## **Unit Testing with NUnit and .NET Core**

Effective unit testing is a cornerstone of building reliable and maintainable software. By isolating and testing the smallest testable parts of an application, developers can catch defects early, ensure code correctness, and gain confidence in their codebase as it evolves.

### **Step 1: Creating the Source Project**

Our initial step involves establishing a simple class library that will house the core logic we intend to test. This project will be named PrimeService. Follow these commands in your terminal:

1. **Create a solution directory:**

| mkdir unit-testing-using-nunit cd unit-testing-using-nunit |
| --- |

This command establishes a new folder to contain our entire solution.

1. **Create the solution file:**

| dotnet new sln |
| --- |

A solution file (.sln) acts as a container for one or more projects, helping organize our development efforts.

1. **Create a directory for the PrimeService project:**

| mkdir PrimeService cd PrimeService |
| --- |

1. **Create the class library project:**

| dotnet new classlib |
| --- |

This command generates a new C# class library project.

1. Rename Class1.cs and define PrimeService:  
   Rename the default Class1.cs file to PrimeService.cs. Update its content as follows. Note that the IsPrime method initially throws a NotImplementedException, adhering to a test-driven development (TDD) mindset where tests are written before the functional code.

| using System;  namespace Prime.Services {  public class PrimeService  {  public bool IsPrime(int candidate)  {  throw new NotImplementedException("Please create a test first.");  }  } } |
| --- |

1. **Add the PrimeService project to the solution:**

| cd .. dotnet sln add PrimeService/PrimeService.csproj |
| --- |

1. This command registers our PrimeService project within the solution file.

### **Step 2: Creating the Test Project**

Having established our source project, we will now set up the dedicated test project, utilizing NUnit as our testing framework.

1. **Create a directory for the test project:**

| mkdir PrimeService.Tests cd PrimeService.Tests |
| --- |

1. **Create a new NUnit test project:**

| dotnet new nunit |
| --- |

This command scaffolds a new NUnit test project, automatically including the necessary NuGet packages: NUnit itself, the NUnit3TestAdapter (which enables test discovery and execution), and Microsoft.NET.Test.Sdk.

1. **Add a project reference from the test project to the source project:**

| dotnet add reference ../PrimeService/PrimeService.csproj |
| --- |

This crucial step allows our test project to access and test the code within the PrimeService class library.

1. **Add the test project to the solution:**

| cd .. dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj |
| --- |

Our test project is now also registered within the solution file.

At this juncture, your solution directory structure should logically resemble the following:

| /unit-testing-using-nunit  unit-testing-using-nunit.sln  /PrimeService  PrimeService.csproj  PrimeService.cs  /PrimeService.Tests  PrimeService.Tests.csproj  UnitTest1.cs |
| --- |

### **Step 3: Creating the First Test**

With both projects established, we are now ready to compose our inaugural unit test. We will focus on the IsPrime method of the PrimeService class, specifically testing its behavior for the input 1.

1. Rename UnitTest1.cs and update its content:  
   Rename PrimeService.Tests/UnitTest1.cs to PrimeService\_IsPrimeShould.cs. Replace its default content with the following NUnit test code:

| using NUnit.Framework; using Prime.Services;  namespace Prime.UnitTests.Services {  [TestFixture] // Marks the class as containing unit tests  public class PrimeService\_IsPrimeShould  {  private PrimeService \_primeService;   [SetUp] // This method is executed before each test method in the fixture  public void SetUp()  {  \_primeService = new PrimeService(); // Initialize the service instance for each test  }   [Test] // Marks this method as a unit test  public void IsPrime\_InputIs1\_ReturnFalse()  {  // Arrange (setup the test conditions)  var result = \_primeService.IsPrime(1);   // Assert (verify the outcome)  Assert.That(result, Is.False, "1 should not be prime");  }  } } |
| --- |

1. **Explanation of NUnit Attributes and Assertions:**
   * [TestFixture]: This attribute decorates a class, signaling to the NUnit test runner that it contains test methods.
   * [SetUp]: Methods adorned with this attribute are executed once before each individual test method within the TestFixture. This is ideal for common initialization tasks, such as creating an instance of the class under test.
   * [Test]: This attribute marks a method as a test case that the NUnit runner should execute.
   * Assert.That: This is NUnit's flexible assertion syntax. Assert.That(actual, constraint, message) verifies that the actual value satisfies a specified constraint. Here, Is.False is the constraint, and the string message provides context on failure.
2. Run the test:  
   Navigate to the solution directory (unit-testing-using-nunit) and execute the following command:

| dotnet test |
| --- |

1. As expected, this test will initially **fail**. This is because our IsPrime method currently throws a NotImplementedException, which is the correct behavior for our test-driven approach.

### **Step 4: Implementing the First Feature**

Now, we will implement the minimal code required to make our first test pass, adhering to the "Red-Green-Refactor" cycle of TDD.

1. Update PrimeService.cs:  
   Modify the IsPrime method in PrimeService.cs to handle the candidate == 1 case:

| public bool IsPrime(int candidate) {  if (candidate == 1)  {  return false; // Correctly handles the test case  }  throw new NotImplementedException("Please create a test first."); // Remaining logic unimplemented } |
| --- |

1. Run the test again:  
   From the solution directory, execute:

| dotnet test |
| --- |

1. This time, the IsPrime\_InputIs1\_ReturnFalse test should **pass**, confirming that our implementation correctly handles the specified scenario.

### **Step 5: Adding More Features**

With our initial test passing, we can now expand our test coverage to include additional cases and then update the PrimeService implementation to satisfy these new requirements.

1. Add [TestCase] attributes to PrimeService\_IsPrimeShould.cs:  
   We will leverage NUnit's [TestCase] attribute to define multiple input values for a single test method, making our test suite more concise and expressive. Add the following to your PrimeService\_IsPrimeShould class:

| // ... existing code ...  [TestCase(-1)] [TestCase(0)] [TestCase(1)] public void IsPrime\_ValuesLessThan2\_ReturnFalse(int value) {  // Arrange  var result = \_primeService?.IsPrime(value); // Use null-conditional operator for safety, though \_primeService is initialized by [SetUp]   // Assert  Assert.That(result, Is.False, $"{value} should not be prime"); }  // ... rest of the class ... |
| --- |

This test method will now be executed three times, once for each value specified by [TestCase], asserting that all inputs less than 2 are correctly identified as non-prime.

1. Update the IsPrime implementation in PrimeService.cs:  
   Modify the method to cover the expanded range of non-prime numbers less than 2:

| public bool IsPrime(int candidate) {  if (candidate < 2) // Handles -1, 0, and 1  {  return false;  }  throw new NotImplementedException("Please create a test first."); // Still awaits full prime logic } |
| --- |

1. Run the tests again:  
   Execute dotnet test from the solution directory.  
     
    All tests should now **pass**. This demonstrates an iterative development process: write a failing test, write minimal code to make it pass, then repeat for new functionalities.

### **Step 6: Finalizing and Organizing Tests**

As your project grows and IsPrime gains its full functionality (e.g., handling true prime numbers, composite numbers, etc.), it is imperative to continue adding comprehensive test cases. For instance, you would add [TestCase] attributes for prime numbers (2, 3, 5, 7) and composite numbers (4, 6, 8, 9, 10).

Maintaining a well-organized test suite is critical. Group related tests within logical TestFixture classes, and ensure test method names are descriptive, clearly stating the scenario being tested and the expected outcome. Unit tests serve as living documentation and a safety net, guaranteeing the correctness of your code and preventing regressions as changes are introduced.

### **Conclusion**

This practical guide has provided a comprehensive introduction to unit testing in C# using NUnit within the .NET Core environment. You have successfully navigated the process of:

* Establishing both a source class library and a dedicated NUnit test project.
* Composing your initial unit test and understanding the fundamental NUnit attributes and assertion mechanisms.
* Implementing the minimal code required to satisfy a failing test, following the principles of Test-Driven Development.
* Expanding your test coverage by leveraging NUnit's [TestCase] attribute for efficient testing of multiple input scenarios.
* Recognizing the importance of ongoing test organization and expansion as your application evolves.