Testing Your Code

Unit Testing Your Code

Testing the code you write is one of the most important and underappreciated practices in programming. There are integration tests, web tests, performance/load tests, and, the one covered in this lesson, unit tests.

Unit Tests test individual methods, your most basic "unit" of code, for completeness and correctness. This type of test drives Test Driven Development, or TDD, and should be small and fast with descriptive names. To be clear, unit tests aren't limited only to TDD, they can and **should** be added at any time during development. Unit tests are a great way to prevent and catch bugs during feature development as well as general refactoring.

Throughout this lesson you will use the xunit testing framework. There is some configuration you will need to do in order to add the Xunit references and run these tests from the `dotnet CLI`. You will need to update your `project.json` file in your test project to be the following.

```
{
  "version": "1.0.0-*",
  "buildOptions": {
    "emitEntryPoint": true
  },
  "dependencies": {
    "xunit": "2.1.0",
    "dotnet-test-xunit": "1.0.0-rc2-build10025"
 },
  "frameworks": {
    "netcoreapp1.0": {
      "dependencies": {
        "Microsoft.NETCore.App": {
          "type": "platform",
          "version": "1.0.0-rc3-004408"
        }
      },
      "imports": [
          "dnxcore50",
          "portable-net45+win8"
        ]
    }
  },
  "testRunner": "xunit"
}
```

Once your `project.json` file looks like above, you can run the following command to add the xunit reference to your project.

```
dotnet restore
```

You will now be able to run all xunit tests in your project by using this command.

```
dotnet test
```

Writing Your First Test

For this test, look back at the methods lesson, and your `ExtensionMethods` class.

```
public static class ExtensionMethods
{
    public static int PlusFive(this int input)
    {
        return input + 5;
    }
}
```

Before writing a test to confirm the result of your extension method above there is something important to consider, **naming**. Good naming for your unit tests can make your code essentially self-documenting. This makes it easier for other developers on your project to know what exactly a test is supposed to do, which means they will know what exactly the "unit" of code is intended for just by the name.

You can now begin to construct a unit test by adding a new class, `ExtensionMethodsPlusFiveShould` , to your test project. In order to use the Xunit framework, you'll need to add the `Xunit` using. Each unit test you create needs to have the `[Fact]` attribute assigned to it in order for the test to be picked up by the test runner. Below is the shell for your unit test.

```
using System;
using Xunit;

public class ExtensionMethodsPlusFiveShould
{
    [Fact]
    public void ReturnFiveMoreThanInput()
    {
        //Arrange

        //Act

        //Assert
    }
}
```

You'll notice the comments of `Arrange`, `Act`, and `Assert` which form the structure of a unit test. The `Arrange` section is where you will set up all of your variables, collections of data, etc. that will be used for the test. The `Act` section is for calling the method you're testing with the data you created from the `Arrange` section. Finally, you will `Assert` what you expect to happen. If the behavior of the method matches your expectation (as defined by Assert statements), the test will pass but if not, it will fail. It's a good practice to put the comments in there while you're beginnning to write unit tests, but over time it will become second nature, so the comments will not be necessary.

For this test, you will need to define an input variable and set up an expected result to assert against in the `Arrange` section. The method you are testing against is the `PlusFive` extension method, so this will be the method/action for your `Act` section. Once you have the result from the `Act` section, you can finally assert that the `expectedResult` matches the `actualResult`.

Initially, it is a good idea to Assert something you don't expect to be true becuase sometimes your test will pass under incorrect conditions, revealing a problem in the test, or in the code. Before you run a passing test set your `expectedResult` to be `0`, a value you know is incorrect.

```
using System;
using Xunit;

public class ExtensionMethodsPlusFiveShould
{
    [Fact]
    public void ReturnFiveMoreThanInput()
    {
        //Arrange
        int input = 10;
        int expectedResult = 0;

        //Act
        int actualResult = input.PlusFive();

        //Assert
        Assert.Equal(expectedResult, actualResult);
    }
}
```

Note

Xunit, like most testing frameworks, has many built in assertion methods for you to use. You can see `Assert.Equal` used above, but there are many more assertions available to you that you can check out here. Once the test is written you can run the `test` command from the dotnet CLI (`dotnet test`). This will produce output like the following.

```
xUnit.net .NET CLI test runner (64-bit .NET Core win10-x64)
  Discovering: test
  Discovered:  test
  Starting:    test
    ConsoleApplication.ExtensionMethodsPlusFiveShould.ReturnFiveMoreThanInput [FAIL]
```

```
      Assert.Equal() Failure
      Expected: 0
      Actual:   15
      Stack Trace:
        C:\dev\test\Program.cs(44,0): at ConsoleApplication.ExtensionMethodsPlusFiveShould.R
  Finished:    test
=== TEST EXECUTION SUMMARY ===
   test  Total: 1, Errors: 0, Failed: 1, Skipped: 0, Time: 0.172s
SUMMARY: Total: 1 targets, Passed: 0, Failed: 1.
```

Now that you have seen the test fail you can update the `expectedResult` to be 15, which you know is the correct result, and rerun the test.

```
using System;
using Xunit;

public class ExtensionMethodsPlusFiveShould
{
    [Fact]
    public void ReturnFiveMoreThanInput()
    {
        //Arrange
        int input = 10;
        int expectedResult = 15;

        //Act
        int actualResult = input.PlusFive();

        //Assert
        Assert.Equal(expectedResult, actualResult);
    }
}
```

Running the test again will produce the following output.

```
xUnit.net .NET CLI test runner (64-bit .NET Core win10-x64)
  Discovering: test
  Discovered:  test
  Starting:    test
  Finished:    test
=== TEST EXECUTION SUMMARY ===
   test  Total: 1, Errors: 0, Failed: 0, Skipped: 0, Time: 0.192s
SUMMARY: Total: 1 targets, Passed: 1, Failed: 0.
```

Notice in the Summary, you have 1 test that Passed and 0 tests that Failed. Congratulations! You created your first unit test!

##Unit Testing an Instance Method

The above example shows how to create a unit test for an extension method, so how does this work for an instance method? The concept is nearly identical, except you'll need to create an instance of the class where your method is. Let's move our `PlusFive` method to a different, non-static, class.

```
public class Utilities
{
    public int PlusFive(int input)
    {
        return input + 5;
    }
}
```

In your test you will need to update the method to create an instance of the `Utilities` class above, and pass in the input.

```
[Fact]
public void ReturnFiveMoreThanInput()
{
    //Arrange
    int input = 10;
    var utilities = new Utilities();
    int expectedResult = 15;

    //Act
    int actualResult = utilities.PlusFive(input);

    //Assert
    Assert.Equal(expectedResult, actualResult);
}
```