Introduction to Unit Testing

Overview

In this lab, you'll learn about unit testing. Unit tests gives you an efficient way to look for logic errors in the methods of your classes. Unit testing has the greatest effect when it's an integral part of your software development workflow. As soon as you write a function or other block of application code, you can create unit tests that verify the behavior of the code in response to standard, boundary, and incorrect cases of input data, and that verify any explicit or implicit assumptions made by the code. In a software development practice known as test-driven development, you create the unit tests before you write the code, so you use the unit tests as both design documentation and functional specifications of the functionality.

Visual Studio has robust support for unit testing, and also supports deep integration with third-party testing tools. In addition, you can leverage the power of Visual Studio Online to manage your projects and run automated tests on your team's behalf.

Objectives

In this hands-on lab, you will learn how to:

- Create unit tests for class libraries
- Create user applications that are highly testable
- Take advantage of advanced Visual Studio features, such as data-driven testing, code coverage analysis, and Microsoft Fakes
- Create a continuous integration environment that automatically runs unit tests upon file check ins

Prerequisites

The following is required to complete this hands-on lab:

- Microsoft Visual Studio 2013
- A Visual Studio Online account (only required for exercise 4)

Exercises

This hands-on lab includes the following exercises:

1. Creating a project and supporting unit tests
2. Unit testing user applications
3. Advanced unit testing features
4. Continuous integration testing with Visual Studio Online

Estimated time to complete this lab: **60** minutes.

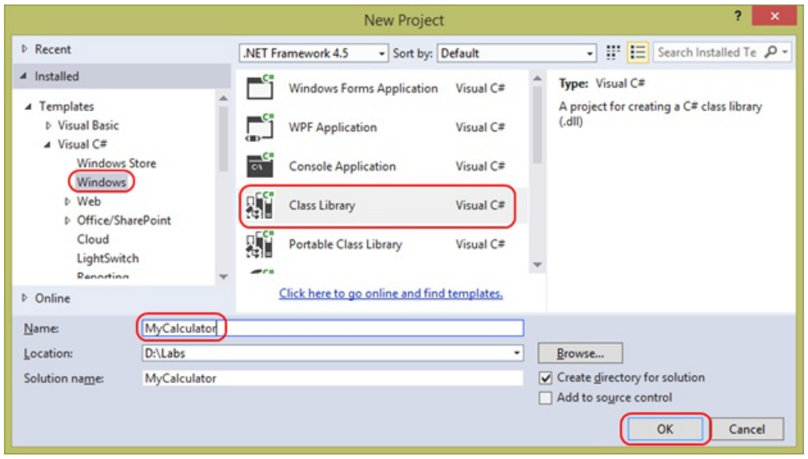
Exercise 1: Creating a project and supporting unit tests

In this exercise, you'll go through the process of creating a new project, as well as some supporting unit tests.

Task 1: Creating a new library

In this task, you'll create a basic calculator library.

1. Open **Visual Studio 2013**.
2. From the main menu, select **File | New | Project**.
3. In the **New Project** dialog, select the **Visual C# | Windows** category and the **Class Library** template. Type **"MyCalculator"** as the **Name** and click **OK**.



.NET Framework 4.5
Sort by: Default
Search Installed Te

Type: Visual C#

A project for creating a C# class library (.dll)

Name: MyCalculator

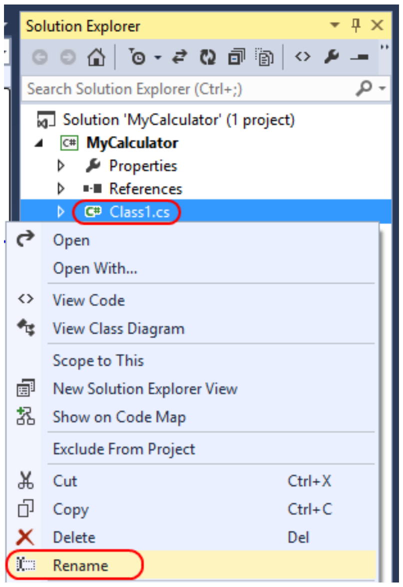
Location: D:\Labs

Solution name: MyCalculator

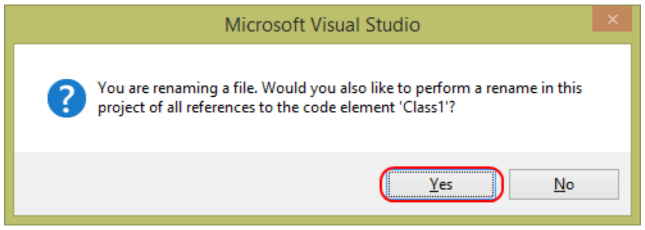
☑ Create directory for solution

☐ Add to source control

4. In **Solution Explorer**, Right-click the **Class1.cs** and select **Rename**. Change the name to **"Calculator.cs"**.



5. When asked to update the name of the class itself, click **Yes**.



You are renaming a file. Would you also like to perform a rename in this project of all references to the code element 'Class1'?

6. Add the following **Add** method to **Calculator.cs**.

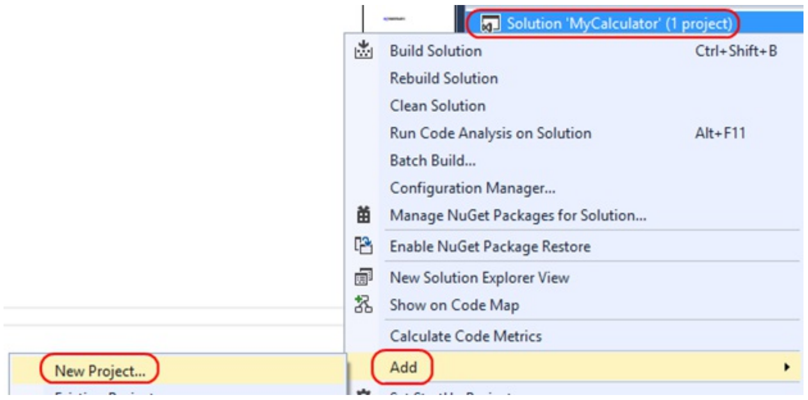
```
public int Add(int first, int second)
{
    return first + second;
}
```

> Note: For the purposes of this lab, all operations will be performed using the **int** value type. In the real world, calculators would be expected to scale to a much greater level of precision. However, the requirements have been relaxed here in order to focus on the unit testing.

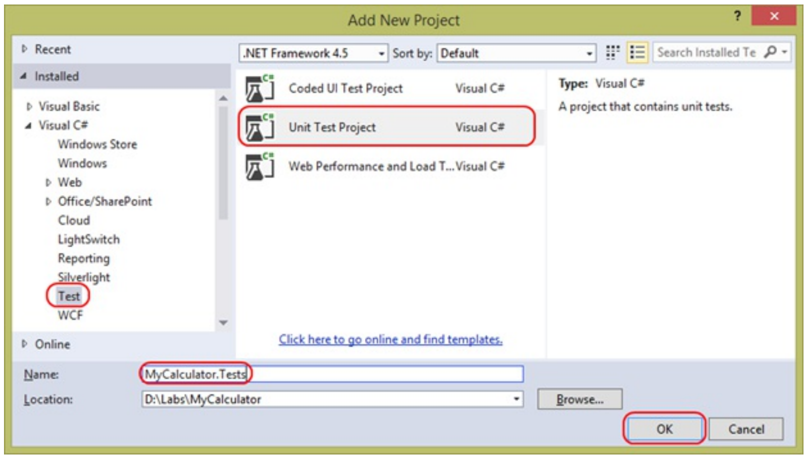
Task 2: Creating a unit test project

In this task, you'll create a new unit test project for your calculator library. Unit tests are kept in their own class libraries, so you'll need to add one to the solution.

1. Right-click the solution node and select **Add | New Project...**.


Solution 'MyCalculator' (1 project)
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Run Code Analysis on Solution Alt+F11
Batch Build...
Configuration Manager...
Manage NuGet Packages for Solution...
Enable NuGet Package Restore
New Solution Explorer View
Show on Code Map
Calculate Code Metrics
New Project...
Add

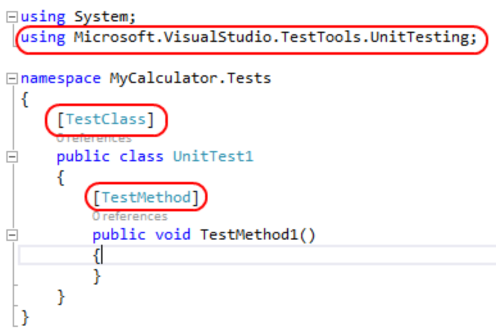
2. Select the **Visual C# | Test** category and the **Unit Test Project** template. Type **"MyCalculator.Tests"** as the **Name** and click **OK**. It's a common practice to name the unit test assembly by adding a **".Tests"** namespace to the name of the assembly it targets.



Type: Visual C#
A project that contains unit tests.

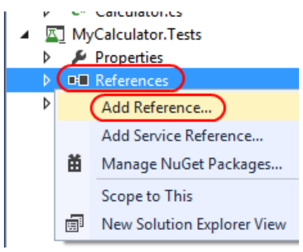
Name: MyCalculator.Tests
Location: D:\Labs\MyCalculator

The wizard will end by opening the default unit test file. There are three key aspects of this class to notice. First, it includes a reference to Visual Studio's unit testing framework. This namespace includes key attributes and classes, such as the **Assert** class that performs value testing. Second, the class itself is attributed with **TestClass**, which is required by the framework to detect classes containing tests after build. Finally, the test method is attributed with **TestMethod** to indicate that it should be run as a test. Any method that is not attributed with this will be ignored by the framework, so it's important to pay attention to which methods do and don't have this attribute.

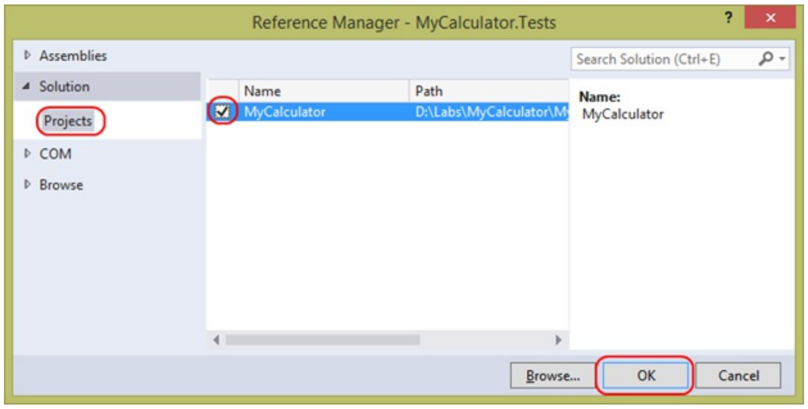


```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace MyCalculator.Tests
{
    [TestClass]
    0 references
    public class UnitTest1
    {
        [TestMethod]
        0 references
        public void TestMethod1()
        {
        }
    }
}
```

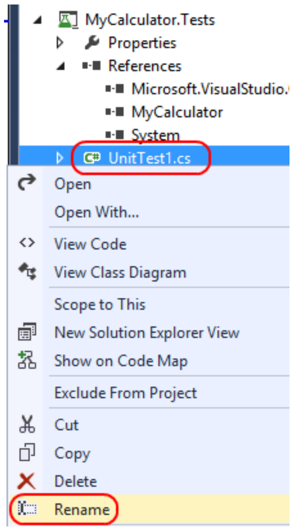
3. The first thing you'll need to do with the unit test project is to add a reference to the project you'll be testing. In **Solution Explorer**, right-click the **References** node of **MyCalculator.Tests** and select **Add Reference…**.



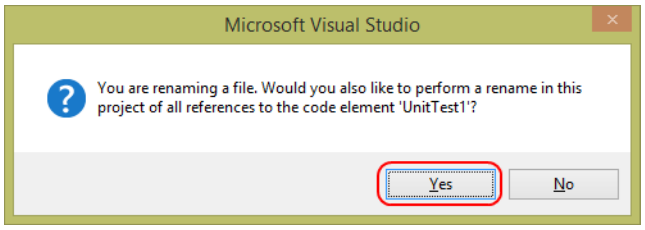
4. Select the **Projects** category in the left pane and check **MyCalculator** in the main pane. Click **OK** to add the reference.



5. To make the project easier to manage, you should rename the default unit test file. In **Solution Explorer**, right-click **UnitTest1.cs** and rename it to **"CalculatorTests.cs"**.



6. When asked to rename the class itself, click **Yes**.



You are renaming a file. Would you also like to perform a rename in this project of all references to the code element 'UnitTest1'?

7. At the top of **CalculatorTests.cs**, add the following **using** directive.

```
using MyCalculator;
```

8. Rename **TestMethod1** to **AddSimple**. As a project grows, you'll often find yourself reviewing a long list of tests, so it's a good practice to give the tests descriptive names. It's also helpful to prefix the names of similar tests with the same string so that tests like **AddSimple**, **AddWithException**, **AddNegative**, etc, all show up together in various views.

```
public void AddSimple()
```

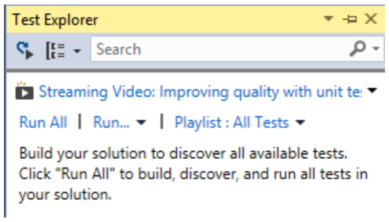
Now you're ready to write the body of your first unit test. The most common pattern for writing unit tests is called **AAA**. This stands for **Arrange, Act, & Assert**. First, initialize the environment. Second, perform the target action. Third, assert that the action resulted the way you intended.

9. Add the following code to the body of the **AddSimple** method.

```
Calculator calculator = new Calculator();
int sum = calculator.Add(1, 2);
Assert.AreEqual(0, sum);
```

This test has only three lines, and each line maps to one of the **AAA** steps. First, the **Calculator** is created. Second, the **Sun** method is called. Third, the **sum** is compared with the expected result. Note that you're designing it to fail at first because the value of **0** is not what the correct result should be. However, it's a common practice to fail a test first to ensure that the test is valid, and then update it to the expected behavior for success. This helps avoid false positives.

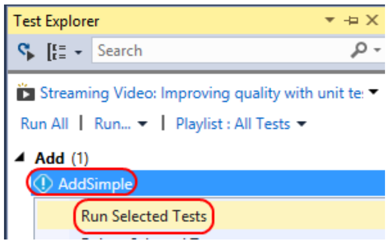
10. From the main menu, select **Test | Windows | Test Explorer**. This will bring up the **Test Explorer**, which acts as a hub for test activity. Note that it will be empty at first because the most recent build does not contain any unit tests.


Test Explorer
Search
Streaming Video: Improving quality with unit te
Run All | Run... | Playlist : All Tests

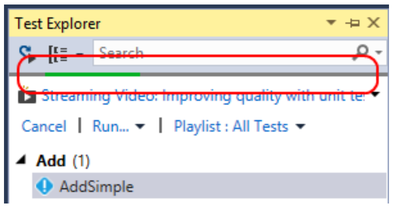
Build your solution to discover all available tests. Click "Run All" to build, discover, and run all tests in your solution.

11. From the main menu, select **Build | Build Solution**.

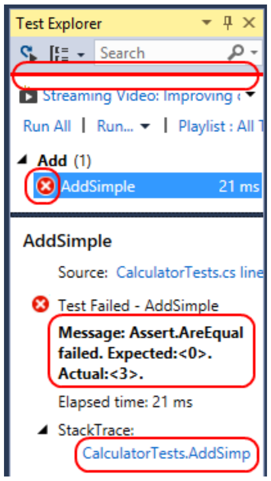
12. The unit test should now appear in **Test Explorer**. Right-click it and select **Run Selected Tests**.


Test Explorer
Search
Streaming Video: Improving quality with unit te
Run All | Run... | Playlist : All Tests
Add (1)
AddSimple
Run Selected Tests

Note that while the test process is running, the progress bar in **Test Explorer** shows an indeterminate green indicator.


Test Explorer
Search
Streaming Video: Improving quality with unit te
Cancel | Run... | Playlist : All Tests
Add (1)
AddSimple

13. However, once the test fails, the bar turns red. This provides a quick and easy way to see the status of your tests. You can also see the status of each individual test based on the icon to its left. Click the test result to see the details about why it failed. If you click the **CalculatorTests.AddSimple** link at the bottom, it will bring you to the method.

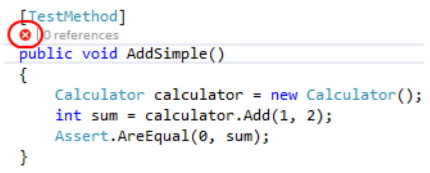

Test Explorer
Search
Streaming Video: Improving
Run All | Run... | Playlist : All T
Add (1)
AddSimple 21 ms
AddSimple
Source: CalculatorTests.cs line
StackTrace:
CalculatorTests.AddSimp

Test Failed - AddSimple

Message: Assert.AreEqual failed. Expected:<0>. Actual:<3>.

Elapsed time: 21 ms

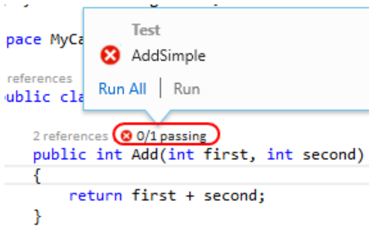
The status indicator also appears in the **CodeLens** directly above the method definition for the test in **CalculatorTests.cs**. Note that **CodeLens** is a feature of Visual Studio Ultimate.



```
[TestMethod]
0 references
public void AddSimple()
{
    Calculator calculator = new Calculator();
    int sum = calculator.Add(1, 2);
    Assert.AreEqual(0, sum);
}
```

And if you switch to **Calculator.cs**, you'll see the unit testing **CodeLens** that indicates the rate of passing (**0/1 passing** here).

14. Click the **CodeLens** to see the results of each test that exercises this method. If you double-click the **AddSimple** test in the **CodeLens** display, it will bring you directly to the test definition.

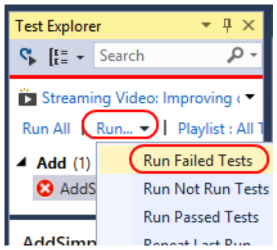


```
pace MyCa
references
ublic cla
    2 references | 0/1 passing
    public int Add(int first, int second)
    {
        return first + second;
    }
```

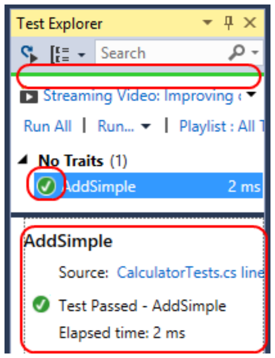
15. In **CalculatorTests.cs**, change the **0** in the **Assert** line to the correct **3**.

```
Assert.AreEqual(3, sum);
```

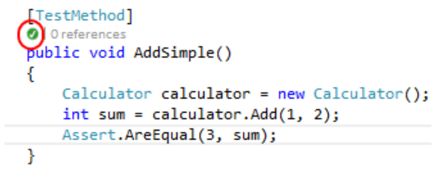
16. In **Test Explorer**, click **Run | Run Failed Tests**. Note that this option only runs the tests that failed in the last pass, which can save time. However, it doesn't run passed tests that may have been impacted by more recent code changes, so be careful when selecting which tests to run.



17. Now that the test passes, the progress bar should provide a solid green. Also, the icon next to the test will turn green. If you click the test, it will provide the basic stats for the test's run.

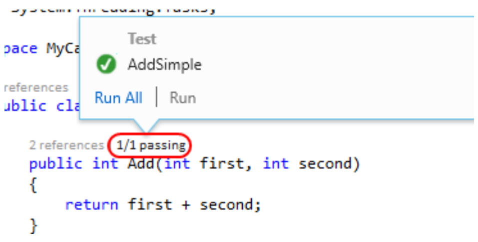


Also note that the **CodeLens** updates above the test's method definition.



```
[TestMethod]
0 references
public void AddSimple()
{
    Calculator calculator = new Calculator();
    int sum = calculator.Add(1, 2);
    Assert.AreEqual(3, sum);
}
```

As well as in the **CodeLens** above the library method being tested in **Calculator.cs**.



```
pace MyCa
references
ublic cla
    2 references | 1/1 passing
    public int Add(int first, int second)
    {
        return first + second;
    }
```

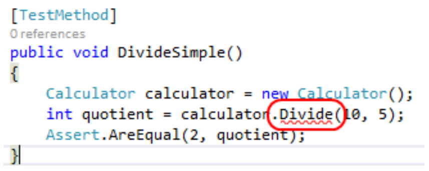
Task 3: Creating a new feature using test-driven development

In this task, you'll create a new feature in the calculator library using the philosophy known as "test-driven development" (TDD). Simply put, this approach encourages that the tests be written before new code is developed, such that the initial tests fail and the new code is not complete until all the tests pass. Some developers find this paradigm to produce great results, while others prefer to write tests during or after a new feature is implemented. It's really up to you and your organization because Visual Studio provides the flexibility for any of these approaches.

1. Add the following method to **CalculatorTests.cs**. It is a simple test that exercises the **Divide** method of the library.

```
[TestMethod]
public void DivideSimple()
{
    Calculator calculator = new Calculator();
    int quotient = calculator.Divide(10, 5);
    Assert.AreEqual(2, quotient);
}
```

2. However, since the **Divide** method has not yet been built, the test cannot be run. However, you can feel confident that this is how it's supposed to work, so now you can focus on implementation.



```
[TestMethod]
0 references
public void DivideSimple()
{
    Calculator calculator = new Calculator();
    int quotient = calculator.Divide(10, 5);
    Assert.AreEqual(2, quotient);
}
```

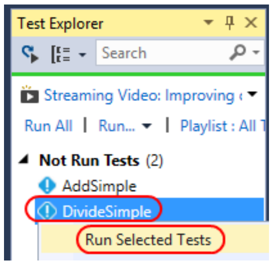
Strictly speaking, the next step should be to implement only the method shell with no functionality. This will allow you to build and run the test, which will fail. However, you can skip ahead here by adding the complete method since it's only one line.

3. Add the **Divide** method to **Calculator.cs**.

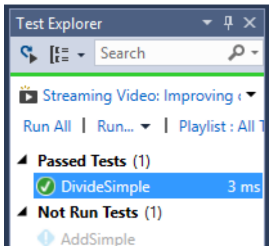
```
public int Divide(int dividend, int divisor)
{
    return dividend / divisor;
}
```

4. From the main menu, select **Build | Build Solution**.

5. In **Test Explorer**, right-click the new **DivideSimple** method and select **Run Selected Tests**.



The test should complete successfully.

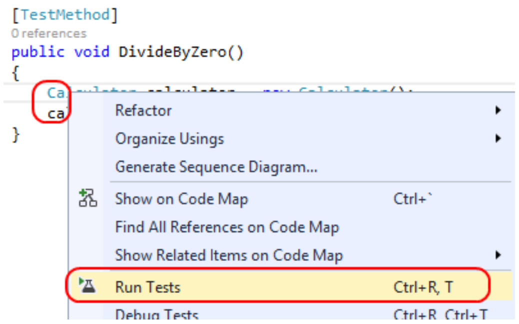


However, there is one edge case to be aware of, which is the case where the library is asked to divide by zero. Ordinarily you would want to test the inputs to the method and throw exceptions as needed, but since the underlying framework will throw the appropriate **DivideByZeroException**, you can take this easy shortcut.

6. Add the following method to **CalculatorTests.cs**. It attempts to divide 10 by 0.

```
[TestMethod]
public void DivideByZero()
{
    Calculator calculator = new Calculator();
    calculator.Divide(10, 0);
}
```

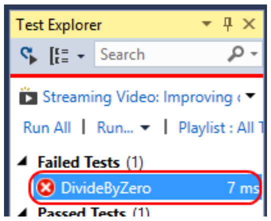
7. Right-click within the body of the test method and select **Run Tests**. This is a convenient way to run just this test.

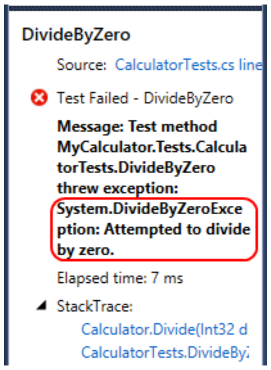


```
[TestMethod]
0 references
public void DivideByZero()
{
    Ca
    ca
}
```

Refactor
Organize Usings
Generate Sequence Diagram...
Show on Code Map Ctrl+`
Find All References on Code Map
Show Related Items on Code Map
Run Tests Ctrl+R, T
Debug Tests Ctrl+R, Ctrl+T

8. The test will run and fail, but that’s expected. If you select the failed test in **Test Explorer**, you’ll see that the **DivideByZeroException** was thrown, which is by design.





DivideByZero

Source: CalculatorTests.cs line

Test Failed - DivideByZero

Message: Test method MyCalculator.Tests.CalculatorTests.DivideByZero threw exception: System.DivideByZeroException: Attempted to divide by zero.

Elapsed time: 7 ms

StackTrace:

Calculator.Divide(Int32 d

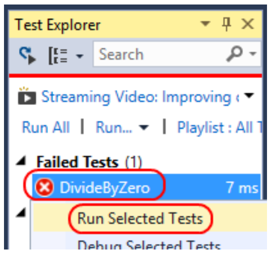
CalculatorTests.DivideBy

However, this is expected behavior, so you can attribute the test method with the **ExpectedException** attribute.

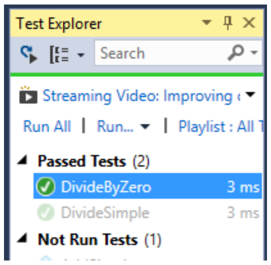
9. Add the following attribute above the definition of **DivideByZero**. Note that it takes the type of exception it’s expecting to be thrown during the course of the test.

```
[ExpectedException(typeof(DivideByZeroException))]
```

10. In **Test Explorer**, right-click the failed test and select **Run Selected Tests**.



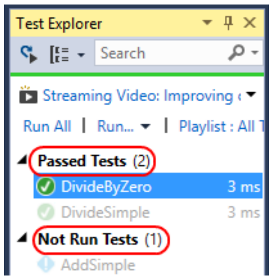
The test should now succeed because it threw the kind of exception it was expecting.



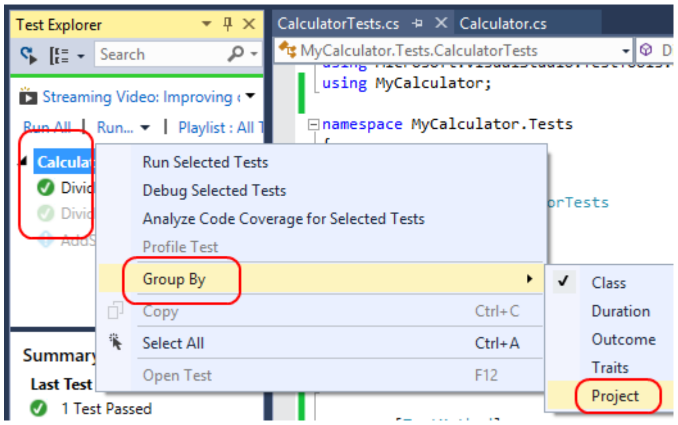
Task 4: Organizing unit tests

In this task, you’ll organize the unit tests created so far using different groupings, as well as create custom traits for finer control over organization.

By default, **Test Explorer** organizes tests into three categories: **Passed Tests**, **Failed Tests**, and **Not Run Tests**. This is useful for most scenarios, especially since developers often only care about the tests that are currently failing.

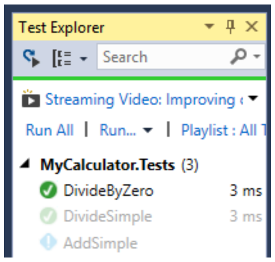


1. However, sometimes it can be useful to group tests by other attributes. Right-click near the tests and select **Group By | Project**.

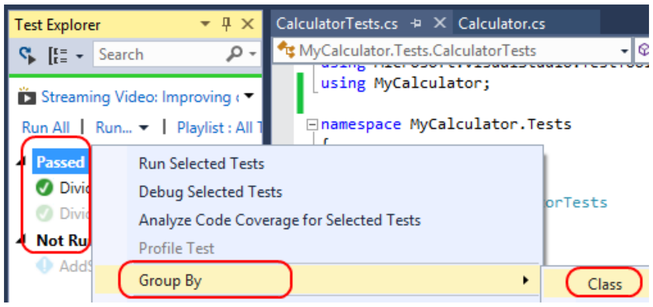


```
using MyCalculator;

namespace MyCalculator.Tests
```

This will organize the tests based on the project they belong to. This can be very useful when navigating huge solutions with many test projects.

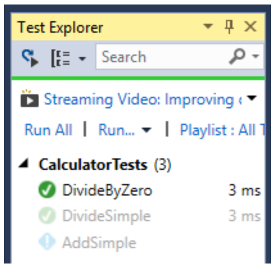


2. Another option is to organize tests by the class they're part of. Right-click near the tests and select **Group By | Class**.

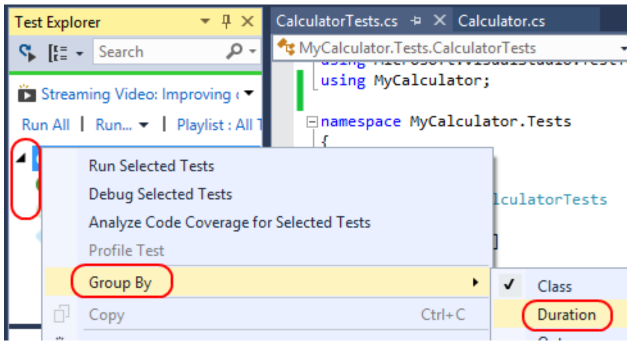


```
using MyCalculator;

namespace MyCalculator.Tests
```

The tests are now grouped by class, which is useful when the classes are cleanly divided across functional boundaries.

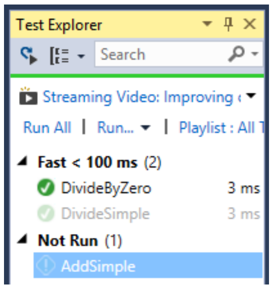


3. Yet another option is to organize tests by how long they take to run. Right-click near the tests and select **Group By | Duration**.

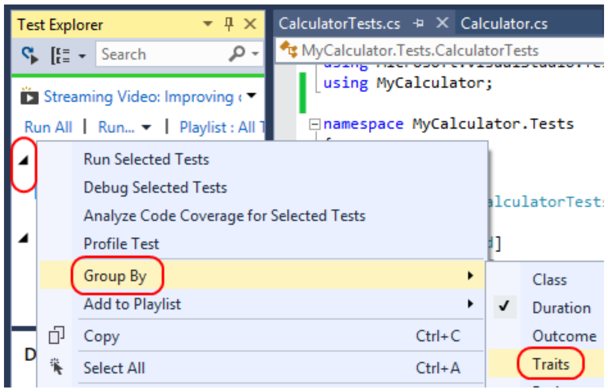


```
using MyCalculator;

namespace MyCalculator.Tests
{
```

Grouping by duration makes it easy to focus on tests that may indicate poorly performing code.

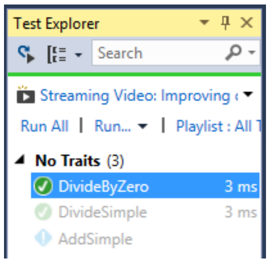


4. Finally, there is another option is to organize tests by their **traits**. Right-click near the tests and select **Group By | Traits**.



```
using MyCalculator;

namespace MyCalculator.Tests
```

By default, a test method doesn't have any traits.

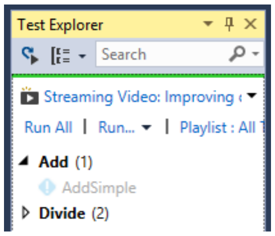


5. However, traits are easy to add using an attribute. Add the following code above the **AddSimple** method. It indicates that this test has the **Add** trait. Note that you may add multiple traits per test, which is useful if you want to tag a test with its functional purpose, development team, performance relevance, etc.

```
[TestCategory("Add")]
```

6. Add **TestCategory** attributes to each of the divide tests, but use the category **Divide**.

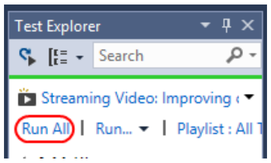
7. From the main menu, select **Build | Build Solution**.



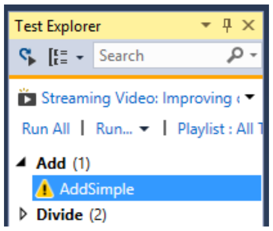
8. Another useful attribute for organizing tests is the **Ignore** attribute. This simply tells the test engine to skip this test was running. Add the following code above the **AddSimple** method.

```
[Ignore]
```

9. In **Test Explorer**, click **Run All** to build and run all tests.



10. Note that **AddSimple** has a yellow exclamation icon now, which indicates that it did not run as part of the last test pass.



There are a few more test attributes that aid in the organization and tracking of unit tests.

- **Description** allows you to specify a description for the test.
- **Owner** specifies the owner of the test.
- **HostType** allows you to specify the type of host the test can run on.
- **Priority** specifies the priority at which this test should run.
- **Timeout** specifies how long the test may run until it times out.
- **WorkItem** allows you to specify the work item IDs the test is associated with.
- **CssProjectStructure** represents the node in the team project hierarchy to which this test corresponds.
- **CssIteration** represents the project iteration to which this test corresponds.

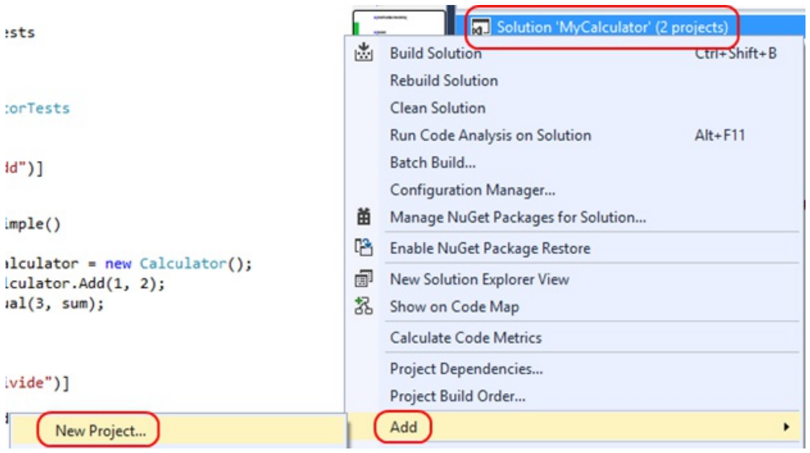
Exercise 2: Unit testing user applications

In this exercise, you'll go through the process of working with unit tests on applications with user interfaces. While it has been historically difficult to build robust tests for applications like these, proper design considerations can drastically improve the testability of virtually any application.

Task 1: Creating a user interface

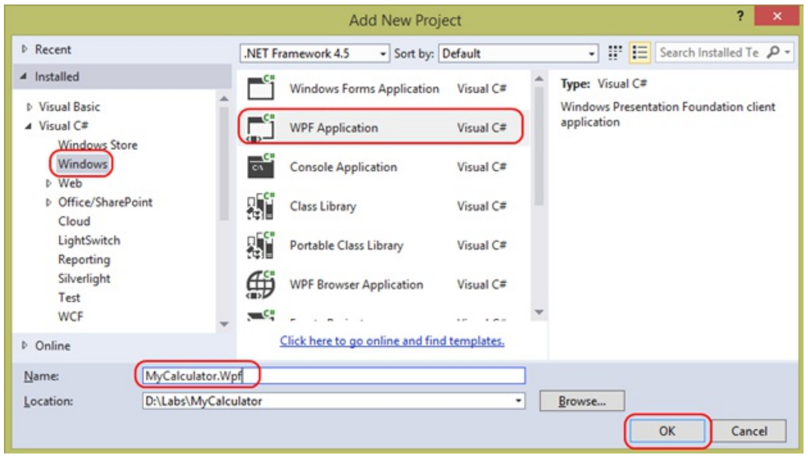
In this task, you'll create a basic user interface for your library. The goal will be to deploy a user interface as quickly as possible, and no special consideration will be given for testability.

1. In **Solution Explorer**, right-click the solution node and select **Add | New Project…**.

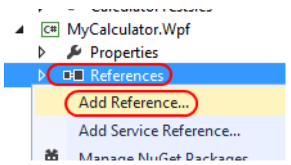

Solution 'MyCalculator' (2 projects)
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Run Code Analysis on Solution Alt+F11
Batch Build...
Configuration Manager...
Manage NuGet Packages for Solution...
Enable NuGet Package Restore
New Solution Explorer View
Show on Code Map
Calculate Code Metrics
Project Dependencies...
Project Build Order...
Add
New Project...

```
:sts

:orTests

ld")]

imple()

ılculator = new Calculator();
lculator.Add(1, 2);
ıal(3, sum);

ivide")]
```

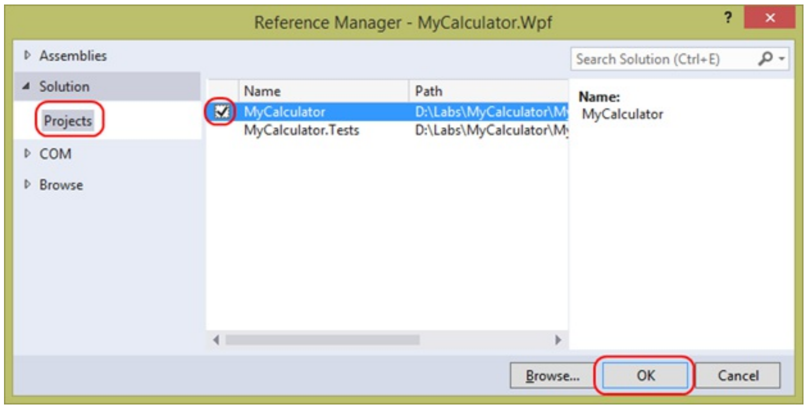
2. Select **Visual C# | Windows** as the category and **WPF Application** as the template. Type **"MyCalculator.Wpf"** as the **Name** and click **OK**.


Add New Project
Recent
Installed
Visual Basic
Visual C#
Windows Store
Windows
Web
Office/SharePoint
Cloud
LightSwitch
Reporting
Silverlight
Test
WCF
Online
.NET Framework 4.5
Sort by: Default
Search Installed Te
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Class Library Visual C#
Portable Class Library Visual C#
WPF Browser Application Visual C#
Click here to go online and find templates.
Type: Visual C#
Windows Presentation Foundation client application
Name: MyCalculator.Wpf
Location: D:\Labs\MyCalculator
Browse...
OK
Cancel

3. The first thing you'll need to do is to add a reference to the **MyCalculator** library. In **Solution Explorer**, right-click the **References** node under **MyCalculator.Wpf** and select **Add Reference…**.



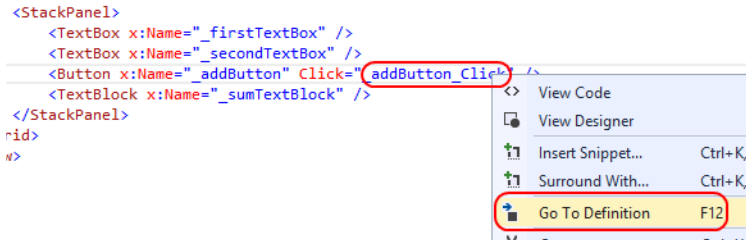
4. Select **Projects** from the left pane and check **MyCalculator** from the assembly list. Click **OK** to add the reference.


Reference Manager - MyCalculator.Wpf
Assemblies
Solution
Projects
COM
Browse
Search Solution (Ctrl+E)
Name
Path
MyCalculator D:\Labs\MyCalculator\M
MyCalculator.Tests D:\Labs\MyCalculator\M
Name:
MyCalculator
Browse...
OK
Cancel

5. The user interface for this calculator application will be very simple with two text boxes, a button, and a text block for the result. Add the following XAML inside the **Grid** node of **MainWindow.xaml**.

```
<StackPanel>
    <TextBox x:Name="_firstTextBox" />
    <TextBox x:Name="_secondTextBox" />
    <Button x:Name="_addButton" Content="Add" Click="_addButton_Click" />
    <TextBlock x:Name="_sumTextBlock" />
</StackPanel>
```

6. The application will access the **MyCalculator** library functionality from the code-behind, so right-click **_addButton_Click** and select **Go To Definition**.

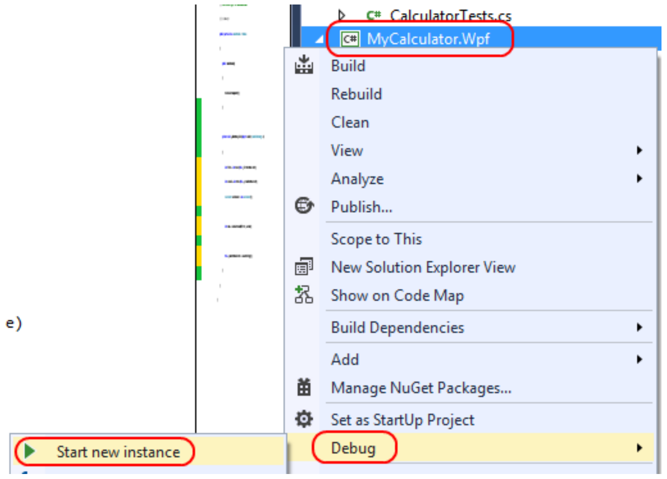


```
<StackPanel>
    <TextBox x:Name="_firstTextBox" />
    <TextBox x:Name="_secondTextBox" />
    <Button x:Name="_addButton" Click="_addButton_Click" />
    <TextBlock x:Name="_sumTextBlock" />
</StackPanel>
rid>
w>
```

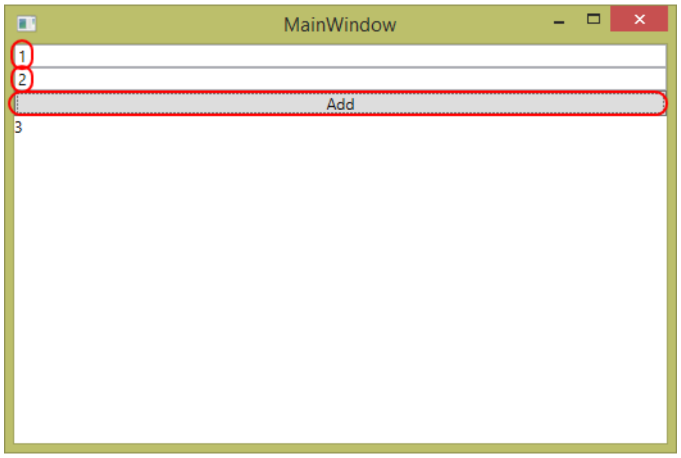
7. Inside **_addButton_Click**, add the following code. It parses the values from the text blocks and uses the calculator library to add them. The result is then placed into the text block.

```
int first = int.Parse(this._firstTextBox.Text);
int second = int.Parse(this._secondTextBox.Text);
Calculator calculator = new Calculator();

int sum = calculator.Add(first, second);

this._sumTextBlock.Text = sum.ToString();
```

8. In **Solution Explorer**, right-click **MyCalculator.Wpf** and select **Debug | Start new instance**. Note that you're only going to run this application once in this lab. If you were running it multiple times, it would be easier to select **Set as StartUp Project** and then press **F5** to run it each time.

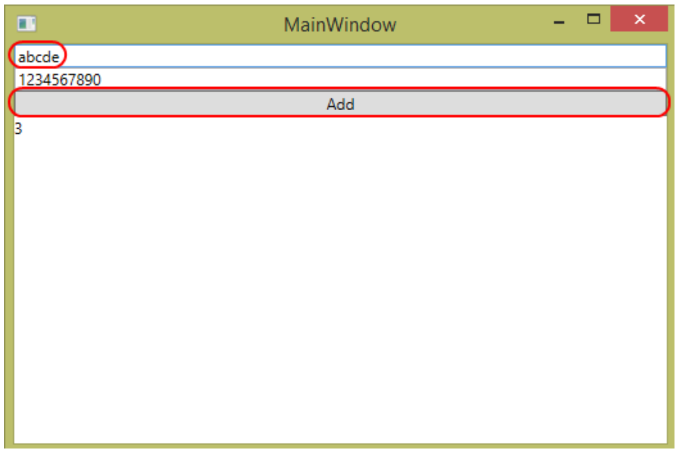


9. The application will build and run. First, use the values **1** and **2** to confirm that **Add** displays **3**. So far so good.

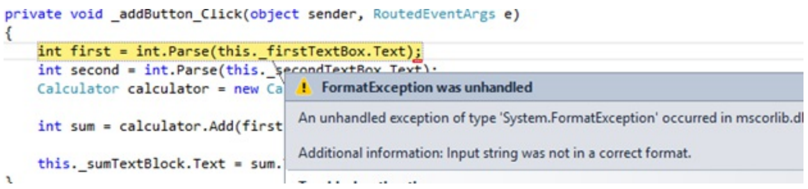


However, while the library supports its functionality fairly well (and you have the unit tests to prove it), there is some untested code that resides in the **MyCalculator.Wpf** project.

10. Replace the value in the first text box with **"abcde"** and click **Add**.



Unfortunately, this results in an exception. In this case, there is a bug in the code-behind. And while you could take the time to fix it, you could never feel high confidence that there aren't other similar bugs lurking under the surface, only to be discovered during actual usage.



```
private void _addButton_Click(object sender, RoutedEventArgs e)
{
    int first = int.Parse(this._firstTextBox.Text);
    int second = int.Parse(this._secondTextBox.Text);
    Calculator calculator = new Ca

    int sum = calculator.Add(first

    this._sumTextBlock.Text = sum.
```

FormatException was unhandled

An unhandled exception of type 'System.FormatException' occurred in mscorlib.dl

Additional information: Input string was not in a correct format.

11. From the main menu of Visual Studio, select **Debug | Stop Debugging** to end the debugging session.

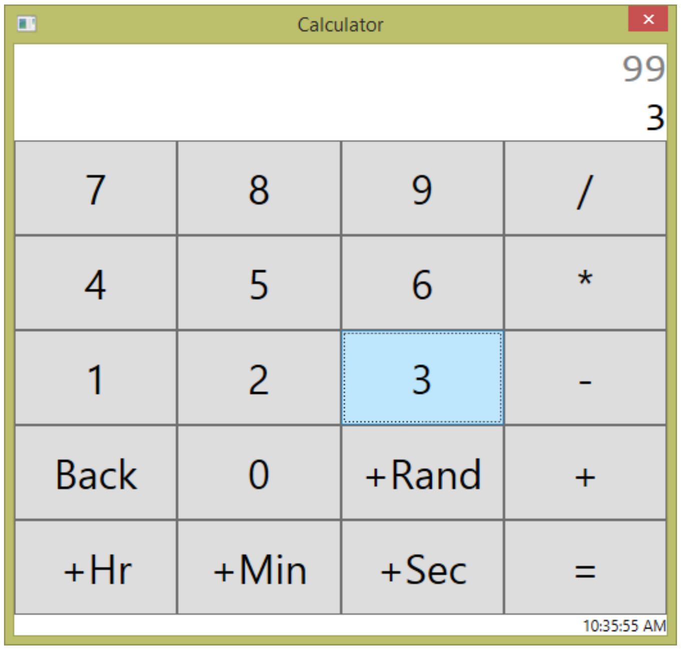
Task 2: Designing apps for testability

In this task, you'll take a look at a similar calculator application that has made use of a Model-View-ViewModel (MVVM) architecture in order to improve testability. Note that MVVM is outside the scope of this lab, so there will not be a focus on its features or guidance on its usage. However, MVVM represents a class of user applications architectures that promote higher quality software through hardened practices and a focus on test support.

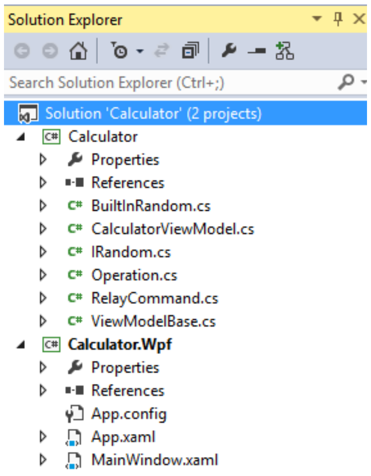
> Note: MVVM is an expansive topic that includes benefits that scale well beyond great testability. For more information, please check out Microsoft's official guidance at http://msdn.microsoft.com/en-us/library/hh848246.aspx.

1. In Visual Studio, open the solution provided with this lab at **Calculator Solution\Calculator.sln**.

2. Press **F5** to build and run the application. It's a simple calculator that supports integer operations. In addition to the basic four functions, it also allows you to add a random number from 1-100 to the current register, as well as add the current time's hours, minutes, or seconds. Take a few moments to play around with the functionality. Close the application when satisfied.

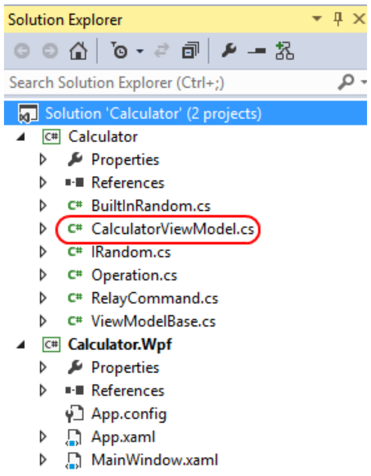


3. In **Solution Explorer**, expand the project nodes to show all top-level files.



This solution has two projects. The **Calculator** project is a class library that contains the logic for the application, and the user interface is contained in **Calculator.Wpf**.

4. Open **CalculatorViewModel.cs**.



In **Calculator**, the main file of interest is the **CalculatorViewModel** class, which is used to perform calculations and communicate with the user interface (from the **Calculator.Wpf** project) via **databinding** and **commanding**.

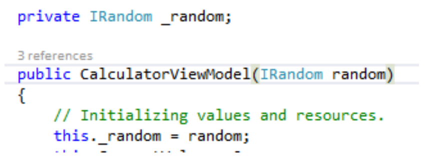
5. Locate the section where the **ICommand** properties are defined.

```
3 references
public ICommand KeyCommand { get; private set; }
6 references
public ICommand AddCommand { get; private set; }
2 references
public ICommand SubtractCommand { get; private set; }
2 references
public ICommand MultiplyCommand { get; private set; }
2 references
public ICommand DivideCommand { get; private set; }
6 references
public ICommand EquateCommand { get; private set; }
2 references
public ICommand BackCommand { get; private set; }
1 reference
public ICommand AddHoursCommand { get; private set; }
1 reference
public ICommand AddMinutesCommand { get; private set; }
2 references
public ICommand AddSecondsCommand { get; private set; }
2 references
public ICommand AddRandomCommand { get; private set; }
```

These properties are all intended to be wired directly up to the user interface. As a result, there is virtually no code-behind. If you want to test what happens when a user clicks a series of buttons (or performs a series of other input actions not leveraged here), you can use the **Execute** method each **ICommand** exposes. Note that this all happens with no user interface present, but it doesn't matter since the **CalculatorViewModel** is abstracted away from the classes using it.

Another important pattern to rely on where possible is known as **Dependency Injection**.

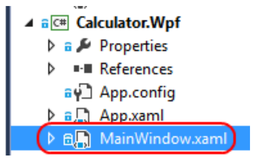
6. Locate the private **IRandom** property, which is just above the constructor.


```
private IRandom _random;

3 references
public CalculatorViewModel(IRandom random)
{
    // Initializing values and resources.
    this._random = random;
```

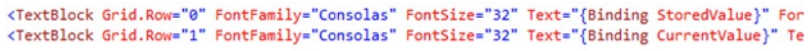
The **IRandom** interface provides an abstraction layer for the library to generate random numbers. While the library itself provides an implementation that uses **System.Random** in **BuiltInRandom**, you could pass in any **IRandom** you want. As a result, you have much more flexibility when it comes to controlling the behavior of the class for isolation and testing purposes.

In **Calculator.Wpf**, virtually all of the functionality is wired up in **MainWindow.xaml**, which binds its user interface (data and buttons) to an instance of **CalculatorViewModel**, which is set in **MainWindow.xaml**.

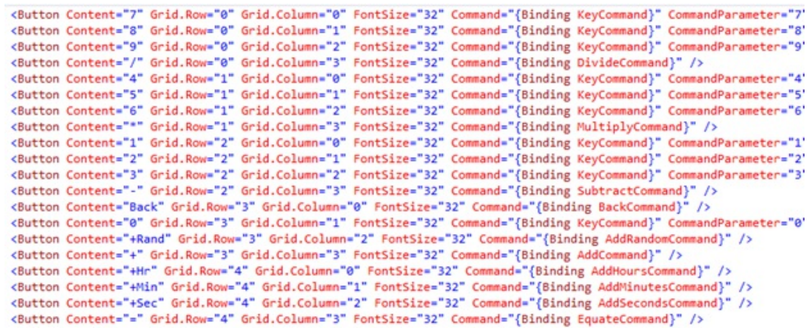
7. From **Solution Explorer**, open **MainWindow.xaml**.



8. Locate the two **TextBlocks** near the top of the file. They display the stored and current registers of the calculator. However, their values are databound to the properties exposed by the **CalculatorViewModel**, so there's no additional code between the XAML and ViewModel.

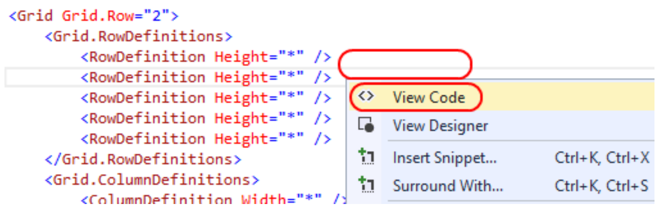

```
<TextBlock Grid.Row="0" FontFamily="Consolas" FontSize="32" Text="{Binding StoredValue}" For
<TextBlock Grid.Row="1" FontFamily="Consolas" FontSize="32" Text="{Binding CurrentValue}" Te
```

9. Locate the long list of **Button** elements.

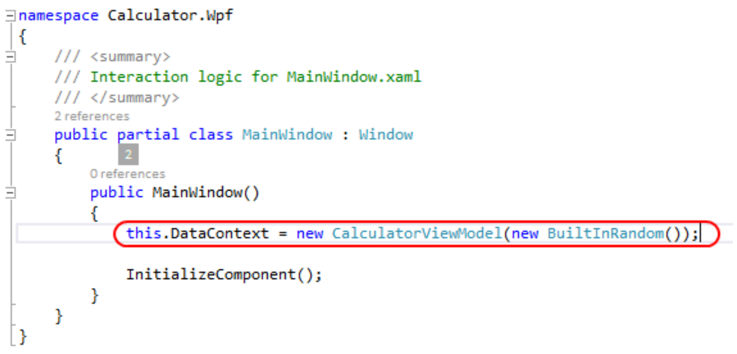

```
<Button Content="7" Grid.Row="0" Grid.Column="0" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="7'
<Button Content="8" Grid.Row="0" Grid.Column="1" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="8'
<Button Content="9" Grid.Row="0" Grid.Column="2" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="9'
<Button Content="/" Grid.Row="0" Grid.Column="3" FontSize="32" Command="{Binding DivideCommand}" />
<Button Content="4" Grid.Row="1" Grid.Column="0" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="4'
<Button Content="5" Grid.Row="1" Grid.Column="1" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="5'
<Button Content="6" Grid.Row="1" Grid.Column="2" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="6'
<Button Content="*" Grid.Row="1" Grid.Column="3" FontSize="32" Command="{Binding MultiplyCommand}" />
<Button Content="1" Grid.Row="2" Grid.Column="0" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="1'
<Button Content="2" Grid.Row="2" Grid.Column="1" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="2'
<Button Content="3" Grid.Row="2" Grid.Column="2" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="3'
<Button Content="-" Grid.Row="2" Grid.Column="3" FontSize="32" Command="{Binding SubtractCommand}" />
<Button Content="Back" Grid.Row="3" Grid.Column="0" FontSize="32" Command="{Binding BackCommand}" />
<Button Content="0" Grid.Row="3" Grid.Column="1" FontSize="32" Command="{Binding KeyCommand}" CommandParameter="0'
<Button Content="+Rand" Grid.Row="3" Grid.Column="2" FontSize="32" Command="{Binding AddRandomCommand}" />
<Button Content="+" Grid.Row="3" Grid.Column="3" FontSize="32" Command="{Binding AddCommand}" />
<Button Content="+Hr" Grid.Row="4" Grid.Column="0" FontSize="32" Command="{Binding AddHoursCommand}" />
<Button Content="+Min" Grid.Row="4" Grid.Column="1" FontSize="32" Command="{Binding AddMinutesCommand}" />
<Button Content="+Sec" Grid.Row="4" Grid.Column="2" FontSize="32" Command="{Binding AddSecondsCommand}" />
<Button Content="=" Grid.Row="4" Grid.Column="3" FontSize="32" Command="{Binding EquateCommand}" />
```

Note how all these buttons have **Commands** set instead of **Click** handlers. These connect directly to the ViewModel as discussed earlier, so there's no additional code to worry about between the XAML and the library you can test directly. Another item to note is the **CommandParameter** property set on the number buttons. These parameters allow reuse of the same command for similar behavior. You could also provide a keyboard hook that uses these same commands to send keys pressed by the user, and almost the entire code path would be the same as used by these buttons.

10. Right-click inside the editor and select **View Code** to navigate to the code-behind file, **MainWindow.xaml.cs**.


```
<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
```
View Code
View Designer
Insert Snippet... Ctrl+K, Ctrl+X
Surround With... Ctrl+K, Ctrl+S

11. There is only one line of code that's been added to this entire project, and it simply sets the **DataContext**. After that, everything is handled between the XAML layer and the ViewModel.



```
namespace Calculator.Wpf
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    2 references
    public partial class MainWindow : Window
    {
        0 references
        public MainWindow()
        {
            this.DataContext = new CalculatorViewModel(new BuiltInRandom());

            InitializeComponent();
        }
    }
}
```

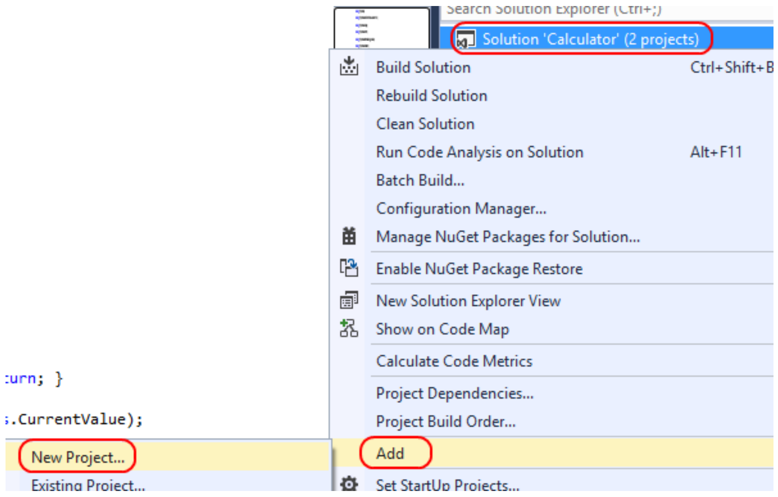
Feel free to take a few minutes to explore the codebase. Keep in mind that the details of the implementation aren't really important in this lab since the focus is on unit testing. The key message is that because this application was designed using MVVM, virtually everything that needs to be tested can be tested in the **Calculator** library, including user behavior such as button clicks.

> Note: Another way to test user interfaces is called **Coded UI Tests**. In a coded UI test, the test machine runs automation that invokes mouse and keyboard actions to test the application. These tests can be used alongside unit tests for greater coverage. For more information on coded UI tests, please visit http://msdn.microsoft.com/en-us/library/dd286726.aspx.

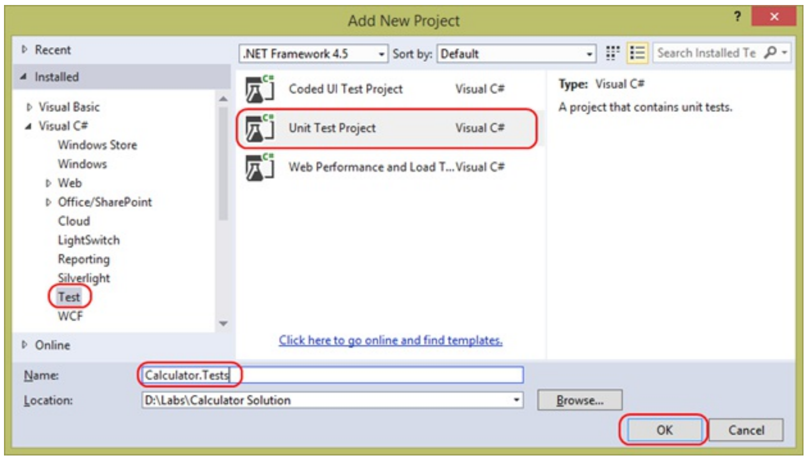
Task 3: Creating tests for the MVVM application

In this task, you'll add a test project and create some more advanced tests for the MVVM application.

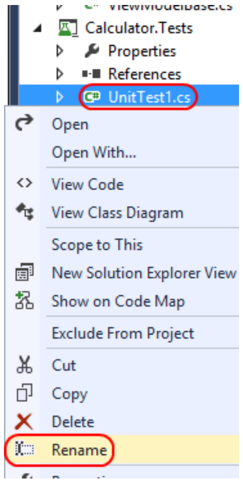
1. In **Solution Explorer**, right-click the solution node and select **Add | New Project…**.



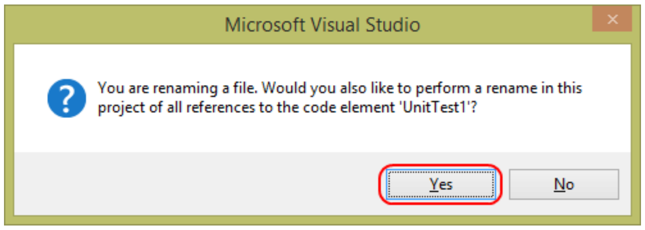
2. Select **Visual C# | Test** as the category and **Unit Test Project** as the template. Type **"Calculator.Tests"** as the **Name** and click **OK**.



3. Right-click **UnitTest1.cs** and select **Rename**.

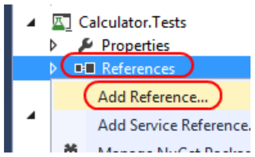


4. Rename the file to **"MathTests.cs"** and click **Enter**. When asked to rename the class itself, click **Yes.**

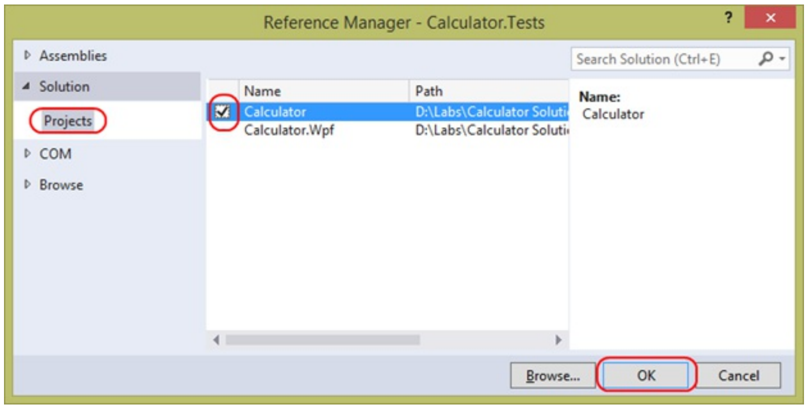

Microsoft Visual Studio

You are renaming a file. Would you also like to perform a rename in this project of all references to the code element 'UnitTest1'?

5. As with the earlier unit test project, you'll need to add a reference to the calculator library being tested. Right-click the **References** node under **Calculator.Tests** and select **Add Reference…**.



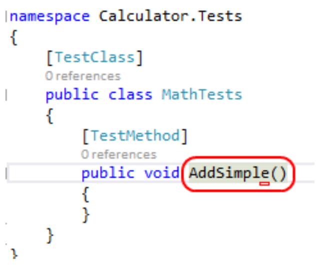
6. Select **Projects** from the left pane and check the **Calculator** assembly. Click **OK** to add a reference to it.



	Name	Path
✓	Calculator	D:\Labs\Calculator Soluti
	Calculator.Wpf	D:\Labs\Calculator Soluti

Name:
Calculator

7. Rename the default **TestMethod1** to **AddSimple**.



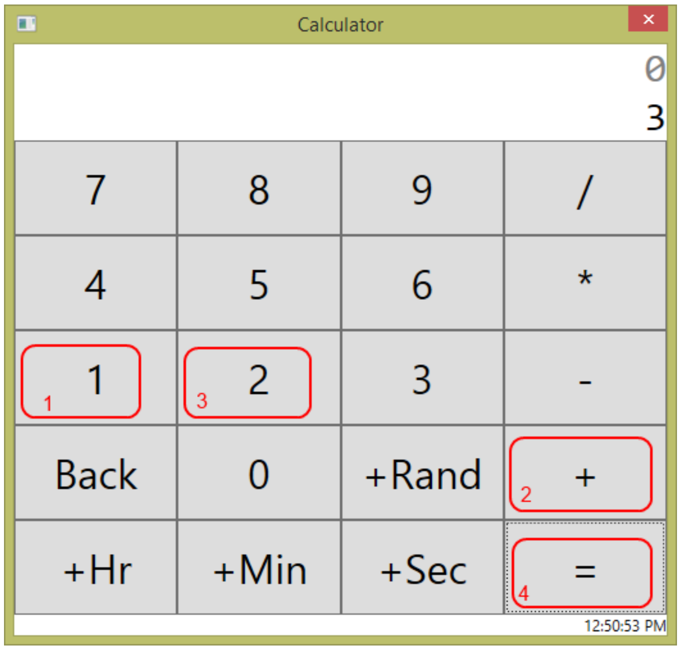
```
namespace Calculator.Tests
{
    [TestClass]
    0 references
    public class MathTests
    {
        [TestMethod]
        0 references
        public void AddSimple()
        {
        }
    }
}
```

Now that everything's set up to start writing the test for **Add** functionality, it's time to write the test itself. As a developer, you'll typically write unit tests for your own code around the same time you write the code itself. However, in this scenario the codebase is inherited without any tests, so you should take a moment to figure out how to interact with the library.

This application uses a fairly strict MVVM approach, so there is no business logic in the WPF codebase. As a result, every button click invokes a command. If you wanted to write a test that closely emulated the way users would interact with the application, you could simply execute those commands as though the unit test were pushing buttons.

8. Press **F5** to run the application.

9. Perform the calculation **1 + 2 =**, which should produce the result **3** as shown.

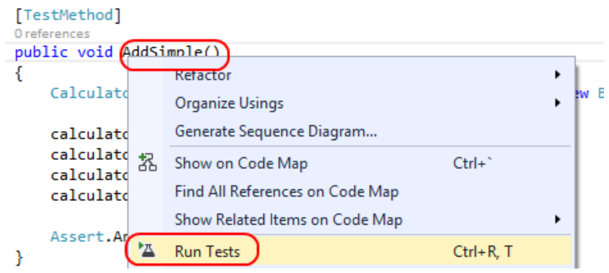


Note that there were four button clicks to get the result. Since all these clicks mapped to buttons in the **CalculatorViewModel** class, you can emulate it exactly.

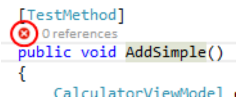
10. Add the following code to the body of **AddSimple**. It'll fail because the sum will actually be **3**.

```
CalculatorViewModel calculator = new CalculatorViewModel(new BuiltInRandom());

calculator.KeyCommand.Execute("1");
calculator.AddCommand.Execute(null);
calculator.KeyCommand.Execute("2");
calculator.EquateCommand.Execute(null);

Assert.AreEqual(0, calculator.CurrentValue);
```

11. Right-click **AddSimple** and select **Run Tests** to run this test.

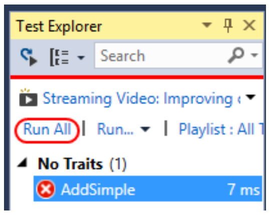

```
[TestMethod]
0 references
public void AddSimple()
{
    Calculato
    
    calculato
    calculato
    calculato
    calculato
    
    Assert.Ar
}
```
Refactor
Organize Usings
Generate Sequence Diagram...
Show on Code Map Ctrl+`
Find All References on Code Map
Show Related Items on Code Map
Run Tests Ctrl+R, T

The test fails as expected.

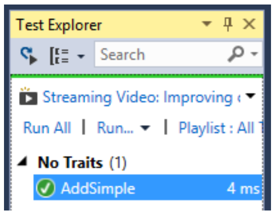

```
[TestMethod]
0 references
public void AddSimple()
{
    CalculatorViewModel
```

12. Change the **0** in the **Assert** line to **3** to fix the test.

13. In **Test Explorer**, click **Run All**.



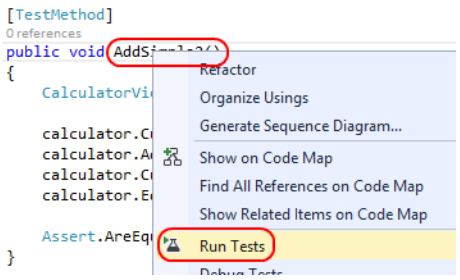
The test now passes.



14. Now add an additional addition test by inserting this code into the test class. Note that instead of using the **KeyCommand** steps, the library was designed to allow you to directly set the **CurrentValue** as a shortcut. It's still a good practice to have tests that confirm the key commanding, but it would be unnecessarily complex to have to enter long numbers via emulated key presses.

```
[TestMethod]
public void AddSimple2()
{
    CalculatorViewModel calculator = new CalculatorViewModel(new BuiltInRandom());

    calculator.CurrentValue = 100;
    calculator.AddCommand.Execute(null);
    calculator.CurrentValue = 200;
    calculator.EquateCommand.Execute(null);

    Assert.AreEqual(300, calculator.CurrentValue);
}
```

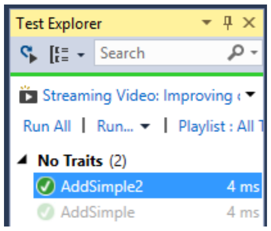
15. Right-click **AddSimple2** and select **Run Tests**.



```
[TestMethod]
0 references
public void AddSimple2()
{
    CalculatorVi

    calculator.C
    calculator.A
    calculator.C
    calculator.E

    Assert.AreEq
}
```

Refactor
Organize Usings
Generate Sequence Diagram...
Show on Code Map
Find All References on Code Map
Show Related Items on Code Map
Run Tests
Debug Tests

The test will succeed.

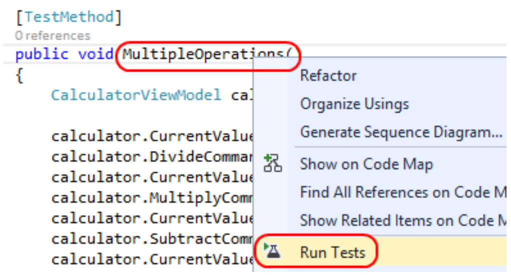


These two tests have given the **Add** functionality a good bit of exercise, but it would be worthwhile to add in some other functionality for the next test.

16. Add the following test to the class. It performs the four basic functions.

```
[TestMethod]
public void MultipleOperations()
{
    CalculatorViewModel calculator = new CalculatorViewModel(new BuiltInRandom());

    calculator.CurrentValue = 10;
    calculator.DivideCommand.Execute(null);
    calculator.CurrentValue = 2;
    calculator.MultiplyCommand.Execute(null);
    calculator.CurrentValue = 7;
    calculator.SubtractCommand.Execute(null);
    calculator.CurrentValue = 12;
    calculator.AddCommand.Execute(null);
    calculator.CurrentValue = 7;
    calculator.EquateCommand.Execute(null);

    Assert.AreEqual(30, calculator.CurrentValue);
}
```

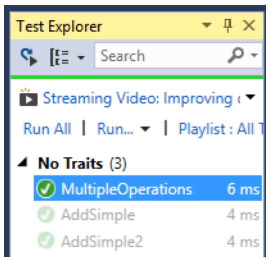
17. Right-click **MultipleOperations** and select **Run Tests**.



```
[TestMethod]
0 references
public void MultipleOperations(
{
    CalculatorViewModel ca

    calculator.CurrentValue
    calculator.DivideComman
    calculator.CurrentValue
    calculator.MultiplyComm
    calculator.CurrentValue
    calculator.SubtractComm
    calculator.CurrentValue
```

Refactor
Organize Usings
Generate Sequence Diagram...
Show on Code Map
Find All References on Code M
Show Related Items on Code M
Run Tests

The test will succeed. However, notice that it likely takes slightly longer due to the extra operations and overhead involved.



Test Explorer
Search
Streaming Video: Improving
Run All | Run... | Playlist : All T
No Traits (3)

MultipleOperations	6 ms
AddSimple	4 ms
AddSimple2	4 ms

Throughout these tests, there has been a bit of redundancy that you can start to factor out. For example, each test is creating a **CalculatorViewModel** using the same **BuiltInRandom** parameter. This isn't necessarily bad form, but since it's so repetitive, you can use a private member and create a new one automatically before the tests.

18. Add the following property near the top of the class.

```
private CalculatorViewModel calculator;
```

19. Add the following initializer method to the class. Note that it's attributed with **TestInitialize**, which indicates that this method should be called prior to each test in the class.

```
[TestInitialize]
public void TestInitialize()
{
    this.calculator = new CalculatorViewModel(new BuiltInRandom());
}
```

In addition to **TestInitialize**, there are five other attributes that allow you to insert a method during the testing lifecycle:

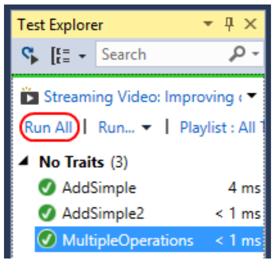
- **TestCleanup** is an attribute for an instance method that runs after each test in that class.
- **ClassInitialize** is an attribute for a static method that runs once prior to any tests in that class.
- **ClassCleanup** is an attribute for a static method that runs once after all the tests in that class have been run.
- **AssemblyInitialize** is an attribute for a static method that runs once prior to any tests in that assembly.
- **AssemblyCleanup** is an attribute for a static method that runs once after all the tests in that assembly have been run.

> Note: For more details on the proper usage of these attributes, as well as their expected method signatures, please visit http://msdn.microsoft.com/en-us/library/Microsoft.VisualStudio.TestTools.UnitTesting.aspx

20. Remove the **CalculatorViewModel** creation line in each of the existing tests. That variable will now reference the instance member.

```
CalculatorViewModel calculator = new CalculatorViewModel(new BuiltInRandom());
```

21. In **Test Explorer**, click **Run All** to run all tests. They should all pass as expected.


Test Explorer
Search
Streaming Video: Improving
Run All | Run... | Playlist : All
No Traits (3)
AddSimple 4 ms
AddSimple2 < 1 ms
MultipleOperations < 1 ms

While MVVM is not the primary focus of this lab, you can see how the up-front investment made in abstracting key components via the MVVM model has paid significant dividends in the form of automated testing. Instead of having to run the UI each time you want to confirm that new code has not introduced a regression bug, you can feel a higher level of confidence that your automated tests provide a great first line of defense.

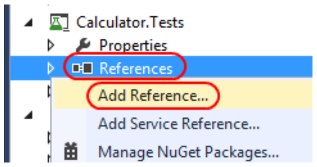
Exercise 3: Advanced unit testing features

In this exercise, you'll learn about some advanced unit testing features in Visual Studio, including data-driven testing, code coverage, and the Fakes framework.

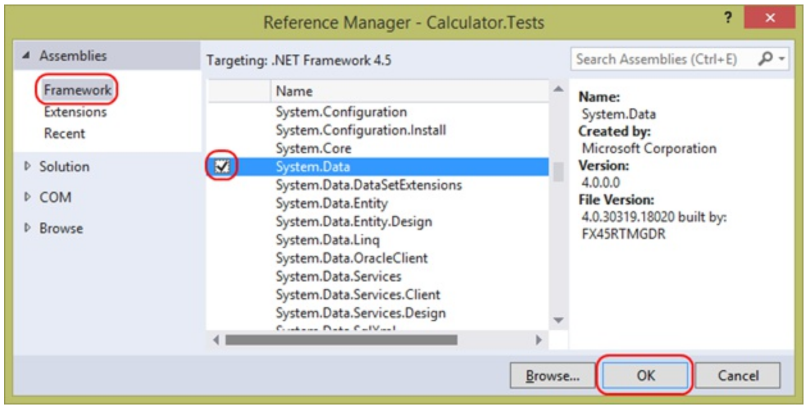
Task 1: Data-driven testing

In this task, you'll create a data-driven test. Data-driven tests provide a great way to perform the same test with different parameters, resulting in broader coverage without duplicated code.

1. Before writing data-driven tests, you'll need to reference an additional assembly that contains some required functionality. In **Solution Explorer**, right-click the **References** node under the **Calculator.Tests** project and select **Add Reference...**.



2. From the **Reference Manager** dialog, select the **Assemblies | Framework** category and check the box next to the **System.Data** assembly. Click **OK** to add the reference.


Reference Manager - Calculator.Tests
Assemblies
Framework
Extensions
Recent
Solution
COM
Browse
Targeting: .NET Framework 4.5
Search Assemblies (Ctrl+E)
Name
System.Configuration
System.Configuration.Install
System.Core
System.Data
System.Data.DataSetExtensions
System.Data.Entity
System.Data.Entity.Design
System.Data.Linq
System.Data.OracleClient
System.Data.Services
System.Data.Services.Client
System.Data.Services.Design
Browse...
OK
Cancel

Name:
System.Data
Created by:
Microsoft Corporation
Version:
4.0.0.0
File Version:
4.0.30319.18020 built by: FX45RTMGDR

3. Add the following code in **MathTests.cs** to create a data-driven test.

```
[TestMethod]
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML",
    "|DataDirectory|\\Tests.xml", "add", DataAccessMethod.Sequential)]
public void AddDataDriven()
{
    int first = int.Parse(this.TestContext.DataRow["first"].ToString());
    int second = int.Parse(this.TestContext.DataRow["second"].ToString());
    int sum = int.Parse(this.TestContext.DataRow["sum"].ToString());

    this.calculator.CurrentValue = first;
    this.calculator.AddCommand.Execute(null);
    this.calculator.CurrentValue = second;
    this.calculator.EquateCommand.Execute(null);
    Assert.AreEqual(sum, this.calculator.CurrentValue);
}
```

Note the use of the **DataSource** attribute, which indicates that this test should be run once per row of data provided in the source.

- The first parameter is the fully qualified name of the data source provider, which is for an XML file here. Alternatively, you can use the appropriate object to support using a database, CSV file, or other source.

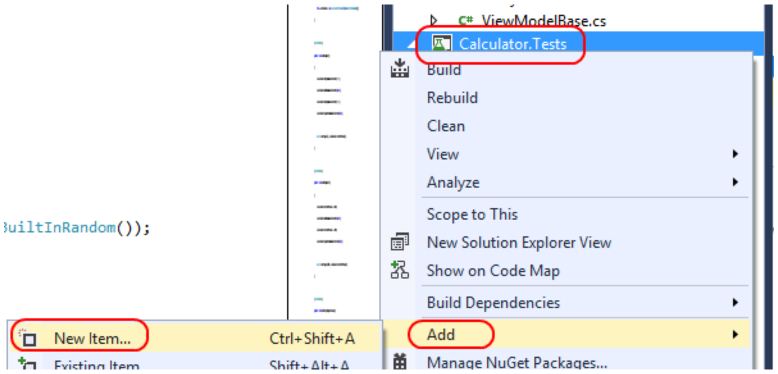
- The second parameter is the connection string, which in this case is simply the test-time path to the XML file.
- The third parameter specifies the table in which the data is stored (or in the case of your XML file, will be the name of the elements directly under the root node).
- The final parameter indicates that the tests should be run using the data in the order it was found (the alternative is to randomize).

  Within the body of the test, the first three lines show accessing the data from the row based on column names, which are “first”, “second”, and the expected “sum”. This will allow you to provide as many test cases as you like without having to rewrite the code based purely on changing parameters.

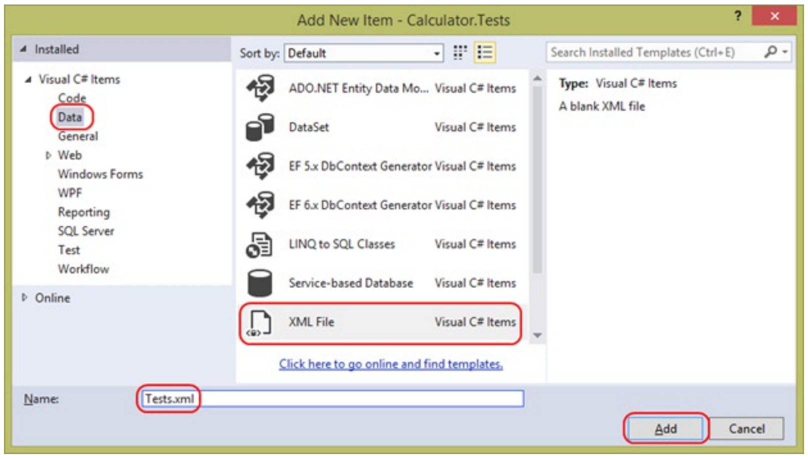
4. Expose a **TestContext** property by adding the following code to **MathTests.cs**. This will get set by the test engine with each row as it’s ready for testing.

```
public TestContext TestContext { get; set; }
```

5. In **Solution Explorer**, right-click the **Calculator.Tests** project node and select **Add | New Item...**.



6. Select the **Visual C# Items | Data** category and the **XML File** template. Type **“Tests.xml”** as the file name and click **Add** to create.

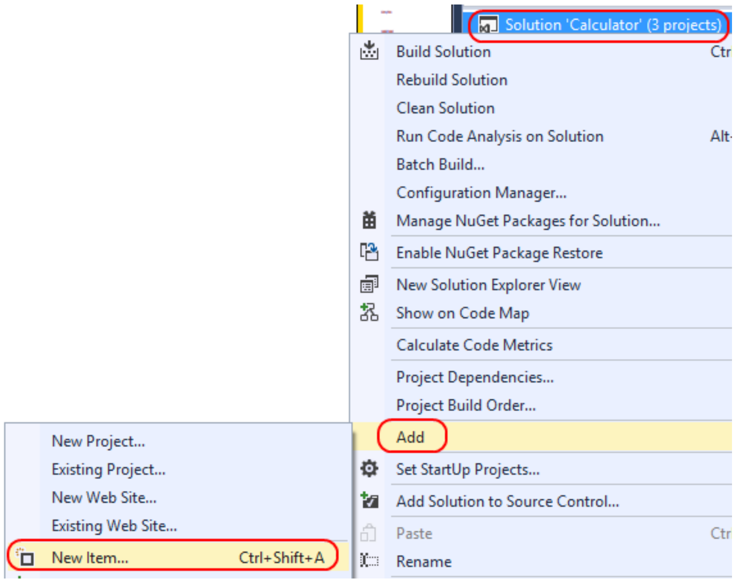

Add New Item - Calculator.Tests
Installed
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
SQL Server
Test
Workflow
Online
Sort by: Default
ADO.NET Entity Data Mo... Visual C# Items
DataSet Visual C# Items
EF 5.x DbContext Generator Visual C# Items
EF 6.x DbContext Generator Visual C# Items
LINQ to SQL Classes Visual C# Items
Service-based Database Visual C# Items
XML File Visual C# Items
Click here to go online and find templates.
Search Installed Templates (Ctrl+E)
Type: Visual C# Items
A blank XML file
Name: Tests.xml
Add
Cancel

7. Add the following XML as the body of the document. When the test runs, it will iterate once with the values **1** and **2** and expect a sum of **3**. Press **Ctrl+S** to save the file.

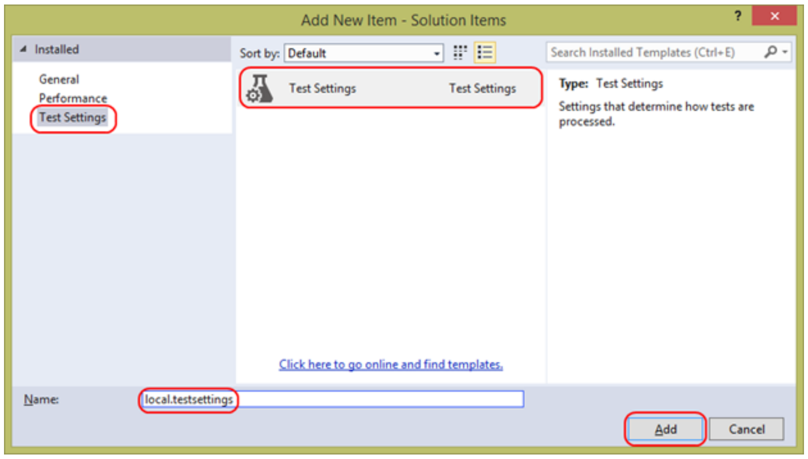
```
<tests>
  <add>
    <first>1</first>
    <second>2</second>
    <sum>3</sum>
  </add>
</tests>
```

   However, there is one additional step required for the test to run, which is to configure the deployment of **Tests.xml** so that the test engine can load it. For this, you will create a **Test Settings** file.

8. In **Solution Explorer**, right-click the solution node and select **Add | New Item...**.



9. Select the **Test Settings** category and the **Test Settings** template. Type **"local.testsettings"** as the **Name** and click **Add**.


Add New Item - Solution Items

Installed: General, Performance, Test Settings

Sort by: Default

Test Settings — Test Settings

Type: Test Settings

Settings that determine how tests are processed.

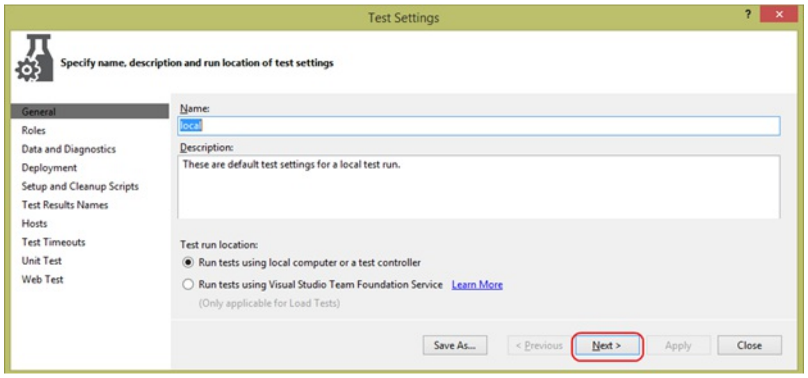
Click here to go online and find templates.

Name: local.testsettings

Add Cancel

The **Test Settings** wizard contains ten steps, although there is only one step needed to configure the deployment. However, you can step through it for the purposes of this lab.

10. On the **General** page, you can configure some metadata about the test settings. Click **Next**.


Test Settings

Specify name, description and run location of test settings

General, Roles, Data and Diagnostics, Deployment, Setup and Cleanup Scripts, Test Results Names, Hosts, Test Timeouts, Unit Test, Web Test

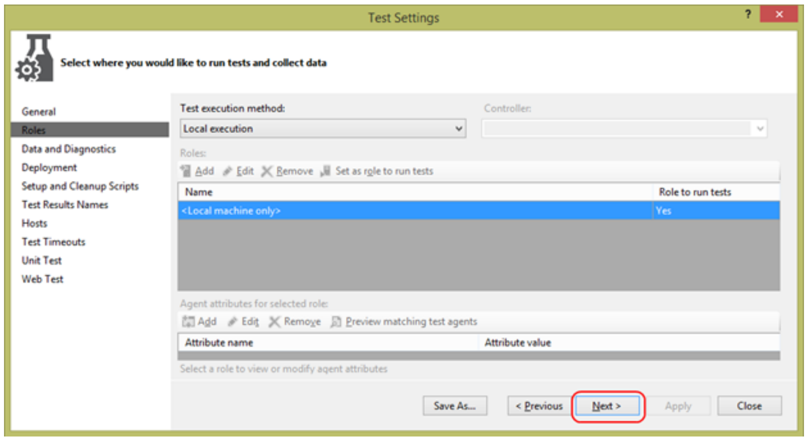
Name: local

Description: These are default test settings for a local test run.

Test run location:
- [✓ Run tests using local computer or a test controller]
- Run tests using Visual Studio Team Foundation Service Learn More (Only applicable for Load Tests)

Save As... < Previous Next > Apply Close

11. On the **Roles** page, you can define different roles and their involvement in running the tests. This isn't applicable here since you only plan to run unit tests that don't require anything special. Click **Next**.



Test execution method: Local execution

Controller:

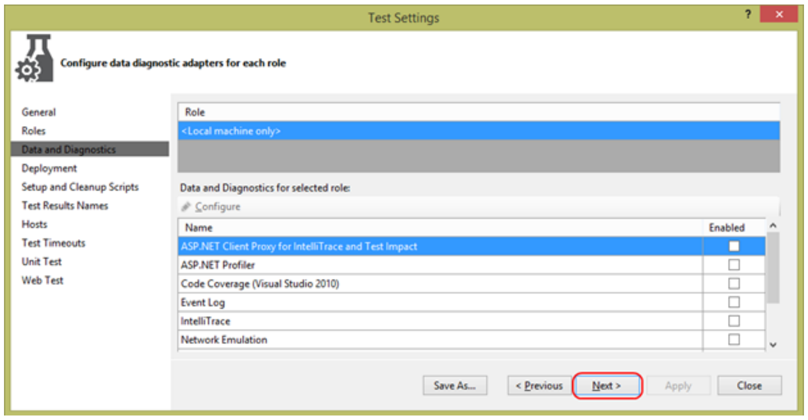
Roles:

Name	Role to run tests
<Local machine only>	Yes

Agent attributes for selected role:

Attribute name	Attribute value

12. On the **Data and Diagnostics** page, you can configure the various ways of collecting data and diagnostics during the test run. Click **Next**.

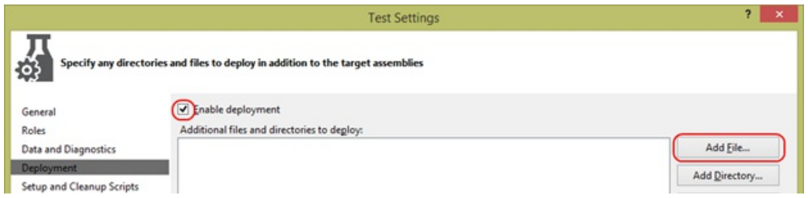


Role
<Local machine only>

Data and Diagnostics for selected role:

Name	Enabled
ASP.NET Client Proxy for IntelliTrace and Test Impact	☐
ASP.NET Profiler	☐
Code Coverage (Visual Studio 2010)	☐
Event Log	☐
IntelliTrace	☐
Network Emulation	☐

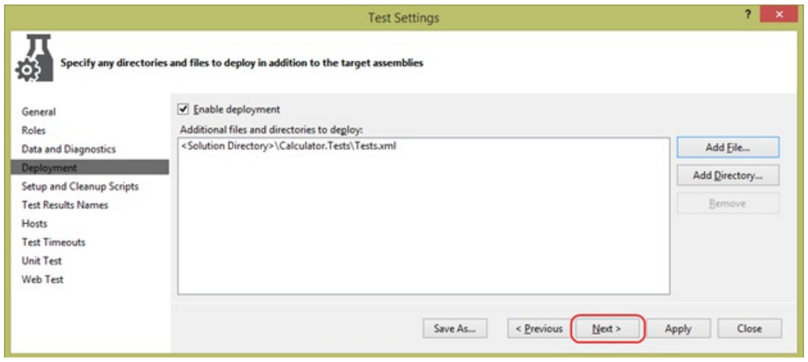
13. On the **Deployment** page, check the **Enable Deployment** box to enable deployment of files (like **Tests.xml**). Next, click **Add File…**.


Test Settings
Specify any directories and files to deploy in addition to the target assemblies
General
Roles
Data and Diagnostics
Deployment
Setup and Cleanup Scripts
Add File...
Add Directory...

☑ Enable deployment

Additional files and directories to deploy:

14. Locate **Tests.xml** and add it. Now it will be deployed to the test working directory when these settings are used. Click **Next**.

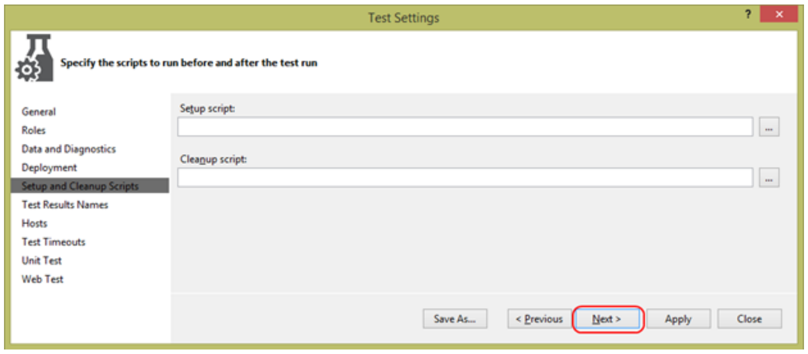


☑ Enable deployment

Additional files and directories to deploy:

<Solution Directory>\Calculator.Tests\Tests.xml

15. On the **Setup and Cleanup Script** page, you can configure scripts to be run before and after your test run. Click **Next**.

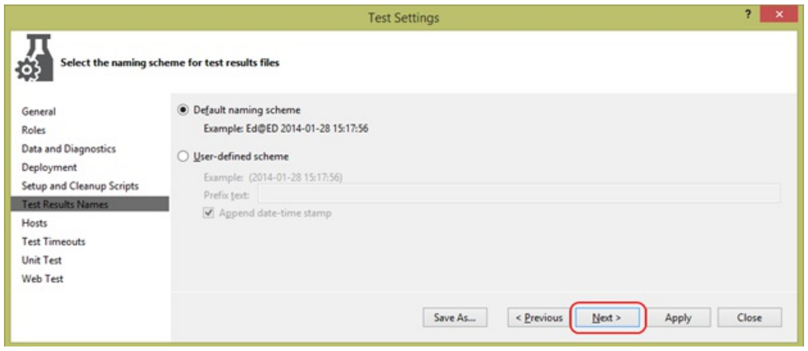

Test Settings

Specify the scripts to run before and after the test run

Setup script:

Cleanup script:

16. On the **Test Results Names** page, you can configure the naming scheme for test runs. The default name includes the user running the test, the machine the test is run on, and the date and time. Click **Next**.

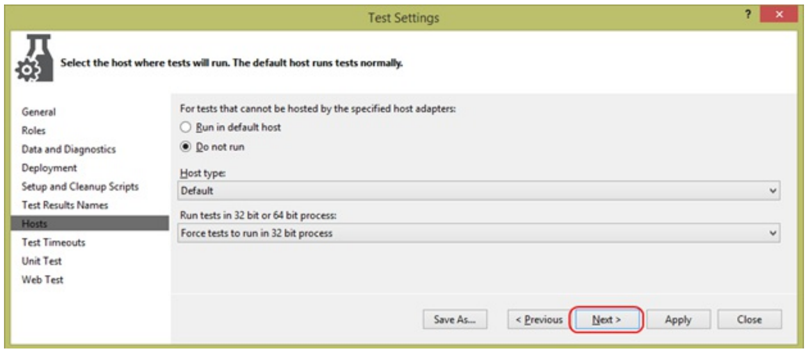

Test Settings

Select the naming scheme for test results files

● Default naming scheme
Example: Ed@ED 2014-01-28 15:17:56

○ User-defined scheme
Example: (2014-01-28 15:17:56)
Prefix text:
✓ Append date-time stamp

17. On the **Hosts** page, you can configure where and how the tests are run. Click **Next**.


Test Settings

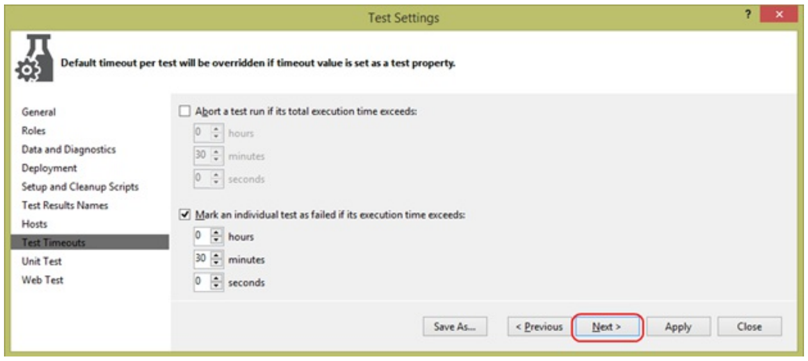
Select the host where tests will run. The default host runs tests normally.

For tests that cannot be hosted by the specified host adapters:
○ Run in default host
● Do not run

Host type:
Default

Run tests in 32 bit or 64 bit process:
Force tests to run in 32 bit process

18. On the **Test Timeouts** page, you can set timeouts for each test as well as the whole test run. Click **Next**.

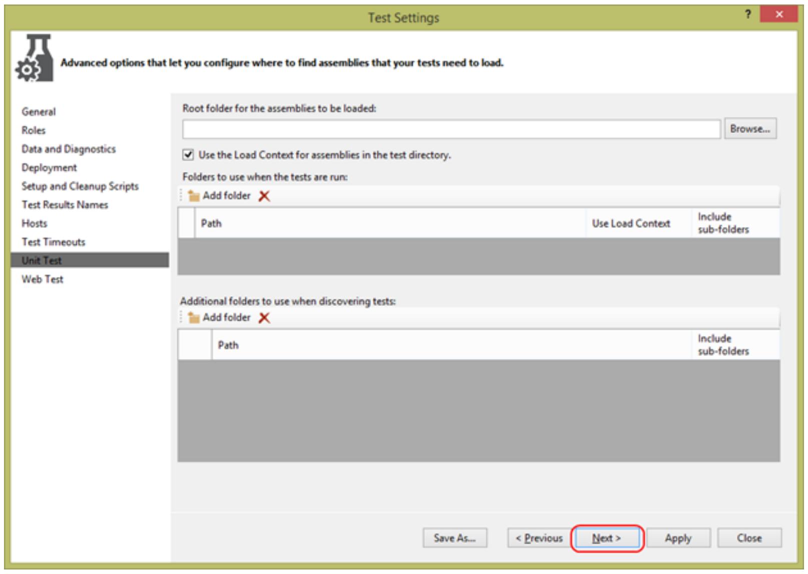

Test Settings

Default timeout per test will be overridden if timeout value is set as a test property.

☐ Abort a test run if its total execution time exceeds:
0 hours
30 minutes
0 seconds

✓ Mark an individual test as failed if its execution time exceeds:
0 hours
30 minutes
0 seconds

19. On the **Unit Test** page, you can configure additional folders containing reference assemblies. This allows you to have additional assemblies referenced by your tests to be kept in once place, rather than getting copied to the test working folder for each run. Click **Next**.


Test Settings

Advanced options that let you configure where to find assemblies that your tests need to load.

Root folder for the assemblies to be loaded:

[✓] Use the Load Context for assemblies in the test directory.

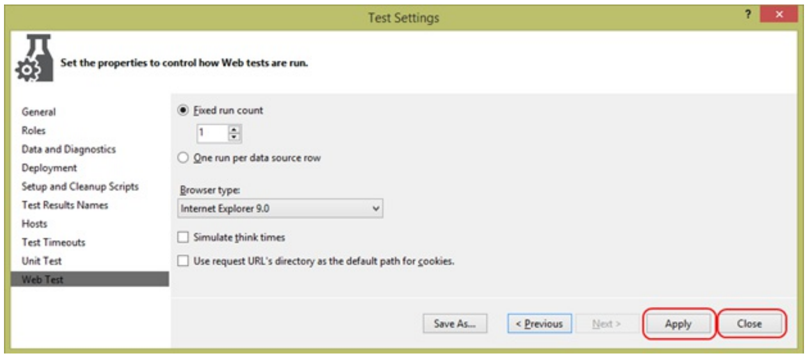
Folders to use when the tests are run:

Path	Use Load Context	Include sub-folders

Additional folders to use when discovering tests:

Path	Include sub-folders

20. On the **Web Test** page, you can configure how Web tests are run. Click **Apply** and then **Close**.


Test Settings

Set the properties to control how Web tests are run.

(●) Fixed run count: 1

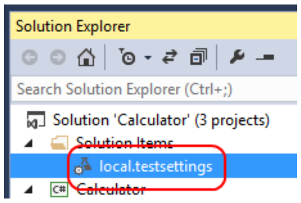
( ) One run per data source row

Browser type: Internet Explorer 9.0

[ ] Simulate think times

[ ] Use request URL's directory as the default path for cookies.

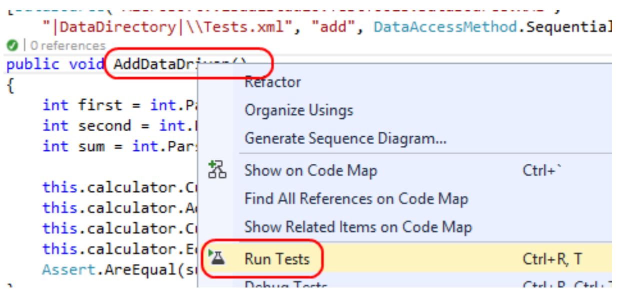
Note that the test settings file is part of the solution, and not specific to any testing project.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Calculator' (3 projects)
Solution Items
local.testsettings
Calculator

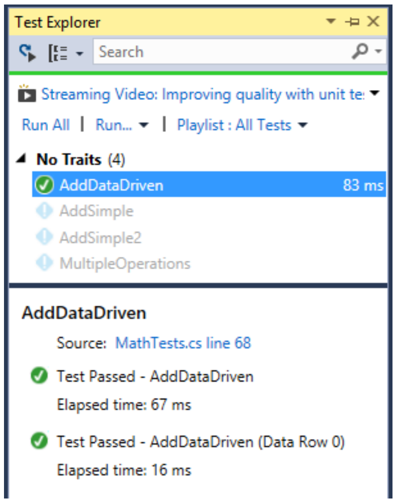
21. From the main menu, select **Test | Test Settings | Select Test Settings File**.

22. Select the **local.testsettings** file you just created, which will be in the root of the solution directory.

23. Right-click the **AddDataDriven** test and select **Run Tests**.


```
    "|DataDirectory|\\Tests.xml", "add", DataAccessMethod.Sequential
 | 0 references
public void AddDataDr
{
    int first = int.P
    int second = int.
    int sum = int.Par

    this.calculator.C
    this.calculator.A
    this.calculator.C
    this.calculator.E
    Assert.AreEqual(s
```
Refactor
Organize Usings
Generate Sequence Diagram...
Show on Code Map Ctrl+`
Find All References on Code Map
Show Related Items on Code Map
Run Tests Ctrl+R, T

24. In **Test Explorer**, you can follow the progress of the test. Once it completes, select the test and notice how the bottom pane shows the overall success, followed by the results from each row. In this case, there is only one.



AddDataDriven

Source: MathTests.cs line 68

Test Passed - AddDataDriven

Elapsed time: 67 ms

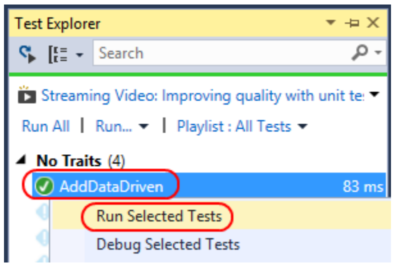
Test Passed - AddDataDriven (Data Row 0)

Elapsed time: 16 ms

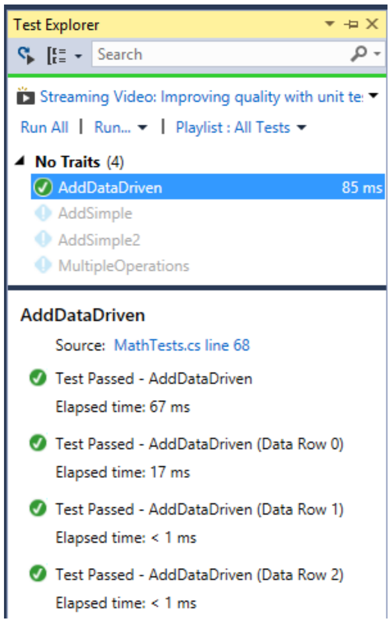
25. In **Tests.xml**, add these two additional records inside the closing tag.

```
<add>
  <first>29</first>
  <second>15</second>
  <sum>44</sum>
</add>
<add>
  <first>10000</first>
  <second>33333</second>
  <sum>43333</sum>
</add>
```

26. In **Test Explorer**, right-click **AddDataDriven** and select **Run Selected Tests**.



After the tests complete, you will notice that there are now three rows of data, and all were successful.



AddDataDriven

Source: MathTests.cs line 68

Test Passed - AddDataDriven

Elapsed time: 67 ms

Test Passed - AddDataDriven (Data Row 0)

Elapsed time: 17 ms

Test Passed - AddDataDriven (Data Row 1)

Elapsed time: < 1 ms

Test Passed - AddDataDriven (Data Row 2)

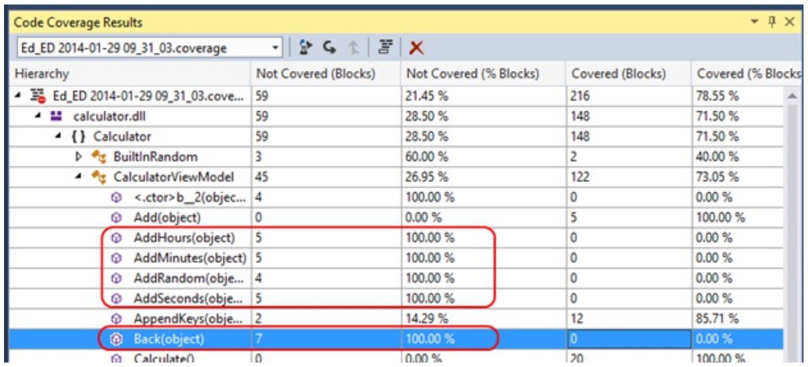
Elapsed time: < 1 ms

Task 2: Analyzing code coverage

In this task, you'll perform a code coverage analysis in order to determine which aspects of the codebase are not covered by unit tests. By identifying the areas of code that are not being exercised during unit testing, you can prioritize the addition of new tests for those areas. However, be cautious of relying too much on the metrics provided by code coverage alone. While it is a great tool for understanding the reach of your tests, it does not necessarily indicate that all scenarios are being properly tested.

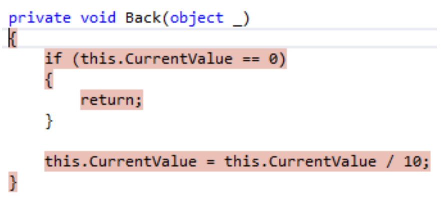
1. From the main menu, select **Test | Analyze Code Coverage | All Tests**. This will build and run all tests.

2. Locate the **Code Coverage Results** panel, which is probably docked to the bottom of Visual Studio. If you expand **calculator.dll | Calculator | CalculatorViewModel**, you can see that there are quite a few methods that are completely uncovered by these tests. Notice how each parent row aggregates the statistics of its children. This gives you a great way to quickly determine the overall code coverage in a class, namespace, or assembly. Double-click the **Back(object)** method to navigate to it.



Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks
Ed_ED 2014-01-29 09_31_03.cove...	59	21.45 %	216	78.55 %
calculator.dll	59	28.50 %	148	71.50 %
Calculator	59	28.50 %	148	71.50 %
BuiltInRandom	3	60.00 %	2	40.00 %
CalculatorViewModel	45	26.95 %	122	73.05 %
<.ctor>b__2(objec...	4	100.00 %	0	0.00 %
Add(object)	0	0.00 %	5	100.00 %
AddHours(object)	5	100.00 %	0	0.00 %
AddMinutes(object)	5	100.00 %	0	0.00 %
AddRandom(obje...	4	100.00 %	0	0.00 %
AddSeconds(obje...	5	100.00 %	0	0.00 %
AppendKeys(obje...	2	14.29 %	12	85.71 %
Back(object)	7	100.00 %	0	0.00 %
Calculate()	0	0.00 %	20	100.00 %

3. In the **Back** method, you can see how the code is shaded red. These lines were not exercised as part of the last test run.



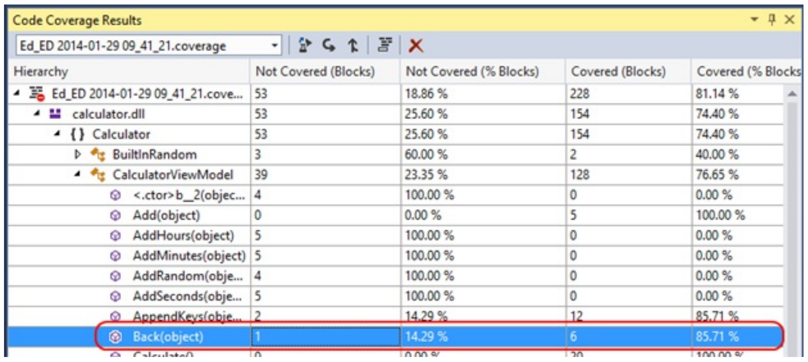
```
private void Back(object _)
{
    if (this.CurrentValue == 0)
    {
        return;
    }

    this.CurrentValue = this.CurrentValue / 10;
}
```

4. In **MathTests.cs**, add the following test. It will exercise the **Back** method via its command.

```
[TestMethod]
public void Back()
{
    calculator.CurrentValue = 100;
    calculator.BackCommand.Execute(null);

    Assert.AreEqual(10, calculator.CurrentValue);
}
```

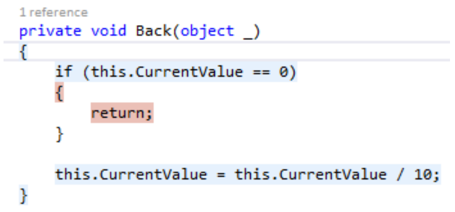
5. From the main menu, select **Test | Analyze Code Coverage | All Tests**. Note that analyzing code coverage invokes a test run, so you don't need to do that separately.

6. In the **Code Coverage Results**, expand the new run out until you find the **Back** method again. Now you can see that it went from having **6** untested lines to only **1.**



Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks
Ed_ED 2014-01-29 09_41_21.cove...	53	18.86 %	228	81.14 %
calculator.dll	53	25.60 %	154	74.40 %
Calculator	53	25.60 %	154	74.40 %
BuiltInRandom	3	60.00 %	2	40.00 %
CalculatorViewModel	39	23.35 %	128	76.65 %
<.ctor>b__2(objec...	4	100.00 %	0	0.00 %
Add(object)	0	0.00 %	5	100.00 %
AddHours(object)	5	100.00 %	0	0.00 %
AddMinutes(object)	5	100.00 %	0	0.00 %
AddRandom(obje...	4	100.00 %	0	0.00 %
AddSeconds(obje...	5	100.00 %	0	0.00 %
AppendKeys(obje...	2	14.29 %	12	85.71 %
Back(object)	1	14.29 %	6	85.71 %
Calculate()	0	0.00 %	20	100.00 %

7. Switch back to **CalculatorViewModel.cs** to see the updated code coverage highlighting. It's now easy to see which lines have been tests (blue) against the one that was not (red).

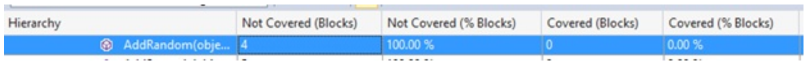


```
1 reference
private void Back(object _)
{
    if (this.CurrentValue == 0)
    {
        return;
    }

    this.CurrentValue = this.CurrentValue / 10;
}
```

Task 3: Using Microsoft Fakes, stubs, & shims

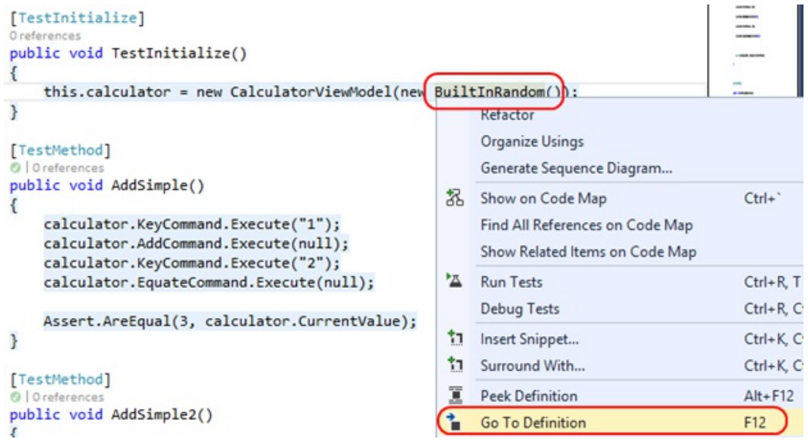
In this task, you'll add a Microsoft Fakes assembly. Microsoft Fakes help you isolate the code you are testing by replacing other parts of the application with stubs or shims. These are small pieces of code that are under the control of your tests. By isolating your code for testing, you know that if the test fails, the cause is there and not somewhere else. Stubs and shims also let you test your code even if other parts of your application are not working yet. Note that Microsoft Fakes requires Visual Studio Ultimate or Visual Studio Premium.

1. In the **Code Coverage Results** panel, note that the **AddRandom** method has no code coverage.



Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
AddRandom(obje...	4	100.00 %	0	0.00 %

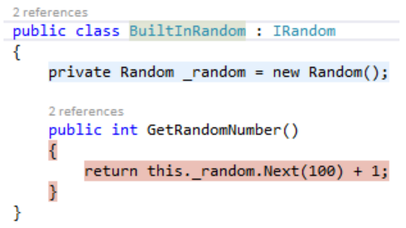
2. Locate the **TestInitialize** method in **MathTests.cs**. This is where the **CalculatorViewModel** is created for use in these tests. Fortunately, the developer who built this class used a dependency injection model, so you can specify which class is providing the random number generation. Right-click **BuiltInRandom** and select **Go To Definition**.



```
[TestInitialize]
0 references
public void TestInitialize()
{
    this.calculator = new CalculatorViewModel(new BuiltInRandom());
}

[TestMethod]
| 0 references
public void AddSimple()
{
    calculator.KeyCommand.Execute("1");
    calculator.AddCommand.Execute(null);
    calculator.KeyCommand.Execute("2");
    calculator.EquateCommand.Execute(null);

    Assert.AreEqual(3, calculator.CurrentValue);
}

[TestMethod]
| 0 references
public void AddSimple2()
```

Refactor
Organize Usings
Generate Sequence Diagram...
Show on Code Map Ctrl+`
Find All References on Code Map
Show Related Items on Code Map
Run Tests Ctrl+R, T
Debug Tests Ctrl+R, C
Insert Snippet... Ctrl+K, C
Surround With... Ctrl+K, C
Peek Definition Alt+F12
Go To Definition F12

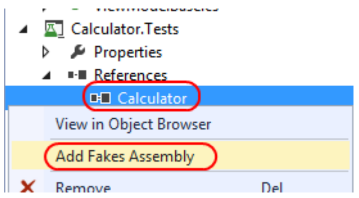
3. **BuiltInRandom** is a fairly simple class. The one method it exposes generates a random number from 1-100. However, there's a potential problem here since the number is unpredictable for test purposes, so there's no good way to rely on this class for **Assert** calls since you won't know what number it's returning.



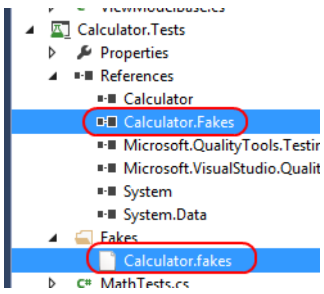
```
2 references
public class BuiltInRandom : IRandom
{
    private Random _random = new Random();

    2 references
    public int GetRandomNumber()
    {
        return this._random.Next(100) + 1;
    }
}
```

Fortunately, the developer had the foresight to use a dependency injection model when defining **CalculatorViewModel**, so you can simply create your own **IRandom** that returns a hardcoded value and pass that in as a parameter. However, this could get messy if there turns out to be a need for different values being returned. Fortunately, this is where **stubs** come in. By adding a Microsoft Fakes assembly for the **Calculator** library, you can quickly and easily "stub out" a valid **IRandom** that you control with a method defined in line with the rest of the code.

4. In **Solution Explorer**, right-click the **Calculator** assembly node under the **References** of **Calculator.Tests** and select **Add Fakes Assembly**.



5. This will generate a **Calculator.Fakes** assembly that has helper classes that allow you to override (or replace) methods and properties of the original types. It also makes it very easy to implement versions of interface classes without having to define all the required methods and properties.



6. At the top of **MathTests.cs**, add the following **using** declaration.

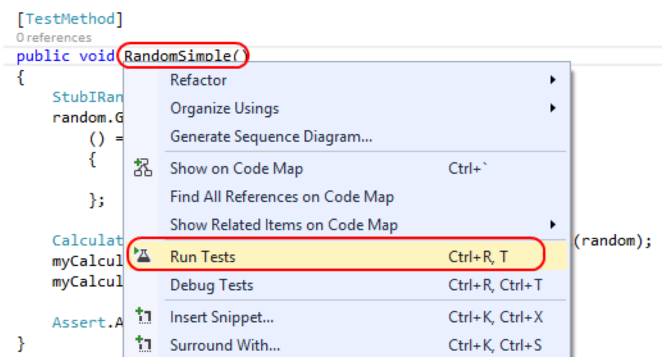
```
using Calculator.Fakes;
```

7. Add the following test to **MathTests.cs**.

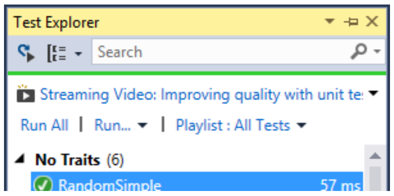
```
public void RandomSimple()
{
    StubIRandom random = new StubIRandom();
    random.GetRandomNumber = () => { return 10; };

    CalculatorViewModel myCalculator = new CalculatorViewModel(random);
    myCalculator.CurrentValue = 100;
    myCalculator.AddRandomCommand.Execute(null);

    Assert.AreEqual(110, myCalculator.CurrentValue);
}
```

The magic in this code lives in the first two lines. First, you're creating a **StubIRandom**, which was generated by Fakes. Next, you're setting its **GetRandomNumber** method to an anonymous delegate that always returns **10**. The rest of the test is fairly straightforward code that creates a new **CalculatorViewModel**, sets its initial value to **100**, and then adds a "random" number to it. However, you've hardcoded that random number to **10**, so you know it'll result in **110** every time.

8. Right-click **RandomSimple** and select **Run Tests**.

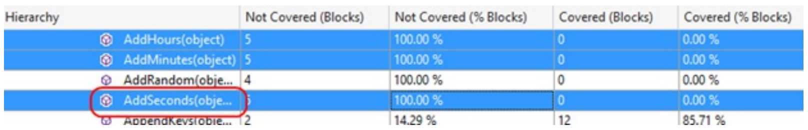

```
[TestMethod]
0 references
public void RandomSimple()
{
    StubIRan
    random.G
        () =
        {

        };

    Calculat                                    (random);
    myCalcul
    myCalcul

    Assert.A
}
```
Refactor
Organize Usings
Generate Sequence Diagram...
Show on Code Map Ctrl+`
Find All References on Code Map
Show Related Items on Code Map
Run Tests Ctrl+R, T
Debug Tests Ctrl+R, Ctrl+T
Insert Snippet... Ctrl+K, Ctrl+X
Surround With... Ctrl+K, Ctrl+S

In **Test Explorer**, the test will proceed and succeed.


Test Explorer
Search
Streaming Video: Improving quality with unit te:
Run All | Run... | Playlist : All Tests
No Traits (6)
RandomSimple 57 ms

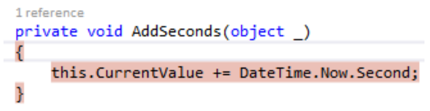
While stubs are great for extending classes and implementing interfaces, sometimes there is a need to override behavior in places you don't have easy access to, such as system calls and static methods. For this, Microsoft Fakes provides **shims**.

9. In the **Code Coverage Results** panel, locate the **AddSeconds** method. Since it doesn't have any code coverage at this point, double-click it to view the code so you can determine the best way to write a test.



Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
AddHours(object)	5	100.00 %	0	0.00 %
AddMinutes(object)	5	100.00 %	0	0.00 %
AddRandom(obje...	4	100.00 %	0	0.00 %
AddSeconds(obje...	5	100.00 %	0	0.00 %
AppendKeys(obje...	2	14.29 %	12	85.71 %

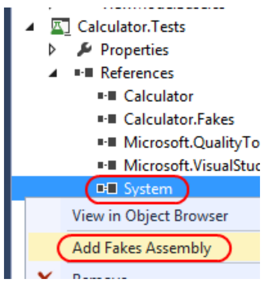
While the code for **AddSeconds** is simple, it unfortunately introduces a problem. Unlike the random functionality from earlier, the developer did not abstract out the time sourcing or provide it as a dependency that could be injected. In a perfect world, you might go into the code and refactor it to use better practices, but for the purposes of this lab, you'll need to test it as-is.



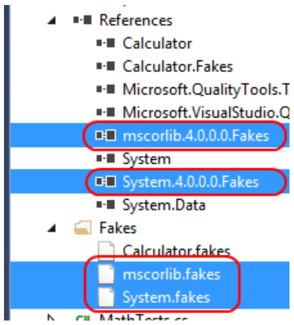
```
1 reference
private void AddSeconds(object _)
{
    this.CurrentValue += DateTime.Now.Second;
}
```

In order to test this method, you'll ultimately need to control what **DateTime.Now** returns, which is a tall order, considering that it's a type that's built into the framework. However, you can create a Fakes assembly for the **System** assembly, which will allow you to shim this property for the scope of the method.

10. In **Solution Explorer**, right-click the **System** assembly node under the **References** of **Calculator.Tests** and select **Add Fakes Assembly**.



Similar to before, Visual Studio creates a Fakes assembly for **System**. Note that it also creates one for **mscorlib**.



11. At the top of **MathTests.cs**, add the following two **using** directives. The first references the new Fakes assembly. The second provides access to the **ShimsContext** class, which is used to scope the period during which a shim applies.

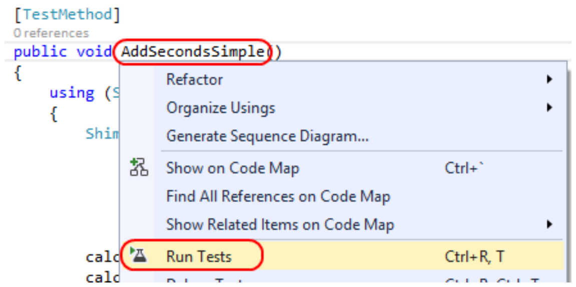
```
using System.Fakes;
using Microsoft.QualityTools.Testing.Fakes;
```

12. Add the following test to **MathTests.cs**.

```
[TestMethod]
public void AddSecondsSimple()
{
    using (ShimsContext.Create())
    {
        ShimDateTime.NowGet =
            () =>
            {
                return new DateTime(2013, 3, 15, 5, 10, 15);
            };

        calculator.CurrentValue = 100;
        calculator.AddSecondsCommand.Execute(null);

        Assert.AreEqual(115, calculator.CurrentValue);
    }
}
```

Note how the test is now enclosed within a **ShimsContext.Create** using block. This ensures that the **ShimDateTime.NowGet** method that overrides the property getter for **DateTime.Now** only applies until the object it returns is disposed at the end of the using block. The rest of the test is pretty standard, and it will work because you are controlling exactly what gets returned from **DateTime.Now**.

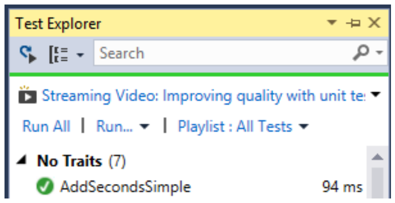
13. Right-click **AddSecondsSimple** and select **Run Tests**.



```
[TestMethod]
0 references
public void AddSecondsSimple()
{
    using (S
    {
        Shim



        calc
        calc
```

Refactor
Organize Usings
Generate Sequence Diagram...
Show on Code Map Ctrl+`
Find All References on Code Map
Show Related Items on Code Map
Run Tests Ctrl+R, T

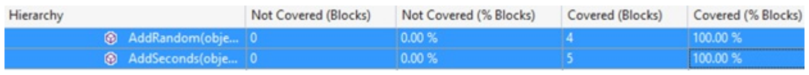
The test will succeed.



Test Explorer
Search
Streaming Video: Improving quality with unit te
Run All | Run... | Playlist : All Tests

No Traits (7)
AddSecondsSimple 94 ms

14. From the main menu, select **Test | Analyze Code Coverage | All Tests**. Now you can see that the test coverage has significantly improved, thanks to Microsoft Fakes.



Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
AddRandom(obje...	0	0.00 %	4	100.00 %
AddSeconds(obje...	0	0.00 %	5	100.00 %

Exercise 4: Continuous integration testing with Visual Studio Online

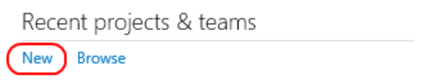
In this exercise, you'll learn about how to set up a continuous integration build & test using Visual Studio Online.

Task 1: Configuring a Visual Studio Online team project

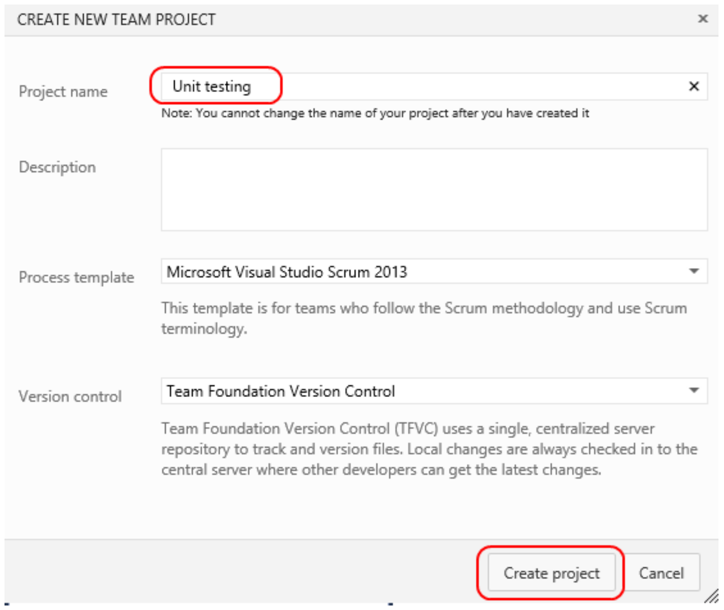
In this task, you'll create a Visual Studio Online team project. This project will ultimately house the code for your project and will later be configured to run the unit tests automatically after a check in. This task will focus on setting up the team project and checking in the current solution.

1. When you created your **Visual Studio Online** account, you set up a domain at visualstudio.com. Navigate to that domain and log in. The URL will be something like https://MYSITE.visualstudio.com.

2. Under the **Recent projects & teams** panel, click the **New** link.



3. In the **Create new team project** dialog, type **"Unit testing"** as the **Project name** and click **Create project**. It may take a few minutes for your project to be created. Note that you can use any name you like for this project, although the screenshots will not match.



CREATE NEW TEAM PROJECT

Project name: Unit testing

Note: You cannot change the name of your project after you have created it

Description:

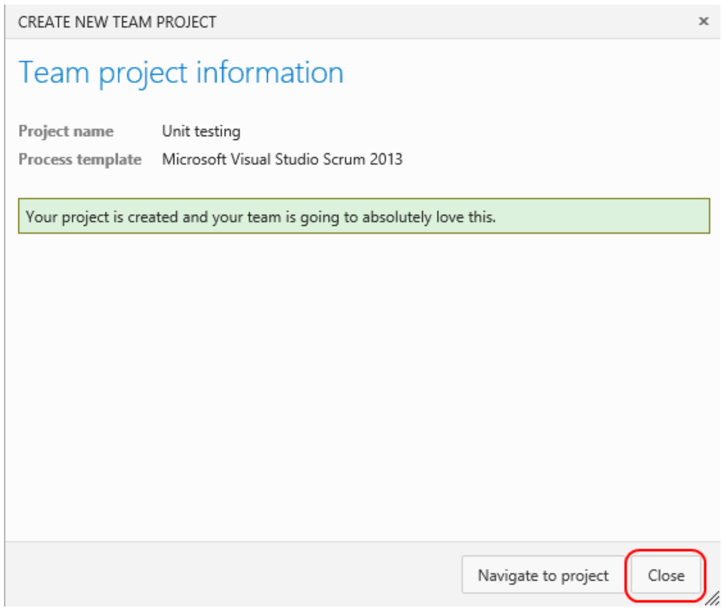
Process template: Microsoft Visual Studio Scrum 2013

This template is for teams who follow the Scrum methodology and use Scrum terminology.

Version control: Team Foundation Version Control

Team Foundation Version Control (TFVC) uses a single, centralized server repository to track and version files. Local changes are always checked in to the central server where other developers can get the latest changes.

4. After the project has been created, click **Close**.

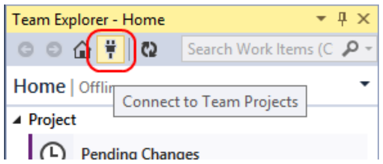


Project name Unit testing

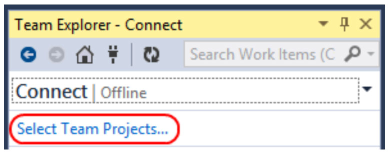
Process template Microsoft Visual Studio Scrum 2013

Your project is created and your team is going to absolutely love this.

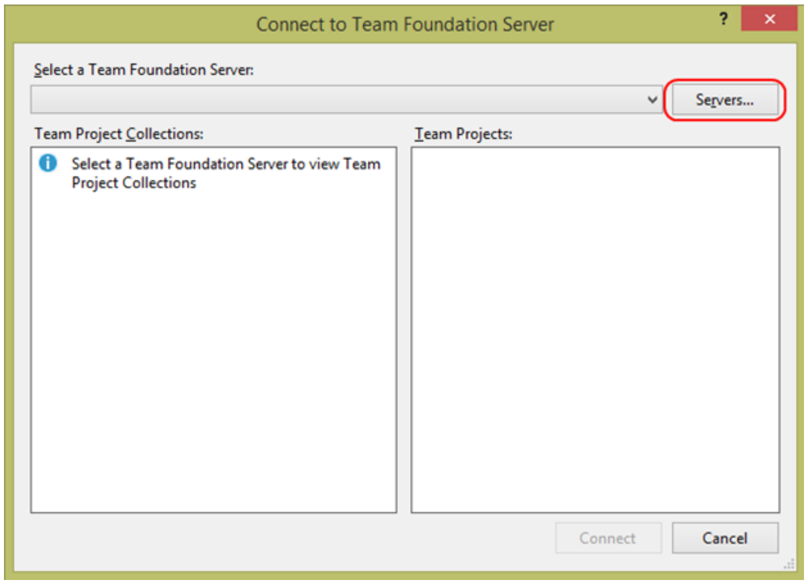
5. In **Visual Studio**, from the main menu, select **Team | Connect to Team Foundation Server…**.
6. In **Team Explorer**, click the **Connect to Team Projects** button.



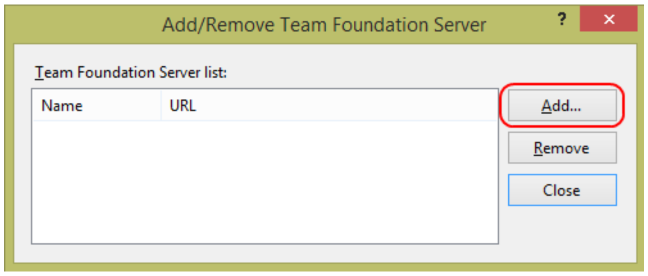
7. Next, click **Select Team Projects…**.



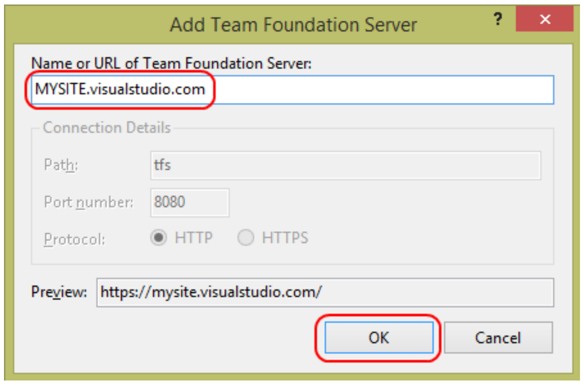
8. In the **Connect to Team Foundation Server** dialog, click **Servers…**.


Connect to Team Foundation Server
Select a Team Foundation Server:
Servers...
Team Project Collections:
Team Projects:
Select a Team Foundation Server to view Team Project Collections
Connect
Cancel

9. In the **Add/Remove Team Foundation Server** dialog, click **Add…**.

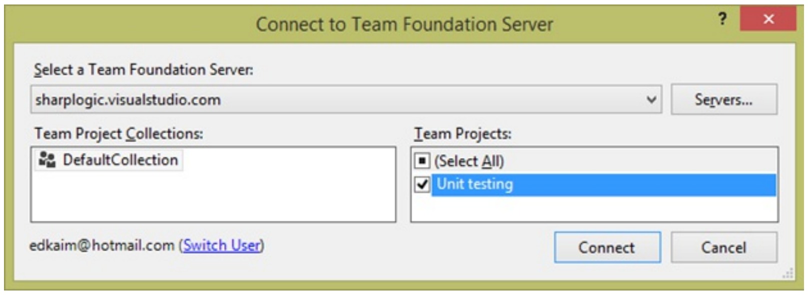

Add/Remove Team Foundation Server
Team Foundation Server list:
Name
URL
Add...
Remove
Close

10. In the **Add Team Foundation Server** dialog, type your **visualstudio.com** domain as the **Name or URL of Team Foundation Server** and click **OK**. If asked to sign in, use the same Microsoft account you used to create the **Visual Studio Online** account.



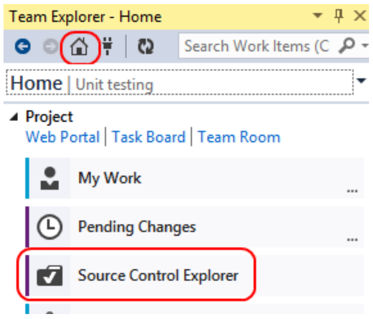
Name or URL of Team Foundation Server: MYSITE.visualstudio.com
Connection Details
Path: tfs
Port number: 8080
Protocol: [● HTTP] ○ HTTPS
Preview: https://mysite.visualstudio.com/

11. In the **Connect to Team Foundation Server** dialog, check the **Unit testing** project and click **Connect**. Visual Studio will now connect to your project and perform some basic configuration to allow access to source code, work items, and other useful project assets.



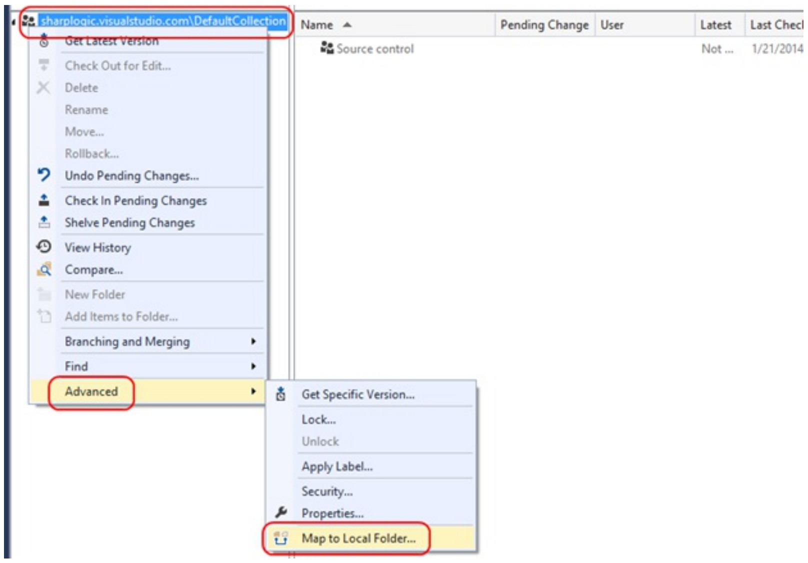
Select a Team Foundation Server: sharplogic.visualstudio.com
Team Project Collections: DefaultCollection
Team Projects: (Select All) [✓ Unit testing]
edkaim@hotmail.com (Switch User)

12. In **Team Explorer**, there are now several options for interacting with the newly created project. Click the **Home** button followed by **Source Control Explorer** to see the source code.

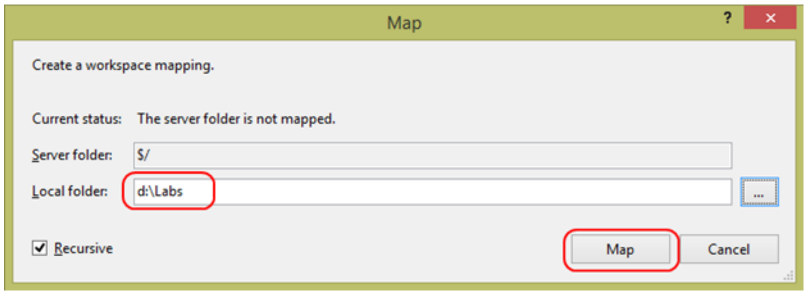


For a new project, the source code repository is empty, with the exception of some default build process templates.

13. In **Source Control Explorer**, right-click the root collection node and select **Advanced | Map to Local Folder…**.



14. In the **Map** dialog, select a local folder where you'd like to keep all your source. Click **Map** to complete the mapping.


Map

Create a workspace mapping.

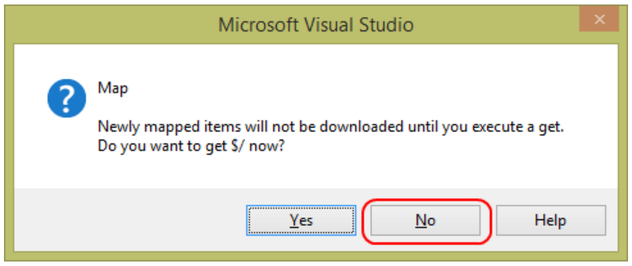
Current status: The server folder is not mapped.

Server folder: $/

Local folder: d:\Labs

☑ Recursive

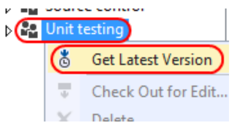
15. After mapping, Visual Studio will provide the option to download the latest version of all files within the mapped project. Click **No** to skip this process for now.


Microsoft Visual Studio

Map

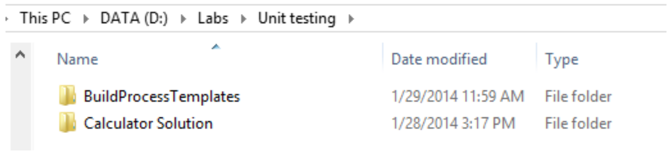
Newly mapped items will not be downloaded until you execute a get.
Do you want to get $/ now?

16. Right-click the **Unit testing** project and select **Get Latest Version**. This will be the way you get the latest version of most projects and files.



17. From the main menu, select **File | Close Solution**.

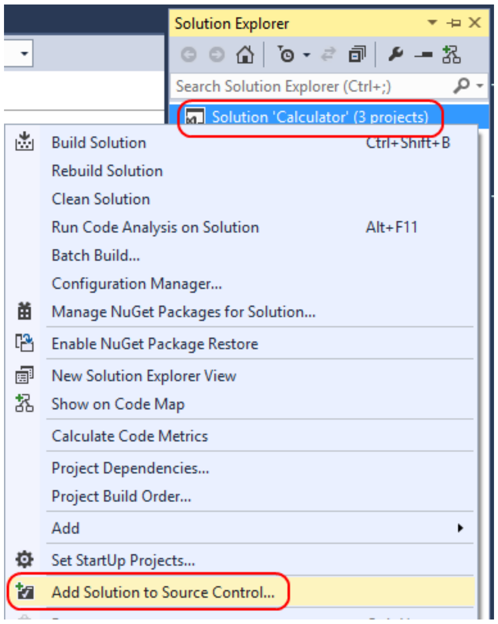
18. Using **Windows Explorer**, move the entire **Calculator Solution** folder into the folder you've mapped for source control. Note that the **Unit testing** team project will be in a **Unit testing** folder inside the map root, inside which there should only be the **BuildProcessTemplates** folder you just downloaded during the "get latest" step. The result should look something like the screenshot below.


This PC › DATA (D:) › Labs › Unit testing ›

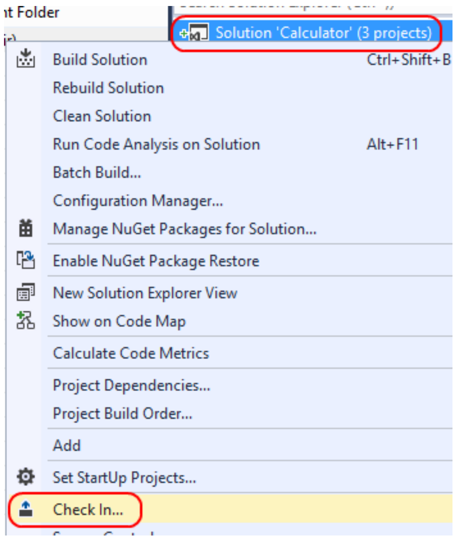
Name	Date modified	Type
BuildProcessTemplates	1/29/2014 11:59 AM	File folder
Calculator Solution	1/28/2014 3:17 PM	File folder

19. In **Visual Studio**, select **File | Open | Project/Solution…** and select the **Calculator.sln** file from the location you just moved it to.

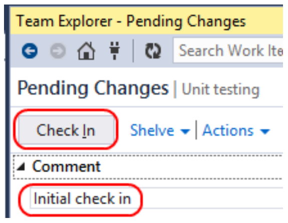
20. In **Solution Explorer**, right-click the solution node and select **Add Solution to Source Control…**.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Calculator' (3 projects)
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Run Code Analysis on Solution Alt+F11
Batch Build...
Configuration Manager...
Manage NuGet Packages for Solution...
Enable NuGet Package Restore
New Solution Explorer View
Show on Code Map
Calculate Code Metrics
Project Dependencies...
Project Build Order...
Add
Set StartUp Projects...
Add Solution to Source Control...

21. Again, right-click the solution node and select **Check In...**.


Solution 'Calculator' (3 projects)
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Run Code Analysis on Solution Alt+F11
Batch Build...
Configuration Manager...
Manage NuGet Packages for Solution...
Enable NuGet Package Restore
New Solution Explorer View
Show on Code Map
Calculate Code Metrics
Project Dependencies...
Project Build Order...
Add
Set StartUp Projects...
Check In...

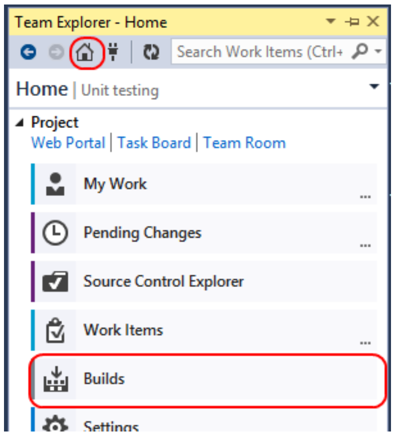
22. In **Team Explorer**, type **"Initial check in"** as the **Comment** and click **Check In**. If prompted to confirm, check the box not to be asked again and click **Yes**.


Team Explorer - Pending Changes
Search Work Ite
Pending Changes | Unit testing
Check In
Shelve
Actions
Comment
Initial check in

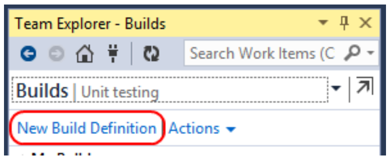
Task 2: Creating a build definition

In this task, you'll create a build definition that instructs Visual Studio Online to build your project and run all available unit tests after a check in. Build definitions go well beyond just defining how the solution is built. They also perform additional tasks before, during, and after the build process, such as running unit tests.

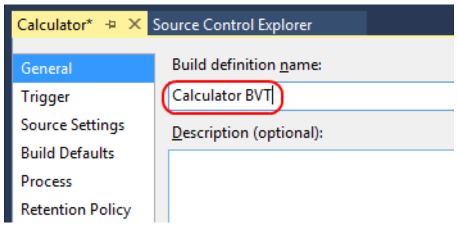
1. In **Team Explorer**, click the **Home** button and then click **Builds**.



2. Click the **New Build Definition** link to create a new build definition.



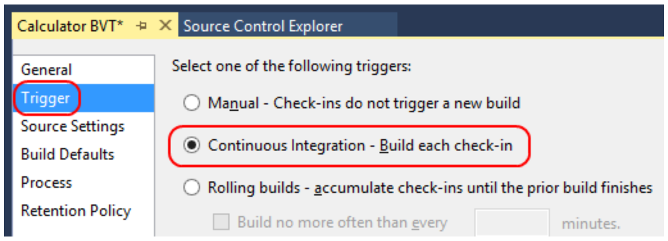
3. On the **General** page of the new build definition, type **"Calculator BVT"** as the **Build definition name**. "BVT" stands for "build verification test" and is a common term for automated builds that perform automated tests. Note that there is also the similarly named "BDT", which refers to "build, deploy, & test", which is a process that performs a deployment after the build, often including resources such as databases, etc, which are then tested.



Build definition name:
Calculator BVT

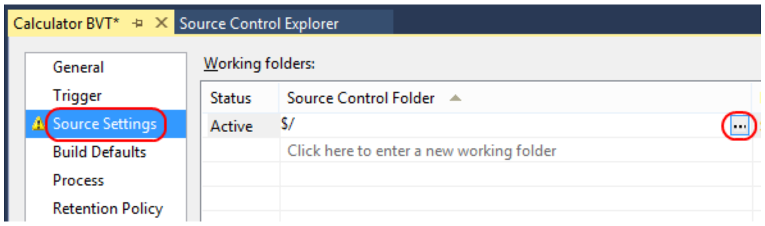
Description (optional):

4. Select the **Trigger** tab and select the option for **Continuous Integration**. This will ensure that this build is run every time a check-in occurs. Note that you can also manually force (or "queue") any build.



Select one of the following triggers:
- Manual - Check-ins do not trigger a new build
- [✓ Continuous Integration - Build each check-in]
- Rolling builds - accumulate check-ins until the prior build finishes
- Build no more often than every minutes.

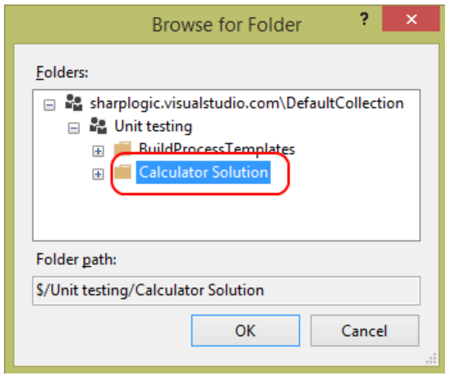
5. Select the **Source Settings** tab. On this page, you can configure the working folders needed to properly run the build. You should always refine this to the minimal source control folders required. Select the first row (the **$/**) and click the **Browse** button that appears on the right side.



Working folders:

Status	Source Control Folder
Active	$/
	Click here to enter a new working folder

6. Navigate down to select the **Unit Testing | Calculator Solution** folder. This is all that's needed for the build. Click **OK**.



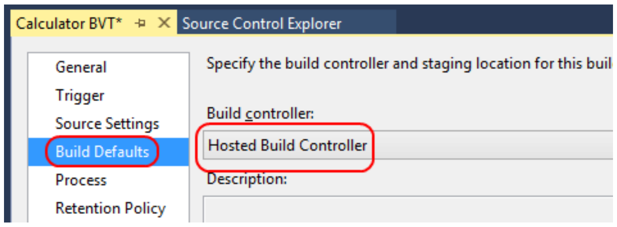
Browse for Folder

Folders:

sharplogic.visualstudio.com\DefaultCollection
Unit testing
BuildProcessTemplates
Calculator Solution

Folder path: $/Unit testing/Calculator Solution

7. Select the **Build Defaults** tab. If running automated builds, you'll need to specify the controller. Since this will be run in the cloud using **Visual Studio Online**, select **Hosted Build Controller** as the **Build controller**.

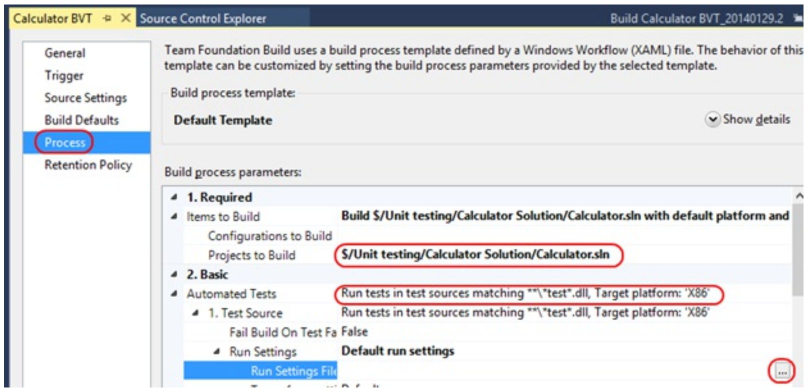


Specify the build controller and staging location for this buil

Build controller: Hosted Build Controller

Description:

8. Select the **Process** tab. This page provides the full configurability for what actually happens in the build process. The default build definition is very useful, and will try to infer what to build (such as when there's only one solution) as well as default to automatically using any assembly with the term **"test"** in it for running automated tests. You can configure all of this, but it works for the purposes of this lab. One additional piece of configuration is to set the **Run Settings File**, so select the field and click the **Browse** button that appears at the right.



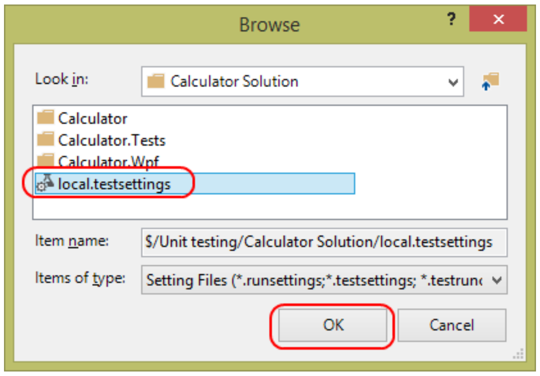
Team Foundation Build uses a build process template defined by a Windows Workflow (XAML) file. The behavior of this template can be customized by setting the build process parameters provided by the selected template.

Build process template: **Default Template**

Build process parameters:

1. Required	
Items to Build	**Build $/Unit testing/Calculator Solution/Calculator.sln with default platform and**
Configurations to Build	
Projects to Build	**$/Unit testing/Calculator Solution/Calculator.sln**
2. Basic	
Automated Tests	Run tests in test sources matching **\*test*.dll, Target platform: 'X86'
1. Test Source	Run tests in test sources matching **\*test*.dll, Target platform: 'X86'
Fail Build On Test Fa	False
Run Settings	**Default run settings**
Run Settings Fil	

9. Select the **local.testsettings** file and click **OK**.



Browse

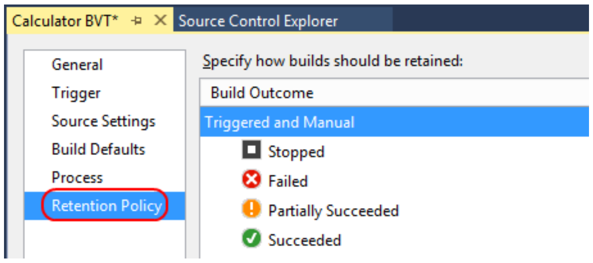
Look in: Calculator Solution

Calculator
Calculator.Tests
Calculator.Wpf
local.testsettings

Item name: $/Unit testing/Calculator Solution/local.testsettings

Items of type: Setting Files (*.runsettings;*.testsettings; *.testrun

10. Select **Retention Policy**. This is the final configuration page and allows you specify which builds to keep.

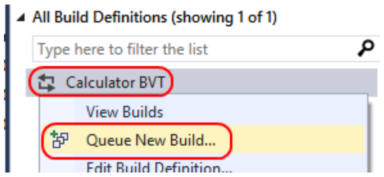


11. Press **Ctrl+S** to save the build definition.

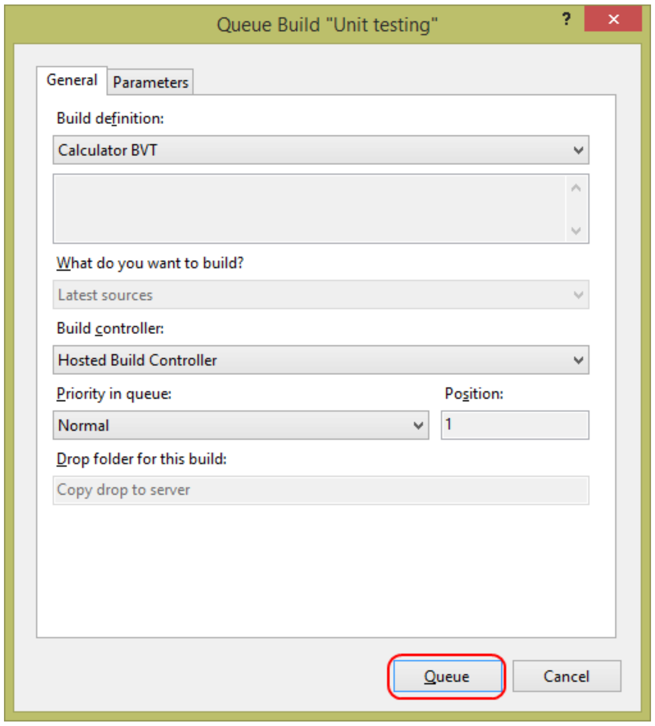
Task 3: Running builds and tests on the server

In this task, you'll use the build definition created in the last task to run builds that also execute the provided unit tests.

1. To test the build definition, you can manually queue a build for the build service. In the **Builds** section of **Team Explorer**, right-click the build and select **Queue New Build…**.



2. In the **Queue Build** dialog, you can customize exactly how the build will be set up. Since you already set the default **Build controller** during the creation of the build definition, you can just click the **Queue** button.



Build definition: Calculator BVT

What do you want to build? Latest sources

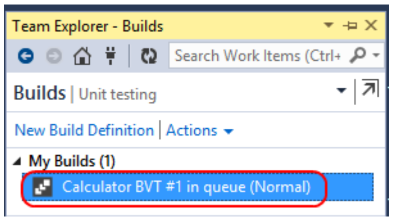
Build controller: Hosted Build Controller

Priority in queue: Normal

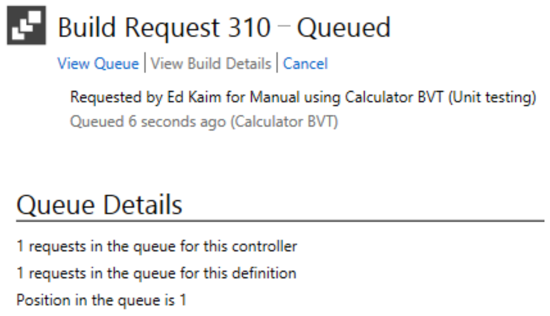
Position: 1

Drop folder for this build: Copy drop to server

3. After the build has been queued, it'll appear in **Team Explorer**. You can double-click the build to open the build overview.



Depending on how much build traffic you're competing with today, it may take a few minutes before your build begins.



Build Request 310 – Queued

View Queue | View Build Details | Cancel

Requested by Ed Kaim for Manual using Calculator BVT (Unit testing)

Queued 6 seconds ago (Calculator BVT)

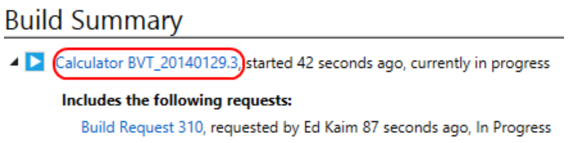
Queue Details

1 requests in the queue for this controller

1 requests in the queue for this definition

Position in the queue is 1

4. Scroll down to the **Build Summary** section and click the link to your build. It should start with **"Calculator BVT".**



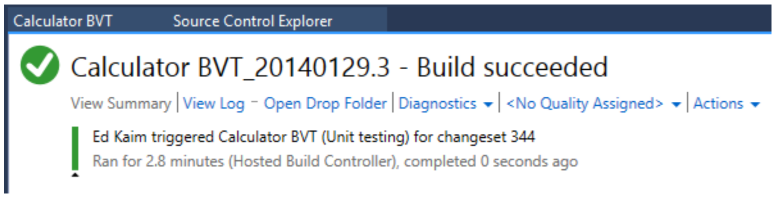
Build Summary

Calculator BVT_20140129.3 started 42 seconds ago, currently in progress

Includes the following requests:

Build Request 310, requested by Ed Kaim 87 seconds ago, In Progress

This will bring you to a detailed view of the build. The page should auto-refresh every 30 seconds. When the build finishes, you should see a page similar to this one.



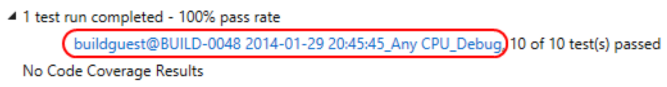
Calculator BVT_20140129.3 - Build succeeded

View Summary | View Log – Open Drop Folder | Diagnostics | <No Quality Assigned> | Actions

Ed Kaim triggered Calculator BVT (Unit testing) for changeset 344

Ran for 2.8 minutes (Hosted Build Controller), completed 0 seconds ago

5. Scroll down to the test run details. This outlines the results of your unit tests. Click the link to open the test results.

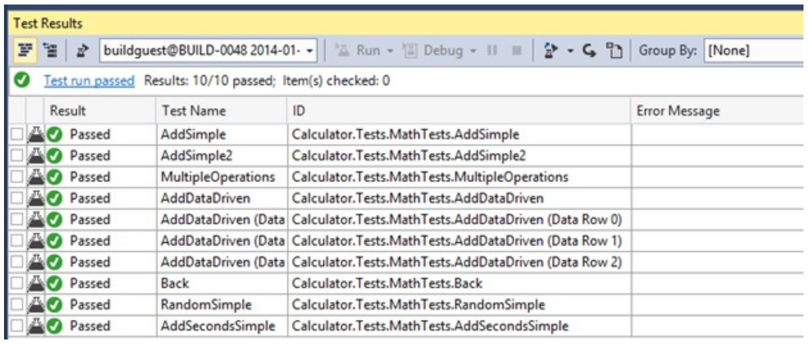


1 test run completed - 100% pass rate

buildguest@BUILD-0048 2014-01-29 20:45:45_Any CPU_Debug 10 of 10 test(s) passed

No Code Coverage Results

The test results will open in Visual Studio and provide the same details you would get if they ran locally.

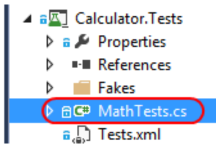


Test run passed Results: 10/10 passed; Item(s) checked: 0

Result	Test Name	ID	Error Message
Passed	AddSimple	Calculator.Tests.MathTests.AddSimple	
Passed	AddSimple2	Calculator.Tests.MathTests.AddSimple2	
Passed	MultipleOperations	Calculator.Tests.MathTests.MultipleOperations	
Passed	AddDataDriven	Calculator.Tests.MathTests.AddDataDriven	
Passed	AddDataDriven (Data	Calculator.Tests.MathTests.AddDataDriven (Data Row 0)	
Passed	AddDataDriven (Data	Calculator.Tests.MathTests.AddDataDriven (Data Row 1)	
Passed	AddDataDriven (Data	Calculator.Tests.MathTests.AddDataDriven (Data Row 2)	
Passed	Back	Calculator.Tests.MathTests.Back	
Passed	RandomSimple	Calculator.Tests.MathTests.RandomSimple	
Passed	AddSecondsSimple	Calculator.Tests.MathTests.AddSecondsSimple	

Now that you've confirmed that the build works properly, it's time to check in a change to confirm that it kicks off automatically.

6. Open **MathTests.cs** if not already open.



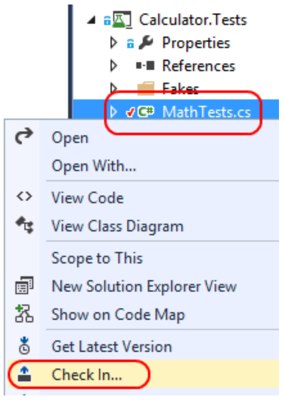
7. Add the following test. It is designed to fail.

```
[TestMethod]
public void Fail()
{
    calculator.CurrentValue = 1;
    calculator.AddCommand.Execute(null);
    calculator.CurrentValue = 1;
    calculator.EquateCommand.Execute(null);

    Assert.AreEqual(0, calculator.CurrentValue);
```

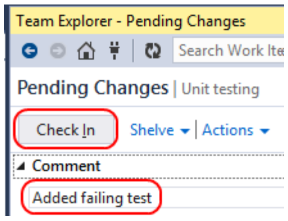
```
  }
```

8. Press **Ctrl+S** to save the file.

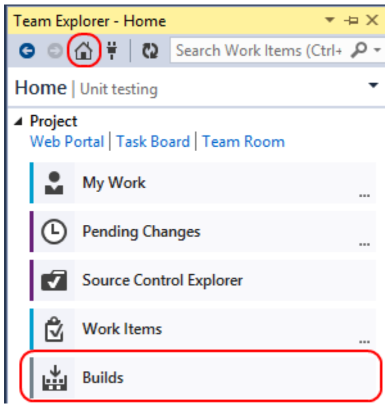
9. In **Solution Explorer**, right-click **MathTests.cs** and select **Check In…**.



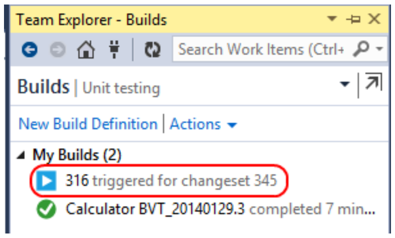
10. In **Team Explorer**, type **"Added failing test"** as the **Comment** and click **Check In** to check in the change.


Team Explorer - Pending Changes
Search Work Ite
Pending Changes | Unit testing
Check In
Shelve
Actions
Comment
Added failing test

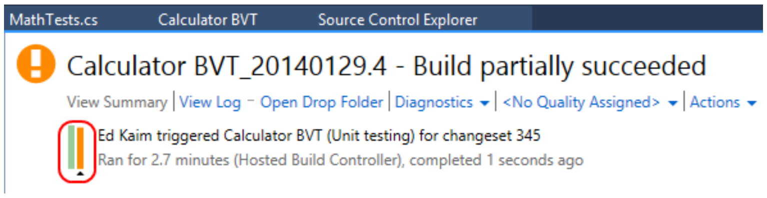
11. Click the **Home** button, followed by the **Builds** button.



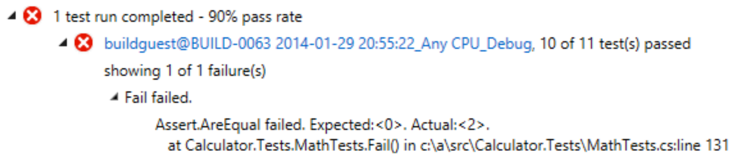
12. There should now be a new build automatically triggered from the check in. Double-click it to open.


Team Explorer - Builds
Search Work Items (Ctrl+
Builds | Unit testing
New Build Definition | Actions
My Builds (2)
316 triggered for changeset 345
Calculator BVT_20140129.3 completed 7 min...

13. Click through to the build summary. When it finishes, you should get a "Build partially succeeded" message. In this case, it means that the build was successful, but that not all the tests passed (which is expected).


MathTests.cs
Calculator BVT
Source Control Explorer
Calculator BVT_20140129.4 - Build partially succeeded
View Summary | View Log - Open Drop Folder | Diagnostics | <No Quality Assigned> | Actions
Ed Kaim triggered Calculator BVT (Unit testing) for changeset 345
Ran for 2.7 minutes (Hosted Build Controller), completed 1 seconds ago

14. Scroll down to find and expand the test results. You'll see that the expected test failed.

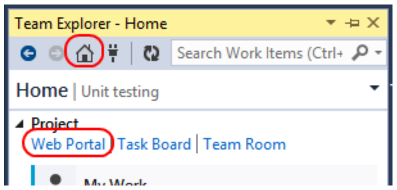


```
1 test run completed - 90% pass rate
    buildguest@BUILD-0063 2014-01-29 20:55:22_Any CPU_Debug, 10 of 11 test(s) passed
    showing 1 of 1 failure(s)
      Fail failed.
          Assert.AreEqual failed. Expected:<0>. Actual:<2>.
            at Calculator.Tests.MathTests.Fail() in c:\a\src\Calculator.Tests\MathTests.cs:line 131
```

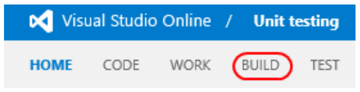
Task 4: Viewing and queuing builds on Visual Studio Online

In this task, you'll explore the build and test support provided by the Visual Studio Online portal.

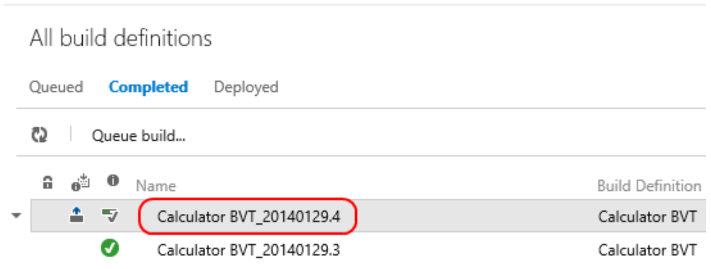
1. In **Team Explorer**, click the **Home** button and then click the **Web Portal** link. This will open a browser window to your project portal.



2. Click the **Build** link from the navigation menu.

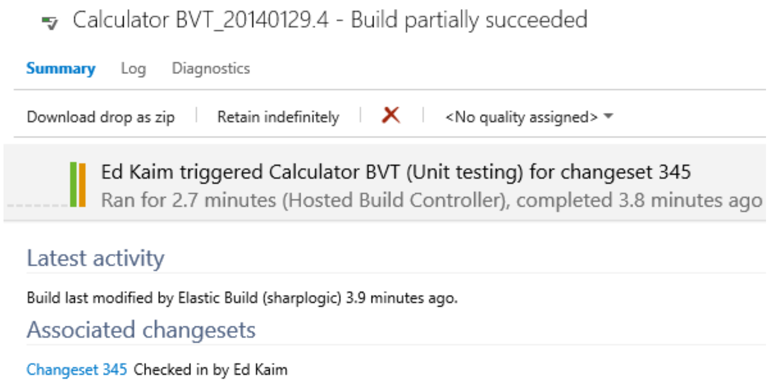


3. By default, this view will provide you with a list of today's completed builds. You can change the filter in the top right corner to customize what builds are shown. You can also queue a build from here, which would present you with similar options to the ones from Visual Studio. Double-click the first build, which is the most recent one.



Name	Build Definition
Calculator BVT_20140129.4	Calculator BVT
Calculator BVT_20140129.3	Calculator BVT

The build results appear in a way that's similar to what you saw in Visual Studio. You also have the ability to download the whole drop as a zip file, mark the build to be retained indefinitely, or even delete the build.



Ed Kaim triggered Calculator BVT (Unit testing) for changeset 345
Ran for 2.7 minutes (Hosted Build Controller), completed 3.8 minutes ago

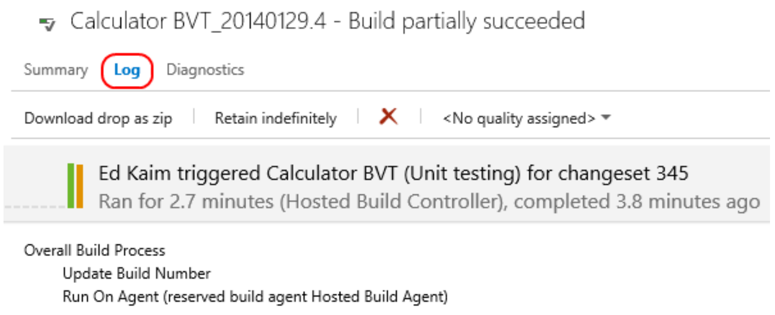
Latest activity

Build last modified by Elastic Build (sharplogic) 3.9 minutes ago.

Associated changesets

Changeset 345 Checked in by Ed Kaim

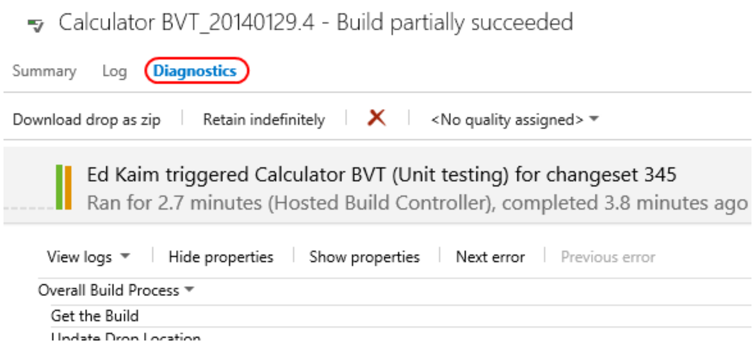
4. Click the **Log** link. The log view provides the steps followed during the build workflow, along with their warnings and errors, if applicable.



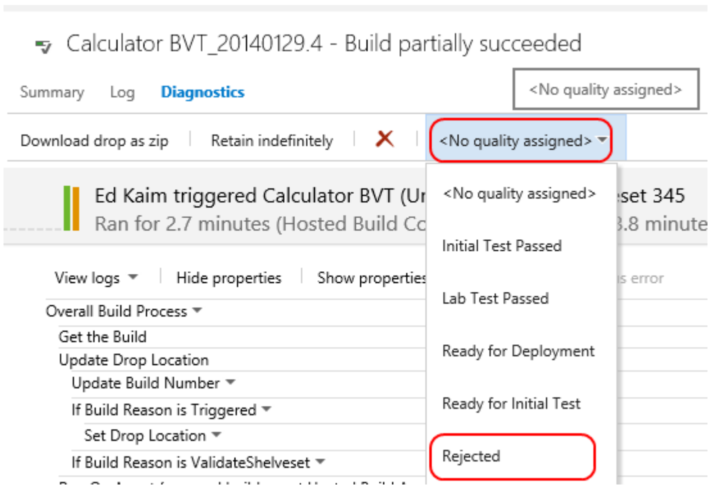
Ed Kaim triggered Calculator BVT (Unit testing) for changeset 345
Ran for 2.7 minutes (Hosted Build Controller), completed 3.8 minutes ago

```
Overall Build Process
    Update Build Number
    Run On Agent (reserved build agent Hosted Build Agent)
```

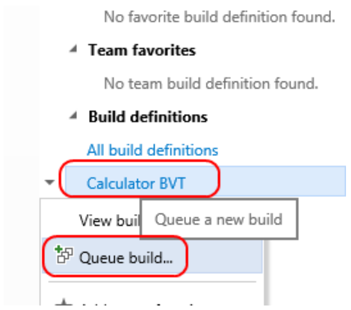
5. Click the **Diagnostics** link. This view is similar to the log view, except that it offers a verbose level of detail regarding each individual action taken and its result. This view is extremely helpful when troubleshooting build definitions.


Calculator BVT_20140129.4 - Build partially succeeded
Summary
Log
Diagnostics
Download drop as zip
Retain indefinitely
<No quality assigned>
Ed Kaim triggered Calculator BVT (Unit testing) for changeset 345
Ran for 2.7 minutes (Hosted Build Controller), completed 3.8 minutes ago
View logs
Hide properties
Show properties
Next error
Previous error
Overall Build Process
Get the Build

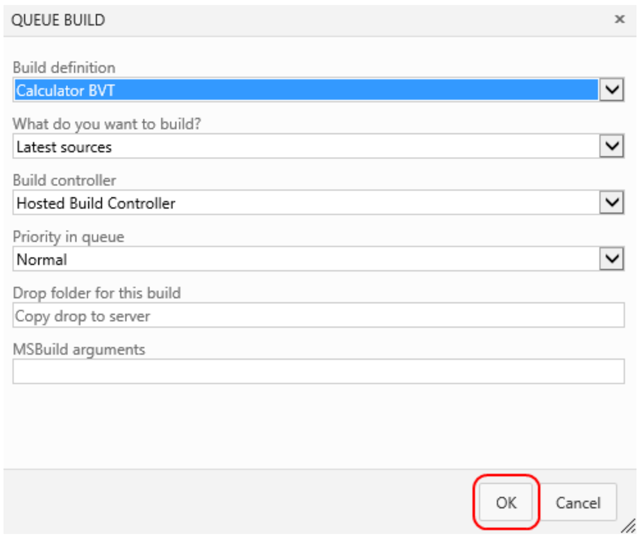
6. Finally, you can assign a quality level to this build. Since it failed, select **Rejected** so that other team members can know to avoid it for deployment.



7. Besides tracking and viewing builds, you can also queue new builds from the portal. In the left panel, locate the **Build Definitions** node. Right-click **Calculator BVT** and select **Queue build…**.



8. In the **Queue Build** dialog, accept the defaults and click **OK**. This will queue a build just like you did earlier from Visual Studio. Note that virtually all of the build queuing and tracking functionality is shared between Visual Studio and the Visual Studio Online project portal.



QUEUE BUILD

Build definition: Calculator BVT

What do you want to build?: Latest sources

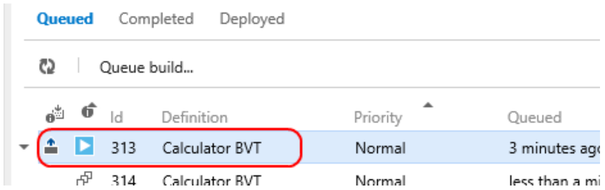
Build controller: Hosted Build Controller

Priority in queue: Normal

Drop folder for this build: Copy drop to server

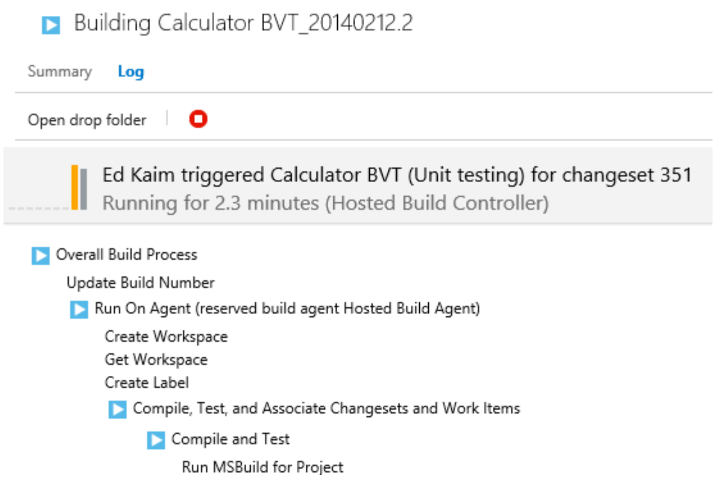
MSBuild arguments:

9. After queuing, locate the new build in the main view and double-click it.



Id	Definition	Priority	Queued
313	Calculator BVT	Normal	3 minutes ag
314	Calculator BVT	Normal	less than a mi

10. Note that you can view build progress in the portal just like in Visual Studio.

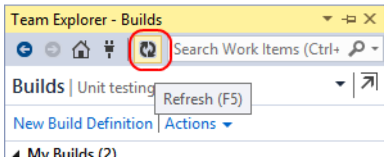


Building Calculator BVT_20140212.2

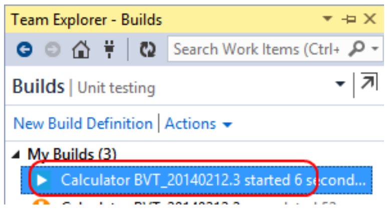
Ed Kaim triggered Calculator BVT (Unit testing) for changeset 351
Running for 2.3 minutes (Hosted Build Controller)

```
Overall Build Process
  Update Build Number
  Run On Agent (reserved build agent Hosted Build Agent)
    Create Workspace
    Get Workspace
    Create Label
    Compile, Test, and Associate Changesets and Work Items
      Compile and Test
        Run MSBuild for Project
```

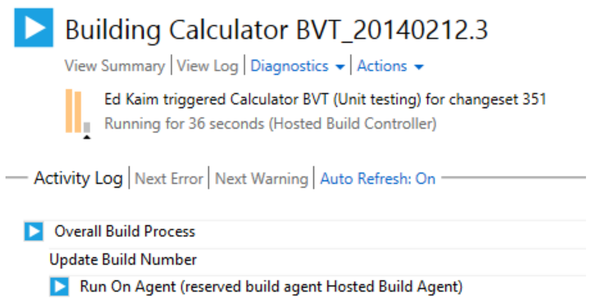
11. Back in **Visual Studio**, switch to the **Builds** section of **Team Explorer** if it's not already open. Otherwise, click the **Refresh** button to refresh the view.



12. The new build will appear at the top. Double-click it to open.



Now you're viewing the same experience you just saw in the portal. In many cases you may find yourself using Visual Studio, but it's great to know that so much of the development experience is still available, even when working from a phone or tablet browser.


Building Calculator BVT_20140212.3

View Summary | View Log | Diagnostics | Actions

Ed Kaim triggered Calculator BVT (Unit testing) for changeset 351

Running for 36 seconds (Hosted Build Controller)

Activity Log | Next Error | Next Warning | Auto Refresh: On

```
Overall Build Process
  Update Build Number
  Run On Agent (reserved build agent Hosted Build Agent)
```