed.

The workaround is to simply uninstall NuGet and then install it from the VS Extension Gallery. See <http://support.microsoft.com/kb/2581019> for more information, or [go directly to the VS hotfix](#).

Note: If Visual Studio won't allow you to uninstall the extension (the Uninstall button is disabled), then you likely need to restart Visual Studio using "Run as Administrator."

NuGet 1.8 Incompatible with Windows XP, hotfix published

Shortly after NuGet 1.8 was released, we learned that a cryptography change in 1.8 broke users on Windows XP.

We have since released a hotfix that addresses this issue. By updating NuGet through the Visual Studio Extension Gallery, you receive this hotfix.

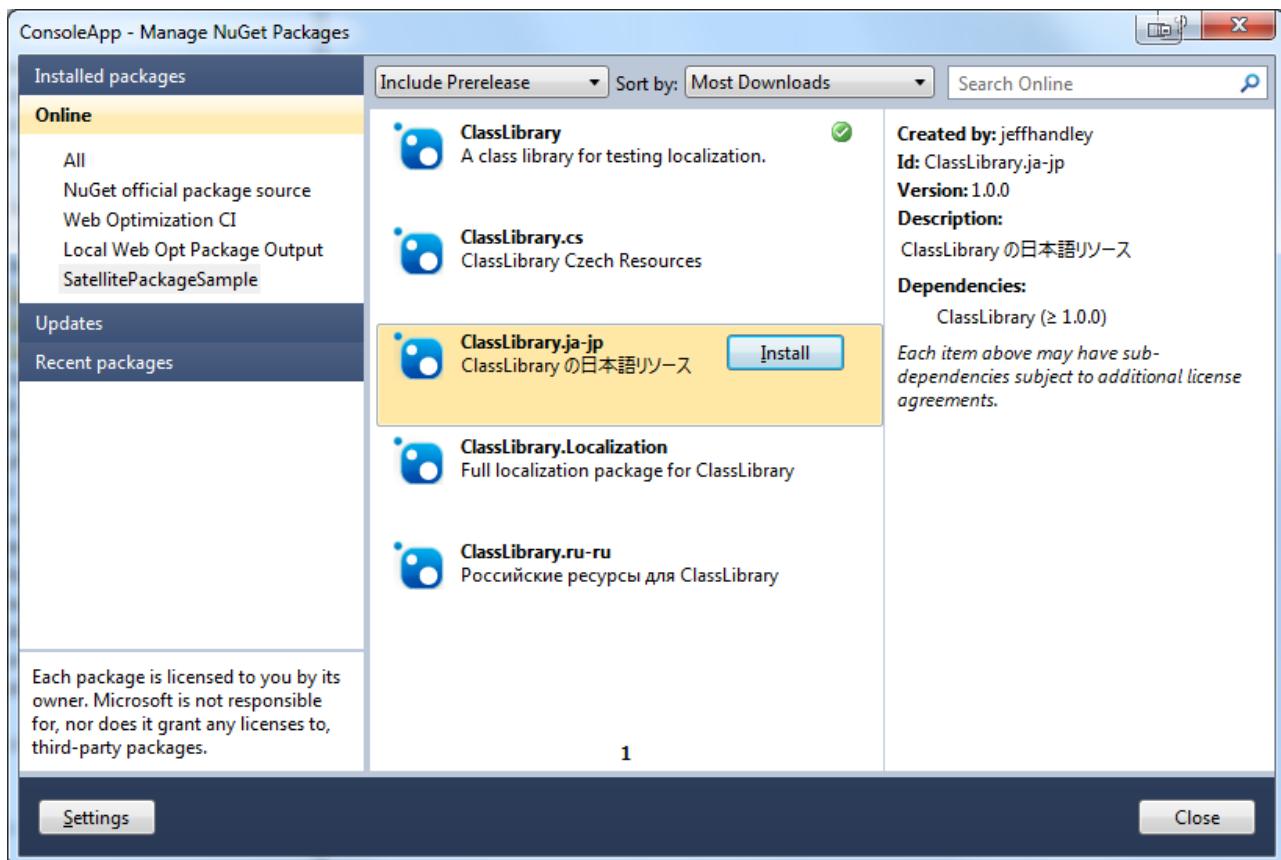
Features

Satellite Packages for Localized Resources

NuGet 1.8 now supports the ability to create separate packages for localized resources, similar to the satellite assembly capabilities of the .NET Framework. A satellite package is created in the same way as any other NuGet package with the addition of a few conventions:

- The satellite package ID and file name should include a suffix that matches one of the standard [culture strings used by the .NET Framework](#).
- In its `.nuspec` file, the satellite package should define a language element with the same culture string used in the ID
- The satellite package should define a dependency in its `.nuspec` file to its core package, which is simply the package with the same ID minus the language suffix. The core package needs to be available in the repository for successful installation.

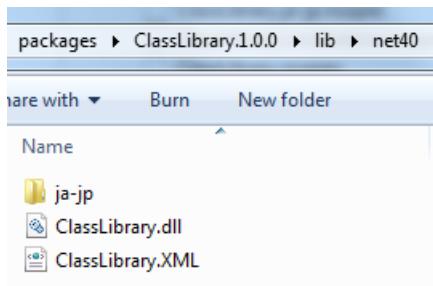
To install a package with localized resources, a developer explicitly selects the localized package from the repository. At present, the NuGet gallery does not give any kind of special treatment to satellite packages.



Because the satellite package lists a dependency to its core package, both the satellite and core packages are pulled into the NuGet packages folder and installed.



Additionally, while installing the satellite package, NuGet also recognizes the culture string naming convention and then copies the localized resource assembly into the correct subfolder within the core package so that it can be picked by the .NET Framework.



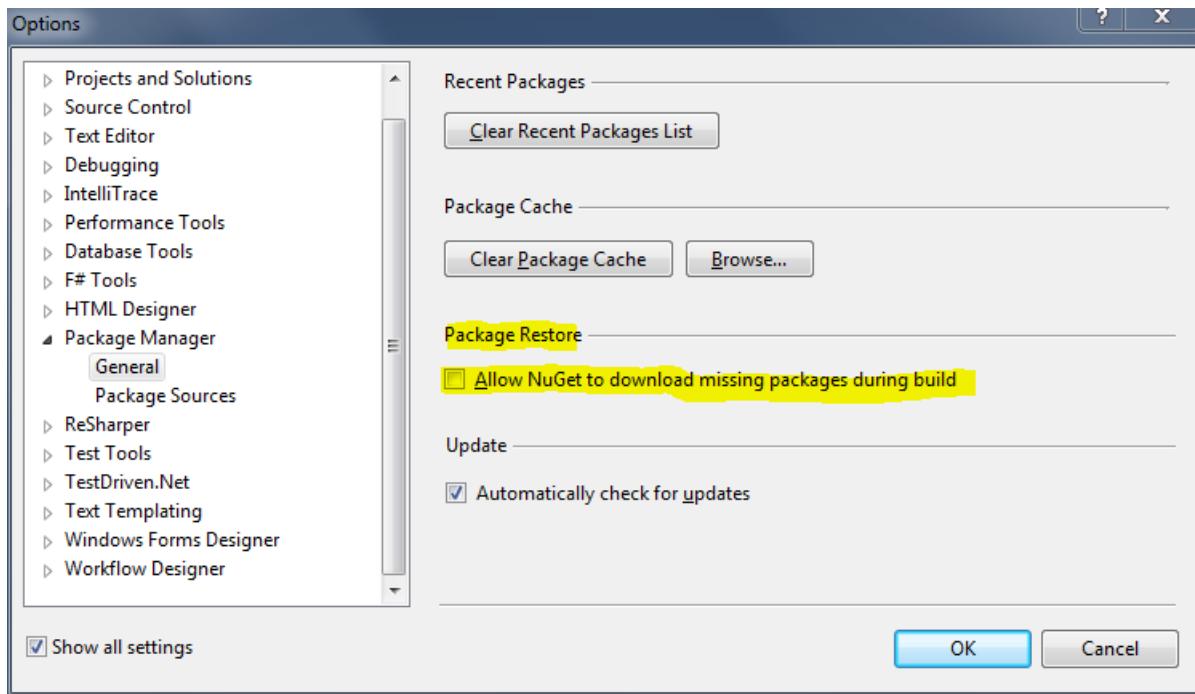
One existing bug to note with satellite packages is that NuGet does not copy localized resources to the `bin` folder for Web site projects. This issue will be fixed in the next release of NuGet.

For a complete sample demonstrating how to create and use satellite packages, see <https://github.com/NuGet/SatellitePackageSample>.

Package Restore Consent

In NuGet 1.8, we laid the groundwork for supporting an important constraint on package restore to protect user privacy. This constraint requires developers building projects and solutions that are using package restore to explicitly consent to package restore's going online to download packages from configured package sources.

There are 2 ways to provide this consent. The first can be found in the package manager configuration dialog as shown below. This method is primarily intended for developer machines.



The second method is to set the environment variable "EnableNuGetPackageRestore" to the value "true". This method is intended for unattended machines such as CI or build servers.

Now, as stated above, we have only laid the groundwork for this feature in NuGet 1.8. Practically, this means that while we've added all of the logic to enable the feature, it's not currently enforced in this version. It will be enabled, however, in the next release of NuGet, so we wanted to make you aware of it as soon as possible so that you can configure your environments appropriately and therefore not be impacted when we start enforce the consent constraint.

For more details, please see the [team blog post](#) on this feature.

nuget.exe Performance Improvements

By modifying the install command to download and install packages in parallel, NuGet 1.8 brings dramatic performance improvements to nuget.exe – and by extension package restore. High level testing shows that performance for installing 6 packages into a project improves by about 35% in NuGet 1.8. Increasing the number of packages to 25 shows a performance gain of about 60%.

Bug Fixes

NuGet 1.8 includes quite a few bug fixes with an emphasis on the package manager console and package restore workflow, particularly as it relates to package restore consent and Windows 8 Express integration. For a full list of work items fixed in NuGet 1.8, please view the [NuGet Issue Tracker for this release](#).

NuGet 1.7 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 1.6 Release Notes](#) | [NuGet 1.8 Release Notes](#)

NuGet 1.7 was released on April 4, 2012.

Known Installation Issue

If you are running VS 2010 SP1, you might run into an installation error when attempting to upgrade NuGet if you have an older version installed.

The workaround is to simply uninstall NuGet and then install it from the VS Extension Gallery. See <http://support.microsoft.com/kb/2581019> for more information.

Note: If Visual Studio won't allow you to uninstall the extension (the Uninstall button is disabled), then you likely need to restart Visual Studio using "Run as Administrator."

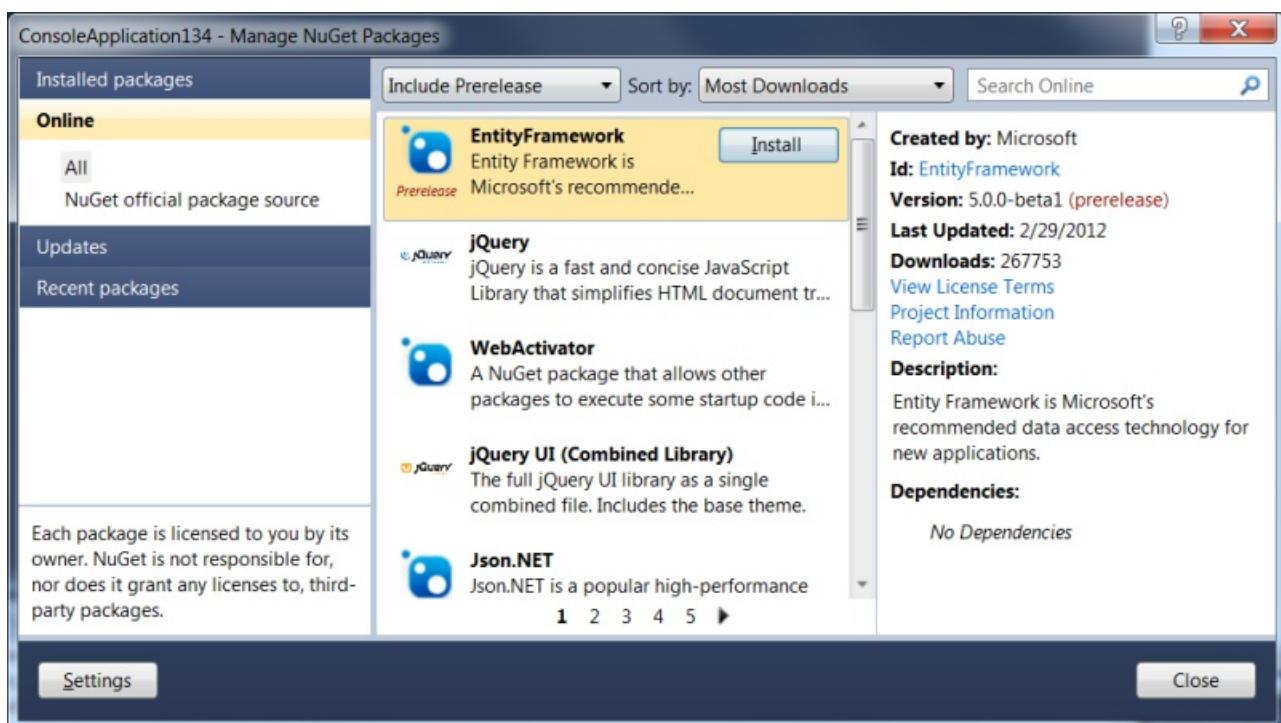
Features

Support opening `readme.txt` file after installation

New in 1.7, if your package includes a `readme.txt` file at the root of the package, NuGet will automatically open this file after it's finished installing your package.

Show prerelease packages in the Manage NuGet packages dialog

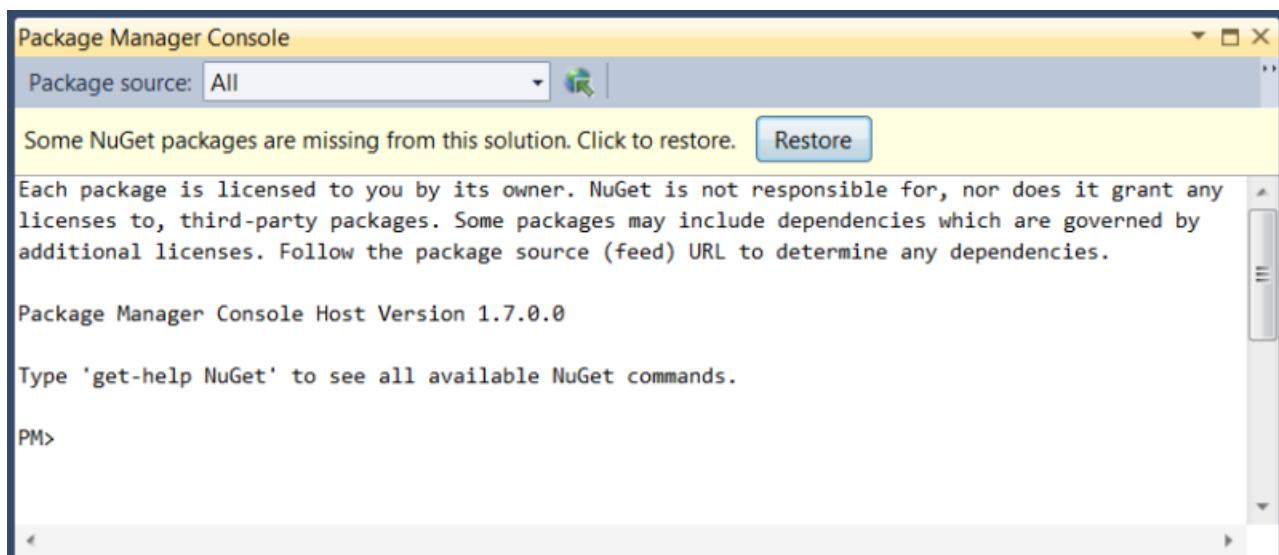
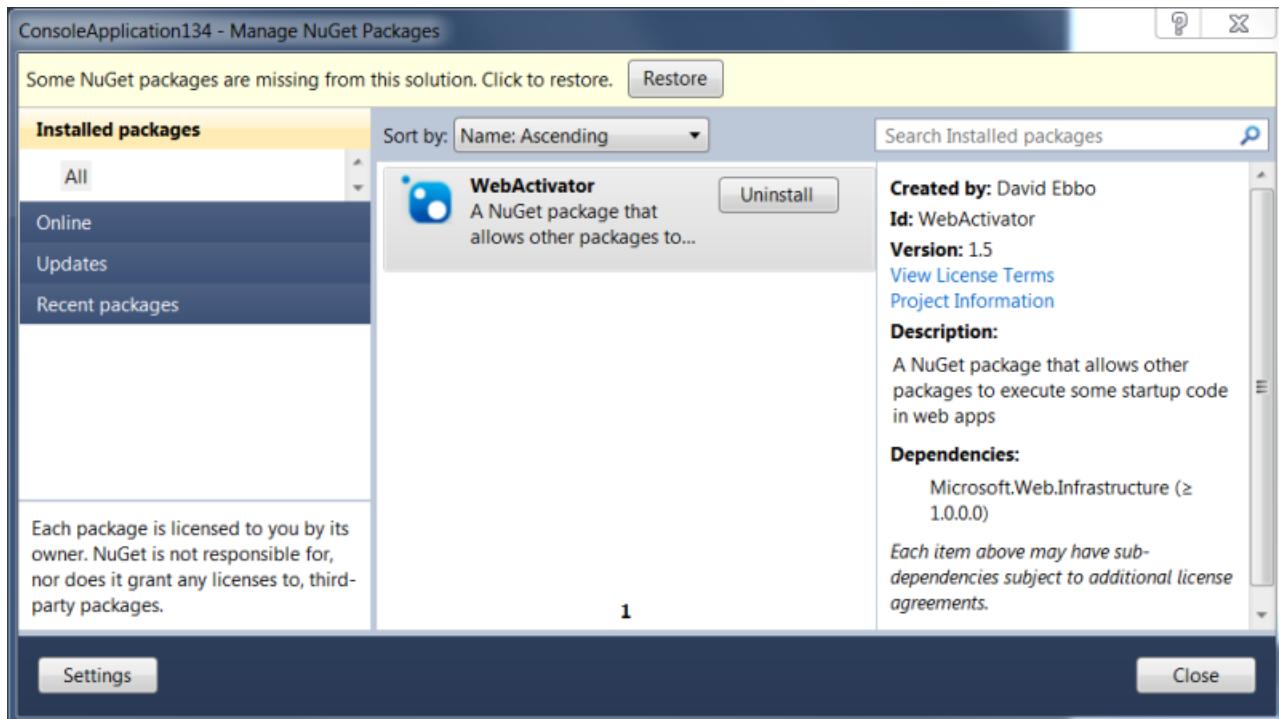
The Manage NuGet Packages dialog now includes a dropdown which provides option to show prerelease packages.



Show Package Restore button when package files are missing

When you open the Package Manager console or the Manager NuGet packages dialog, NuGet will check if the current solution has enabled the Package Restore mode and if any package files are missing from the `packages`

folder. If these two conditions are met, NuGet will notify you and will show a convenient Restore button. Clicking this button will trigger NuGet to restore all the missing packages.



Add solution-level packages.config file

In previous versions of NuGet, each project has a `packages.config` file which keeps track of what NuGet packages are installed in that project. However, there was no similar file at the solution level to keep track of solution-level packages. As a result, there was no way to restore solution-level packages. This feature is now implemented in NuGet 1.7. The solution-level `packages.config` file is placed under the `.nuget` folder under solution root and will store only solution-level packages.

Remove New-Package command

Due to low usage, the New-Package command has been removed. Developers are recommended to use `nuget.exe` or the handy NuGet Package Explorer to create packages.

Bug Fixes

NuGet 1.7 has fixed many bugs around the Package Restore workflow and Network/Source Control scenarios.

For a full list of work items fixed in NuGet 1.7, please view the [NuGet Issue Tracker for this release](#).

NuGet 1.6 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 1.5 Release Notes](#) | [NuGet 1.7 Release Notes](#)

NuGet 1.6 was released on December 13, 2011.

Known Installation Issue

If you are running VS 2010 SP1, you might run into an installation error when attempting to upgrade NuGet if you have an older version installed.

The workaround is to simply uninstall NuGet and then install it from the VS Extension Gallery. See <http://support.microsoft.com/kb/2581019> for more information.

Note: If Visual Studio won't allow you to uninstall the extension (the Uninstall button is disabled), then you likely need to restart Visual Studio using "Run as Administrator."

Features

Support for Semantic Versioning and Prerelease Packages

NuGet 1.6 introduces support for Semantic Versioning (SemVer). For more details on how it uses SemVer, read the [Versioning documentation](#).

Using NuGet Without Checking In Packages (Package Restore)

NuGet 1.6 now has first class support for the workflow in which NuGet packages are not added to source control, but instead are restored at build time if missing. For more details, read the [Using NuGet without committing packages to source control](#) topic.

Item Templates That Install NuGet Packages

Building on the work to support preinstalled NuGet package to Visual Studio project templates, NuGet 1.6 also adds support for Visual Studio item templates. Item templates can have associated NuGet packages that are installed when the template is invoked.

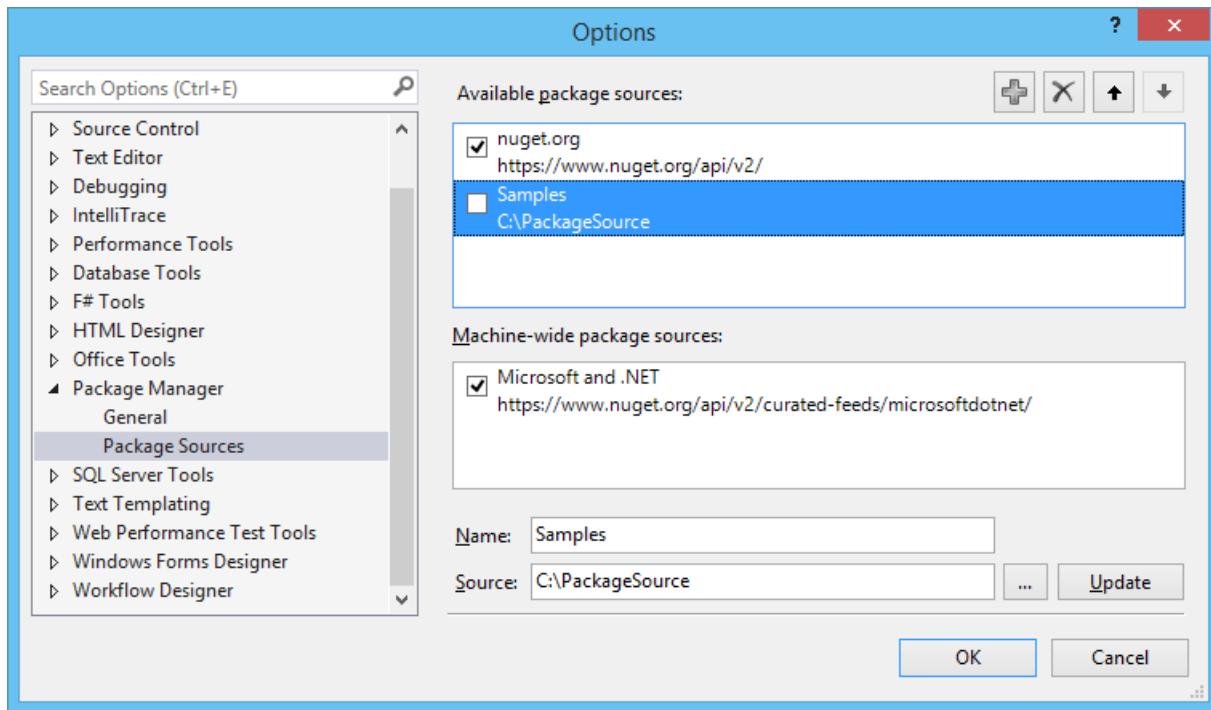
For more details on how to change a project/item template to install NuGet packages, read the [Packages in Visual Studio Templates](#) topic.

Support for disabling package sources

When multiple package sources are configured, NuGet will look in each one for packages during installation of a package and its dependencies. A package source that is down for some reason can severely slow down NuGet.

Prior to NuGet 1.6, you could remove the package source, but then you have to remember the details for when you want to add it back in.

NuGet 1.6 allows unchecking a package source to disable it, but keep it around.



Bug Fixes

NuGet 1.6 had a total of 106 work items fixed. 95 of those were classified as bugs and 10 of those were features.

For a full list of work items fixed in NuGet 1.6, please view the [NuGet Issue Tracker for this release](#).

NuGet 1.5 Release Notes

9/4/2018 • 3 minutes to read • [Edit Online](#)

[NuGet 1.4 Release Notes](#) | [NuGet 1.6 Release Notes](#)

NuGet 1.5 was released on August 30, 2011.

Features

Project Templates with Preinstalled NuGet Packages

When creating a new ASP.NET MVC 3 project template, the jQuery script libraries included in the project are actually placed there by installing NuGet packages.

The ASP.NET MVC 3 project template includes a set of NuGet packages that get installed when the project template is invoked. This ability to include NuGet packages with a project template is now a feature of NuGet that *any* project template can now take advantage of.

For more details about this feature, read this [blog post by the developer of the feature](#).

Explicit Assembly References

Added a new `<references />` element used to explicitly specify which assemblies within the package should be referenced.

For example, if you add the following:

```
<references>
  <reference file="xunit.dll" />
  <reference file="xunit.extensions.dll" />
</references>
```

Then only the `xunit.dll` and `xunit.extensions.dll` will be referenced from the appropriate [framework/profile subfolder](#) of the `lib` folder even if there are other assemblies in the folder.

If this element is omitted, then the usual behavior applies, which is to reference every assembly in the `lib` folder.

What is this feature used for?

This feature supports design-time only assemblies. For example, when using Code Contracts, the contract assemblies need to be next to the runtime assemblies that they augment so that Visual Studio can find them, but the contract assemblies should not actually be referenced by the project and should not be copied into the `bin` folder.

Likewise, the feature can be used to for unit test frameworks such as XUnit which need its tools assemblies to be located next to the runtime assemblies, but excluded from project references.

Added ability to exclude files in the .nuspec

The `<file>` element within a `.nuspec` file can be used to include a specific file or a set of files using a wildcard. When using a wildcard, there's no way to exclude a specific subset of the included files. For example, suppose you want all text files within a folder except a specific one.

```
<files>
  <file src="*.txt" target="content\docs" exclude="admin.txt" />
</files>
```

Use semicolons to specify multiple files.

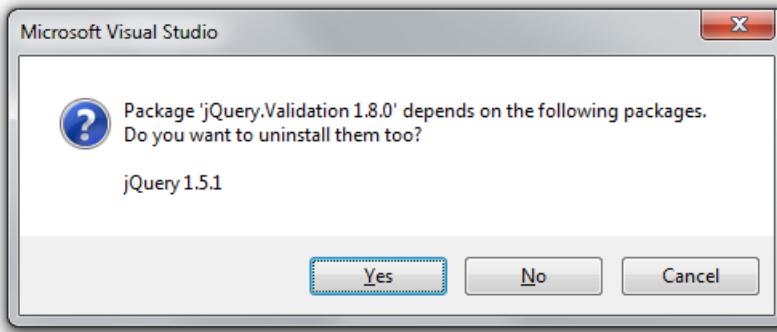
```
<files>
  <file src="*.txt" target="content\docs" exclude="admin.txt;log.txt" />
</files>
```

Or use a wild card to exclude a set of files such as all backup files

```
<files>
  <file src="tools\*.*" target="tools" exclude="*.bak" />
</files>
```

Removing packages using the dialog prompts to remove dependencies

When uninstalling a package with dependencies, NuGet prompts, allowing the removal of a package's dependencies along with the package.



Get-Package command improvement

The `Get-Package` command now supports a `-ProjectName` parameter. So the command

```
Get-Package -ProjectName A
```

will list all packages installed in project A.

Support for Proxies that require authentication

When using NuGet behind a proxy that requires authentication, NuGet will now prompt for proxy credentials. Entering credentials allows NuGet to connect to the remote repository.

Support for Repositories that require authentication

NuGet now supports connecting to [private repositories](#) that require basic or NTLM authentication.

Support for Digest authentication will be added in a future release.

Performance improvements to the nuget.org repository

We've made several performance improvements to the nuget.org gallery to make package listing and searching faster.

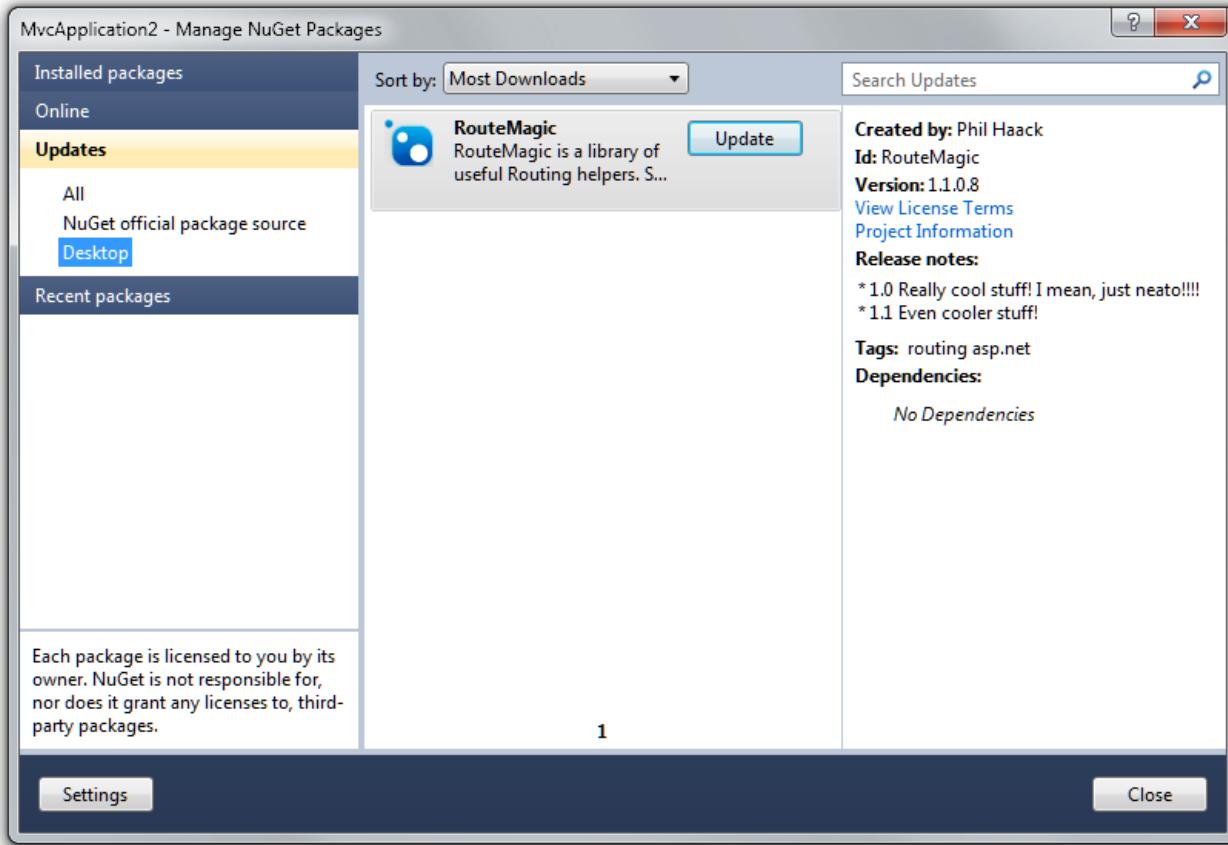
Solution dialog project filtering

In the Solution-level dialog, when prompting for what projects to install, we only show projects that are compatible

with the selected package.

Package Release Notes

NuGet packages now include support for release notes. The release notes only show up when viewing *Updates* for a package, so it doesn't make sense to add them to your first release.



To add release notes to a package, use the new `<releaseNotes />` metadata element in your NuSpec file.

.nuspec <files /> improvement

The `.nuspec` file now allows empty `<files />` element, which tells nuget.exe not to include any file in the package.

Bug Fixes

NuGet 1.5 had a total of 107 work items fixed. 103 of those were marked as bugs.

For a full list of work items fixed in NuGet 1.5, please view the [NuGet Issue Tracker for this release](#).

Bug fixes worth noting:

- [Issue 1273](#): Made `packages.config` more version control friendly by sorting packages alphabetically and removing extra whitespace.
- [Issue 844](#): Version numbers are now normalized so that `Install-Package 1.0` works on a package with the version `1.0.0`.
- [Issue 1060](#): When creating a package using nuget.exe, the `-Version` flag overrides the `<version />` element.

NuGet 1.4 Release Notes

8/15/2019 • 6 minutes to read • [Edit Online](#)

[NuGet 1.3 Release Notes](#) | [NuGet 1.5 Release Notes](#)

NuGet 1.4 was released on June 17, 2011.

Features

Update-Package improvements

NuGet 1.4 introduces a lot of improvements to the Update-Package command making it easier to keep packages at the same version across multiple projects in a solution. For example, when upgrading a package to the latest version, it's very common to want all projects with that package installed to be updated to the same version.

The `Update-Package` command now makes it easier to:

Update all packages in a single project

```
Update-Package -Project MvcApplication1
```

Update a package in all projects

```
Update-Package PackageId
```

Update all packages in all projects

```
Update-Package
```

Perform a "safe" update on all packages

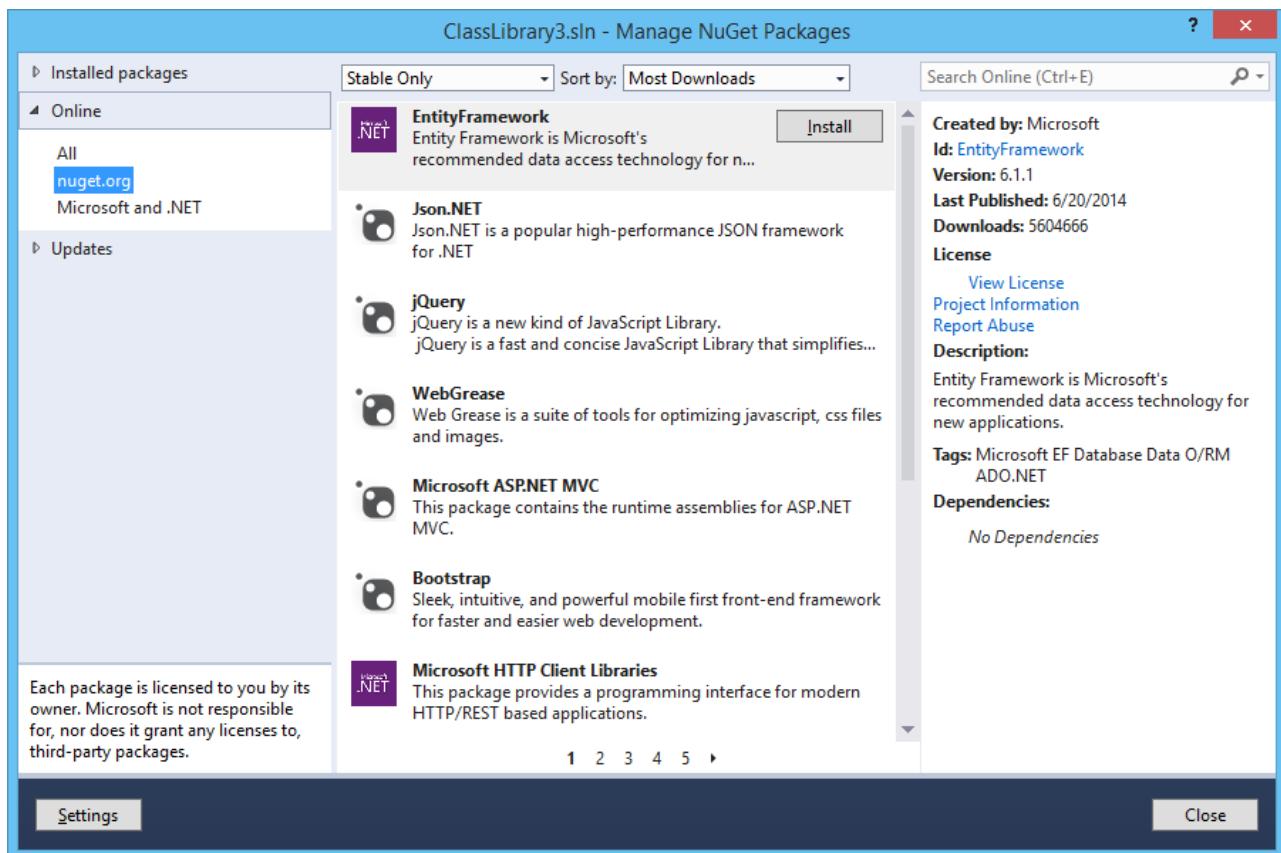
The `-Safe` flag constrains upgrades to only versions with the same Major and Minor version component. For example, if version 1.0.0 of a package is installed, and versions 1.0.1, 1.0.2, and 1.1 are available in the feed, the `-Safe` flag updates the package to 1.0.2. Upgrading without the `-Safe` flag would upgrade the package to the latest version, 1.1.

```
Update-Package -Safe
```

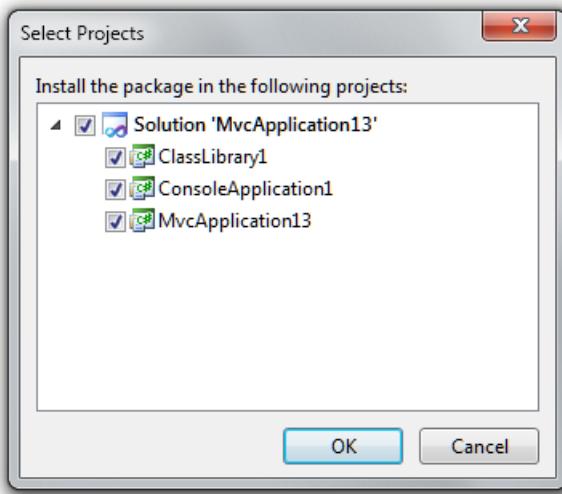
Managing Packages at the Solution Level

Prior to NuGet 1.4, installing a package into multiple projects was cumbersome using the dialog. It required launching the dialog once per project.

NuGet 1.4 adds support for installing/uninstalling/updating packages in multiple projects at the same time. Simply launch the by right clicking the Solution and selecting the **Manage NuGet Packages** menu option.



Notice that in the title bar of the dialog, the name of the solution is displayed, not the name of a project. Package operations now provide a list of checkboxes with the list of projects the operation should apply to.



For more details, see the topic on [Managing Packages for the Solution](#).

Constraining Upgrades To Allowed Versions

By default, when running the `Update-Package` command on a package (or updating the package using dialog), it will be updated to the latest version in the feed. With the new support for updating all packages, there may be cases in which you want to lock a package to a specific version range. For example, you may know in advance that your application will only work with version 2.* of a package, but not 3.0 and above. In order to prevent accidentally updating the package to 3, NuGet 1.4 adds support for constraining the range of versions that packages can be upgraded to by hand editing the `packages.config` file using the new `allowedVersions` attribute.

For example, the following example shows how to lock the `SomePackage` package the version range 2.0 - 3.0 (exclusive). The `allowedVersions` attribute accepts values using the [version range format](#).

```

<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="SomePackage" version="2.1.0" allowedVersions="[2.0, 3.0)" />
</packages>

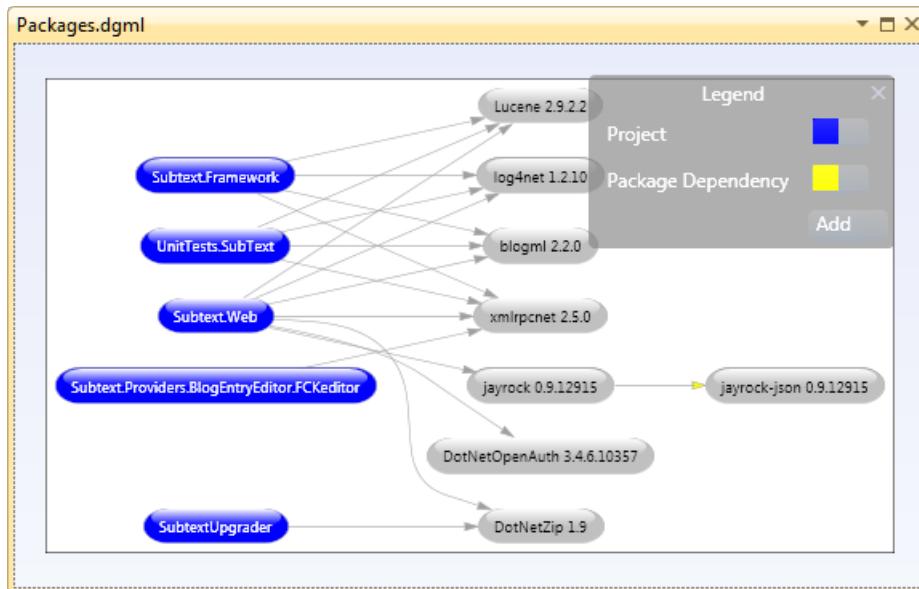
```

Note that in 1.4, locking a package to a specific version range must be hand-edited. In NuGet 1.5 we plan to add support for placing this range via the `Install-Package` command.

Package Visualizer

The new package visualizer, launched via the **Tools -> Library Package Manager -> Package Visualizer** menu option, enables you to easily visualize all the projects and their package dependencies within a solution.

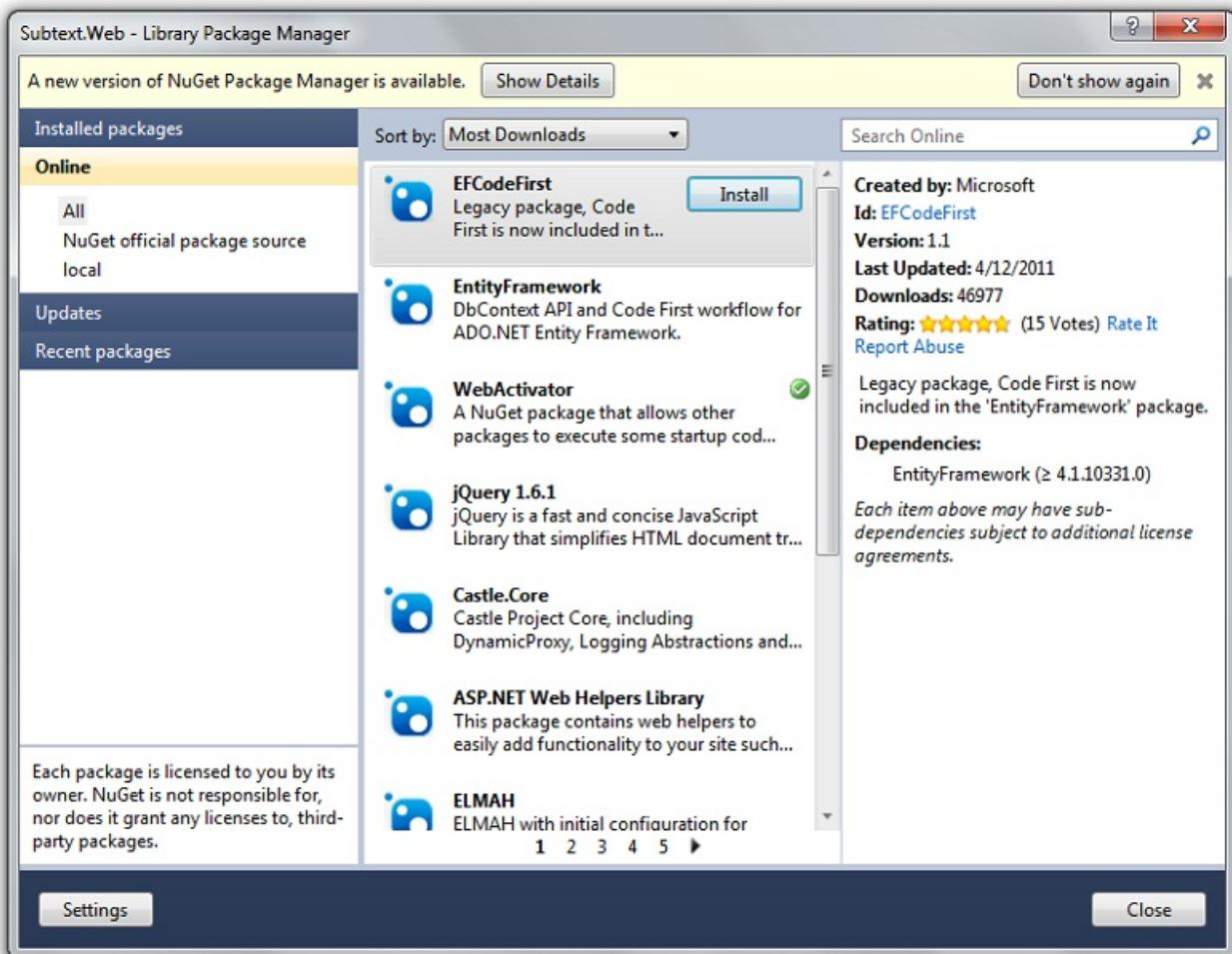
Important Note: *This feature takes advantage of the DGML support in Visual Studio. Creating the visualization is only supported in Visual Studio Ultimate. Viewing a DGML diagram is only supported in Visual Studio Premium or Higher.*



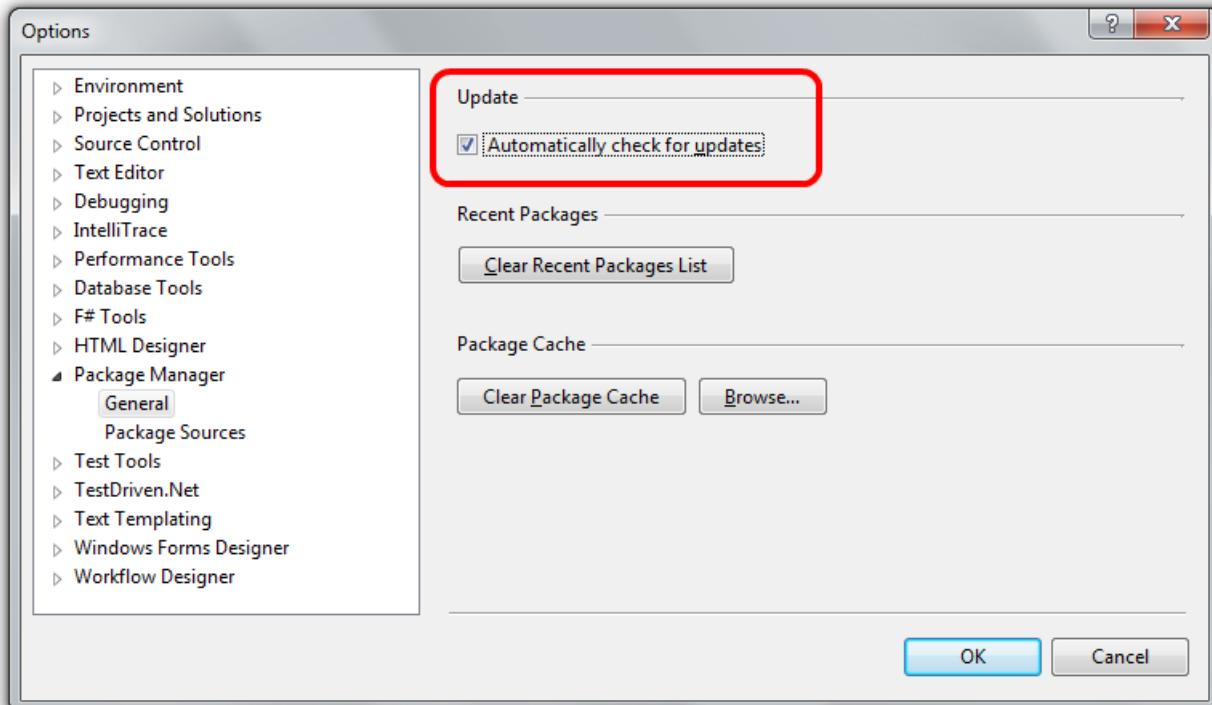
Automatic Update Check for the NuGet Dialog

Some versions of NuGet introduce new features expressed via the `.nuspec` file which are not understood by older versions of the NuGet dialog. One example is the introduction in NuGet 1.4 for [specifying framework assemblies](#). Because of this, it's important to use the latest version of NuGet to ensure you can use packages taking advantage of the latest features. To make updates to NuGet more visible, the NuGet dialog contains logic to highlight when a newer version is available.

Note: The check is only made if the **Online** tab has been selected in the current session.



To turn off the automatic check for updates, go to the NuGet settings dialog and uncheck **Automatically check for updates**.



This feature was actually added in NuGet 1.3, but would not be visible, of course, until an update to 1.3, such as NuGet 1.4, was made available.

Package Manager Dialog Improvements

- **Menu names improved:** Menu options to launch the dialog have been renamed for clarity. The menu option is now **Manage NuGet Packages**.
- **Details pane shows latest update date:** The NuGet dialog displays the date of the latest update in the details pane for a package when the **Online** or **Updates** tab is selected.
- **List of tags displayed:** The Nuget dialog displays tags.

Powershell Improvements

- **Signed PowerShell scripts:** NuGet includes signed Powershell scripts enabling usage in more restrictive environments.
- **Prompting Support:** The Package Manager Console now supports prompting via the `$host.ui.Prompt` and `$host.ui.PromptForChoice` commands.
- **Package Source Names:** Supplying the name of a package source is supported when specifying a package source using the `-Source` flag.

nuget.exe Command line improvements

- **NuGet Custom Commands:** nuget.exe is extensible via custom commands using MEF.
- **Simpler the workflow for creating symbol packages:** The `-Symbols` flag can be applied to a normal convention based folder structure creating a symbols package by only including the source and `.pdb` files within the folder.
- **Specifying Multiple Sources:** The `NuGet install` command supports specifying multiple sources using semi-colons as a delimiter or by specifying `-Source` multiple times.
- **Proxy Authentication Support:** NuGet 1.4 adds support for prompting for user credentials when using NuGet behind a proxy that requires authentication.
- **nuget.exe Update Breaking Change:** The `-Self` flag is now required for nuget.exe to update itself. `nuget.exe Update` now takes in a path to the `packages.config` file and will attempt to update packages. Note that this update is limited in that it will not: ** Update, add, remove content in the project file. ** Run Powershell scripts within the package.

NuGet Server Support for Pushing Packages using nuget.exe

NuGet includes a simple way to host a [lightweight web based NuGet repository](#) via the `NuGet.Server` NuGet package. With NuGet 1.4, the lightweight server supports pushing and deleting packages using nuget.exe. The latest version of `NuGet.Server` adds a new `appSetting`, named `apiKey`. When the key is omitted or left blank, pushing packages to the feed is disabled. Setting the apiKey to a value (ideally a strong password) enables pushing packages using nuget.exe.

```

<appSettings>
    <!-- Set the value here to allow people to push/delete packages from the server.
        NOTE: This is a shared key (password) for all users. -->
    <add key="apiKey" value="" />
</appSettings>

```

Support for Windows Phone Tools Mango Edition

NuGet is now supported in the release candidate version of Windows Phone Tools for Mango. Currently, Windows Phone Tools does not have support for the Visual Studio Extension manager so to install NuGet for Windows Phone Tools, you may need to download and run the VSIX manually.

To uninstall NuGet for Windows Phone Tools, run the following command.

```
vsixinstaller.exe /uninstall:NuPackToolsVsix.Microsoft.67e54e40-0ae3-42c5-a949-fddff5739e7a5
```

Bug Fixes

NuGet 1.4 had a total of 88 work items fixed. 71 of those were marked as bugs.

For a full list of work items fixed in NuGet 1.4, please view the [NuGet Issue Tracker for this release](#).

Bug fixes worth noting:

- [Issue 603](#): Package dependencies across different repositories resolves correctly when specifying a specific package source.
- [Issue 1036](#): Adding `NuGet Pack SomeProject.csproj` to post-build event no longer causes an infinite loop.
- [Issue 961](#): `-Source` flag supports relative paths.

NuGet 1.4 Update

Shortly after the release of NuGet 1.4, we found a couple of issues that were important to fix. The specific version number of this update to 1.4 is 1.4.20615.9020.

Bug Fixes

- [Issue 1220](#): Update-Package doesn't execute `install.ps1` / `uninstall.ps1` in all projects when there is more than one project
- [Issue 1156](#): Package Manager Consol stuck on W2K3/XP (when Powershell 2 is not installed)

NuGet 1.3 Release Notes

9/4/2018 • 2 minutes to read • [Edit Online](#)

[NuGet 1.2 Release Notes](#) | [NuGet 1.4 Release Notes](#)

NuGet 1.3 was released on April 25, 2011.

New Features

Streamlined Package Creation with symbol server integration

The NuGet team partnered with the folks at [SymbolSource.org](#) to offer a really simple way of publishing your sources and PDB's along with your package. This allows consumers of your package to step into the source for your package in the debugger. For more details, read [Creating and Publishing a Symbol Package](#) The easy way to publish NuGet packages with sources. You can also watch a live demonstration of this feature as part of the NuGet in Depth talk at Mix11. This feature is fully demonstrated starting at the 20 minute mark of the video.

`Open-PackagePage` **Command**

This command makes it easy to get to the project page for a package from within the Package Manager Console. It also provides options to open the license URL and the report abuse page for the package. The syntax for the command is:

```
Open-PackagePage -Id <string> [-Version] [-Source] [-License] [-ReportAbuse] [-PassThru]
```

The `-PassThru` option is used to return the value of the specified URL.

Examples:

```
PM> Open-PackagePage Ninject
```

Opens a browser to the project URL specified in the Ninject package.

```
PM> Open-PackagePage Ninject -License
```

Opens a browser to the license URL specified in the Ninject package.

```
PM> Open-PackagePage Ninject -ReportAbuse
```

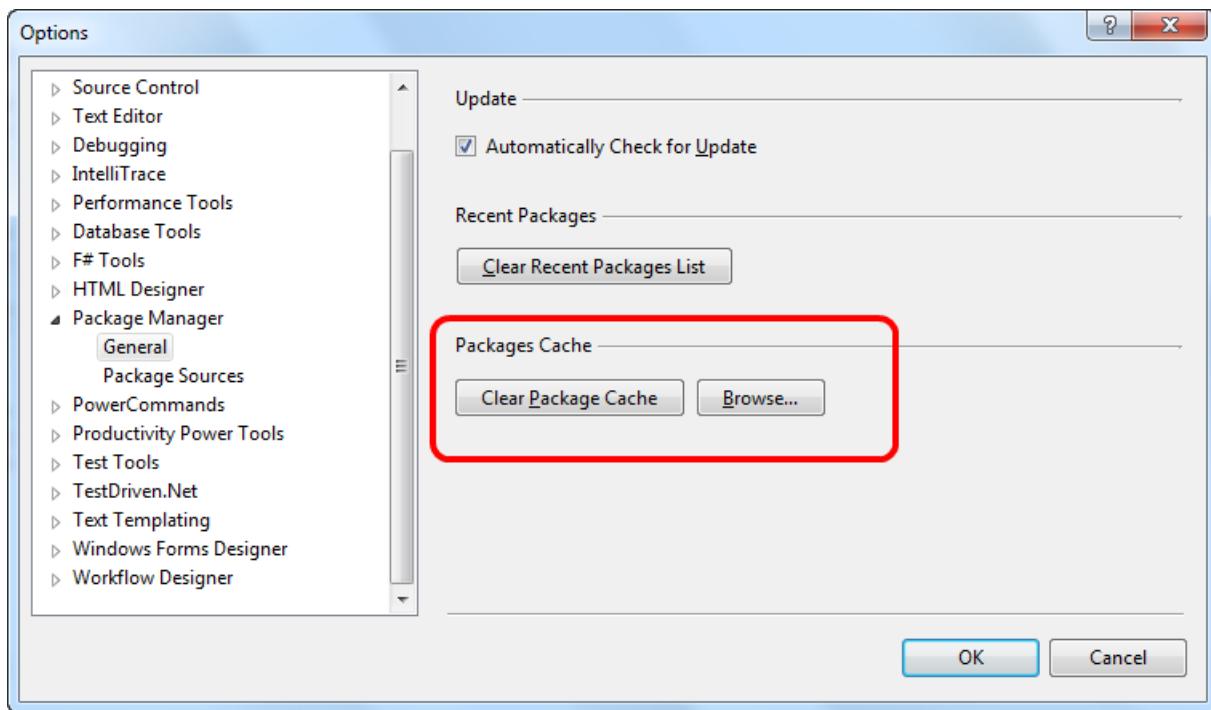
Opens a browser to the URL at the current package source used to report abuse for the specified package.

```
PM> $url = Open-PackagePage Ninject -License -WhatIf -PassThru
```

Assigns the license URL to the variable, \$url, without opening the URL in a browser.

Performance Improvements

NuGet 1.3 introduces a lot of performance improvements. NuGet 1.3 avoids downloading the same version of a package multiple times by including a local per-user cache. The cache can be accessed and cleared via the Package Manager Settings dialog:



Other performance improvements include adding support for HTTP compression and improving the package installation speed within Visual Studio.

Visual Studio and nuget.exe uses the same list of package sources

Prior to NuGet 1.3, the list of package sources used by nuget.exe and the NuGet Visual Studio Add-In were not stored in the same place. NuGet 1.3 now uses the same list in both places. The list is stored in `NuGet.Config` and stored in the AppData folder.

nuget.exe Ignores Files and Folders that start with '.' by default

In order to make NuGet work well with source control systems such Subversion and Mercurial, nuget.exe ignores folders and files that start with the '.' character when creating packages. This can be overridden using two new flags:

- **-NoDefaultExcludes** is used to override this setting and include all files.
- **-Exclude** is used to add other files/folders to exclude using a pattern. For example, to exclude all files with the '.bak' file extension

```
nuget Pack MyPackage.nuspec -Exclude **\*.bak
```

Note: the pattern is not recursive by default.

Support for WiX Projects and the .NET Micro Framework

Thanks to community contributions, NuGet includes support for WiX project types as well as the .NET Micro Framework.

Bug Fixes

For a full list of bug fixes, please view the [NuGet Issue Tracker for this release](#).

Bug fixes worth noting

- Packages with source files work in both Websites and in Web Application Projects. For Websites, source files are copied into the `App_Code` folder

NuGet 1.2 Release Notes

6/28/2019 • 3 minutes to read • [Edit Online](#)

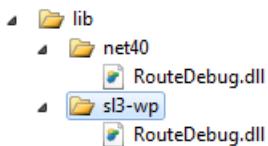
[NuGet 1.0 and 1.1 Release Notes](#) | [NuGet 1.3 Release Notes](#)

NuGet 1.2 was released on March 30, 2011.

New Features

Framework Profile Support

From the start, NuGet supported having libraries target different frameworks. But now packages may contain assemblies that target specific profiles such as the Windows Phone profile. To target a specific profile of a framework, append a dash followed by the profile abbreviation. For example, to target SilverLight running on a Windows Phone (aka Windows Phone 7), you can put an assembly in the sl3-wp folder as demonstrated in the following screenshot.



You might ask why we didn't just choose to use "wp7" as the moniker. In part, we're anticipating that Windows Phone 7 might run a newer version of Silverlight in the future, in which case you may need to be more specific about which framework profile you're targeting.

Automatically Add Binding Redirects

When installing a package with strong named assemblies, NuGet can now detect cases where the project requires binding redirects to be added to the configuration file in order for the project to compile and add them automatically. Part 3 of David Ebbo's blog post series on NuGet Versioning entitled "[Unification via Binding Redirects](#)" covers the purpose of this feature in more details.

Specifying Framework Assembly References (GAC)

In some cases, a package may depend on an assembly that's in the .NET Framework. Strictly speaking, it's not always necessary that the consumer of your package reference the framework assembly. But in some cases, it's important, such as when the developer needs to code against types in that assembly in order to use your package. The new `frameworkAssemblies` element, a child element of the metadata element, allows you to specify a set of `frameworkAssembly` elements pointing to a Framework assembly in the GAC. Note the emphasis on Framework assembly. These assemblies are not included in your package as they are assumed to be on every machine as part of the .NET Framework. The following table lists attributes of the `frameworkAssembly` element.

ATTRIBUTE	DESCRIPTION
assemblyName	<i>Required.</i> Name of the assembly such as <code>System.Net</code> .
targetFramework	<i>Optional.</i> Allows specifying a framework and profile name (or alias) that this framework assembly applies to such as "net40" or "sl4". Uses the same format described in Supporting Multiple Target Frameworks .

```
<frameworkAssemblies>
  <frameworkAssembly assemblyName="System.ComponentModel.DataAnnotations" targetFramework="net40" />
  <frameworkAssembly assemblyName="System.ServiceModel" targetFramework="net40" />
</frameworkAssemblies>
```

nuget.exe now is able to store API Key credentials

When using the nuget.exe command line tool, you can now use the SetApiKey command to store your API key. That way, you won't need to specify it every time you push a package. For more details on saving your API key with nuget.exe, [read the documentation on publishing a package](#).

Package Explorer

Package Explorer has been updated to support NuGet 1.2. For more information, check out the [Package Explorer release notes](#).

Other features/fixes

The previous list were the most noticeable of the many features we implemented and bugs we fixed. All in all, we implemented/fixes [59 work items](#) in this release.

Known Issues

- **1.2 Package incompatibility:** Packages built with the latest version of the command line tool, nuget.exe (> 1.2) will not work with older versions of the NuGet VS Add-in (such as 1.1). If you run into an error message stating something about incompatible schema, you are running into this error. Please update NuGet to the latest version.
- **NuGet.Server incompatibility:** If you're hosting an internal NuGet feed using the NuGet.Server project, you'll need to update that project with the latest version of NuGet.Server.
- **Signature Mismatch Error:** If you run into an error during an upgrade with a message about a Signature Mismatch, you need to uninstall NuGet first and then install it. This is listed in our [Known Issues page](#) which provides more details. The issue only affects those running Visual Studio 2010 SP1 and have a version of NuGet 1.0 installed that was incorrectly signed. This version was only made available from the CodePlex website for a brief period so this issue shouldn't affect too many people.

NuGet 1.0 and 1.1 Release Notes

9/4/2018 • 10 minutes to read • [Edit Online](#)

NuGet 1.2 Release Notes

NuGet 1.0 was released on January 13, 2011. NuGet 1.1 was released on February 12, 2011.

Overview

This document contains the release notes for the various releases of NuGet 1.0 grouped according to major preview release.

NuGet includes the following components:

- *NuGet.Tools.vsix* * which consists of:
 - **Add Library Package Dialog** * Dialog within Visual Studio used to browse and install packages.
 - **Package Manager Console** * Powershell based console within Visual Studio.
- *NuGet Command Line Tool* * Tool used to create (and eventually publish) packages.

The NuGet Tools Visual Studio Extension (*NuGet.Tools.vsix*) requires:

- Visual Studio 2010 or Visual Web Developer 2010 Express.

The NuGet Command Line Tool requires:

- .NET Framework Version 4

Installation

To use this [latest release](#):

- First uninstall your older build. You need to run VS as administrator to do this.
- Remove all the existing feeds that you have.
- Add a new feed pointing to <http://go.microsoft.com/fwlink/?LinkId=206669>.

NuGet 1.1

The list of issues fixed in this release [can be found here](#)

NuGet 1.0 RTM

One issue was fixed for RTM since the RC.

- [Issue 474: Removing Packages Affects All Project In Solution](#)

Release Candidate

The following are the changes made in this Release Candidate since CTP 2. Visit the Issue Tracker to see the full list of bugs.

- [Updating Package from Console does not update dependencies.](#)
- [Adding package picks up bin not package reference \(CTP1\)](#)
- [Updating a package leaves broken references](#)

- Get-Package -Updates fails in the dialog, or when the 'All' aggregate source is selected in the console
- Getting package verification errors
- Warn users when a package cannot be installed from the Add Package Dialog
- Get-Package -Updates throws when updating large number of packages
- Improve error handling when nuspec files are authored incorrectly
- NuGet pack ignores specified files
- Removing the second-to-last package source and then clicking "Move Down" crashes VS
- Remove assembly reference while installing packages
- InvalidOperationException when opening Settings dialog
- Access Key for Package Source in Package Manager Console doesn't work
- NuGet VS Settings Dialog Access Keys Give Focus to Wrong Fields
- Package ID intellisense should not query too many items
- Failure adding package to project with a dot character in the Project name
- Issue with specified files in nuspec
- Correct official feed should get registered when using newer build
- Tags should use spaces instead of #
- IPackageMetadata lacks some useful information
- Add Report Abuse Link to the Dialog
- Using App_Data to unzip packages breaks in Visual Studio
- Implement Tags
- PackageBuilder allows empty package with no dependencies to be created
- Add Owners Field for the Package
- Update the VSIX manifest to say NuGet Package Manager rather than VSIX Tools
- Get-Package command throws error when All source is selected
- Allow ordering of package sources in Options dialog
- Update-Package does not remove older version
- Implement Version Range Specification for Dependencies
- Visual Studio crashes when clicking "Add new package"
- Display Downloads and Ratings in the Add Package Dialog
- Changing between package sources in the Dialog doesn't update active source
- Remove Key Binding for Package Manager Console Window
- Install-Package is not recognized as the name of a cmdlet...
- Installing a package from a local feed the dependencies on regular feeds are not resolved
- RemoveDependencies should skip dependencies that are still in use
- If cancelling page navigation, user cannot navigate to a different page while the original page request returns
- Investigate performance of NuPack.Server for serving feeds with large number of packages.
- The second time I filter for a package it uses the "New" package source, instead of the previously selected source.
- Default package source should be selected when selecting the "Online" tab on the dialog.
- List-Package should show installed packages by default
- Assembly Reference HintPaths
- Exception while opening Package Manager Console
- Console intellisense downloads entire feed
- 'Default' package source should be renamed to 'Active'
- Package sources UI: pressing OK should add the new source if Name/Source fields are non-empty
- Dialog becomes super slow when the number of installed packages is large

- Support Binding Redirects for Strong Named Assemblies
- Add Package Reference... UI to include drop down for Package source
- NuPack needs to support config transform agnostically of the config file name
- Allows BasePath to be Overridden in NuPack.exe
- Package Source Fallback Behavior
- Crash on GUI
- Add sorting options to Add Package Dialog
- shortcut key to clear the Package Manager Console
- PowerConsole causes NuPack Console to fail
- Console and Add Package Dialog should set user agent in requests
- Set version number of the VSIX and NuPack.exe in the build.
- Hide common PowerShell parameters from -?
- Add -detailed help for console commands
- Add Package Dialog Should Allow Choosing the Current Package Source
- Move NuPack.Core classes into different namespaces
- Add help to cmdlets
- Verify hash from feed after package download

CTP 2

The following are the most significant changes made in CTP 2:

- Switched the package feed from ATOM to an OData service endpoint: If you upgrade to the CTP2 version of NuGet, be sure to add the following URL as a package source: <https://feed.nuget.org/ctp2/odata/v1/>.
- Renamed the Add-Package command to *Install-Package*.
- Updated the `.nuspec` Format. The `.nuspec` format now includes the *iconUrl* field for specifying a 32x32 png icon which will show up in the Add Package Dialog. So be sure to set that to distinguish your package. The `.nuspec` format also includes the new *projectUrl* field which you can use to point to a web page that provides more information about your package.

This build will not work with old `.nupkg` files. If you get null reference exceptions, you're using an old `.nupkg` file and need to rebuild it with the updated [NuGet command line tool](#).

The following is a list of features and bugs that were fixed for NuGet CTP 2 (does not include bugs for minor code cleanups etc.).

- [Error unpacking package assemblies when specifying the TargetFramework for an assembly.](#)
- [Make NuPack Console window more discoverable](#)
- [ILMerge the nupack.exe release](#)
- [Better error/exception handling](#)
- [\[Nupack.Core\]: PackageManager should gracefully handle feed-related errors](#)
- [Need a new icon for the console](#)
- [Localize strings in the Dialog](#)
- [NuPack caches downloaded .nupack files in memory](#)
- [NuPack Console: Change the default shortcut for displaying console](#)
- [ProjectSystem should support default values for common properties](#)
- [Running nupack.exe in a folder with just one nuspec file should use that nuspec](#)
- [Project Menu Shows Up Even When No Project/Solution Is Loaded](#)
- [build.cmd fails on a clean clone of the codebase](#)
- [Updates available feature](#)

- Dialog: Adding a package through the dialog removes the prompt in the console
- Adding a package by clicking 'Install' is often slow, with no visual feedback
- There is no way to discover which of my installed packages have updates.
- There is no way to update an installed package in the dialog.
- There is no way to uninstall an installed package in the dialog
- "Add Package Reference..." appears on the context menu of installed references
- After updating a package from the console, it shows both the old version and the new version as installed
- The activity in the console, when using the dialog, disappears after use
- Cleanup command line parsing in nupack.exe
- Add a friendly name to package sources
- Update .nuspec to support including package icons
- Feed UI doesn't allow copying the URL
- Better remove-package error handling.
- Typing in Console Window depends on cursor focus
- Error messages look awful
- The performance of Remove-Package for a package that isn't installed is bad
- Removing a package fails when there are no package sources
- Remove-Package fails when the package source is unavailable
- Add Title to the package metadata and the feed.
- Add the -Source parameter back to Add-Package
- List-Package should have a -Source parameter
- Update NuPack.Server to require NuPack User Agent To Download Package
- License Acceptance Dialog Must List Licenses For All Dependencies That Require Acceptance
- Log an error when a package throws in the feed
- NuPack.exe should not allow an empty <licenseurl> element
- Rename List-Package to Get-Package, Add-Package to Install-Package, and Remove-Package to Uninstall-Package
- Using the Add Package Reference menu item from the Solution Navigator crashes Visual Studio
- "Available package sources" label is missing a colon
- Make .nuspec xml element casing consistently camel cased
- The NuPack VSIX's manifest needs to turn on the 'admin' bit
- If you run List-Package with no feeds, you get null ref error
- nuget.exe: specify destination path
- Powershell Errors Opening Package Management Console on WinXP
- VS Crashes while trying to load package list
- allow meta packages (no files, only dependencies)
- Convert Powershell Script to Powershell 2.0 Module
- PathResolver should discard path portion preceding wildcard characters when target is specified
- No dependencies
- Error installing Elmah
- Config transforms don't work correctly with <configsections>
- The variable '\$global:projectCache' cannot be retrieved because it has not been set
- Add MSBuild task for creating NuPack packages
- list-package needs to support searching/filtering
- Always display a link to license if the package author provides a license URL
- Occasional "Access Denied" exception with Remove-Package

- Unit Tests Failing: InvalidPackageIsExcludedFromFeedItems & CreatingFeedConvertsPackagesToAtomEntries
- Allow for a fallback/default set of files if a specific framework version cannot be found
- Add Package Reference... UI cannot remove a package
- Add Package Reference crashes studio when one or more project is unloaded
- Config transform does not appear to work on web.debug.config file
- init.ps1 not firing on custom package
- When adding paths to the feedlist, the default button is set to OK, so if I press ENTER it automatically closes
- Attempt to uninstall a dependency will crash VS if attempted 2 times in a row
- Display the Project URL in the Add Package dialog
- Default the Add-Package dialog to Installed Packages
- Change Add Package Dialog menu item.
- Rename namespaces and assemblies
- Rename the NuPack Project to NuGet
- Add the following text under the list of dependencies
- Change the license acceptance text in the License Acceptance Dialog
- Change the text in the License Acceptance Dialog above the list of packages
- OData doesn't work with an fwlink URL
- Package Manager UI: Over aggressive caching of package count used for paging
- NuPack / NuGet -> Package Manager Console error
- Add Package Dialog shows License Acceptance For Already Installed Packaged

CTP 1

The following is a list of features and bugs that were fixed for NuGet CTP 1.

- Package extension should be renamed to .nupack
- Move package file into folder
- Merge install & Add PS commands
- Create aliases for Verb-Noun cmdlets
- NuPack gets confused when switching solution in VS
- We should hide the 'packages' solution folder by default
- Add support for token replacement in content items.
- NuPack.UI should use the PackageSource API
- [Nupack.Core]: PackageManager marks packages as installed prior to installing them
- Deleting default project from solution still shows the deleted project as default
- New-Package fails with "Cannot add part for the specified URI because it's already in the package."
- Remove "NuPack" strings from Visual Studio GUI
- Add Apache Header To a COPYRIGHT.txt file
- Remove Update-PackageSource Command
- Package Manager unusable when loading profile throws an exception
- init.ps1, install.ps1 and uninstall.ps1 need to receive additional state
- Combine Console and GUI Packages Into One Package
- Xml transform logic doesn't work if applied to XML that isn't at the root
- Manage package sources settings dialog not updating the NuPack console
- NuPack Console UI: Rename 'Package feed' drop-down list to 'Package source'
- NuPack Console Options: Rename 'Repository UI' to be consistent with NuPack Console
- Add-Package fails against a website that was opened from IIS or a URL
- Package Manager Source Doesn't Work With FwLink

- Set the default package source
- When adding package sources in option, when only one source is supplied, assume it's the default.
- The Dialog UI shows fake "recent" packages
- Options: Clicking cancel does not cancel changes
- Add Package Reference Dialog Search should be case insensitive
- Fix company metadata in AssemblyInfo.cs files
- Version number for the VSIX
- Remove-Package: Using -? displays help twice
- Execute install/uninstall packages for project level packages
- Server unable to create feed when one nupack fails validation
- Need to Replace NuPack Icons
- NTLM http proxy does not authenticate to the package feed.
- The dialog doesn't always start centered in the VS window
- Many of the fields in a packages details are not being populated in the dialog
- Dialog UI doesn't show Authors' names
- Why -Version for Remove-Package
- Remove the Recent tab on the Dialog UI
- VS crash when right click on solution folder after opening Dialog UI at least one.
- Change the -Local parameter of List-Package to -Installed
- Rename packages.xml to NuPack.config
- Console forces cursor to the end of line
- Remove-Package intellisense is broken
- Add RequireLicenseAcceptance Flag to .nuspec and Feed
- Add LicenseUrl to .nuspec Format and Package Feed
- Clicking Install For Package That Requires Acceptance Should Show Acceptance Dialog
- Add Disclaimer Text to the Add Package Dialog
- Add Disclaimer When the Package Console is run the first time
- Display Disclaimer After Installing Package In The Console
- Rename the .nupack extension to .nupkg

NuGet frequently-asked questions

8/23/2019 • 5 minutes to read • [Edit Online](#)

For frequently-asked questions pertaining to NuGet.org, such as NuGet.org account questions, see [NuGet.org frequently-asked questions](#).

What is required to run NuGet?

All the information around both UI and command-line tools is available in the [Install guide](#).

Does NuGet support Mono?

The command-line tool, `nuget.exe`, builds and runs under Mono 3.2+ and can create packages in Mono.

Although `nuget.exe` works fully on Windows, there are known issues on Linux and OS X. Refer to [Mono issues](#) on GitHub.

A [graphical client](#) is available as an add-in for MonoDevelop.

How can I determine what a package contains and whether it's stable and useful for my application?

The primary source for learning about a package is its listing page on nuget.org (or another private feed). Each package page on nuget.org includes a description of the package, its version history, and usage statistics. The **Info** section on the package page also contains a link to the project's web site where you typically find many examples and other documentation to help you learn how the package is used.

For more information, see [Finding and choosing packages](#).

NuGet in Visual Studio

How is NuGet supported in different Visual Studio products?

- Visual Studio on Windows supports the [Package Manager UI](#) and the [Package Manager Console](#).
- Visual Studio for Mac has built-in NuGet capabilities as described on [Including a NuGet package in your project](#).
- Visual Studio Code (all platforms) does not have any direct NuGet integration. Use the [NuGet CLI](#) or the [dotnet CLI](#).
- Azure DevOps provides [a build step to restore NuGet packages](#). You can also [host private NuGet package feeds on Azure DevOps](#).

How do I check the exact version of the NuGet tools that are installed?

In Visual Studio, use the **Help > About Microsoft Visual Studio** command and look at the version displayed next to **NuGet Package Manager**.

Alternatively, launch the Package Manager Console (**Tools > NuGet Package Manager > Package Manager Console**) and enter `$host` to see information about NuGet including the version.

What programming languages are supported by NuGet?

NuGet generally works for .NET languages and is designed to bring .NET libraries into a project. Because it also supports MSBuild and Visual Studio automation in some project types, it also supports other projects and languages to various degrees.

The most recent version of NuGet supports C#, Visual Basic, F#, WiX, and C++.

What project templates are supported by NuGet?

NuGet has full support for a variety of project templates like Windows, Web, Cloud, SharePoint, Wix, and so on.

How do I update packages that are part of Visual Studio templates?

Go to the **Updates** tab in the Package Manager UI and select **Update All**, or use the `Update-Package` command from the Package Manager Console.

To update the template itself, you need to manually update the template repository. See [Xavier Decoster's blog](#) on this subject. Note that this is done at your own risk, because manual updates might corrupt the template if the latest version of all dependencies are not compatible with each other.

Can I use NuGet outside of Visual Studio?

Yes, NuGet works directly from the command line. See the [Install guide](#) and the [CLI reference](#).

NuGet command line

How do I get the latest version of NuGet command line tool?

See the [Install guide](#). To check the current installed version of the tool, use `nuget help`.

What is the license for nuget.exe?

You are allowed to redistribute nuget.exe under the terms of the MIT license. You are responsible for updating and servicing any copies of nuget.exe that you choose to redistribute.

Is it possible to extend the NuGet command line tool?

Yes, it's possible to add custom commands to `nuget.exe`, as described in [Rob Reynold's post](#).

NuGet Package Manager Console (Visual Studio on Windows)

How do I get access to the DTE object in the Package Manager console?

The top-level object in the Visual Studio automation object model is called the DTE (Development Tools Environment) object. The console provides this through a variable named `$DTE`. For more information, see [Automation Model Overview](#) in the Visual Studio Extensibility documentation.

I try to cast the \$DTE variable to the type DTE2, but I get an error: Cannot convert the "EnvDTE.DTEClass" value of type "EnvDTE.DTEClass" to type "EnvDTE80.DTE2". What's wrong?

This is a known issue with how PowerShell interacts with a COM object. Try the following:

```
`$dte2 = Get-Interface $dte ([EnvDTE80.DTE2])`
```

`Get-Interface` is a helper function added by the NuGet PowerShell host.

Creating and publishing packages

How do I list my package in a feed?

See [Creating and publishing a package](#).

I have multiple versions of my library that target different versions of the .NET Framework. How do I build a single package that supports this?

See [Supporting Multiple .NET Framework Versions and Profiles](#).

How do I set up my own repository or feed?

See the [Hosting packages overview](#).

How can I upload packages to my NuGet feed in bulk?

See [Bulk publishing NuGet packages](#) (jeffhandly.com).

Working with packages

What is the difference between a project-level package and a solution-level package?

A solution-level package (NuGet 3.x+) is installed only once in a solution and is then available for all projects in the solution. A project-level package is installed in each project that uses it. A solution-level package might also install new commands that can be called from within the Package Manager Console.

Is it possible to install NuGet packages without Internet connectivity?

Yes, see Scott Hanselman's Blog post [How to access NuGet when nuget.org is down \(or you're on a plane\)](#) (hanselman.com).

How do I install packages in a different location from the default packages folder?

Set the `repositoryPath` setting in `Nuget.Config` using `nuget config -set repositoryPath=<path>`.

How do I avoid adding the NuGet packages folder into to source control?

Set the `disableSourceControlIntegration` in `Nuget.Config` to `true`. This key works at the solution level and hence need to be added to the `$(solutiondir)\.nuget\Nuget.Config` file. Enabling package restore from Visual Studio creates this file automatically.

How do I turn off package restore?

See [Enabling and disabling package restore](#).

Why do I get an "Unable to resolve dependency error" when installing a local package with remote dependencies?

You need to select the **All** source when installing a local package into the project. This aggregates all the feeds instead of using just one. The reason this error appears is that users of a local repository often want to avoid accidentally installing a remote package due to corporate policies.

I have multiple projects in the same folder, how can I use separate packages.config files for each project?

In most projects where separate projects live in separate folders, this is not a problem as NuGet identifies the `packages.config` files in each project. With NuGet 3.3+ and multiple projects in the same folder, you can insert the name of the project into the `packages.config` filenames use the pattern `packages.{project-name}.config`, and NuGet uses that file.

This is not an issue when using `PackageReference`, as each project file contains its own list of dependencies.

I don't see nuget.org in my list of repositories, how do I get it back?

- Add `https://api.nuget.org/v3/index.json` to your list of sources, or
- Delete `%appdata%\nuget\NuGet.Config` (Windows) or `~/.nuget/NuGet/NuGet.Config` (Mac/Linux) and let NuGet re-create it.

Identify the project format

8/15/2019 • 2 minutes to read • [Edit Online](#)

NuGet works with all .NET projects. However, the project format (SDK-style or non-SDK-style) determines some of the tools and methods that you need to use to consume and create NuGet packages. SDK-style projects use the [SDK attribute](#). It is important to identify your project type because the methods and tools you use to consume and create NuGet packages are dependent on the project format. For non-SDK-style projects, the methods and tools are also dependent on whether or not the project has been migrated to [PackageReference](#) format.

Whether your project is SDK-style or not depends on the method used to create the project. The following table shows the default project format and the associated CLI tool for your project when you create it using Visual Studio 2017 and later versions.

PROJECT	DEFAULT PROJECT FORMAT	CLI TOOL	NOTES
.NET Standard	SDK-style	dotnet CLI	Projects created prior to Visual Studio 2017 are non-SDK-style. Use nuget.exe CLI.
.NET Core	SDK-style	dotnet CLI	Projects created prior to Visual Studio 2017 are non-SDK-style. Use nuget.exe CLI.
.NET Framework	Non-SDK-style	nuget.exe CLI	.NET Framework projects created using other methods may be SDK-style projects. For these, use dotnet CLI instead.
Migrated .NET project	Non-SDK-style	To create packages, use msbuild -t:pack to create packages.	To create packages, msbuild -t:pack is recommended. Otherwise, use the dotnet CLI . Migrated projects are not SDK-style projects.

Check the project format

If you're unsure whether the project is SDK-style format or not, look for the [SDK attribute](#) in the [`<Project>`](#) element in the project file (For C#, this is the `*.csproj` file). If it is present, the project is an SDK-style project.

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
  <Authors>authorname</Authors>
  <PackageId>mypackageid</PackageId>
  <Company>mycompanyname</Company>
</PropertyGroup>

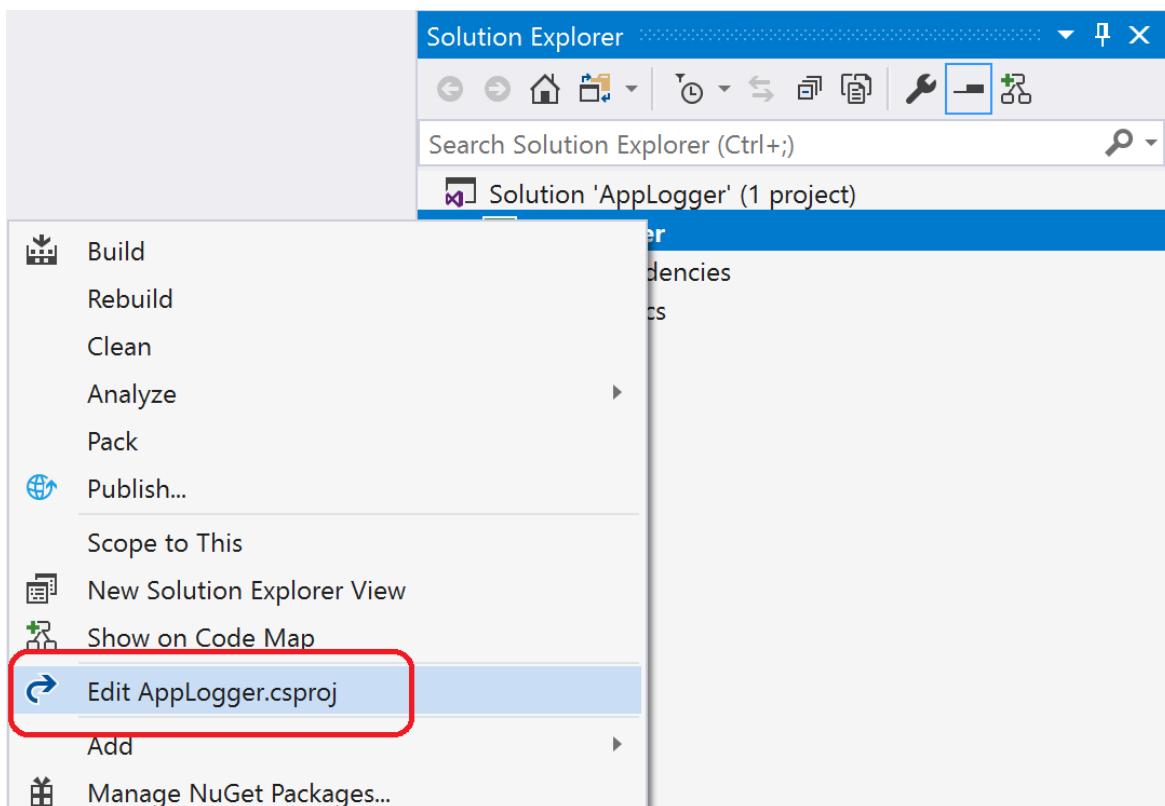
</Project>
```

Check the project format in Visual Studio

If you are working in Visual Studio, you can quickly check the project format using one of the following methods:

- Right-click the project in Solution Explorer and select **Edit myprojectname.csproj**.

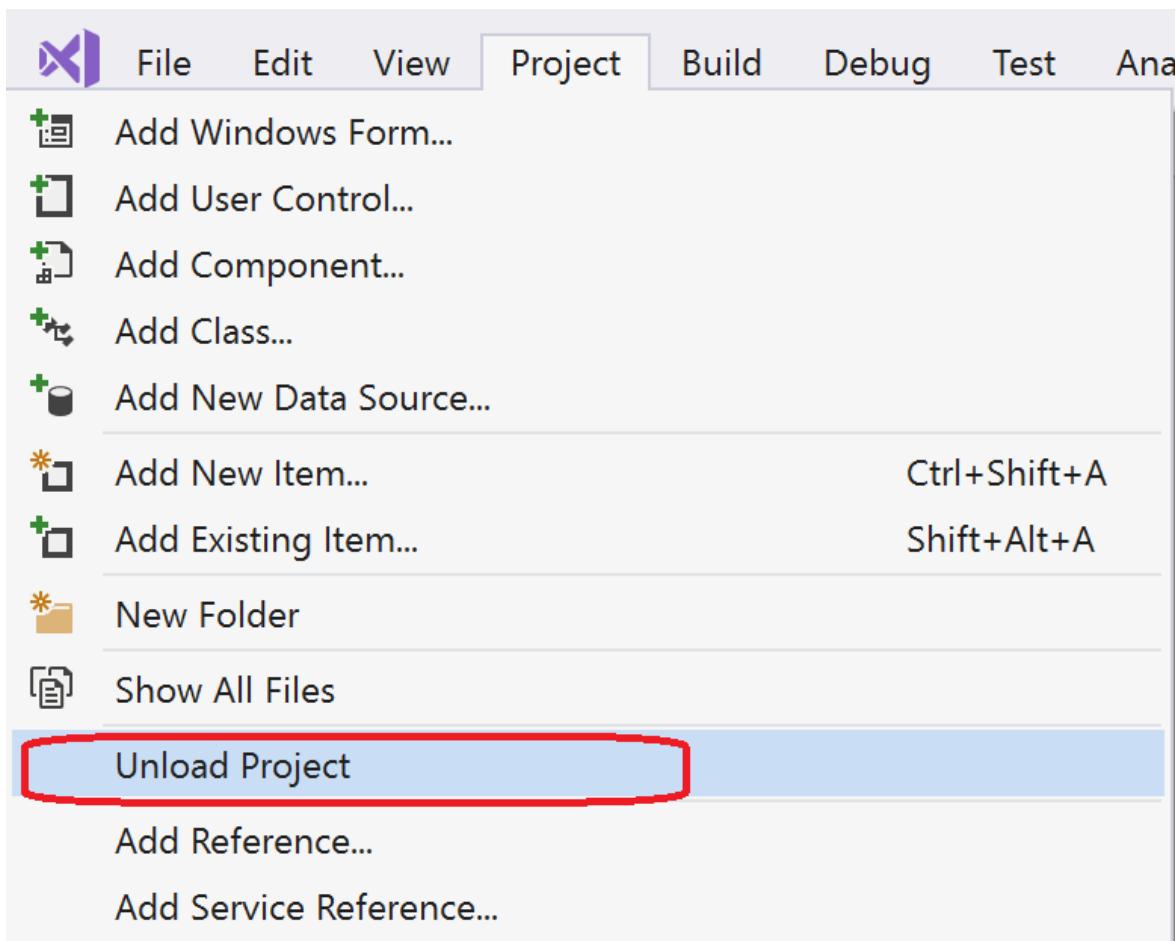
This option is only available starting in Visual Studio 2017 for projects that use the SDK-style attribute. Otherwise, use the other method.



An SDK-style project shows the [SDK attribute](#) in the project file.

- From the **Project** menu, choose **Unload Project** (or right-click the project and choose **Unload Project**).

This project will not include the SDK attribute in the project file. It is not an SDK-style project.



Then, right-click the unloaded project and choose **Edit myprojectname.csproj**.

See also

- [Create .NET Standard Packages with dotnet CLI](#)
- [Create .NET Standard Packages with Visual Studio](#)
- [Create and publish a .NET Framework package \(Visual Studio\)](#)
- [NuGet pack and restore as MSBuild targets](#)

Overview of NuGet.org

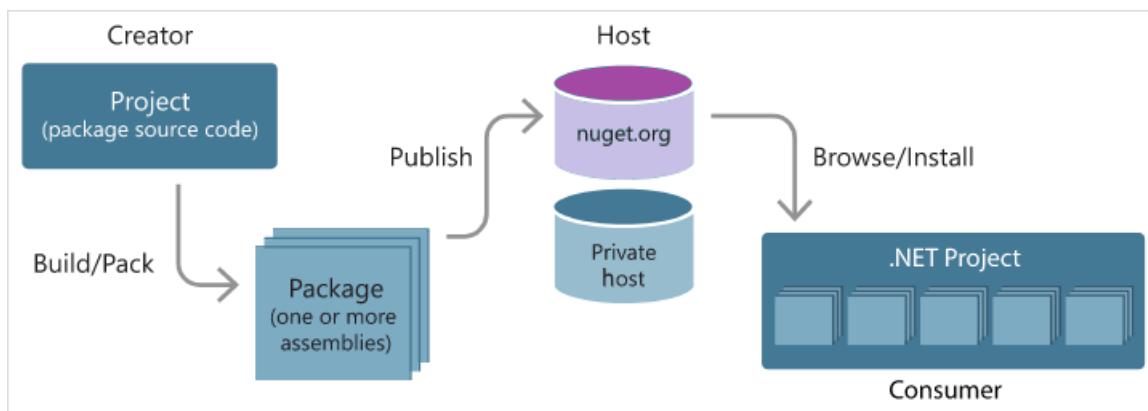
6/28/2019 • 2 minutes to read • [Edit Online](#)

NuGet.org is a public host of NuGet packages that are employed by millions of .NET and .NET Core developers every day.

Role of NuGet.org in the NuGet ecosystem

In its role as a public host, NuGet.org itself maintains the central repository of over 100,000 unique packages at nuget.org. NuGet.org is not the only possible host for packages. The NuGet technology also enables you to host packages privately in the cloud (such as on Azure DevOps), on a private network, or even on just your local file system. If you are interested in a different host or hosting option, see [Hosting your own NuGet feeds](#).

NuGet.org, like any host for NuGet packages, serves as the point of connection between package *creators* and package *consumers*. Creators build useful NuGet packages and publish them. Consumers then search for useful and compatible packages on accessible hosts, downloading and including those packages in their projects. Once installed in a project, the packages' APIs are available to the rest of the project code.



Accounts

To publish packages on NuGet.org, you first create an [individual \(user\) account](#). This becomes your identity on NuGet.org.

NuGet.org also allows you to create an [organization account](#). An organization account has one or more individual accounts as its members. Members can manage a set of packages while maintaining a single identity for ownership. Through your individual account, you can be a member of any number of organizations.

A package can belong to an organization account like it can belong to an individual account. Package consumers don't see any difference between an individual account or the organization account: both appear as package [owners](#).

API keys

Once you have a NuGet package (`.nupkg` file) to publish, you publish it to NuGet.org using either the `nuget.exe` CLI or the `dotnet.exe` CLI, along with an [API key](#) acquired from NuGet.org.

When you [publish a package](#), you include the API key value in the CLI command.

ID prefixes

When you publish packages, you can reserve and protect your identity by [reserving ID prefixes](#). When installing a package, package consumers are provided with additional information indicating that the package they are consuming is not deceptive in its identifying properties.

API endpoint for NuGet.org

To use NuGet.org as a package repository with NuGet clients, you should use the following V3 API endpoint:

`https://api.nuget.org/v3/index.json`

Older clients can still use the V2 protocol to reach NuGet.org. However, please note, NuGet clients 3.0 or later will have slower and less reliable service using the V2 protocol:

`https://www.nuget.org/api/v2` (**The V2 protocol is deprecated!**)