Tooling in .Net Core

**Abstract**

You can now manage all aspects of working with solutions and projects using the new command-line tooling. The principle command is dotnet. Using this command you can create solutions, create projects and add them to the solution, add project dependencies, add NuGet dependencies—the works!

All this is done so that you no longer need to rely on VS for all your building needs. We now have full integration with MSBuild.

This document covers only the very basics. You’re going to have to google the rest.

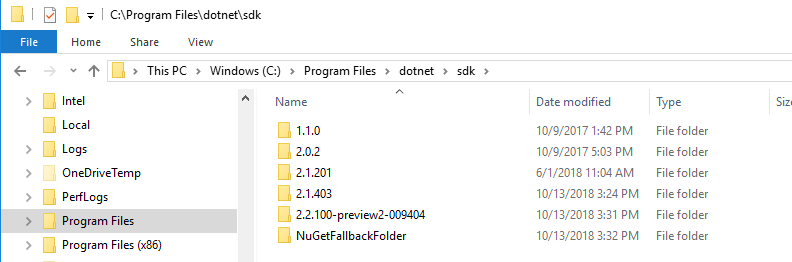
# Setup

How to get the tooling (two options):

1. (preferred option for dev environments) All the tooling you’ll need comes when you install VS 2017 (you can probably pick it up with the community edition). You CANNOT use VS 2015!
2. (preferred option for a build machine) You can download it manually:
   * <https://dot.net> 🡺 <https://microsoft.com/net>
   * Look for the sdk.

## Which SDKs do you have installed?

On Windows, the dot-net sdk gets installed under program files/dotnet/sdk.



On Linux, type “which dotnet” to see the folder name, then go to that folder (should be something like /usr/local/share/dotnet). You can then go into the sdk folder and use ls to see what you’ve got.

The main tooling command is dotnet. (think of it like npm, where you typed npm start, npm test, etc.)

To see what version you’re on, you can use --version.

C:\Git\Projects\DemoProjects>dotnet --version

2.2.100-preview2-009404

You can get help:

dotnet -h

This will give you a quick summary of commands.

# Managing projects and solutions

## Creating a new solution

## .NET Core Project Overview

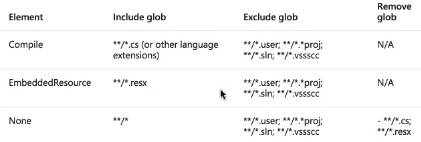
So a bit of history:

1. Before .NET Core, we used .csproj files, which were nice, but super-duper complicated.
2. So they invented a new project.json file that has a much simpler structure, but more importantly, they used a convention-based approach, and they also borrowed a lot from NPM, which didn’t rely on specifying every damn thing in the world. But there were still problems.
3. So they decided to take what they learned with .csproj files, and just simplify everything. They took all the goodness and all the lessons that they’d learned, and migrated them into the new .csproj format, which is basically the same old format with a ton of implicit conventions to cut down on the verbosity.

And boy did they cut things down. It turns out VS / MSBuild really didn’t need to be explicitly told every stinking thing in the universe. These tools could figure out plenty on their own.

So the new benefits include:

1. **Implicit inclusion of source files.** If it’s in the same folder or a sub-folder of your .csproj file, it’ll get built. Here is a screenshot:



1. **Streamlined package references.** Only explicit package references are required.
   1. This now works like NPM, where if package A needs package B, you only need to include package A, and NuGet will get all the dependencies.
   2. And like NPM, you can also specify a version number.
2. **You can modify the .csproj file directly,** without needing VS or a tool to do that.
   1. You can manually change versions of NuGet packages.
   2. You can manually change the targeted framework(s)
   3. NuGet is now fully integrated into MSBuild
3. Create a NuGet package from a project file
4. **Cross compiling a single peoject.** You can simultaneously build multiple frameworks! Change the <TargetFramework> element to <TargetFrameworks>, and give a ; delimited list of frameworks, like so:



1. **Cross platform support.** It’s now open source, and you can guild on linux.
2. **Projects are composable.** You can have projects that reference other projects, or you can have projects inherit settings from parent projects.

## Creating a new project

So, to create a new project, you shell out, and use the dotnet command.

You’ll need to initialize the global package cache, so type this:

dotnet new -h

If it hasn’t been initialized yet, it will do its thing, then it will show you a list of project templates that you can use:

Templates Short Name Language Tags

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

Console Application console [C#], F#, VB Common/Console

Class library classlib [C#], F#, VB Common/Library

Unit Test Project mstest [C#], F#, VB Test/MSTest

NUnit 3 Test Project nunit [C#], F#, VB Test/NUnit

NUnit 3 Test Item nunit-test [C#], F#, VB Test/NUnit

xUnit Test Project xunit [C#], F#, VB Test/xUnit

Razor Page page [C#] Web/ASP.NET

MVC ViewImports viewimports [C#] Web/ASP.NET

MVC ViewStart viewstart [C#] Web/ASP.NET

ASP.NET Core Empty web [C#], F# Web/Empty

ASP.NET Core Web App (Model-View-Controller) mvc [C#], F# Web/MVC

ASP.NET Core Web App razor [C#] Web/MVC/Razor Pages

ASP.NET Core with Angular angular [C#] Web/MVC/SPA

ASP.NET Core with React.js react [C#] Web/MVC/SPA

ASP.NET Core with React.js and Redux reactredux [C#] Web/MVC/SPA

Razor Class Library razorclasslib [C#] Web/Razor/Library/Razor Class Library

ASP.NET Core Web API webapi [C#], F# Web/WebAPI

global.json file globaljson Config

NuGet Config nugetconfig Config

Web Config webconfig Config

Solution File sln Solution

Examples:

dotnet new mvc --auth Individual

dotnet new nunit-test

dotnet new --help

When creating a project from one of these template, you can use the long name (enclosed with quotes), or you can use the short name. (you can install other templates—you can even create project templates of your own. W00t!)

so, pick a template name and a folder location, and away you go

dotnet new reactredux -o AspReactDemo

The project name and the folder name are the same.

Historically with VS you had a wizard to guide you through creating a project, and it gave you tons of options. But you don’t have that any more with the command line. Instead, each project template has a number of options which you can specify. To get help on these, you use the --help (or -h) command.

So, if I wanted to see what options I have with the reactredux template, I could type this command:

dotnet new reactredux -h

And that gives me options galore. So, for instance, we can specify an alternate namespace and an alternate folder name:

dotnet new reactredux -o AspReactDemo -n MyNamespace

### Adding packages to your project

Adding packages is pretty simple. Start by going into the same folder as your .csproj file:

dotnet add package <package-name>

### Building your project

Before you can build anything, you’ll need to fetch all the dependencies. Use the following command:

dotnet restore

And then you can run it:

dotnet run

Or alternatively you can straight-up build the project:

dotnet build

# Static code analysis

## StyleCop

StyleCop checks C# code for conformance to StyleCop's recommended coding styles and a subset of Microsoft's .NET Framework Design Guidelines. The rules are classified into the following categories:

* Documentation
* Layout
* Maintainability
* Naming
* Ordering
* Readability
* Spacing

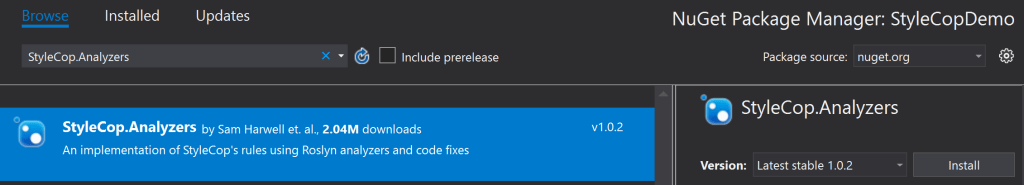
Documentation for the latest version (.NET Core 2 / VS 2017) can be found on GitHub:

<https://github.com/DotNetAnalyzers/StyleCopAnalyzers>

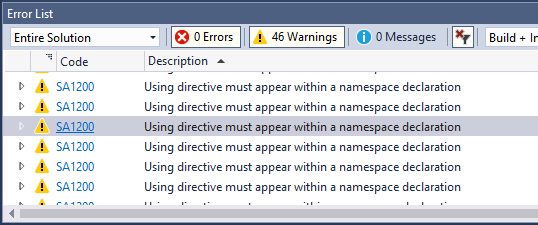
### Setup

**StyleCop is installed on a per-project basis.** There is a VS extension that you can install if you like, but it’s not required with VS 2017. The extension gets you a few extra context menu items and such, so that you can run StyleCop without kicking off a full build.

You merely add StyleCop to your project as a NuGet package, and that’s it. You need to install StyleCop.Analyzers:



Once this is added, all you have to do is re-build your project and they’ll show up as warnings in the Error List window:



So, you’ll want to configure the rules, because a lot of them feel a little pointless, and some of them are directly OPPOSITE of what people normally do.

### Setting up stylecop.json

You’ll need to set up a stylecop.json file, and this file should be checked in with the other files for your project. For VS 2017 this is a two-step process.

Step 1: The easiest way to add a stylecop.json configuration file to a new project is using a code fix provided by the project. To invoke the code fix, open any file where SA1633 is reported (The file header is missing or not located at the top of the file.) and press Ctrl+. to bring up the Quick Fix menu. From the menu, select Add StyleCop settings file to the project.

Or, you can create a new stylecop.json file in the root of your project folder and copy-paste this text:

{

  "$schema": "https://raw.githubusercontent.com/DotNetAnalyzers/StyleCopAnalyzers/master/StyleCop.Analyzers/StyleCop.Analyzers/Settings/stylecop.schema.json",

  "settings": {

    "documentationRules": {

      "companyName": "PlaceholderCompany"

    }

  }

}

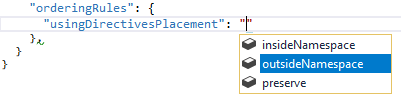
Step 2: Once you have that stylecop.json added to your project and saved, go to the properties for the file:

1. Open Solution Explorer
2. Click the file
3. Properties (Alt+Enter)

Set the build action to C# analyzer additional file. If you’re not using .NET Core then set the type to Additional file.

### Configuring rule options in stylecop.json

Once you have stylecop.json set up, all you have to do is open it and start typing, and the intelliSense™ will kick in. Everything goes under the “settings” property.

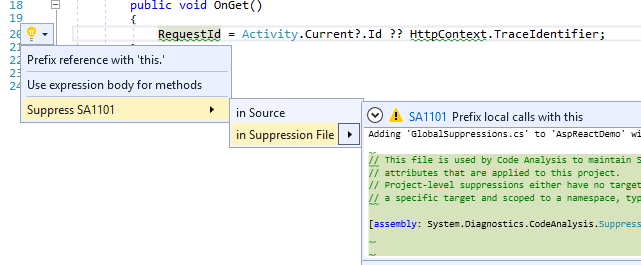


### Disabling rules

Click on an error in the errors window. That will take you to the offending line of code. Now press Ctrl+. to get the Quick Fix menu and then select Supress SAxxxx.

You have a couple options:

This will create a file called GlobalSuppressions.cs, which has a bunch of assembly-level attributes.



You will have to modify the rule suppression. By default, Vs will add the rule to be specific to that single offense in that single class, on that single line.

* To apply the rule for the whole file, use a #pragma instead of using GlobalSuppressions.cs
* To make the rule apply for that class only, set Scope to “type”, and modify the Target to refer to a class name.
* To make a rule apply for the entire project, erase the Scope parameter and the Target parameter.

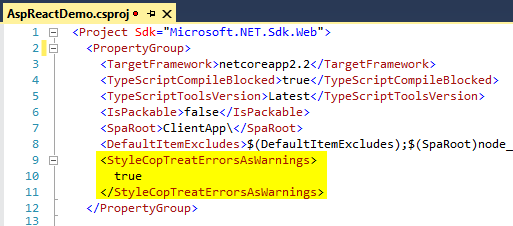
### Treating warnings as errors

I got this working, but it’s not recognizing my config, so it’s treating all errors as warnings regardless of what’s been suppressed or not.

You’ll need to install another NuGet package:

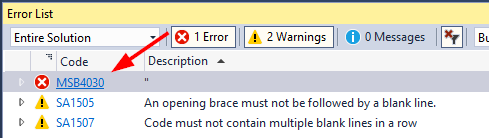
StyleCop.MSBuild

Now you need to edit your .csproj file (so unload it, then open it in an editor). Add this to your Project/PropertyGroup element:



NOTE: the newer StyleCop built on the Roslyn analyzer will ignore generated files for you. There are a lot of blog-posts that’ll warn you about generated files, but you can ignore them.

When you build, you’ll see this (which is only sort of helpful. Ugh…):



Clicking the error will take you to a cryptic config file (StyleCop.MSBuild.targets)