




.Net / NuGet - Dependency Management Research [April 2017]

This document contains a brief overview of NuGet flow, issues, practices etc. As these concept are wide and complex, not all details are within this text. For more information, please review the official online documentation. Also, please see the corresponding  [POC](#)

1. Nuget Basics

NuGet is a free and open-source package manager designed for the Microsoft development platforms.

It is distributed as a Visual Studio extension and can also be used from the CLI.

[Finding](#) and evaluating a package is mostly done through the public collection on [nuget.org](https://www.nuget.org), where details such as compatibility and popularity are found.

NuGet remembers the identity and version number of each installed package, recording it in either `packages.config` or `project.json` in your project root, depending on project type. [[installing](#), [caching](#), [configuration](#)]

2. Managing Requirements

1. NuGet 2.X

With NuGet 2.x, a project's dependencies are written to the `packages.config` file as a flat list.

2. NuGet 3.x

With NuGet 2.x, a project's dependencies are written to the `project.json` file as a flat list.

[\[Converting package.config to project.json\]](#)

3. Dependency Resolution

1. NuGet 2.X

NuGet 2.x will attempt to resolve dependency conflicts during the installation of each individual package, which includes package restore. 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 will compare the versions of

Package B being requested and attempt to find a version that will satisfy all version constraints. Specifically, NuGet will select the lower major.minor version that satisfies dependencies.

By default, NuGet 2.7 and earlier resolves the highest patch version (using the major.minor.patch.build convention). [NuGet 2.8](#) changes this behavior to look for the lowest patch version by default, and allows you to control this setting through the `DependencyVersion` attribute in `Nuget.Config` and the `-DependencyVersion` switch on the command line.

On the downside, the NuGet 2.x process for resolving dependencies gets complicated for larger dependency graphs. This is especially true during package restore, because each new package installation requires a traversal of the whole graph and raises the chance for version conflicts. When a conflict occurs, package restoration is stopped, leaving the project in a partially-restored state, especially with potential modifications to the project file itself.

This and other challenges is why dependency resolution was overhauled in NuGet 3.x, as described in the next section.

2. NuGet 3.x

As dependencies are installed into a project, NuGet 3.x adds them to a flat package graph in `project.json` in which conflicts are resolved ahead of time. This is referred to as transitive restore. Reinstalling or restoring packages is then simply a process of downloading the packages listed in the graph, resulting in faster and more predictable builds. This allows developers explicitly declare which package versions they depend on, without worrying about their down-level dependencies. [\[dependency resolution\]](#)

Dependency Resolution Rules

NuGet 3.x applies four main rules to resolve dependencies: lowest applicable version, floating versions, nearest-wins, cousin dependencies.

- **Lowest Applicable Version**

NuGet 3.x restores the lowest possible version of a package as defined by its dependencies. This rule also applied to dependencies on the application or the class library unless declared as floating.

- **Floating Versions**

When a floating version constraint is specified then NuGet will resolve the highest version of a package that matches the version pattern, for example `6.0.*` will get the highest version of a package that starts with `6.0`.

- **Nearest Wins**

When the package graph for an application contains different versions of the same package, the package that's closest to the application in the graph will be used and others will be ignored. This allows an application to override any particular package version in the dependency graph.

[Applying the Nearest Wins rule can [downgrading](#) the package version]

- **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.

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. This can be solved using the exclude attribute. [\[excluding references\]](#)

Lock File and MSBuild

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.lock.json` alongside `project.json`. 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 lock file is temporary and does not need to be added to source control; it's listed by default in both `.gitignore` and `.tfignore`.

Package References

Package references, using the `PackageReference` node, allow you to manage NuGet dependencies directly in .NET Core and .NET Standard project files, without needing a separate `packages.config` or `project.json` file. This approach also allows you to use MSBuild conditions to choose package references per target framework, configuration, platform, or other groupings. It also allows for fine-grained control over dependencies and content flow. In terms of behavior and dependency resolution, it is the same as using `project.json`.

Package references are currently supported for only .NET Core and .NET Standard projects in Visual Studio 2017.