Unit Testing - let's do this!

In a previous article we explored the why of Unit Testing. We discussed the main benefits and outlined reasons you and your team should be using Unit Testing as one aspect of your overall Testing Strategy. Implementing Unit Testing gives you benefits, such as having confidence in code, striving for better coding practices, and producing improved code. Let's continue and dive a little deeper to understand Unit Testing and how we might begin coding with unit tests.

There are some basic principles that help make Unit Testing effective. An individual unit test should be:

- Fast: tests should run quickly as fast feedback is important – we are talking milliseconds!
- Isolated: tests should only run unit code and have no external dependencies. For example, they shouldn't include database connections or API calls to outside services.
- Repeatable and consistent: each run of the same test should produce the same result. Inconsistent results make debugging and code fixes difficult.
- Reliable and meaningful: the tests are code and as such should be production ready like the rest of your code base. They need to be maintained going forward.
- Readability: tests should be easy to understand and maintain.
- Self-checking: tests should automatically pass or fail without human interaction.

With these principles in mind, what do we write as far as code is concerned? What does a unit test look like?

Tests typically include a setup stage (which may be a separate method), an execution stage and an assertion stage. They are often expressed as "Arrange", "Act, and "Assert". An esteemed colleague introduced the steps as "Given", "When", and "Then" which is a bit more generic and informal.

Whichever suits you, these steps help give unit tests a consistent and predictable structure.

- Arrange/Given - arrange the conditions, create objects, or setup mocks to use for the test.
- Act/When - execute the method being tested.
- Assert/Then - assert on a test of the execution result – this is the actual test!

Let's look at a simple example in code that demonstrates this predictable structure, in C# using the NUnit framework:


```
using NUnit.Framework;

namespace Example
{
    public static class AdditionExample
    {
        private static int AddIntegers(int a, int b)
        {
            return a + b;
        }
    }
}

namespace Example
{
    [TestFixture]
    public class TestAdditionExample
    {
        [Test]
        public void TestAddIntegers()
        {
            // Given
            int a = 1;
            int b = 2;

            // When
            int result = AdditionExample.Add(a, b);

            // Then
            Assert.AreEqual(3, result);
        }
   }
}
```

So, here's a unit test that's going to ensure that our simple function can add two numbers and gives the expected result. The TestAddIntegers method has:

- The "Given" which sets up two integer variables that will be added by the "Add" method.
- The "When" is calling or executing the "Add" method being tested.
- The "Then" asserts that the result is what was expected. If this passes, the method works as expected. If it fails, there's a problem to investigate.

This is a structure you can follow with your own code and start implementing in a solid and productive way.

Chances are you'll already have code that you could write a unit test for, code that isn't tightly coupled or with deep dependencies to the rest of your code. Some of this code could be at the backbone of important internal or business logic. These are usually named "utility" methods and can

be the target to get started. Utility methods that are discrete, not coupled to any other module, have a single responsibility, accept a small number of parameters, and give a simple result are perfect subjects for your first unit test.

Let's take another example - a string formatter for building a web address where the C# code is as follows:

```
public static class RequestUtilities
{
    public static string GetFullUrl(string baseurl, string route)
    {
        var baseUrl = baseurl.TrimEnd('/');
        var relativeUrl = route.TrimStart('/');

        return $"{baseUrl}/{relativeUrl}";
    }
}
```

This method has no dependencies (hence being static), takes two string parameters and has a simple return value – there's a very good chance this type of method is lurking in your code base ready for unit testing.

Taking this method, we can write a positive assertion test on it passing:

```
    [Test]
    public void TestAddIntegers()
    {
        // Given
        string baseurl = "https://www.testapi.com/";
        string route   = "/api/testroute/";
        string expectedUrl = "https://www.testapi.com/api/testroute/";

        // When
        string result = RequestUtilities.GetFullUrl(baseurl, route);

        // Then
        Assert.AreEqual(expectedUrl, result);
    }
```

This is a simple example, but hopefully you can see how you'd write other tests with and without the starting or trailing forward slashes to try and break the code in "GetFullUrl". (Using NUnit, as we are here, you could use the "TestCase" attribute to compress and write several tests together).

Now you have a start: Let's now say we've identified some of these methods and we've written tests to cover their logic. Placing these tests into a separate project or solution now gives you a suite of unit tests.

These can be run fast locally, so it's quick and easy to check and validate your code. The project can be built and run in a step of our Automated Continuous Integration Builds. With that completed, you can now tackle more code and add more tests.

Unit Testing is one part of the puzzle of testing software, and it can add to your development confidence. Hopefully these small examples give you an idea on how to start. With time and practice it becomes easier to develop and clearer to see how Unit Testing is important for day-to-day coding with measurable benefits.

If your business would like help with Unit Testing, or coding in general, contact a Fujitsu Data & AI specialist now.

