Unit testing C# in .NET Core using dotnet test and xUnit

4 minutes to read • +15

This tutorial shows how to build a solution containing a unit test project and source code project. To follow the tutorial using a pre-built solution, view or download the sample code. For download instructions, see Samples and Tutorials.

Create the solution

In this section, a solution is created that contains the source and test projects. The completed solution has the following directory structure:

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
        PrimeService.cs
        PrimeService.csproj
    /PrimeService.Tests
        PrimeService_IsPrimeShould.cs
        PrimeServiceTests.csproj
```

The following instructions provide the steps to create the test solution. See Commands to create test solution for instructions to create the test solution in one step.

- Open a shell window.
- Run the following command:

```
dotnet new sln -o unit-testing-using-dotnet-test
```

The dotnet new sln command creates a new solution in the *unit-testing-using-*

dotnet-test directory.

- Change directory to the *unit-testing-using-dotnet-test* folder.
- Run the following command:

```
dotnet new classlib -o PrimeService
```

The dotnet new classlib command creates a new class library project in the *PrimeService* folder. The new class library will contain the code to be tested.

- Rename *Class1.cs* to *PrimeService.cs*.
- Replace the code in *PrimeService.cs* with the following code:

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Not implemented.");
        }
    }
}
```

- The preceding code:
    - Throws a NotImplementedException with a message indicating it's not implemented.
    - Is updated later in the tutorial.
- In the *unit-testing-using-dotnet-test* directory, run the following command to add the class library project to the solution:

```
dotnet sln add ./PrimeService/PrimeService.csproj
```

- Create the *PrimeService.Tests* project by running the following command:

```
dotnet new xunit -o PrimeService.Tests
```

- The preceding command:
  - Creates the *PrimeService.Tests* project in the *PrimeService.Tests* directory. The test project uses xUnit as the test library.
  - Configures the test runner by adding the following `<PackageReference />` elements to the project file:
    - "Microsoft.NET.Test.Sdk"
    - "xunit"
    - "xunit.runner.visualstudio"

- Add the test project to the solution file by running the following command:

```
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

- Add the `PrimeService` class library as a dependency to the *PrimeService.Tests* project:

```
dotnet add ./PrimeService.Tests/PrimeService.Tests.csproj reference
./PrimeService/PrimeService.csproj
```

Commands to create the solution

This section summarizes all the commands in the previous section. Skip this section if you've completed the steps in the previous section.

The following commands create the test solution on a windows machine. For macOS and Unix, update the `ren` command to the OS version of `ren` to rename a file:

```
dotnet new sln -o unit-testing-using-dotnet-test
```

```
cd unit-testing-using-dotnet-test
dotnet new classlib -o PrimeService
ren .\PrimeService\Class1.cs PrimeService.cs
dotnet sln add ./PrimeService/PrimeService.csproj
dotnet new xunit -o PrimeService.Tests
dotnet add ./PrimeService.Tests/PrimeService.Tests.csproj reference
./PrimeService/PrimeService.csproj
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

Follow the instructions for "Replace the code in *PrimeService.cs* with the following code" in the previous section.

Create a test

A popular approach in test driven development (TDD) is to write a test before implementing the target code. This tutorial uses the TDD approach. The `IsPrime` method is callable, but not implemented. A test call to `IsPrime` fails. With TDD, a test is written that is known to fail. The target code is updated to make the test pass. You keep repeating this approach, writing a failing test and then updating the target code to pass.

Update the *PrimeService.Tests* project:

- Delete *PrimeService.Tests/UnitTest1.cs*.
- Create a *PrimeService.Tests/PrimeService_IsPrimeShould.cs* file.
- Replace the code in *PrimeService_IsPrimeShould.cs* with the following code:

```
using Xunit;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    public class PrimeService_IsPrimeShould
    {
        private readonly PrimeService _primeService;

        public PrimeService_IsPrimeShould()
        {
            _primeService = new PrimeService();
        }

        [Fact]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            var result = _primeService.IsPrime(1);

            Assert.False(result, "1 should not be prime");
        }
    }
}
```

The `[Fact]` attribute declares a test method that's run by the test runner. From the *PrimeService.Tests* folder, run `dotnet test`. The dotnet test command builds both projects and runs the tests. The xUnit test runner contains the program entry point to run the tests. `dotnet test` starts the test runner using the unit test project.

The test fails because `IsPrime` hasn't been implemented. Using the TDD approach, write only enough code so this test passes. Update `IsPrime` with the following code:

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Not fully implemented.");
}
```

Run `dotnet test`. The test passes.

Add more tests

Add prime number tests for 0 and -1. You could copy the preceding test and change the following code to use 0 and -1:

```
var result = _primeService.IsPrime(1);

Assert.False(result, "1 should not be prime");
```

Copying test code when only a parameter changes results in code duplication and test bloat. The following xUnit attributes enable writing a suite of similar tests:

- `[Theory]` represents a suite of tests that execute the same code but have different input arguments.
- `[InlineData]` attribute specifies values for those inputs.

Rather than creating new tests, apply the preceding xUnit attributes to create a single theory. Replace the following code:

```
[Fact]
public void IsPrime_InputIs1_ReturnFalse()
{
    var result = _primeService.IsPrime(1);

    Assert.False(result, "1 should not be prime");
}
```

with the following code:

```
[Theory]
[InlineData(-1)]
[InlineData(0)]
[InlineData(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.False(result, $"{value} should not be prime");
}
```

In the preceding code, `[Theory]` and `[InlineData]` enable testing several values less than two. Two is the smallest prime number.

Run `dotnet test`, two of the tests fail. To make all of the tests pass, update the `IsPrime` method with the following code:

```
public bool IsPrime(int candidate)
{
    if (candidate < 2)
```

```
        {
            return false;
        }
        throw new NotImplementedException("Not fully implemented.");
    }
```

Following the TDD approach, add more failing tests, then update the target code. See the finished version of the tests and the complete implementation of the library.

The completed `IsPrime` method is not an efficient algorithm for testing primality.

Additional resources

- xUnit.net official site
- Testing controller logic in ASP.NET Core
- dotnet add reference

Is this page helpful?

Yes No