

**ANG-450** *POC Document*

|  |  |
| --- | --- |
| POC Title: | **.NET template & nuget package** |
| Created by: | Md. Sahidul Islam, Jr. Software Engineer |
| Reviewed by: | Md. Mahedee Hasan, Head of Soft. Dev Eng |
| Approved by: | Md. Mahedee Hasan, Head of Soft. Dev Eng |
| Confidentiality level: | Internal |

Page Break

**DOCUMENT CHANGE HISTORY**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Author** | **Description of change** |
| 21-Sept-2022 | 1.0 | Md. Sahidul Islam | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[.NET Item Template 1](#_Toc114644829)

[Prerequisites 1](#_Toc114644830)

[Create the required folders 2](#_Toc114644831)

[Create an item template 2](#_Toc114644832)

[Create the template config 3](#_Toc114644833)

[Test the item template 4](#_Toc114644834)

[Uninstall the template 5](#_Toc114644835)

[.NET Project Template 6](#_Toc114644836)

[Create a project template 6](#_Toc114644837)

[Modify Program.cs 6](#_Toc114644838)

[Build the project 6](#_Toc114644839)

[Create the template config 7](#_Toc114644840)

[Test the project template 8](#_Toc114644841)

[Uninstall the template 9](#_Toc114644842)

[Create a template package 9](#_Toc114644843)

[Create a template package project 9](#_Toc114644844)

[Build and install 12](#_Toc114644845)

[Uninstall the template package 13](#_Toc114644846)

[Creating a library package 14](#_Toc114644847)

[Prerequisites 14](#_Toc114644848)

[Create a class library project 14](#_Toc114644849)

[Add package metadata to the project file 14](#_Toc114644850)

[Run the pack command 15](#_Toc114644851)

[Automatically generate package on build 16](#_Toc114644852)

[Publish the package 16](#_Toc114644853)

[Acquire your API key 16](#_Toc114644854)

[Publish with dotnet nuget push 17](#_Toc114644855)

[Publish errors 17](#_Toc114644856)

[Manage the published package 18](#_Toc114644857)

[Hosting your own NuGet feeds 19](#_Toc114644858)

[Gitlab Package Registry: 20](#_Toc114644859)

[Add the Package Registry as a source for NuGet packages 21](#_Toc114644860)

[Add a source with the NuGet CLI 22](#_Toc114644861)

[Add a source with Visual Studio 22](#_Toc114644862)

[Add a source with the .NET CLI 27](#_Toc114644863)

[Add a source with a configuration file 28](#_Toc114644864)

[Publish a NuGet package 30](#_Toc114644865)

[Publish a package with the NuGet CLI 30](#_Toc114644866)

[Publish a package with the .NET CLI 30](#_Toc114644867)

[Publish a NuGet package by using CI/CD 30](#_Toc114644868)

[Publishing a package with the same name or version 31](#_Toc114644869)

[Install packages 32](#_Toc114644870)

[Install a package with the NuGet CLI 32](#_Toc114644871)

[Install a package with the .NET CLI 32](#_Toc114644872)

**.NET template & nuget package**

## .NET Item Template

## Prerequisites

* [.NET 6.0 SDK](https://dotnet.microsoft.com/download) or a later version.
* Read the reference article [Custom templates for dotnet new](https://docs.microsoft.com/en-us/dotnet/core/tools/custom-templates).

The reference article explains the basics about templates and how they're put together. Some of this information will be reiterated here.

* Open a terminal and navigate to the working\templates folder.

## Create the required folders

This series uses a "working folder" where your template source is contained and a "testing folder" used to test your templates. The working folder and testing folder should be under the same parent folder.

First, create the parent folder, the name does not matter. Then, create a subfolder named working. Inside of the working folder, create a subfolder named templates.

Next, create a folder under the parent folder named test. The folder structure should look like the following.

parent\_folder

├───test

└───working

└───templates

## Create an item template

An item template is a specific type of template that contains one or more files. These types of templates are useful when you want to generate something like a config, code, or solution file. In this example, you'll create a class that adds an extension method to the string type.

In your terminal, navigate to the working\templates folder and create a new subfolder named extensions. Enter the folder.

working

└───templates

└───extensions

Create a new file named CommonExtensions.cs and open it with your favorite text editor. This class will provide an extension method named Reverse that reverses the contents of a string. Paste in the following code and save the file:

using System;

namespace System

{

public static class StringExtensions

{

public static string Reverse(this string value)

{

var tempArray = value.ToCharArray();

Array.Reverse(tempArray);

return new string(tempArray);

}

}

}

Now that you have the content of the template created, you need to create the template config at the root folder of the template.

## Create the template config

Templates are recognized by a special folder and config file that exist at the root of your template. In this tutorial, your template folder is located at working\templates\extensions.

When you create a template, all files and folders in the template folder are included as part of the template except for the special config folder. This config folder is named .template.config.

First, create a new subfolder named .template.config, enter it. Then, create a new file named template.json. Your folder structure should look like this:

working

└───templates

└───extensions

└───.template.config

template.json

Open the template.json with your favorite text editor and paste in the following JSON code and save it.

{

"$schema": "http://json.schemastore.org/template",

"author": "Me",

"classifications": [ "Common", "Code" ],

"identity": "ExampleTemplate.StringExtensions",

"name": "Example templates: string extensions",

"shortName": "stringext",

"tags": {

"language": "C#",

"type": "item"

}

}

This config file contains all the settings for your template. You can see the basic settings, such as name and shortName, but there's also a tags/type value that is set to item. This categorizes your template as an item template. There's no restriction on the type of template you create. The item and project values are common names that .NET recommends so that users can easily filter the type of template they're searching for.

The classifications item represents the **tags** column you see when you run dotnet new and get a list of templates. Users can also search based on classification tags. Don't confuse the tags property in the \*.json file with the classifications tags list. They're two different things unfortunately named similarly. The full schema for the template.json file is found at the [JSON Schema Store](http://json.schemastore.org/template). For more information about the template.json file, see the [dotnet templating wiki](https://github.com/dotnet/templating/wiki).

Now that you have a valid .template.config/template.json file, your template is ready to be installed. In your terminal, navigate to the extensions folder and run the following command to install the template located at the current folder:

* **On Windows**: dotnet new --install .\
* **On Linux or macOS**: dotnet new --install ./

This command outputs the list of templates installed, which should include yours.

The following template packages will be installed:

<root path>\working\templates\extensions

Success: <root path>\working\templates\extensions installed the following templates:

Templates Short Name Language Tags

-------------------------------------------- ------------------- --------

Example templates: string extensions stringext [C#] Common/Code

## Test the item template

Now that you have an item template installed, test it. Navigate to the test/ folder and create a new console application with dotnet new console. This generates a working project you can easily test with the dotnet run command. dotnet new console

You get output similar to the following.

The template "Console Application" was created successfully.

Processing post-creation actions...

Running 'dotnet restore' on C:\test\test.csproj...

Restore completed in 54.82 ms for C:\test\test.csproj.

Restore succeeded.

Run the project with dotnet run, You get the following output.

Hello World!

Next, run dotnet new stringext to generate the CommonExtensions.cs from the template. dotnet new stringext

You get the following output.

The template "Example templates: string extensions" was created successfully.

Change the code in Program.cs to reverse the "Hello World" string with the extension method provided by the template.

Console.WriteLine("Hello World!".Reverse());

Run the program again and you'll see that the result is reversed.

dotnet run

You get the following output.

!dlroW olleH

Congratulations! You created and deployed an item template with .NET. In preparation for the next part of this tutorial series, you must uninstall the template you created. Make sure to delete all files from the test folder too. This will get you back to a clean state ready for the next major section of this tutorial.

## Uninstall the template

In your terminal, navigate to the extensions folder and run the following command to uninstall the template located at the current folder:

* **On Windows**: dotnet new --uninstall .\
* **On Linux or macOS**: dotnet new --uninstall ./

This command outputs a list of the templates that were uninstalled, which should include yours.

Success: <root path>\working\templates\extensions was uninstalled.

At any time, you can use dotnet new --uninstall to see a list of installed template packages, including for each template package the command to uninstall it.

## .NET Project Template

## Create a project template

Project templates produce ready-to-run projects that make it easy for users to start with a working set of code. .NET includes a few project templates such as a console application or a class library. In this example, you'll create a new console project that replaces the standard "Hello World" console output with one that runs asynchronously.

In your terminal, navigate to the working\templates folder and create a new subfolder named consoleasync. Enter the subfolder and run dotnet new console to generate the standard console application. You'll be editing the files produced by this template to create a new template.

working

└───templates

└───consoleasync

consoleasync.csproj

Program.cs

## Modify Program.cs

Open up the program.cs file. The standard console project doesn't asynchronously write to the console output, so let's add that. Change the code to the following and save the file:

await Console.Out.WriteAsync("Hello World with C#");

## Build the project

Before you complete a project template, you should test it to make sure it compiles and runs correctly.

In your terminal, run the following command. dotnet run

You get the following output.

Hello World with C#

You can delete the obj and bin folders created by using dotnet run. Deleting these files ensures your template only includes the files related to your template and not any files that result from a build action.

Now that you have the content of the template created, you need to create the template config at the root folder of the template.

## Create the template config

Templates are recognized in .NET by a special folder and config file that exist at the root of your template. In this tutorial, your template folder is located at working\templates\consoleasync.

When you create a template, all files and folders in the template folder are included as part of the template except for the special config folder. This config folder is named .template.config.

First, create a new subfolder named .template.config, enter it. Then, create a new file named template.json. Your folder structure should look like this.

ConsoleCopy

working

└───templates

└───consoleasync

└───.template.config

template.json

Open the template.json with your favorite text editor and paste in the following json code and save it.

{

"$schema": "http://json.schemastore.org/template",

"author": "Me",

"classifications": [ "Common", "Console" ],

"identity": "ExampleTemplate.AsyncProject",

"name": "Example templates: async project",

"shortName": "consoleasync",

"tags": {

"language": "C#",

"type": "project"

}

}

This config file contains all of the settings for your template. You can see the basic settings such as name and shortName but also there's a tags/type value that's set to project. This designates your template as a project template. There's no restriction on the type of template you create. The item and project values are common names that .NET recommends so that users can easily filter the type of template they're searching for.

The classifications item represents the **tags** column you see when you run dotnet new and get a list of templates. Users can also search based on classification tags. Don't confuse the tags property in the json file with the classifications tags list. They're two different things unfortunately named similarly. The full schema for the template.json file is found at the [JSON Schema Store](http://json.schemastore.org/template). For more information about the template.json file, see the [dotnet templating wiki](https://github.com/dotnet/templating/wiki).

Now that you have a valid .template.config/template.json file, your template is ready to be installed. Before you install the template, make sure that you delete any extra folders and files you don't want included in your template, like the bin or obj folders. In your terminal, navigate to the consoleasync folder and run dotnet new --install .\ to install the template located at the current folder. If you're using a Linux or macOS operating system, use a forward slash: dotnet new --install ./.

dotnet new --install .\

This command outputs a list of the installed templates, which should include yours.

The following template packages will be installed:

<root path>\working\templates\consoleasync

Success: <root path>\working\templates\consoleasync installed the following templates:

Templates Short Name Language Tags

-------------------------------------------- ------------------- ------------ ----------------------

Example templates: async project consoleasync [C#] Common/Console

### Test the project template

Now that you have a project template installed, test it.

1. Navigate to the test folder
2. Create a new console application with the following command which generates a working project you can easily test with the dotnet run command.

dotnet new consoleasync

You get the following output.

The template "Example templates: async project" was created successfully.

1. Run the project using the following command.

dotnet run

You get the following output.

Hello World with C#

Congratulations! You created and deployed a project template with .NET. In preparation for the next part of this tutorial series, you must uninstall the template you created. Make sure to delete all files from the test folder too. This will get you back to a clean state ready for the next major section of this tutorial.

### Uninstall the template

In your terminal, navigate to the consoleasync folder and run the following command to uninstall the template located in the current folder:

* **On Windows**: dotnet new --uninstall .\
* **On Linux or macOS**: dotnet new --uninstall ./

This command outputs a list of the templates that were uninstalled, which should include yours.

Success: <root path>\working\templates\consoleasync was uninstalled.

At any time, you can use dotnet new --uninstall to see a list of installed template packages, including for each template package the command to uninstall it.

## Create a template package

## Create a template package project

A template package is one or more templates packaged into a NuGet package. When you install or uninstall a template package, all templates contained in the package are added or removed, respectively. The previous parts of this tutorial series only worked with individual templates. To share a non-packed template, you have to copy the template folder and install via that folder. Because a template package can have more than one template in it, and is a single file, sharing is easier.

Template packages are represented by a NuGet package (.nupkg) file. And, like any NuGet package, you can upload the template package to a NuGet feed. The dotnet new --install command supports installing template package from a NuGet package feed. Additionally, you can install a template package from a .nupkg file directly.

Normally you use a C# project file to compile code and produce a binary. However, the project can also be used to generate a template package. By changing the settings of the .csproj, you can prevent it from compiling any code and instead include all the assets of your templates as resources. When this project is built, it produces a template package NuGet package.

The package you'll create will include the [item template](https://docs.microsoft.com/en-us/dotnet/core/tutorials/cli-templates-create-item-template) and [package template](https://docs.microsoft.com/en-us/dotnet/core/tutorials/cli-templates-create-project-template) previously created. Because we grouped the two templates into the working\templates\ folder, we can use the working folder for the .csproj file.

In your terminal, navigate to the working folder. Create a new project and set the name to templatepack and the output folder to the current folder.

dotnet new console -n templatepack -o .

The -n parameter sets the .csproj filename to templatepack.csproj. The -o parameter creates the files in the current directory. You should see a result similar to the following output.

The template "Console Application" was created successfully.

Processing post-creation actions...

Running 'dotnet restore' on .\templatepack.csproj...

Restore completed in 52.38 ms for C:\working\templatepack.csproj.

Restore succeeded.

The new project template generates a Program.cs file. You can safely delete this file as it's not used by the templates.

Next, open the templatepack.csproj file in your favorite editor and replace the content with the following XML:

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<PackageType>Template</PackageType>

<PackageVersion>1.0</PackageVersion>

<PackageId>AdatumCorporation.Utility.Templates</PackageId>

<Title>AdatumCorporation Templates</Title>

<Authors>Me</Authors>

<Description>Templates to use when creating an application for Adatum Corporation.</Description>

<PackageTags>dotnet-new;templates;contoso</PackageTags>

<TargetFramework>netstandard2.0</TargetFramework>

<IncludeContentInPack>true</IncludeContentInPack>

<IncludeBuildOutput>false</IncludeBuildOutput>

<ContentTargetFolders>content</ContentTargetFolders>

<NoWarn>$(NoWarn);NU5128</NoWarn>

<NoDefaultExcludes>true</NoDefaultExcludes>

</PropertyGroup>

<ItemGroup>

<Content Include="templates\\*\*\\*" Exclude="templates\\*\*\bin\\*\*;templates\\*\*\obj\\*\*" />

<Compile Remove="\*\*\\*" />

</ItemGroup>

</Project>

The settings under <PropertyGroup> in the XML snippet are broken into three groups.

The first group deals with properties required for a NuGet package. The three <Package\*> settings have to do with the NuGet package properties to identify your package on a NuGet feed. Specifically the <PackageId> value is used to uninstall the template package with a single name instead of a directory path. It can also be used to install the template package from a NuGet feed. The remaining settings, such as <Title> and <PackageTags>, have to do with metadata displayed on the NuGet feed. For more information about NuGet settings, see [NuGet and MSBuild properties](https://docs.microsoft.com/en-us/nuget/reference/msbuild-targets).

**Note**

To ensure that the template package appears in dotnet new --search results, set <PackageType> to Template.

In the second group, the <TargetFramework> setting ensures that MSBuild executes properly when you run the pack command to compile and pack the project.

The third group includes settings that have to do with configuring the project to include the templates in the appropriate folder in the NuGet pack when it's created:

* The <NoWarn> setting suppresses a warning message that doesn't apply to template package projects.
* The <NoDefaultExcludes> setting ensures that files and folders that start with a . (like .gitignore) are part of the template. The default behavior of NuGet packages is to ignore those files and folders.

<ItemGroup> contains two items. First, the <Content> item includes everything in the templates folder as content. It's also set to exclude any bin folder or obj folder to prevent any compiled code (if you tested and compiled your templates) from being included. Second, the <Compile> item excludes all code files from compiling no matter where they're located. This setting prevents the project that's used to create the template package from trying to compile the code in the templates folder hierarchy.

## Build and install

Save the project file. Before building the template package, verify that your folder structure is correct. Any template you want to pack should be placed in the templates folder, in its own folder. The folder structure should look similar to the following hierarchy:

working

│ templatepack.csproj

└───templates

├───extensions

│ └───.template.config

│ template.json

└───consoleasync

└───.template.config

template.json

The templates folder has two folders: extensions and consoleasync.

In your terminal, from the working folder, run the dotnet pack command. This command builds your project and creates a NuGet package in the working\bin\Debug folder, as indicated by the following output:

C:\working> dotnet pack

Microsoft (R) Build Engine version 16.8.0+126527ff1 for .NET

Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 123.86 ms for C:\working\templatepack.csproj.

templatepack -> C:\working\bin\Debug\netstandard2.0\templatepack.dll

Successfully created package 'C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg'.

Next, install the template package with the dotnet new --install PATH\_TO\_NUPKG\_FILE command.

C:\working> dotnet new -i C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg

The following template packages will be installed:

C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg

Success: AdatumCorporation.Utility.Templates::1.0.0 installed the following templates:

Templates Short Name Language Tags

-------------------------------------------- ------------------- ------------ ----------------------

Example templates: string extensions stringext [C#] Common/Code

Example templates: async project consoleasync [C#] Common/Console/C#9

If you uploaded the NuGet package to a NuGet feed, you can use the dotnet new --install PACKAGEID command where PACKAGEID is the same as the <PackageId> setting from the .csproj file. This package ID is the same as the NuGet package identifier.

## Uninstall the template package

No matter how you installed the template package, either with the .nupkg file directly or by NuGet feed, removing a template package is the same. Use the <PackageId> of the template you want to uninstall. You can get a list of templates that are installed by running the dotnet new --uninstall command.

C:\working> dotnet new --uninstall

Template Instantiation Commands for .NET CLI

Currently installed items:

Microsoft.DotNet.Common.ProjectTemplates.2.2

Details:

NuGetPackageId: Microsoft.DotNet.Common.ProjectTemplates.2.2

Version: 1.0.2-beta4

Author: Microsoft

Templates:

Class library (classlib) C#

Class library (classlib) F#

Class library (classlib) VB

Console Application (console) C#

Console Application (console) F#

Console Application (console) VB

Uninstall Command:

dotnet new --uninstall Microsoft.DotNet.Common.ProjectTemplates.2.2

... cut to save space ...

AdatumCorporation.Utility.Templates

Details:

NuGetPackageId: AdatumCorporation.Utility.Templates

Version: 1.0.0

Author: Me

Templates:

Example templates: async project (consoleasync) C#

Example templates: string extensions (stringext) C#

Uninstall Command:

dotnet new --uninstall AdatumCorporation.Utility.Templates

Run dotnet new --uninstall AdatumCorporation.Utility.Templates to uninstall the template package. The command will output information about what template packages were uninstalled.

Congratulations! You've installed and uninstalled a template package.

## Creating a library package

## Prerequisites

1. Install the [.NET Core SDK](https://www.microsoft.com/net/download/), 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](https://www.nuget.org/users/account/LogOn?returnUrl=%2F) 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, .fsproj or .vbproj depending on the language you're using) and add the following minimal properties inside the existing <PropertyGroup> tag, changing the values as appropriate:

XMLCopy

<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).

1. Add any optional properties described on [NuGet metadata properties](https://docs.microsoft.com/en-us/dotnet/core/tools/csproj#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:

OutputCopy

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.

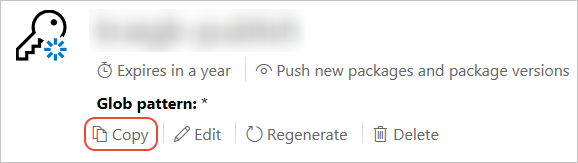
Packages published to nuget.org are also publicly visible to other developers unless you unlist them. To host packages privately, see [**Hosting packages**](https://docs.microsoft.com/en-us/nuget/hosting-packages/overview).

### Acquire your API key

1. [Sign into your nuget.org account](https://www.nuget.org/users/account/LogOn?returnUrl=%2F) or create an account if you don't have one already.

For more information on creating your account, see [Individual accounts](https://docs.microsoft.com/en-us/nuget/nuget-org/individual-accounts).

1. Select your user name (on the upper right), then select **API Keys**.
2. 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.)
3. Once the key is created, select **Copy** to retrieve the access key you need in the CLI:



**Warning**

**Always keep your API key a secret!** Treat your API key as a password that allows anyone to manage packages on your behalf. You should delete or regenerate your API key if it is accidentally revealed.

**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.

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](https://docs.microsoft.com/en-us/nuget/nuget-org/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 --api-key qz2jga8pl3dvn2akksyquwcs9ygggg4exypy3bhxy6w6x6 --source https://api.nuget.org/v3/index.json

1. dotnet displays the results of the publishing process:

OutputCopy

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](https://docs.microsoft.com/en-us/dotnet/core/tools/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:

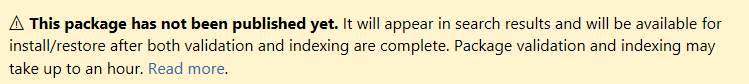
Response status code does not indicate success: 403 (The specified API key is invalid,

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

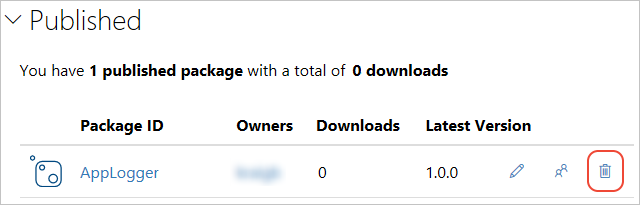
From your profile on nuget.org, select **Manage Packages** to see the one you just published. You also receive a confirmation email. Note that it might take a while for your package to be indexed and appear in search results where others can find it. During that time your package page shows the message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



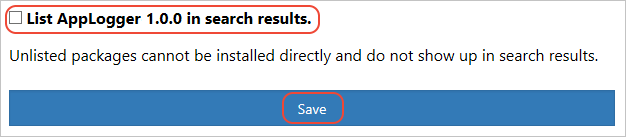
And that's it! You've just published your first NuGet package to nuget.org that other developers can use in their own projects.

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:



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



## 

## Hosting your own NuGet feeds

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](https://docs.microsoft.com/en-us/nuget/hosting-packages/local-feeds).
* NuGet.Server: Packages are made available through a local HTTP server. For details, see [NuGet.Server](https://docs.microsoft.com/en-us/nuget/hosting-packages/nuget-server). Usages:

nuget sources add -Name tc -Source https://xxxxxxxxx/ -Username username -Password password

nuget install xxxxxxxx -source tc

nuget sources remove -Name tc

* NuGet Gallery: Packages are hosted on an Internet server using the [NuGet Gallery Project](https://github.com/NuGet/NuGetGallery#build-and-run-the-gallery-in-arbitrary-number-easy-steps) (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](https://www.visualstudio.com/docs/package/nuget/publish) and [GitHub package registry](https://help.github.com/articles/configuring-nuget-for-use-with-github-package-registry) that support remote private feeds. Below is a list of such products:

* [Artifactory](https://www.jfrog.com/artifactory/) from JFrog.
* [Azure Artifacts](https://www.visualstudio.com/docs/package/nuget/publish), which is also available on Team Foundation Server 2017 and later.
* [BaGet](https://github.com/loic-sharma/BaGet), an open-source implementation of NuGet V3 server built on ASP.NET Core
* [Bytesafe](https://docs.bytesafe.dev/package-managers/nuget/) A fully managed package and supply chain security platform
* [Cloudsmith](https://cloudsmith.io/l/nuget-feed/), a fully managed package management SaaS
* [Gitea](https://gitea.io/), an open-source, self-hostable Git service supports NuGet as a [package registry](https://docs.gitea.io/en-us/packages/nuget/)
* [GitHub package registry](https://help.github.com/articles/configuring-nuget-for-use-with-github-package-registry)
* [GitLab Package Registry](https://docs.gitlab.com/ee/user/packages/nuget_repository/)
* [LiGet](https://github.com/ai-traders/liget), an open-source implementation of NuGet V2 server that runs on kestrel in docker
* [MyGet](https://myget.org/)
* [Nexus Repository OSS](https://www.sonatype.com/nexus-repository-oss) from Sonatype.
* [NuGet Server (Open Source)](https://github.com/svenkle/nuget-server), an open-source implementation similar to Inedo's NuGet Server
* [NuGet Server](http://nugetserver.net/), a community project from Inedo
* [ProGet](https://inedo.com/proget) from Inedo
* [Sleet](https://github.com/emgarten/sleet), an open-source NuGet V3 static feed generator
* [TeamCity](https://www.jetbrains.com/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](https://docs.microsoft.com/en-us/nuget/consume-packages/install-use-packages-visual-studio#package-sources), or from the command line using [nuget sources](https://docs.microsoft.com/en-us/nuget/reference/cli-reference/cli-ref-sources). The path to a source can be a local folder pathname, a network name, or a URL.

## Gitlab Package Registry:

The required minimum versions are:

* [NuGet CLI 5.1 or later](https://www.nuget.org/downloads). If you have [Visual Studio](https://visualstudio.microsoft.com/vs/), the NuGet CLI is probably already installed.
* Alternatively, you can use [.NET SDK 3.0 or later](https://dotnet.microsoft.com/download/dotnet/3.0), which installs the NuGet CLI.
* NuGet protocol version 3 or later.

To use the GitLab endpoint for NuGet Packages, choose an option:

* **Project-level**: Use when you have few NuGet packages and they are not in the same GitLab group.
* **Group-level**: Use when you have many NuGet packages in different projects within the same GitLab group.

Some features such as [publishing](https://docs.gitlab.com/ee/user/packages/nuget_repository/#publish-a-nuget-package) a package are only available on the project-level endpoint.

When asking for versions of a given NuGet package name, the GitLab Package Registry returns a maximum of 300 most recent versions.

Because of how NuGet handles credentials, the Package Registry rejects anonymous requests on the group-level endpoint. To work around this limitation, set up [authentication](https://docs.gitlab.com/ee/user/packages/nuget_repository/#add-the-package-registry-as-a-source-for-nuget-packages).

## Add the Package Registry as a source for NuGet packages

To publish and install packages to the Package Registry, you must add the Package Registry as a source for your packages.

Prerequisites:

* Your GitLab username.
* A personal access token or deploy token. For repository authentication:
  + You can generate a [personal access token](https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html) with the scope set to api.
  + You can generate a [deploy token](https://docs.gitlab.com/ee/user/project/deploy_tokens/index.html) with the scope set to read\_package\_registry, write\_package\_registry, or both.
* A name for your source.
* Depending on the [endpoint level](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages) you use, either:
  + Your project ID, which is found on your project’s home page.
  + Your group ID, which is found on your group’s home page.

You can now add a new source to NuGet with:

* [NuGet CLI](https://docs.gitlab.com/ee/user/packages/nuget_repository/#add-a-source-with-the-nuget-cli)
* [Visual Studio](https://docs.gitlab.com/ee/user/packages/nuget_repository/#add-a-source-with-visual-studio)
* [.NET CLI](https://docs.gitlab.com/ee/user/packages/nuget_repository/#add-a-source-with-the-net-cli)
* [Configuration file](https://docs.gitlab.com/ee/user/packages/nuget_repository/#add-a-source-with-a-configuration-file)

### Add a source with the NuGet CLI

#### Project-level endpoint

A project-level endpoint is required to publish NuGet packages to the Package Registry. A project-level endpoint is also required to install NuGet packages from a project.

To use the [project-level](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages) NuGet endpoint, add the Package Registry as a source with nuget:

nuget source Add -Name <source\_name> -Source "https://gitlab.example.com/api/v4/projects/<your\_project\_id>/packages/nuget/index.json" -UserName <gitlab\_username or deploy\_token\_username> -Password <gitlab\_personal\_access\_token or deploy\_token>

* <source\_name> is the desired source name.

For example:

nuget source Add -Name "GitLab" -Source "https://gitlab.example.com/api/v4/projects/10/packages/nuget/index.json" -UserName carol -Password 12345678asdf

#### Group-level endpoint

To install a NuGet package from a group, use a group-level endpoint.

To use the [group-level](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages) NuGet endpoint, add the Package Registry as a source with nuget:

nuget source Add -Name <source\_name> -Source "https://gitlab.example.com/api/v4/groups/<your\_group\_id>/-/packages/nuget/index.json" -UserName <gitlab\_username or deploy\_token\_username> -Password <gitlab\_personal\_access\_token or deploy\_token>

* <source\_name> is the desired source name.

For example:

nuget source Add -Name "GitLab" -Source "https://gitlab.example.com/api/v4/groups/23/-/packages/nuget/index.json" -UserName carol -Password 12345678asdf

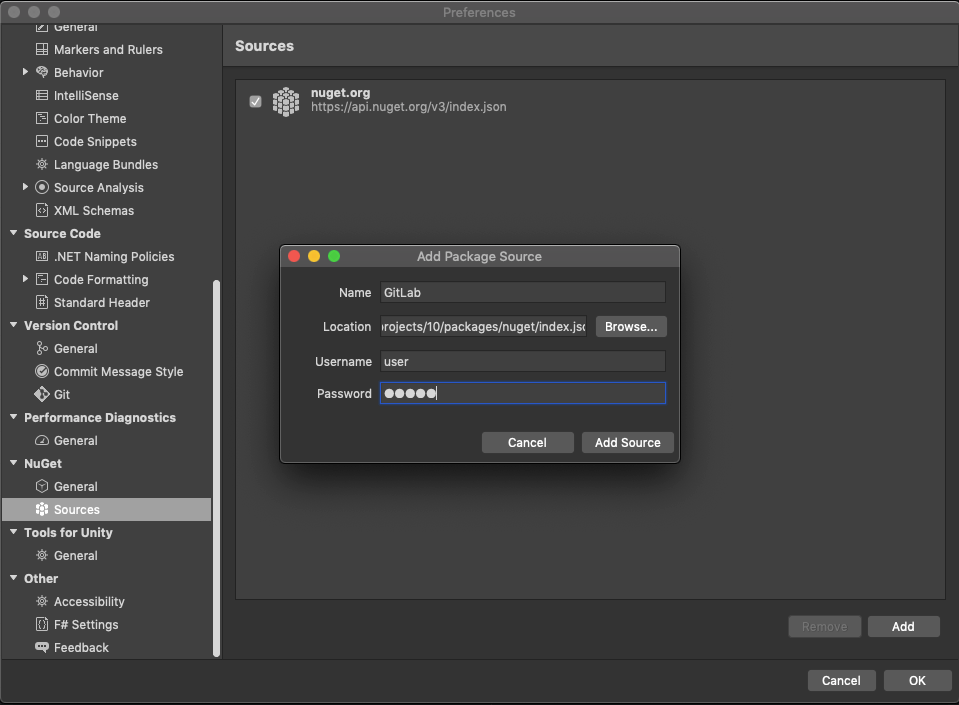
### Add a source with Visual Studio

#### Project-level endpoint

A project-level endpoint is required to publish NuGet packages to the Package Registry. A project-level endpoint is also required to install NuGet packages from a project.

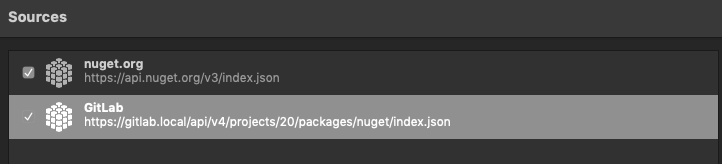
To use the [project-level](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages) NuGet endpoint, add the Package Registry as a source with Visual Studio:

1. Open [Visual Studio](https://visualstudio.microsoft.com/vs/).
2. In Windows, select **File > Options**. On macOS, select **Visual Studio > Preferences**.
3. In the **NuGet** section, select **Sources** to view a list of all your NuGet sources.
4. Select **Add**.
5. Complete the following fields:
   * **Name**: Name for the source.
   * **Location**: https://gitlab.example.com/api/v4/projects/<your\_project\_id>/packages/nuget/index.json, where <your\_project\_id> is your project ID, and gitlab.example.com is your domain name.
   * **Username**: Your GitLab username or deploy token username.
   * **Password**: Your personal access token or deploy token.

[](https://docs.gitlab.com/ee/user/packages/nuget_repository/img/visual_studio_adding_nuget_source.png)

1. Select **Save**.

The source is displayed in your list.

[](https://docs.gitlab.com/ee/user/packages/nuget_repository/img/visual_studio_nuget_source_added.png)

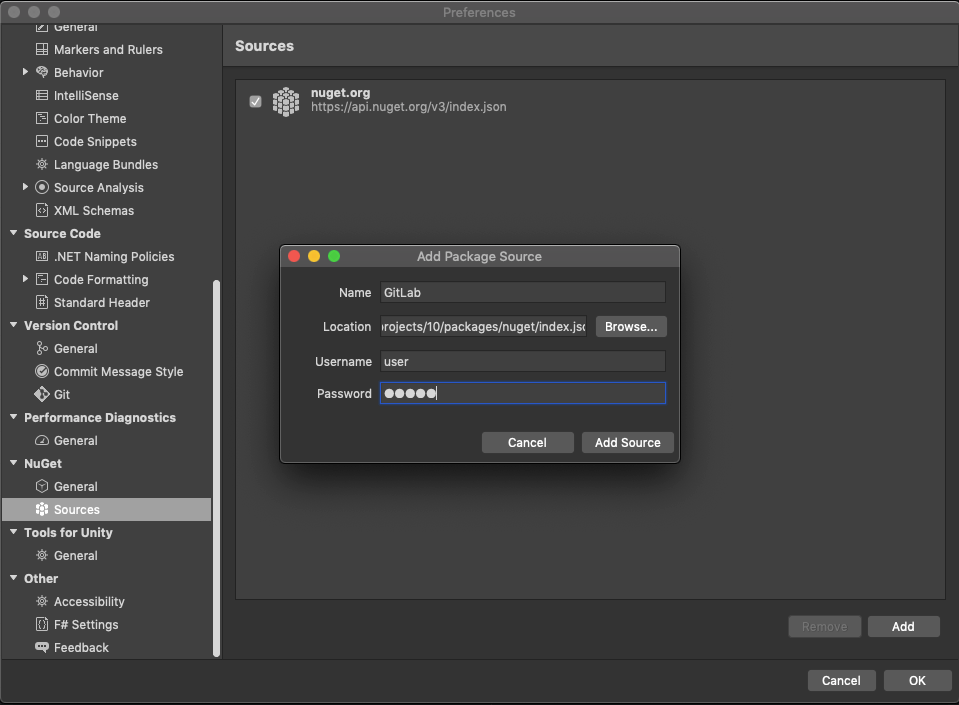
If you get a warning, ensure that the **Location**, **Username**, and **Password** are correct.

#### Group-level endpoint

To install a package from a group, use a group-level endpoint.

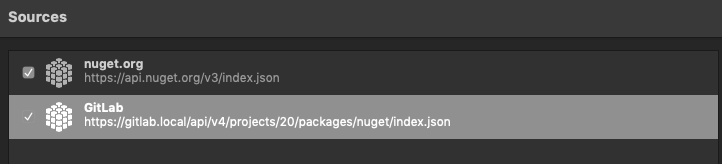
To use the [group-level](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages) NuGet endpoint, add the Package Registry as a source with Visual Studio:

1. Open [Visual Studio](https://visualstudio.microsoft.com/vs/).
2. In Windows, select **File > Options**. On macOS, select **Visual Studio > Preferences**.
3. In the **NuGet** section, select **Sources** to view a list of all your NuGet sources.
4. Select **Add**.
5. Complete the following fields:
   * **Name**: Name for the source.
   * **Location**: https://gitlab.example.com/api/v4/groups/<your\_group\_id>/-/packages/nuget/index.json, where <your\_group\_id> is your group ID, and gitlab.example.com is your domain name.
   * **Username**: Your GitLab username or deploy token username.
   * **Password**: Your personal access token or deploy token.

[](https://docs.gitlab.com/ee/user/packages/nuget_repository/img/visual_studio_adding_nuget_source.png)

1. Select **Save**.

The source is displayed in your list.

[](https://docs.gitlab.com/ee/user/packages/nuget_repository/img/visual_studio_nuget_source_added.png)

If you get a warning, ensure that the **Location**, **Username**, and **Password** are correct.

### Add a source with the .NET CLI

#### Project-level endpoint

A project-level endpoint is required to publish NuGet packages to the Package Registry. A project-level endpoint is also required to install NuGet packages from a project.

To use the [project-level](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages) NuGet endpoint, add the Package Registry as a source with nuget:

dotnet nuget add source "https://gitlab.example.com/api/v4/projects/<your\_project\_id>/packages/nuget/index.json" --name <source\_name> --username <gitlab\_username or deploy\_token\_username> --password <gitlab\_personal\_access\_token or deploy\_token>

* <source\_name> is the desired source name.
* --store-password-in-clear-text might be necessary depending on your operating system.

For example:

dotnet nuget add source "https://gitlab.example.com/api/v4/projects/10/packages/nuget/index.json" --name gitlab --username carol --password 12345678asdf

#### Group-level endpoint

To install a NuGet package from a group, use a group-level endpoint.

To use the [group-level](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages) NuGet endpoint, add the Package Registry as a source with nuget:

dotnet nuget add source "https://gitlab.example.com/api/v4/groups/<your\_group\_id>/-/packages/nuget/index.json" --name <source\_name> --username <gitlab\_username or deploy\_token\_username> --password <gitlab\_personal\_access\_token or deploy\_token>

* <source\_name> is the desired source name.
* --store-password-in-clear-text might be necessary depending on your operating system.

For example:

dotnet nuget add source "https://gitlab.example.com/api/v4/groups/23/-/packages/nuget/index.json" --name gitlab --username carol --password 12345678asdf

### Add a source with a configuration file

#### Project-level endpoint

A project-level endpoint is required to:

* Publish NuGet packages to the Package Registry.
* Install NuGet packages from a project.

To use the [project-level](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages) Package Registry as a source for .NET:

1. In the root of your project, create a file named nuget.config.
2. Add this content:
3. <?xml version="1.0" encoding="utf-8"?>
4. <configuration>
5. <packageSources>
6. <clear />
7. <add key="gitlab" value="https://gitlab.example.com/api/v4/projects/<your\_project\_id>/packages/nuget/index.json" />
8. </packageSources>
9. <packageSourceCredentials>
10. <gitlab>
11. <add key="Username" value="%GITLAB\_PACKAGE\_REGISTRY\_USERNAME%" />
12. <add key="ClearTextPassword" value="%GITLAB\_PACKAGE\_REGISTRY\_PASSWORD%" />
13. </gitlab>
14. </packageSourceCredentials>
15. </configuration>
16. Configure the necessary environment variables:
17. export GITLAB\_PACKAGE\_REGISTRY\_USERNAME=<gitlab\_username or deploy\_token\_username>
18. export GITLAB\_PACKAGE\_REGISTRY\_PASSWORD=<gitlab\_personal\_access\_token or deploy\_token>

#### Group-level endpoint

To install a package from a group, use a group-level endpoint.

To use the [group-level](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages) Package Registry as a source for .NET:

1. In the root of your project, create a file named nuget.config.
2. Add this content:
3. <?xml version="1.0" encoding="utf-8"?>
4. <configuration>
5. <packageSources>
6. <clear />
7. <add key="gitlab" value="https://gitlab.example.com/api/v4/groups/<your\_group\_id>/-/packages/nuget/index.json" />
8. </packageSources>
9. <packageSourceCredentials>
10. <gitlab>
11. <add key="Username" value="%GITLAB\_PACKAGE\_REGISTRY\_USERNAME%" />
12. <add key="ClearTextPassword" value="%GITLAB\_PACKAGE\_REGISTRY\_PASSWORD%" />
13. </gitlab>
14. </packageSourceCredentials>
15. </configuration>
16. Configure the necessary environment variables:
17. export GITLAB\_PACKAGE\_REGISTRY\_USERNAME=<gitlab\_username or deploy\_token\_username>
18. export GITLAB\_PACKAGE\_REGISTRY\_PASSWORD=<gitlab\_personal\_access\_token or deploy\_token>

## Publish a NuGet package

Prerequisite:

* Set up the [source](https://docs.gitlab.com/ee/user/packages/nuget_repository/#https://docs.gitlab.com/ee/user/packages/nuget_repository/#add-the-package-registry-as-a-source-for-nuget-packages) with a [project-level endpoint](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages).

When publishing packages:

* The Package Registry on GitLab.com can store up to 5 GB of content. This limit is [configurable for self-managed GitLab instances](https://docs.gitlab.com/ee/administration/instance_limits.html#package-registry-limits).
* If you publish the same package with the same version multiple times, each consecutive upload is saved as a separate file. When installing a package, GitLab serves the most recent file.
* When publishing packages to GitLab, they aren’t displayed in the packages user interface of your project immediately. It can take up to 10 minutes to process a package.

### Publish a package with the NuGet CLI

Prerequisites:

* [A NuGet package created with NuGet CLI](https://docs.microsoft.com/en-us/nuget/create-packages/creating-a-package).
* Set a [project-level endpoint](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages).

Publish a package by running this command:

nuget push <package\_file> -Source <source\_name>

* <package\_file> is your package filename, ending in .nupkg.
* <source\_name> is the [source name used during setup](https://docs.gitlab.com/ee/user/packages/nuget_repository/#add-a-source-with-the-nuget-cli).

### Publish a package with the .NET CLI

Prerequisites:

* [A NuGet package created with .NET CLI](https://docs.microsoft.com/en-us/nuget/create-packages/creating-a-package-dotnet-cli).
* Set a [project-level endpoint](https://docs.gitlab.com/ee/user/packages/nuget_repository/#use-the-gitlab-endpoint-for-nuget-packages).

Publish a package by running this command:

dotnet nuget push <package\_file> --source <source\_name>

* <package\_file> is your package filename, ending in .nupkg.
* <source\_name> is the [source name used during setup](https://docs.gitlab.com/ee/user/packages/nuget_repository/#add-a-source-with-the-net-cli).

For example:

dotnet nuget push MyPackage.1.0.0.nupkg --source gitlab

### Publish a NuGet package by using CI/CD

[Introduced](https://gitlab.com/gitlab-org/gitlab/-/issues/36424) in GitLab 13.3.

If you’re using NuGet with GitLab CI/CD, a CI job token can be used instead of a personal access token or deploy token. The token inherits the permissions of the user that generates the pipeline.

This example shows how to create a new package each time the main branch is updated:

1. Add a deploy job to your .gitlab-ci.yml file:
2. image: mcr.microsoft.com/dotnet/core/sdk:3.1
3. stages:
4. - deploy
5. deploy:
6. stage: deploy
7. script:
8. - dotnet pack -c Release
9. - dotnet nuget add source "${CI\_API\_V4\_URL}/projects/${CI\_PROJECT\_ID}/packages/nuget/index.json" --name gitlab --username gitlab-ci-token --password $CI\_JOB\_TOKEN --store-password-in-clear-text
10. - dotnet nuget push "bin/Release/\*.nupkg" --source gitlab
11. only:
12. - main
13. environment: production
14. Commit the changes and push it to your GitLab repository to trigger a new CI/CD build.

### Publishing a package with the same name or version

When you publish a package with the same name or version as an existing package, the existing package is overwritten.

## Install packages

If multiple packages have the same name and version, when you install a package, the most recently-published package is retrieved.

To install a NuGet package from the Package Registry, you must first [add a project-level or group-level endpoint](https://docs.gitlab.com/ee/user/packages/nuget_repository/#add-the-package-registry-as-a-source-for-nuget-packages).

### Install a package with the NuGet CLI

By default, nuget checks the official source at nuget.org first. If you have a NuGet package in the Package Registry with the same name as a package at nuget.org, you must specify the source name to install the correct package.

Install the latest version of a package by running this command:

nuget install <package\_id> -OutputDirectory <output\_directory> \

-Version <package\_version> \

-Source <source\_name>

* <package\_id> is the package ID.
* <output\_directory> is the output directory, where the package is installed.
* <package\_version> The package version. Optional.
* <source\_name> The source name. Optional.

### Install a package with the .NET CLI

If you have a package in the Package Registry with the same name as a package at a different source, verify the order in which dotnet checks sources during install. This is defined in the nuget.config file.

Install the latest version of a package by running this command:

dotnet add package <package\_id> \

-v <package\_version>

* <package\_id> is the package ID.
* <package\_version> is the package version. Optional.

[NuGet](https://asainternational.sharepoint.com/sites/KB/Information%20Technology/Forms/AllItems.aspx?id=%2Fsites%2FKB%2FInformation%20Technology%2FR%26D%2FDocs%2FNuGet%20templates%20%26%20Packages&viewid=1a03b43c%2D2976%2D44ab%2Db16a%2Df4497d46e398)

References:

<https://learn.microsoft.com/en-us/dotnet/core/tutorials/cli-templates-create-template-package>

<https://learn.microsoft.com/en-us/dotnet/core/tutorials/cli-templates-create-item-template?source=recommendations>

<https://learn.microsoft.com/en-us/dotnet/core/tutorials/cli-templates-create-project-template>

<https://learn.microsoft.com/en-us/dotnet/core/tutorials/cli-templates-create-template-package>

<https://learn.microsoft.com/en-us/nuget/create-packages/creating-a-package>

<https://docs.gitlab.com/ee/user/packages/>

<https://docs.gitlab.com/ee/user/packages/nuget_repository/>

<https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-nuget-registry>

<https://learn.microsoft.com/en-us/nuget/create-packages/set-package-type?tabs=dotnet>