How to create reusable packages for .NET Core using VS Code, C# and NuGet

> Creating a reusable package of code is great. It allows you to be faster up and running next time you are about to build something. It can also help your colleagues or why not the entire planet if you upload your package to a public available repository.

Disclaimer: this tutorial is *largely* based on https://softchris.github.io/pages/dotnet-nuget.html#create-a-nuget-package.
It is modified to only contain the necessary concepts learned in this lesson.

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**.

So NuGet is an open-source package manager designed for the Microsoft development platform and was introduced in 2010.

> Sweet, is there a lot of packages somewhere out there ready for me to use?

Yes, they all reside here:

> https://www.nuget.org/

You can also add the package to your project through a simple command in the terminal or using a UI if you have a full-blown IDE like Visual Studio, for example.

Let's learn how to create such a package, a so-called nuget for .NET Core, using **VS Code**.

In this article we are looking to go through the following:

- **Why** create a package. It's important to know why we are creating a package as opposed to just creating a library for example.
- **Creating** a library, here we will create a library, add some code and some tests. You should always test to your code.
- **NuGet** package creation, here we will create a NuGet by using a terminal command
- **Install** your package locally, we want to ensure that our package works locally. We can easily do that by installing it to one of our existing projects.

#Why create a package

Ok, so you want a package. Do you know why you want one? A lot of the times creating something reusable means creating a library. You might be a program vendor or you might be an OSS (open-source software) developer though or you got another good reason. Using NuGet to distribute your code means that **anyone** on the planet can use your code.

Are you:

- intellectually curious how this is done in .NET Core?
- do you want to create something reusable that you want anyone to access and install?

If the answer is yes to any of the above you seem ready to learn.

#Creating a library

The package is meant to be something reusable, like a library that we can add a reference to in our existing project. Because we are using .NET Core for this, we will be using the terminal. We will do the following:

1. **Create** a class library type project
2. **Add** the class library reference to the solution

Create a class library Next we will create the class library. This is where our reusable code will live.

```
dotnet new classlib -o mathnextlevel
```

#Add some code

Ok, we have a solution, a class library, but we are missing the meat, the code we want the world to use.

Let's open up our solution and add some code.

Rename our `Class1.cs` file to `Start.cs`. Now let's replace whatever code is in there with this:

```
namespace mathnextlevel
{
 public class Start
 {
  public int Add(int a, int b)
  {
    return a + b;
  }

  public int Sub(int a, int b)
  {
   return a - b;
  }
 }
}
```

Behold, the most impressive calculator known to man. 😉

Good, we are ready for turning this into a NuGet package now that our code looks like its working.

#Create a NuGet package

We need to do the following to create a NuGet package:

1. **Add** some meta information to the library project
2. **Invoke** the `pack` command, this will create the package

Adding meta information Now this is about adding meta data information to your project. Meta information are things like name, author, company and so on. You will need to add this as entry in the **mathnextlevel.csproj** file in the XML tag **`PropertyGroup`**. It can look like this:

```
<TargetFramework>netcoreapp3.1</TargetFramework>
<PackageId>mathnextlevel</PackageId>
<Version>1.0.0</Version>
<Authors>Student name</Authors>
<Company>CCCP dotnet inc.</Company>
```

The `PackageId` + `Version` will determine the name of your package. Next let's create our package. Let's navigate to our library directory and type:

```
cd mathnextlevel
dotnet pack
```

We can see from the above image that we got a package created in the `bin/Debug` folder called `mathnextlevel.1.0.0.nupkg.` We will use this information in our next section where we learn to install the package locally.

#Install your package locally

Let's start with the WHY? We need to test the package locally before we can push it nuget.org, that's just using sound judgment, we don't want to push anything broken.

So how do we do this? Well, we need a project to test it on. Let's scaffold a console application

```
dotnet new console -o app
```

Next, lets now install our package. We do this in two steps:

1. **Point out** the local source, this is about instructing the target project where this package can be found on your computer, so if it fails to find it on NuGet it knows to look at your computer next.
2. **Add the package**, this will install the package in the target project

#Point out the local source

We need to go into the target projects `.csproj` file. We want to test out our NuGet package on our console app called `app` so we open up `app.csproj` and under `PropertyGroup` we add a tag called `RestoreSources`. Here we point out both the path to our local NuGet package and the NuGet stream. It should look like so:

```
<RestoreSources>$(RestoreSources);absolute-path-to-my-solution/library/bin/Debug;https://api.nuget.org/v3/index.json</RestoreSources>
```

You need to replace `absolute-path-to-my-solution/library/bin/Debug` above with the absolute path to where your package is located.

#Add the package

Now that we pointed out where NuGet can find our package we are ready to install it. This is as simple as calling (from the console app root):

```
dotnet add package mathnextlevel
```

`mathnextlevel` is the name of the package we give it in the meta tags.

Inspecting our `app.csproj` file we can see that we now get the following entry:

```
<PackageReference Include="mathnextlevel" Version="1.0.1" />
```

Now lets try out our installed package. Change `Program.cs` in your app project to the following code:

```
using System;
using library;

namespace app
{
  class Program
  {
    static void Main(string[] args)
    {
      var start = new Start();
      Console.WriteLine(string.Format("Hello! {0}",
start.Add(2, 2)));
    }
  }
}
```

Test the program with:

```
dotnet run
```

#Working with different versions

As an author of a NuGet package you probably want to update your library sooner or later and add more functionality or correct bugs. Depending on what kind of update you do - you should increment the version number differently. Let's look at the version numbers in general and talk about a concept semantic version:

```
1.0.0
```

The leftmost value is called a major version. This should only be incremented if we do really major changes, including breaking the interface like, removing or renaming code. Updating it means we go from `1.0.0` to `2.0.0`.
The middle value is called a minor version. You use this when you do things like adding more functionality, like adding more methods. Updating it means we go from `1.0.0` to `1.1.0`.
The rightmost value is called a patch version. You should use this to do small changes. Those types of changes are usually because you want to fix an error, i.e patching the code. Updating it means we go from `1.0.0` to `1.0.1`

#Upgrading

Ok, so you've decided on upgrading your package. That means that you will need to take the following steps:

1. **Add** new functions to our library project
2. **Increment** the minor version of the package
3. **Generate** a new NuGet package from our library project
4. **Install** the new version on our console project app

Add new functions

Let's go to our library project. We will add code to handle multiplications, like so:

```
public int Multiply(int a, int b)
{
 return a * b;
}
```

the code in `Start.cs` should now look like so:

```
using System;
namespace library
{
 public class Start
 {
  public int Add(int a, int b)
  {
     return a + b;
  }
  public int Sub(int a, int b)
  {
    return a - b;
  }
  public int Multiply(int a, int b)
  {
     return a * b;
  }
 }
}
```

Increment the minor version

Open up mathnextlevel`.csproj` and find the XML tag `Version` and make sure it looks like this:

```
<Version>1.1.1</Version>
```

Generate a new NuGet package

For this, we will navigate ourselves to the library directory and run `dotnet pack`.

This should create the file `mathnextlevel.1.1.1.nupkg`

Install the new version

To install the new version we need to open up our `app.csproj` and increment the version. We need to find the XML tag `PackageReference` and change the attribute `Version` to `1.1.1` like so:

```
<PackageReference Include="math-chris" Version="1.1.1" />
```

Thereafter we can run `dotnet restore` in our app directory.

Your terminal should say something like this:

```
Installing math-chris 1.1.1
```

This means that you got the new code. Let's ensure that is the case by testing it out. Remember, you should have access to the `Multiply()` function. Open up the `Program.cs` file and change Add to Multiply:

#Managing versions

Sometimes you have the following scenario:

- **Fix a mistake**, You produced a package but realize you made a mistake in the code that you want to fix AND you want to stay on the same version
- **Go to a lower version**, You want to move back to a previous version of the package

In both these cases, you need to deal with your local NuGet cache. The way it currently works is that you can always increment a version and do a `dotnet restore`. Going back means you need to clear the cache first, as it would otherwise hold on to the newest version. You do that by the following command:

```
dotnet nuget locals all -clear
```

Now you can change `Version` attribute to a lower one in `app.csproj` and go to the terminal and your app directory and do `dotnet restore`. Your code, in `Program.cs`, should now be yelling at you saying you don't have a `Multiply()` method and that's a GOOD thing, it means the whole process is working as it should.

Remember, you can always change the `Version` back to `1.1.1` do a `dotnet restore` and your code will work fine again 😃

#Summary

We've come to the final stop. Time to recap what we've learned. We've managed to learn:

- Why a package might be a good thing to create
- How to create a package
- How to test the package locally.

Hopefully, this will make you inspired to create packages of your own 😃