**Data Driven Unit Test**

**Introduction**

Unit testing is one of the most important step in any software development life cycle, however most of the times we developer do not pay much attention to it. Although there are different unit testing frameworks available to effectively and easily write unit test for the code modules, some of the features are still not widely used during unit testing. One of such feature is data driven unit testing. In this article, we will cover –

* What is data driven unit test?
* What are its advantages over regular unit testing technique?
* How to implement it?

**What is data driven unit test?**

Data driven unit test is a way to setup a unit test method to retrieve values from external source like database / XML file / CSV file etc., to test the method. The unit test method then runs for each row in the external data source which makes it easy to test variety of inputs using a single method.

**Data driven unit test advantages**

1. The most important advantage of writing data driven unit test is testing variety of inputs using a single unit test method easily. Let’s take a simplest example of **Calculator** class which exposes two methods – **Add** & **Subtract**. Now, it’s obvious that you need to test these methods for different inputs like positive numbers, negative numbers, large numbers etc. You can either write different unit test methods for each of these scenarios, however doing this way adds a lot of noise to your code and it also makes code file unnecessarily large. Data driven unit test technique makes it really easy to test such methods by storing inputs in an external file and retrieve it during method execution.
2. Since inputs to system under test can be externalized, overall cost of test automation is cheap as compared to traditional technique.
3. Supports different external data source format like XML / Database / CSV etc. which provides flexibility during implementation.

**How to implement it?**

Before we dive into the implementation details of data driven unit test, I assume that you have basic understanding of unit testing and how to create unit test project in Visual Studio. If you are not familiar with it, I strongly recommend you to read [Writing Unit Tests for the .NET Framework](http://msdn.microsoft.com/en-us/library/hh598960.aspx) before reading further.

Since the focus of this article is on data driven unit testing, first we will write few unit tests without the data driven approach and then we will refactor same code using data driven approach. I hope this will help you to understand its advantages clearly.

Just to avoid any complexity and focus on the approach, we will take very simple example of Calculator class and one single method – Add. Below defined code contains the complete Calculator class definition.

public class Calculator

{

    public int Add(int firstNumber, int secondNumber)

    {

        return firstNumber + secondNumber;

    }

}

Now, let’s add few unit tests to test the Add function declared above. Since we need to test this function for different inputs, we need to define separate unit tests for each of the data inputs. Below defined code contains four unit tests which passes different input values to Add function and asserts its output.

[TestMethod]

public void Add\_Two\_Integer\_Returns\_Integer()

{

    var calculator = new Calculator();

    var result = calculator.Add(10, 20);

    Assert.AreEqual(30, result);

}

[TestMethod]

public void Add\_Two\_Zeroes\_Returns\_Zero()

{

    var calculator = new Calculator();

    var result = calculator.Add(0,  0);

    Assert.AreEqual(0, result);

}

[TestMethod]

public void Add\_Positive\_And\_Nagative\_Numbers()

{

    var calculator = new Calculator();

    var result = calculator.Add(20,  -10);

    Assert.AreEqual(10, result);

}

[TestMethod]

public void Add\_Two\_Nagative\_Numbers()

{

    var calculator = new Calculator();

    var result = calculator.Add(-20,  -10);

    Assert.AreEqual(-30, result);

}

If you observe carefully, most of the code in above unit tests is duplicate except the input values. Now, you might think just for the simple Calculator class we had to implement four unit tests and that too did not cover all the different combinations of input values, in a real world application writing such unit tests would be very time consuming and difficult to maintain. Well, that’s where data driven test approach shines.

Now, let’s refactor above source code using data driven approach. As I explained earlier, we need to store different input values into an external data source. In this case we are storing it in an XML file [Tests.xml] as shown below. You have full freedom to define the structure of the XML file and don’t have to stick with the XML attribute naming convention like add or firstNumber etc.

<tests>

  <add>

    <firstNumber>10</firstNumber>

    <secondNumber>100</secondNumber>

    <expected>110</expected>

  </add>

  <add>

    <firstNumber>100</firstNumber>

    <secondNumber>1000</secondNumber>

    <expected>1100</expected>

  </add>

  <add>

    <firstNumber>3</firstNumber>

    <secondNumber>4</secondNumber>

    <expected>7</expected>

  </add>

  <add>

    <firstNumber>0</firstNumber>

    <secondNumber>0</secondNumber>

    <expected>0</expected>

  </add>

</tests>

Now let’s write the unit test which will read this XML file and will execute the code for each of the input values [row] stored in the XML file.

First thing you need to do is to define a TestContext class instance in your project, which allows you to extract the input values from external data source, XML file in this case. Next, we you need to decorate the unit test method with a DataSource attribute.

The DataSource attribute specifies the connection string for the data source and the name of the table that you use in the test method. The exact information in the connection string differs, depending on what kind of data source you are using. In this example, we used a XML file. DataSource attribute has different overloaded constructors, the one we are using below accepts provider information, name of XML file, table name and data access method. Note that each ‘add’ block in the XML file is represented as table, hence we have passed ‘add’ as third argument to DataSource attribute.

Next, we are iterating over the XML file using TestContext.DataRow attribute to get the value of firstNumber, secondNumber and expected XML attribute and executing the add method from Calculator class. In the last statement, we are simply asserting the expected and result values. Note that, since we have different input values specified in the XML file, Run\_Data\_Driven\_Test tests actually gets executed for all those inputs.

public class CalculatorTests

{

public TestContext TestContext { get; set; }

[TestMethod]

[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML", "Tests.xml",  "add", DataAccessMethod.Sequential)]

public void Run\_Data\_Driven\_Test()

{

var calculator = new Calculator();

   var first = int.Parse(TestContext.DataRow["firstNumber"].ToString());

   var second = int.Parse(TestContext.DataRow["secondNumber"].ToString());

   var expected = int.Parse(TestContext.DataRow["expected"].ToString());

   var result = calculator.Add(first, second);

  Assert.AreEqual(expected, result);

}

}

Now, if you want to test your code for additional data inputs, you just need to make changes to the XML file and rerun the unit test. I hope by now you have realized that advantages of data driven approach over traditional unit testing approach.

**Summary**

In this article we have covered basic concepts of data driven unit test and how it can be useful during application development. We have also covered how to set it up in your .Net project using an XML file as external data source. I hope it was useful to you.

Thanks.