A boilerplate for Visual Studio Code C# solution .Net Core with unit tests

Ambientes Virtuais de Execução

2017

In the following sections we present a step by step guide to setup a VS Code solution with libraries (Primes and Fibonacci), a console application (App) and unit tests (Primes.Tests and Fibonacci.Tests).

If you just want to check the final result, then jump to Quick Setup after the Installation.

Table of contents:

Installation

1. .NET Core SDK
2. Visual Studio Code (VS Code)
3. Open VS Code and click on Extensions (last icon of left side bar or `Ctrl + Shif + X`)
4. Install the extension "*C# for Visual Studio Code (powered by OmniSharp*"

Quick Setup

Clone this repo and run the following commands in the root folder:

- `code .` to open VS Code for this solution
- Type `Ctr + Shif + B` to build the solution
- Type `Ctr + F5` to run `App` project
- Select the `Debug` icon on left side bar and then `.Net Core Xunt tests` to run the unit tests.

Commands list

In this guide we use the following commands:

- `dotnet new` to create an up-to-date `project.json` file with NuGet dependencies.
- `dotnet restore`, which calls into NuGet to restore the tree of dependencies.
- `dotnet build` to compile source files.

- `dotnet <assembly.dll>` to run the target application.

or:

- `dotnet run` to run your application from the source code. Relies on `dotnet build` to build source inputs before launching the program.
- `code .` to open Visual Studio Code for current location.

Folders structure

Create the following projects folders structure (NOTE: create only folders and **not the files**):

```
/<solution name>
|__global.json
|__/src
   |__/<library name>
      |__Source Files
      |__project.json
   |.../ Other libraries or applications
|__/test
   |__/<library name>.Test
      |__Test Files
      |__project.json
```

Each library , application, or unit test project will contain its own folder with its own `project.json` file inside it. In the next section we will initialize those `project.json` files.

Setup libraries and application projects

1. CD into library directory (e.g. `src/Fibonacci`)
2. Run `dotnet new -t lib` to create the source project.
3. Check the `project.json` which contains dependencies necessary to build the library.
4. Rename `Library.cs` (e.g.`FibSupplier.cs`)
5. Open library C# file and rename the class according to the new file name (e.g. `class FibSupplier {…}`)
6. Rename the namespace according to the directory name (e.g. `namespace Fibonacci {…}`)

Repeat these steps for each library or application. **For applications suppress the `-t lib` option** on `dotnet new` command. **Advice**: You should create an application because it will simplify the build configuration in VS Code.
In `boilerplate` solution we have 3 projects in `src` folder, corresponding to `Fibonacci`, `Primes` and `App`.
NOTE: For small demos without unit tests this cheasheet finishes here. CD to your application folder and run `dotnet restore`, `dotnet build` and then `dotnet run`.

6. For each project that depends of other projects you must refer those projects in the `dependencies` property of `project.json`. For instance, the `project.json` of `App` has the following dependencies:

```
"dependencies": {
  "Fibonacci": {"target": "project"},
  "Primes": {"target": "project"}
}
```

The following steps are **optional** because we will build the entire solution in the last section. However if you want to try each project individually you can follow next steps for each project:

7. Run `dotnet restore`, which calls into NuGet to restore the tree of dependencies.
8. Check the `project.lock.json` files that contains a complete set of the graph of NuGet dependencies.
9. Run `dotnet build` to compile source files.
10. For applications run `dotnet run` that calls `dotnet <assembly.dll>` to run the target application (e.g. `dotnet src/App/bin/Debug/netcoreapp1.0/App.dll`)

Note: If your `App` refers all project libraries, then you just need to build the `App` project because `dotnet build` ensures to build target projects.

Setup unit tests project

1. CD into unit tests project directory (e.g. `test/Fibonacci.Tests`)
2. Run `dotnet new -t xunittest`. Check the generated `project.json`, which includes the test runner and dependencies for `xunit` and `dotnet-test-xunit` Nuget libraries.
3. Rename `Tests.cs` (e.g. `FibSupplierTests.cs`)
4. Open tests C# file and rename the class according to the new file name (e.g. `class FibSupplierTests{…}`)
5. Rename the namespace according to the directory name (e.g. `namespace Fibonacci.Tests{…}`)
6. Add the dependency to `Fibonacci` project. Edit `src/Fibonacci.Tests/project.json` file and add the property `"Fibonacci": {"target": "project"}` to the `dependencies` object that will look like Figure 1

The following steps are **optional** because we will build the entire solution in the last section. However if you want to try your tests follow next steps:

7. In the root directory create a global.json that contains the names of your src and test directories (e.g. { "projects": ["src", "test"] })
8. Run `dotnet restore`, which calls into NuGet to restore the tree of dependencies.
9. Check the `project.lock.json` files that contains a complete set of the graph of NuGet dependencies.
10. Run `dotnet build` to compile source files.
11. Execute `dotnet test` to run the tests from the console. The xunit test runner has the program entry point to run your tests from the Console. `dotnet test` starts the test runner, and provides a command line argument to the testrunner indicating the assembly that contains your tests.

Figure 1:

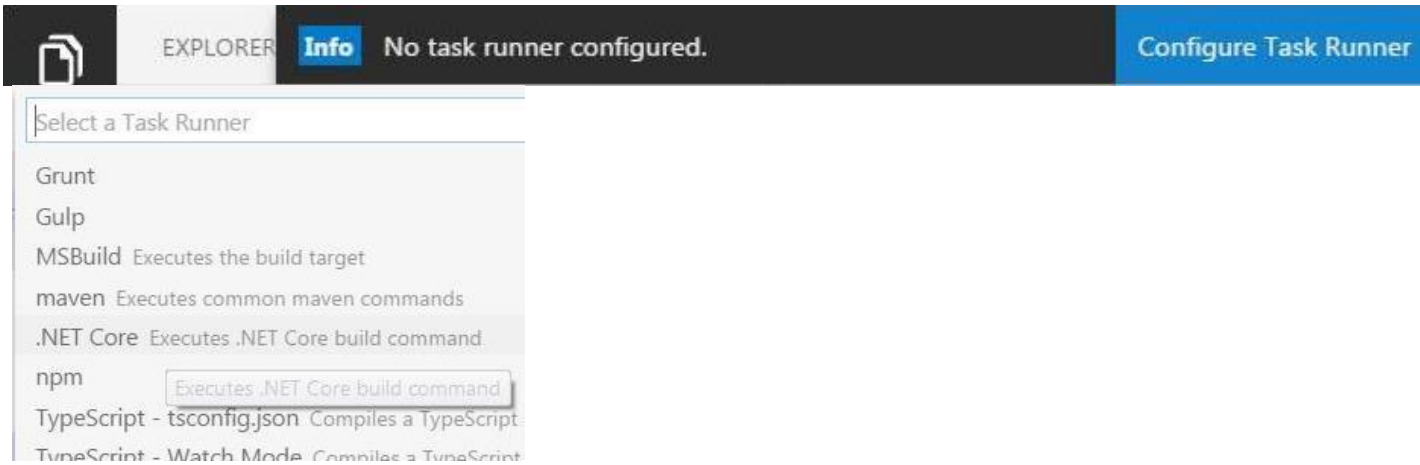
```
"dependencies": {
  "System.Runtime.Serialization.Primitives": "4.1.1",
  "xunit": "2.1.0",
  "dotnet-test-xunit": "1.0.0-*",
  "Fibonacci": {"target": "project"}
}
```

Notice that you do not include any directory path to the `Fibonacci` project, because you created the project structure to match the expected organization of `src` and `test`. The `"target": "project"` element informs NuGet that it should look in project directories, not in the NuGet feed. Without this key, you might download a package with the same name as your internal library.

Setup solution build

1. In the root directory create a `global.json` that contains the names of your `src` and `test` directories (e.g. `{ "projects": ["src", "test"] }`)
2. Run `dotnet restore`, which calls into NuGet to restore the tree of dependencies (you can skip this step if you have already run it individually for each project previously).
3. Check the `project.lock.json` files which appear on each project folder and contain a complete set of the graph of NuGet dependencies.
4. Run `code .` on root directory that will open the Visual Studio Code for your solution
5. Create the task runner file (`.vscode/tasks.json`) following one of the next options:
    - Option A: Click `Yes` if appears the Warning message `Required assets to build and debug are missing from….`

    - Option B: Type `Ctr + Shif + B` and choose `Configure Task Runner` and then `.NET Core -- Executes .NET Core build command`

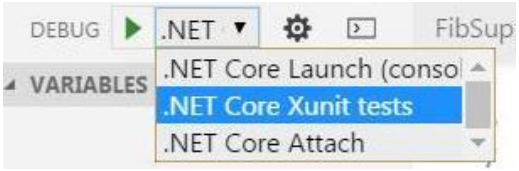


6. Open `.vscode/tasks.json` and add to `args` array the paths to your projects (e.g. `"${workspaceRoot}/src/App/project.json"` (use `\\` in Wndows)). Because `App` depends of all other projects it will build the `Fibonacci` and `Primes` projects first.
7. If your solution includes an application (with property `"emitEntryPoint": true` in `buildOptions` of `project.json`) then run your application with `F5` or `Ctr + F5` (without debug).
8. Add a new entry in `launch.json` to run all your xunit tests. Add the following item to `configurations` property of `launch.json`:

```
"configurations": [
      {
          "name": ".NET Core Xunit tests",
          "type": "coreclr",
          "request": "launch",
          "preLaunchTask": "build",
          "program": " /usr/local/share/dotnet/dotnet ",
          "args": ["test"],
          "cwd": "${workspaceRoot}/test/Fibonacci.Tests",
          "externalConsole": false,
          "stopAtEntry": false,
          "internalConsoleOptions": "openOnSessionStart"
      },
     …
]
```

For windows users in `program` set to `C:\\Program Files\\dotnet\\dotnet.exe`

9. To run xunit tests select the `Debug` icon on left side bar and then `.Net Core Xunt tests` -- this is the name you gave in step 8.



10. You can run or debug each test individually by clicking on the corresponding option over each unit test method

```
[Fact]
0 references | run test | debug test
public void FuncTest()
{
    int[] expected = {0,
    var fibs = ToIterator
    Assert.Equal(expected
```