

Contents

[Testing tools in Visual Studio](#)

[Unit testing](#)

[Unit test tools overview](#)

[Get started](#)

[Unit test basics](#)

[Create a unit test project](#)

[Create Unit Tests command](#)

[IntelliTest](#)

[How to: Generate unit tests with IntelliTest](#)

[Overview](#)

[Getting started](#)

[Index to sections](#)

[Test generation](#)

[Dynamic symbolic execution](#)

[Exploration bounds](#)

[Attribute glossary](#)

[Settings waterfall](#)

[Static helper classes](#)

[Warnings and errors](#)

[Install third-party unit test frameworks](#)

[Test Explorer](#)

[Run unit tests with Test Explorer](#)

[Test Explorer FAQ](#)

[Run tests from the command line](#)

[Run a unit test as a 64-bit process](#)

[Configure unit tests by using a .runsettings file](#)

[Write unit tests for managed code](#)

[Walkthrough: Create and run unit tests for managed code](#)

[Walkthrough: Test-driven development](#)

- [Use the MSTest API in unit tests](#)
- [Use the assert classes](#)
- [Microsoft Fakes](#)
 - [Isolate code under test](#)
 - [Use stubs to isolate parts of your app](#)
 - [Use shims to isolate your app from other assemblies](#)
 - [Code generation, compilation, and naming conventions](#)
- [Data-driven unit tests](#)
 - [Create a data-driven unit test](#)
 - [Use a configuration file to define a data source](#)
- [Unit tests for generic methods](#)
- [Configure unit tests to target an earlier version of .NET](#)
- [Unit tests for C/C++ code](#)
 - [Write unit tests for C/C++ code](#)
 - [Use the Microsoft Unit Testing Framework for C++](#)
 - [Use Google C++ Testing Framework](#)
 - [Use Boost.Test](#)
 - [Use CTest](#)
 - [Write unit tests for C/C++ DLLs](#)
 - [Walkthrough: Writing Unit tests for C++ DLLs](#)
 - [Microsoft Unit Testing Framework for C++ API Reference](#)
- [Create and run unit tests for UWP apps](#)
 - [Unit test a C++ UWP DLL](#)
 - [Unit test Visual C# code in UWP apps](#)
 - [Walkthrough: Create and run unit tests for UWP apps](#)
- [Code coverage](#)
 - [Use code coverage to determine how much code is being tested](#)
 - [Customize code coverage analysis](#)
 - [Troubleshoot code coverage](#)
- [Live Unit Testing](#)
 - [Introduction](#)
 - [What's new](#)

[Get started](#)

[Configure and use](#)

[FAQ](#)

[Web performance and load testing](#)

[Quickstart: Create a load test project](#)

[Load test configuration](#)

[Configure load tests](#)

[Scenarios and test mixes](#)

[Edit scenarios](#)

[Scenario properties](#)

[Test mix](#)

[Test mix models](#)

[Overview](#)

[Edit test mix models](#)

[Browser mix](#)

[Network mix](#)

[Think profile](#)

[Load patterns](#)

[Load patterns](#)

[Step ramp time property](#)

[Start delays](#)

[Test iterations](#)

[Percentage of new users](#)

[Pacing delay distribution](#)

[Counter sets](#)

[Edit counter sets](#)

[Add counters](#)

[Custom counter sets](#)

[Manage counter sets](#)

[Add a threshold rule](#)

[Run settings](#)

[Edit run settings](#)

- [Run settings properties](#)
- [Add additional run settings](#)
- [Context parameters](#)
- [Logging settings](#)
- [Modify logging settings](#)
- [Failed tests](#)
- [Timing details storage](#)
- [Test iterations](#)
- [Sample rate](#)
- [Select a run setting](#)
 - [Select a run setting](#)
 - [Command line](#)
- [Walkthrough: Create and run a load test](#)
- [Web performance tests](#)
 - [Create a web service test](#)
 - [Generate a coded Web performance test](#)
 - [Add a data source to a Web performance test](#)
 - [Dynamic parameters](#)
- [Test controllers and test agents](#)
 - [Overview](#)
 - [Requirements](#)
 - [Manage test controllers and test agents](#)
 - [How to specify test agents](#)
 - [Assign roles](#)
 - [Configure ports](#)
 - [Bind to a network adapter](#)
 - [Run tests that interact with the desktop](#)
 - [Timeout periods](#)
 - [Troubleshoot](#)
- [Customize load tests](#)
 - [Load test API](#)
 - [Web performance test API](#)

[Create a custom extraction rule](#)

[Create a custom validation rule](#)

[Custom code and plug-ins](#)

[Create custom code and plug-ins](#)

[Create a Web performance test plug-in](#)

[Create a request level plug-in](#)

[Create a load test plug-in](#)

[Create a recorder plug-in](#)

[Create a custom HTTP body editor](#)

[Diagnostics for load tests](#)

[Diagnostic data adapters](#)

[Create a diagnostic data adapter](#)

[How to create](#)

[Collect diagnostics](#)

[Collect diagnostics using test settings](#)

[Create test settings](#)

[Configure ASP.NET profiler](#)

[Configure network emulation](#)

[Collect IntelliTrace data](#)

[Screen and voice recordings](#)

[Test results](#)

[Analyze results](#)

[Access results](#)

[Summary overview](#)

[Tables view](#)

[Tables view](#)

[Web page response time](#)

[Graphs view](#)

[Graphs view](#)

[Add and delete counters](#)

[Create custom graphs](#)

[Use the legend](#)

- [How to zoom in](#)
- [Details view](#)
 - [Details view](#)
 - [Analyze virtual user activity](#)
 - [Walkthrough: Use the activity chart](#)
- [Threshold rule violations](#)
- [Compare results and analyze trends](#)
- [Create a report in Excel](#)
- [Create a report in Word](#)
- [Results repositories](#)
 - [Results repositories](#)
 - [Select a repository](#)
 - [Import results](#)
 - [Export results](#)
 - [Delete results](#)
- [Create an add-in for the results viewer](#)
- [UI automation using Coded UI test](#)
 - [Overview](#)
 - [Walkthrough: Create, Edit and Maintain a Coded UI Test](#)
 - [Test UWP Apps with Coded UI Tests](#)
 - [Set a Unique Automation Property for UWP Controls](#)
 - [Use HTML5 Controls in Coded UI Tests](#)
 - [Create a Data-Driven Coded UI Test](#)
 - [Wait For Specific Events During Playback](#)
 - [Use Different Web Browsers with Coded UI Tests](#)
 - [Edit Coded UI Tests Using the Coded UI Test Editor](#)
 - [Analyze Coded UI Tests Using Coded UI Test Logs](#)
 - [Anatomy of a Coded UI Test](#)
 - [Best Practices for Coded UI Tests](#)
 - [Test a Large Application with Multiple UI Maps](#)
 - [Enable Coded UI Testing of Your Controls](#)
 - [Supported Configurations and Platforms](#)

Extend the Coded UI Test Framework

Test lab management

Use a lab environment

Use build or release management for automated testing

Install test agents and test controllers

Testing tools in Visual Studio

1/1/2020 • 2 minutes to read • [Edit Online](#)

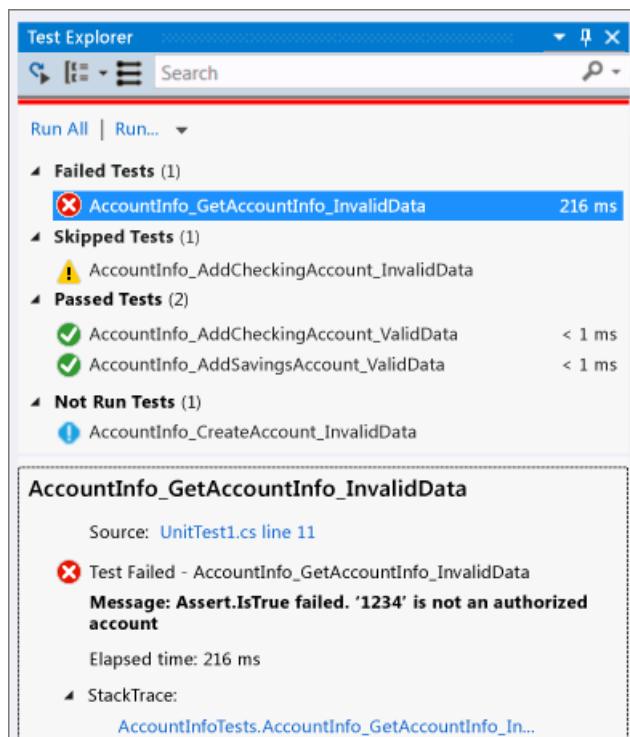
Visual Studio testing tools can help you and your team develop and sustain high standards of code excellence.

NOTE

Unit testing is available in all editions of Visual Studio. Other testing tools, such as Live Unit Testing, IntelliTest, and Coded UI Test, are only available in Visual Studio Enterprise edition. For more information about editions see [Compare Visual Studio IDEs](#).

Test Explorer

The **Test Explorer** window helps developers create, manage, and run unit tests. You can use the Microsoft unit test framework or one of several third-party and open source frameworks.



Test Explorer

17 6 5 6

Test	Duration	Error Message
>UserSentimentAnalysis.Tests (17)	83 ms	
>UserSentimentAnalysis.Tests (17)	83 ms	
EmojiTests (5)	6 ms	
EmojiClothingTest	5 ms	
EmojiExtraSpecialCharatersTest	< 1 ms	
EmojiFaceSearchTest	1 ms	
EmojiHeartsTest	< 1 ms	
TearsOfJoyTest	< 1 ms	
HomeControllerTests (6)		
TwitterDataModelTests (6)	77 ms	
ActiveCognitiveServiceTest	74 ms	Assert.AreEqual failed. Expected...
AverageTweetTest	3 ms	Test method UserSentimentAna...
GetTweetSentimentTest	< 1 ms	Assert.AreEqual failed. Expected...
HistoricalTweetTest	< 1 ms	Assert.AreEqual failed. Expected...
MultipleAverageTweetTest	< 1 ms	
ParseTweetTest	< 1 ms	Assert.AreEqual failed. Expected...

Group Summary

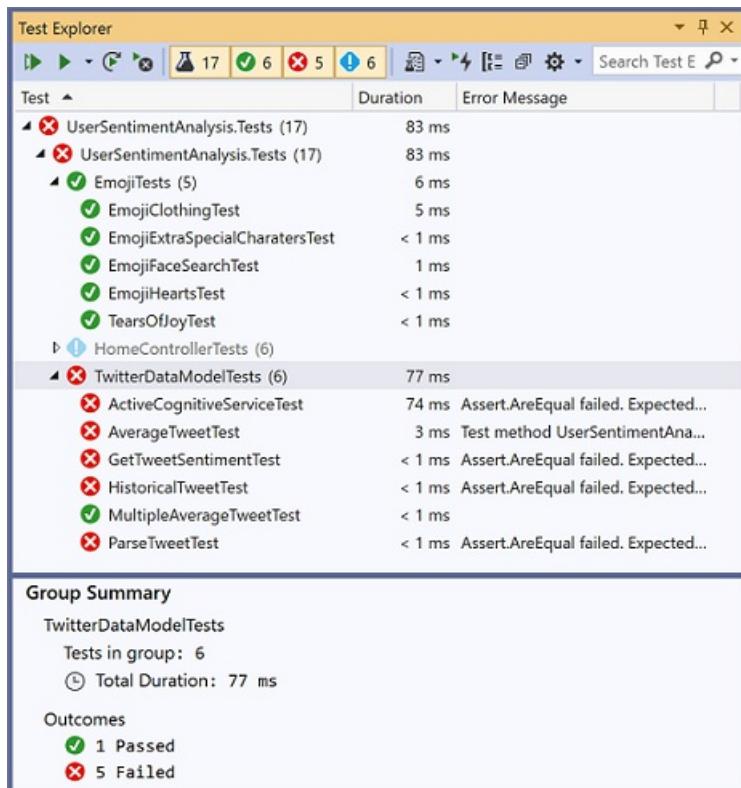
TwitterDataModelTests

Tests in group: 6

Total Duration: 77 ms

Outcomes

1 Passed
5 Failed



- [Get started with unit testing](#)
- [Run unit tests with Test Explorer](#)
- [Test Explorer FAQ](#)
- [Install third-party unit test frameworks](#)

Visual Studio is also extensible and opens the door for third-party unit testing adapters such as NUnit and xUnit.net. In addition, the code clone capability goes hand-in-hand with delivering high-quality software by helping you identify blocks of semantically similar code that may be candidates for common bug fixes or refactoring.

 ChutzpahDemo_20120705.5 - Build succeeded

[View Summary](#) | [View Log](#) - [Open Drop Folder](#) | [Diagnostics](#) ▾ | <No Quality Assigned> ▾ | [Actions](#) ▾

 Mathew Aniyan triggered ChutzpahDemo (UnitTest) for changeset 249
Ran for 87 seconds (Hosted Build Controller), completed 4 days ago

Latest Activity

Build last modified by Elastic Build (mathewan) 4 days ago.

Request Summary

[Request 55](#), requested by Mathew Aniyan 4 days ago, Completed

Summary

Debug | Any CPU

- ▷ 0 error(s), 2 warning(s)
- ▷ \$/UnitTest/Sources/ChutzpahDemo/ChutzpahDemo.sln compiled
- ◀ 1 test run completed - 100% pass rate
[buildguest@WIN-1CQH2N24PSU 2012-07-05 10:06:44_Any CPU_Debug](#), 3 of 3 test(s) passed

No Code Coverage Results

Test Results

Test Results			
	Result	Test Name	ID
	Passed	hello test	hello test::C:\a\Binaries\tests.hello test
	Passed	a basic test	a basic test::C:\a\Binaries\tests.a basic test
	Passed	add 5	add 5::C:\a\Binaries\tests.add 5

Live Unit Testing

Live Unit Testing automatically runs unit tests in the background, and graphically displays code coverage and test results in the Visual Studio code editor.

IntelliTest

IntelliTest automatically generates unit tests and test data for your managed code. IntelliTest improves coverage and dramatically reduces the effort to create and maintain unit tests for new or existing code.

IntelliTest Exploration Results - stopped

TaxCalculator.CalculateExemption

5 Passed, 2 Failed, 10/10 blocks, 0/0 asserts, 13 runs

	target	employee	ic	result(target)
1	new TaxCalc	null		null
2	new TaxCalc	new Employee(An	new HRAexe	new TaxCal
3	new TaxCalc	new Employee(An	new HRAexe	new TaxCal
4	new TaxCalc	new Employee(An	new HRAexe	new TaxCal
5	new TaxCalc	new Employee(An	new HRAexe	new TaxCal
6	new TaxCalc	new Employee(An	new HRAexe	
7	new TaxCalc	new Employee(An	new HRAexe	new TaxCal

Details

```
[TestMethod]
[PexGeneratedBy(typeof(TaxCalculator))]
[PexRaisedException(typeof(NullReferenceException))]
public void CalculateExemptionForRentThresholdException453()
{
    uint u;
    TaxCalculator s0 = new TaxCalculator();
    u = this.CalculateExemptionForRentThresholdException453();
}
```

- Generate unit tests for your code with IntelliTest

- [IntelliTest – One test to rule them all](#)
- [IntelliTest reference manual](#)

Code coverage

[Code coverage](#) determines what proportion of your project's code is actually being tested by coded tests such as unit tests. To guard effectively against bugs, your tests should exercise or "cover" a large proportion of your code.

Code coverage analysis can be applied to both managed and unmanaged (native) code.

Code coverage is an option when you run test methods using Test Explorer. The results table shows the percentage of the code that was run in each assembly, class, and method. In addition, the source editor shows you which code has been tested.

- [Use code coverage to determine how much code is being tested](#)
- [Unit testing, code coverage and code clone analysis with Visual Studio \(Lab\)](#)
- [Customize code coverage analysis](#)

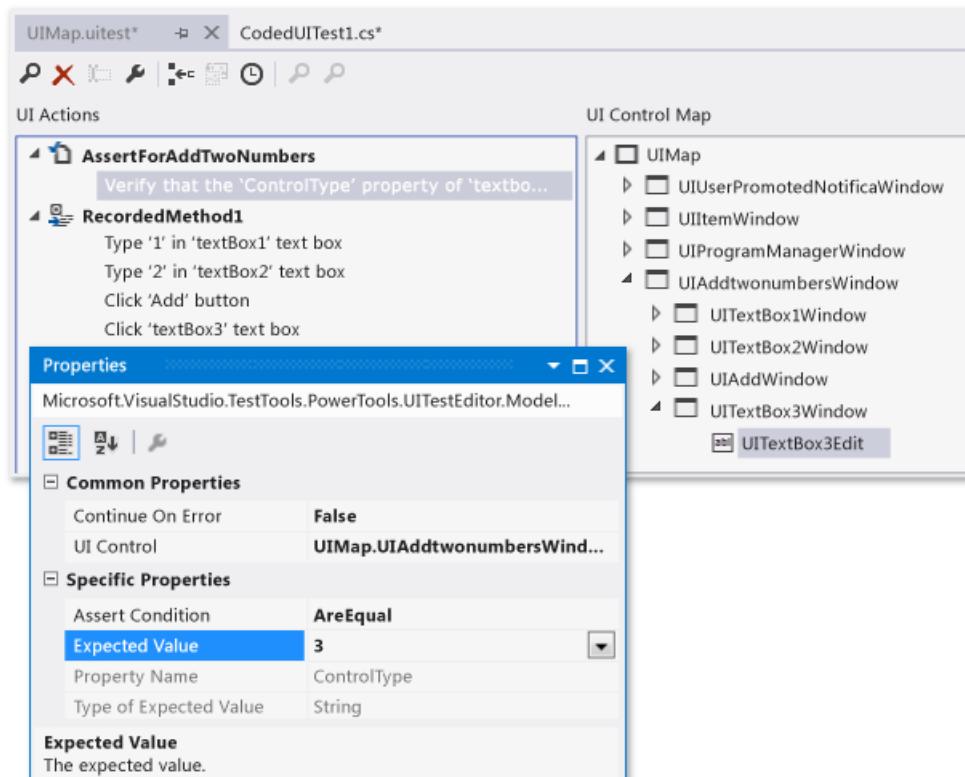
Microsoft Fakes

[Microsoft Fakes](#) help you isolate the code you're testing by replacing other parts of the application with stubs or shims.

User interface testing with Coded UI and Selenium

Coded UI tests provide a way to create fully automated tests to validate the functionality and behavior of your application's user interface. They can automate UI testing across a variety of technologies, including XAML-based UWP apps, browser apps, and SharePoint apps.

Whether you choose best-of-breed Coded UI Tests or generic browser-based UI testing with Selenium, Visual Studio provides all the tools you need.



- [Use UI automation to test your code](#)
- [Get started creating, editing, and maintaining a coded UI test](#)

- [Test UWP apps with coded UI tests](#)
- [Introduction to coded UI tests with Visual Studio Enterprise \(Lab\)](#)

Load testing

[Load testing](#) simulates load on a server application by running unit tests and web performance tests.

Related scenarios

- [Exploratory & manual testing \(Azure Test Plans\)](#)
- [Load testing \(Azure Test Plans\)](#)
- [Continuous testing \(Azure Test Plans\)](#)
- [Code analysis tools](#)

Unit test your code

1/1/2020 • 2 minutes to read • [Edit Online](#)

Unit tests give developers and testers a quick way to look for logic errors in the methods of classes in C#, Visual Basic, and C++ projects.

The unit test tools include:

- **Test Explorer**—Run unit tests and see their results in **Test Explorer**. You can use any unit test framework, including a third-party framework, that has an adapter for **Test Explorer**.
- **Microsoft unit test framework for managed code**—The Microsoft unit test framework for managed code is installed with Visual Studio and provides a framework for testing .NET code.
- **Microsoft unit test framework for C++**—The Microsoft unit test framework for C++ is installed as part of the **Desktop development with C++** workload. It provides a framework for testing native code. Google Test, Boost.Test, and CTest frameworks are also included, and third-party adapters are available for additional test frameworks. For more information, see [Write unit tests for C/C++](#).
- **Code coverage tools**—You can determine the amount of product code that your unit tests exercise from one command in Test Explorer.
- **Microsoft Fakes isolation framework**—The Microsoft Fakes isolation framework can create substitute classes and methods for production and system code that create dependencies in the code under test. By implementing the fake delegates for a function, you control the behavior and output of the dependency object.

You can also use [IntelliTest](#) to explore your .NET code to generate test data and a suite of unit tests. For every statement in the code, a test input is generated that will execute that statement. A case analysis is performed for every conditional branch in the code.

Key tasks

Use the following articles to help with understanding and creating unit tests:

TASKS	ASSOCIATED TOPICS
Quickstarts and walkthroughs: Learn about unit testing in Visual Studio from code examples.	- Walkthrough: Create and run unit tests for managed code - Quickstart: Test-driven development with Test Explorer - How to: Add unit tests to C++ apps
Unit testing with Test Explorer: Learn how Test Explorer can help create more productive and efficient unit tests.	- Unit test basics - Create a unit test project - Run unit tests with Test Explorer - Install third-party unit test frameworks
Unit test C++ code	- Write unit tests for C/C++
Isolating unit tests	- Isolate code under test with Microsoft Fakes
Use code coverage to identify what proportion of your project's code is tested: Learn about the code coverage feature of Visual Studio testing tools.	- Use code coverage to determine how much code is being tested

TASKS	ASSOCIATED TOPICS
<p>Perform stress and performance analysis by using load tests: Learn how to create load tests to help isolate performance and stress issues in your application.</p>	<ul style="list-style-type: none"> - Quickstart: Create a load test project - Load testing (Azure Test Plans and TFS)
<p>Set quality gates: Learn how to create quality gates to enforce that tests are run before code is checked in or merged.</p>	<ul style="list-style-type: none"> - Check-in policies (Azure Repos TFVC)
<p>Set testing options: Learn how to configure test options, for example, where test results are stored.</p>	Configure unit tests by using a .runsettings file

API reference documentation

- [Microsoft.VisualStudio.TestTools.UnitTesting](#) describes the UnitTesting namespace, which provides attributes, exceptions, asserts, and other classes that support unit testing.
- [Microsoft.VisualStudio.TestTools.UnitTesting.Web](#) describes the UnitTesting.Web namespace, which extends the UnitTesting namespace by providing support for ASP.NET and web service unit tests.

See also

- [Improve code quality](#)

Get started with unit testing

1/1/2020 • 4 minutes to read • [Edit Online](#)

Use Visual Studio to define and run unit tests to maintain code health, ensure code coverage, and find errors and faults before your customers do. Run your unit tests frequently to make sure your code is working properly.

Create unit tests

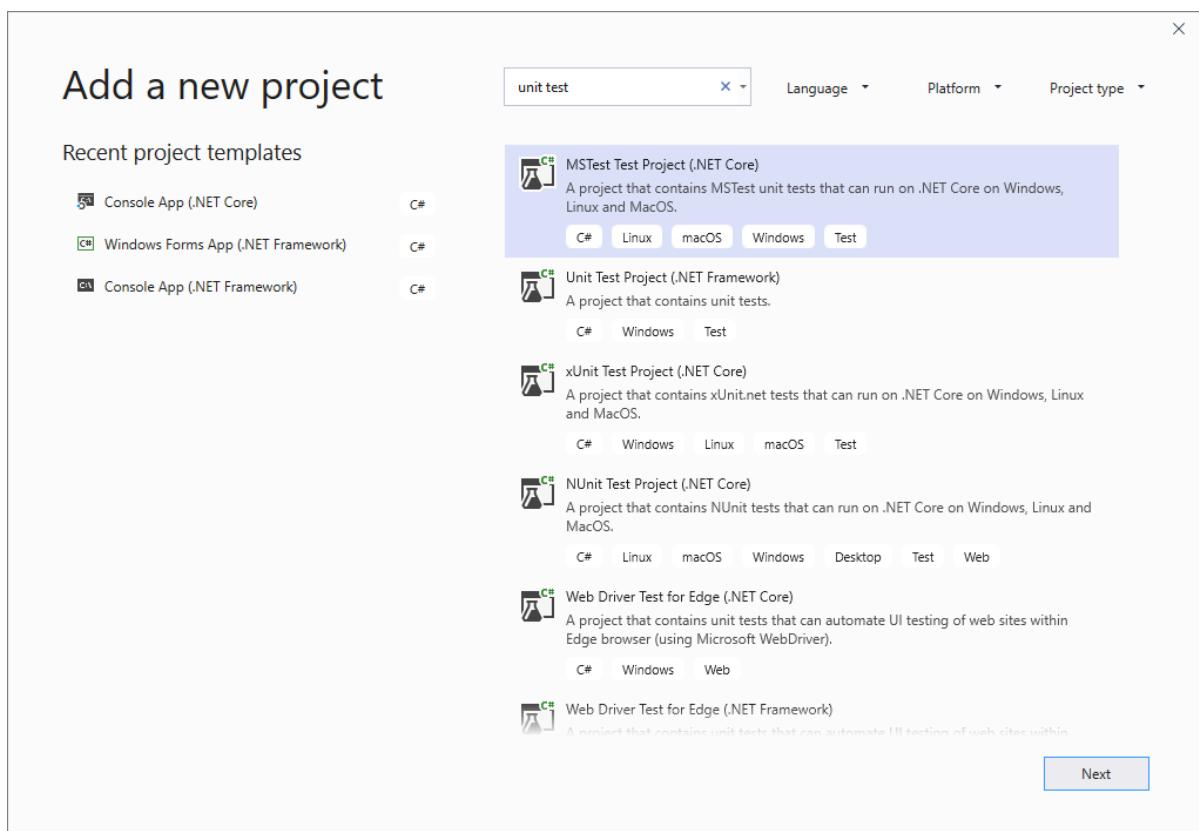
This section describes at a high level how to create a unit test project.

1. Open the project that you want to test in Visual Studio.

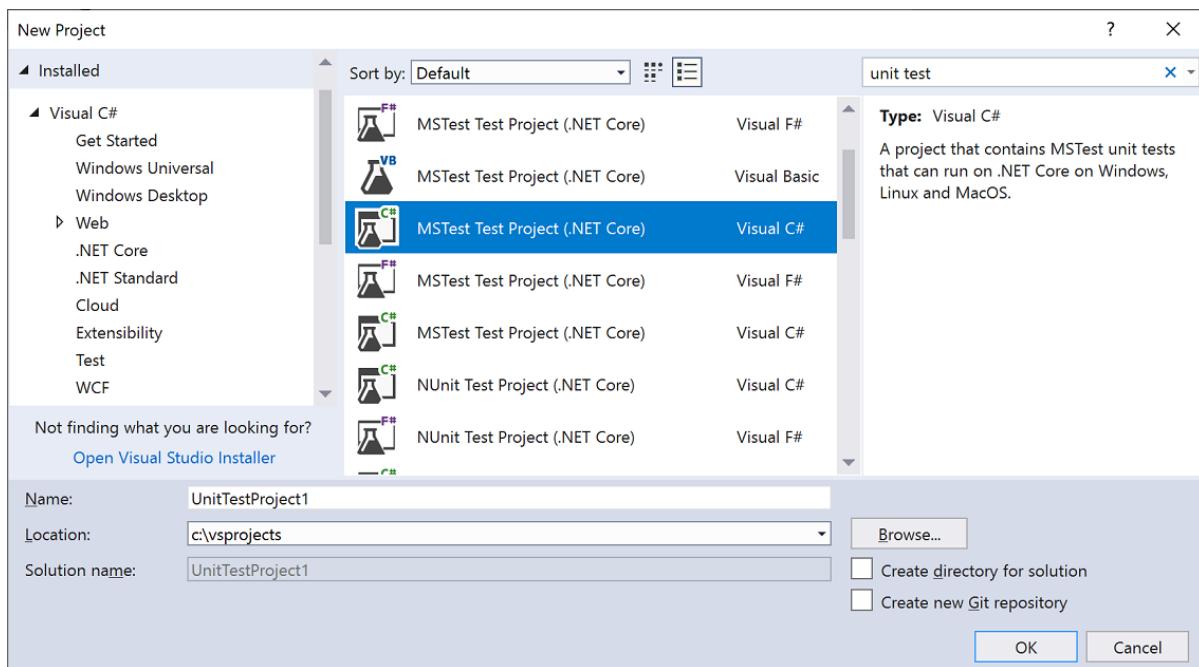
For the purposes of demonstrating an example unit test, this article tests a simple "Hello World" project. The sample code for such a project is as follows:

```
public class Program
{
    public static void Main()
    {
        Console.WriteLine("Hello World!");
    }
}
```

2. In **Solution Explorer**, select the solution node. Then, from the top menu bar, select **File > Add > New Project**.
3. In the new project dialog box, find a unit test project template for the test framework you want to use and select it.

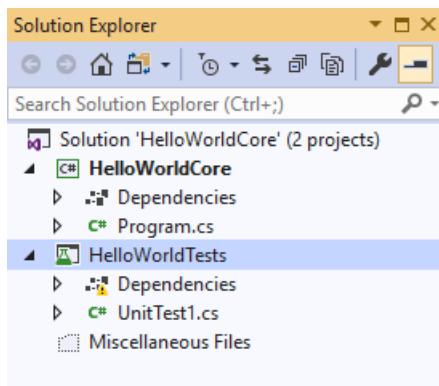


Click **Next**, choose a name for the test project, and then click **Create**.

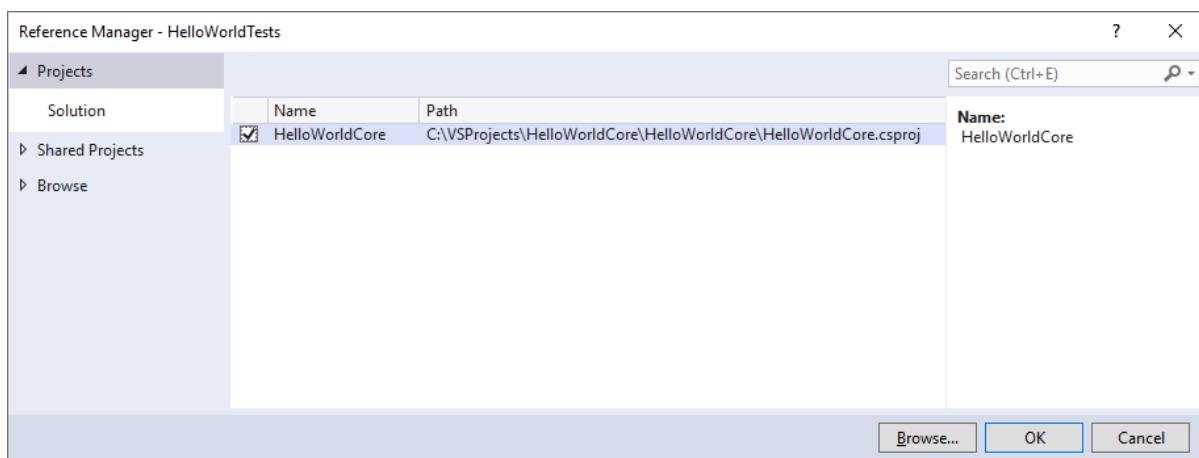


Choose a name for the test project, and then click **OK**.

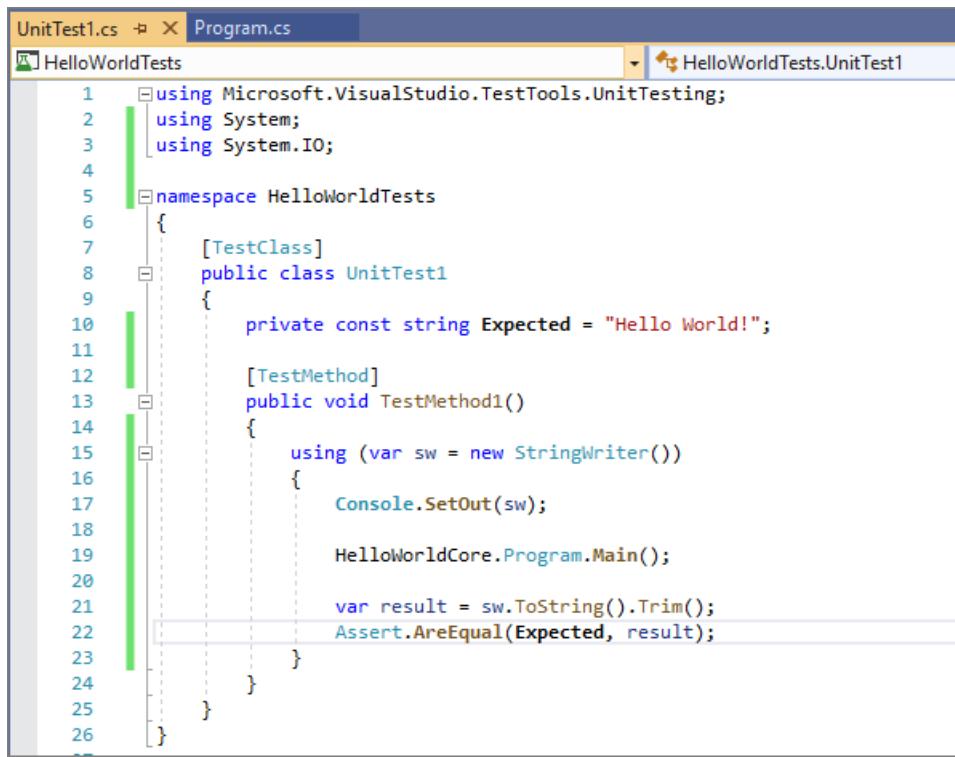
The project is added to your solution.



4. In the unit test project, add a reference to the project you want to test by right-clicking on **References** or **Dependencies** and then choosing **Add Reference**.
5. Select the project that contains the code you'll test and click **OK**.



6. Add code to the unit test method.



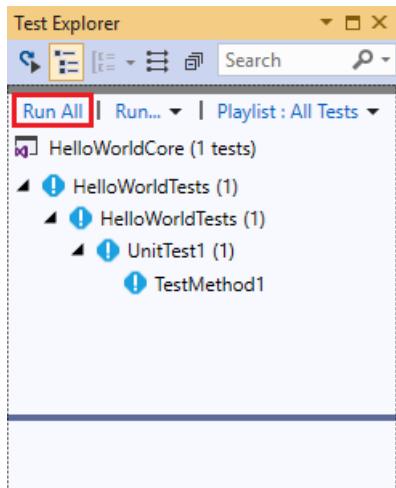
```
1  using Microsoft.VisualStudio.TestTools.UnitTesting;
2  using System;
3  using System.IO;
4
5  namespace HelloWorldTests
6  {
7      [TestClass]
8      public class UnitTest1
9      {
10         private const string Expected = "Hello World!";
11
12         [TestMethod]
13         public void TestMethod1()
14         {
15             using (var sw = new StringWriter())
16             {
17                 Console.SetOut(sw);
18
19                 HelloWorldCore.Program.Main();
20
21                 var result = sw.ToString().Trim();
22                 Assert.AreEqual(Expected, result);
23             }
24         }
25     }
26 }
```

TIP

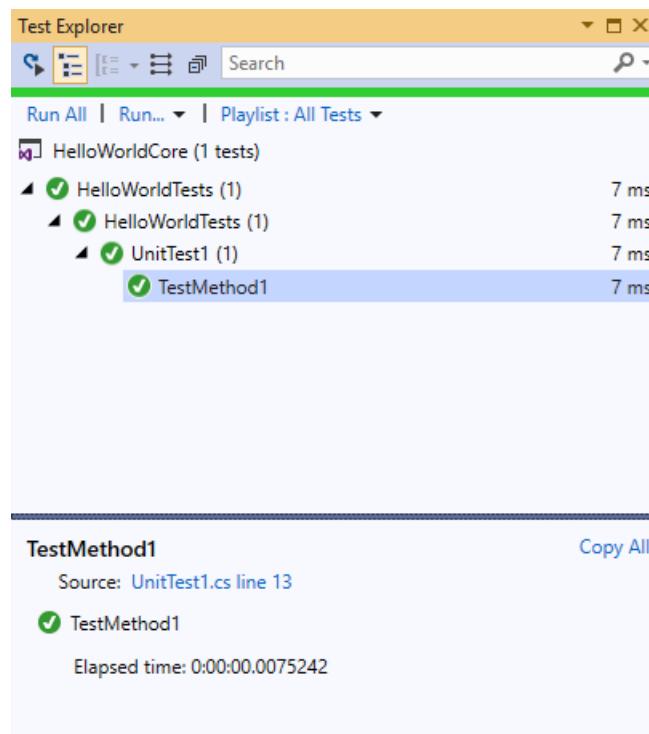
For a more detailed walkthrough of creating unit tests, see [Create and run unit tests for managed code](#).

Run unit tests

1. Open **Test Explorer** by choosing **Test > Windows > Test Explorer** from the top menu bar.
2. Run your unit tests by clicking **Run All**.



After the tests have completed, a green check mark indicates that a test passed. A red "x" icon indicates that a test failed.



TIP

You can use [Test Explorer](#) to run unit tests from the built-in test framework (MSTest) or from third-party test frameworks. You can group tests into categories, filter the test list, and create, save, and run playlists of tests. You can also debug tests and analyze test performance and code coverage.

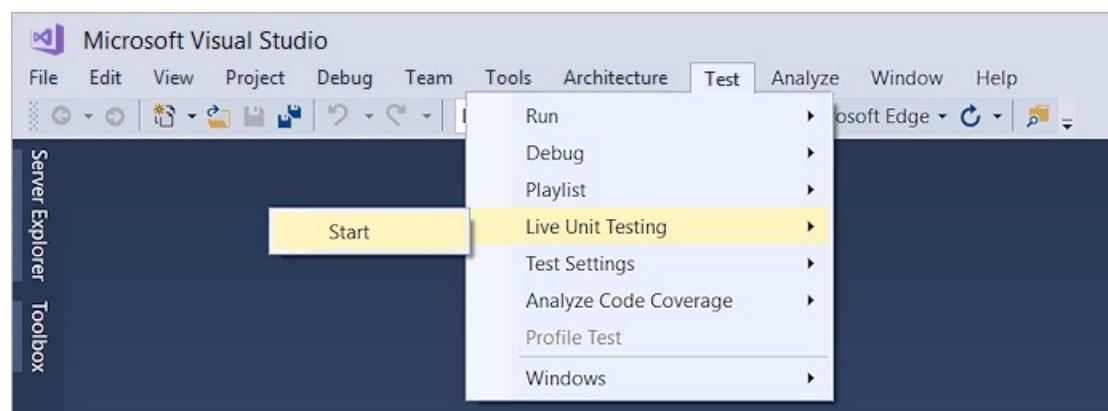
View live unit test results

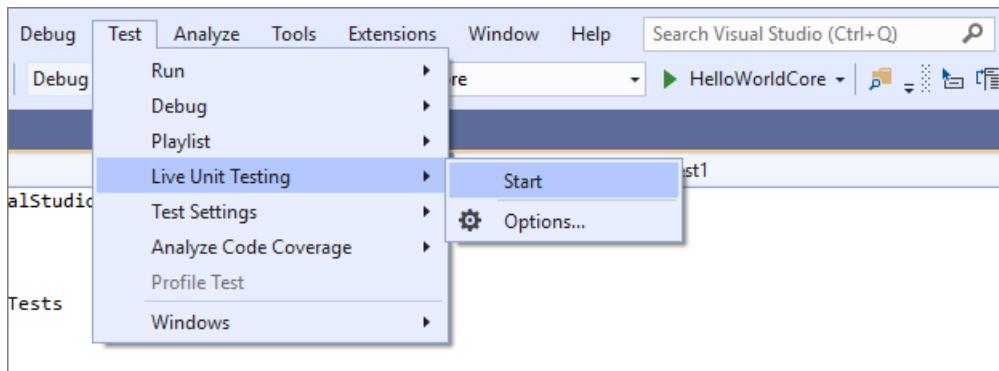
If you are using the MSTest, xUnit, or NUnit testing framework in Visual Studio 2017 or later, you can see live results of your unit tests.

NOTE

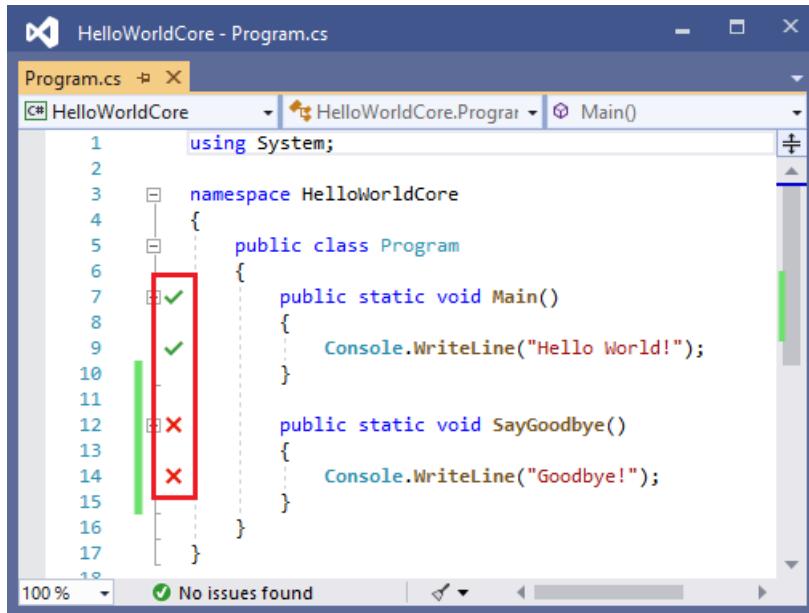
Live unit testing is available in Enterprise edition only.

1. Turn live unit testing from the **Test** menu by choosing **Test > Live Unit Testing > Start**.

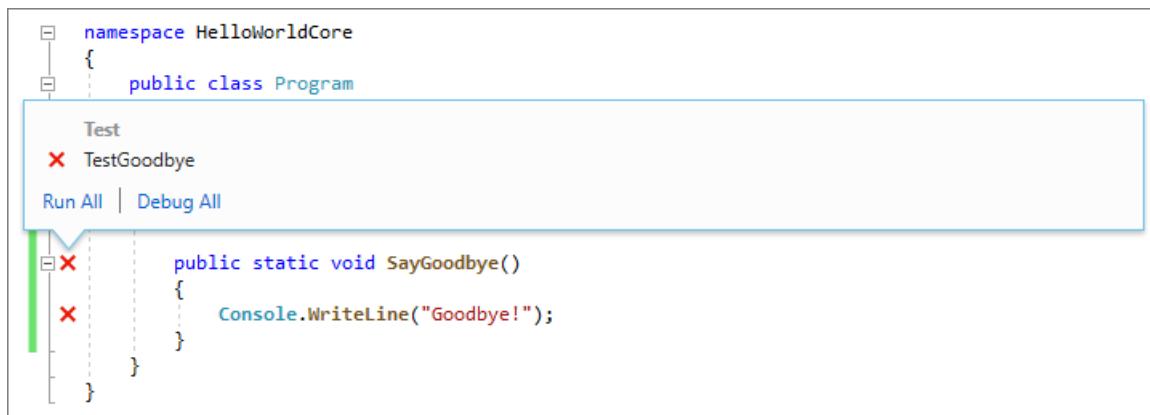




2. View the results of the tests within the code editor window as you write and edit code.



3. Click a test result indicator to see more information, such as the names of the tests that cover that method.



For more information about live unit testing, see [Live unit testing](#).

Generate unit tests with IntelliTest

When you run IntelliTest, you can see which tests are failing and add any necessary code to fix them. You can select which of the generated tests to save into a test project to provide a regression suite. As you change your code, rerun IntelliTest to keep the generated tests in sync with your code changes. To learn how, see [Generate unit tests for your code with IntelliTest](#).

TIP

IntelliTest is only available for managed code that targets the .NET Framework.

IntelliTest Exploration Results - stopped				
Triangle.ClassifyBySideLengths(int[])		Run	Stop	0 Warnings
Result	lengths	result	Summary/Exception	Error Message
✗	1 null		NullReferenceException	Object refer...
✗	2 {}		IndexOutOfRangeException	Index was out...
✗	3 {0}		IndexOutOfRangeException	Index was out...
✗	4 {0, 0}		IndexOutOfRangeException	Index was out...
✓	5 {0, 0, 0}	Invalid		
✓	6 {5, 538, 0}	Invalid		
✓	7 {67, 0, 0}	Invalid		
✓	8 {422, 536, 6...}	Scalene		
✓	9 {528, 413, 5...}	Isosceles		
✓	10 {2, 2, 3}	Isosceles		
✓	11 {1, 512, 512}	Isosceles		
✓	12 {512, 512, 5...	Equilateral		

▷ Details:

✗ Stack trace:

```
System.NullReferenceException...
at Triangle.ClassifyBySideLengths...
at TriangleTest.ClassifyBySideLengths...
```

Analyze code coverage

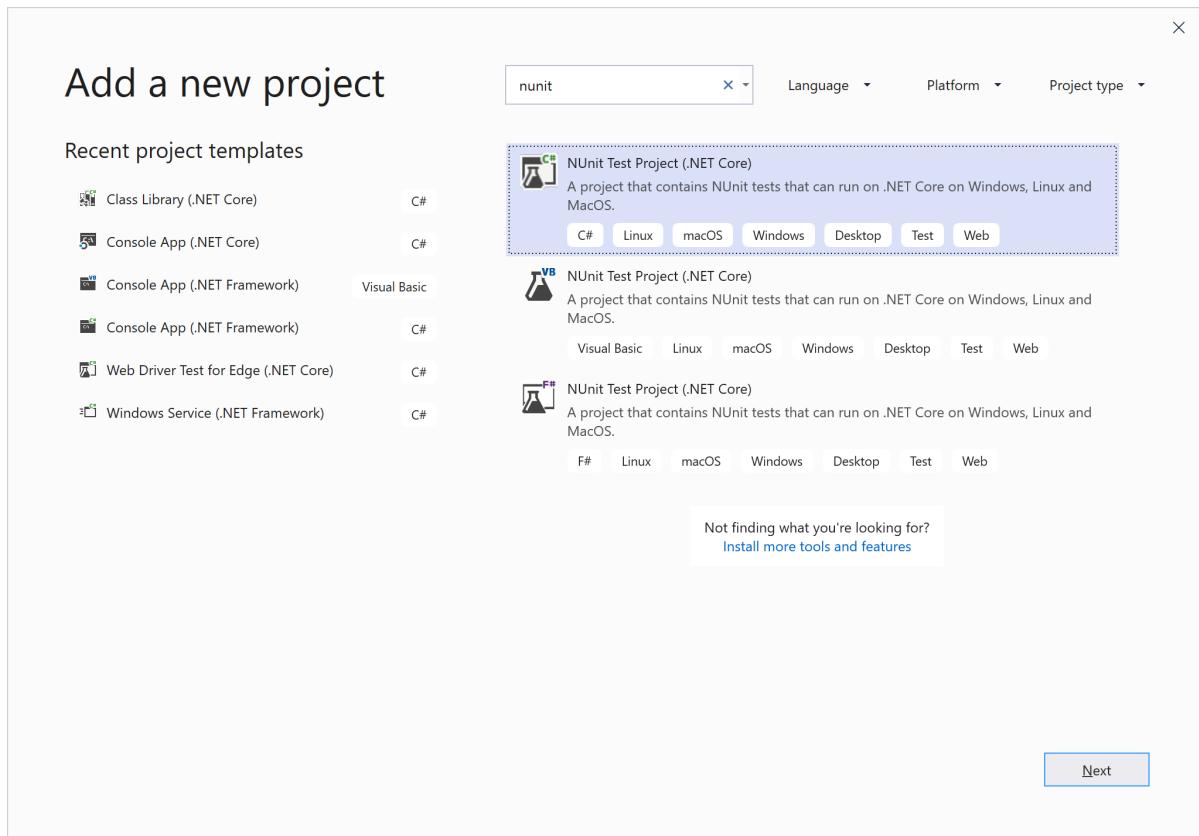
To determine what proportion of your project's code is actually being tested by coded tests such as unit tests, you can use the code coverage feature of Visual Studio. To guard effectively against bugs, your tests should exercise a large proportion of your code. To learn how, see [Use code coverage to determine how much code is being tested](#).

Use a third-party test framework

You can run unit tests in Visual Studio by using third-party test frameworks such as Boost, Google, and NUnit. Use the **NuGet Package Manager** to install the NuGet package for the framework of your choice. Or, for the NUnit and xUnit test frameworks, Visual Studio includes preconfigured test project templates that include the necessary NuGet packages.

To create unit tests that use **NUnit**:

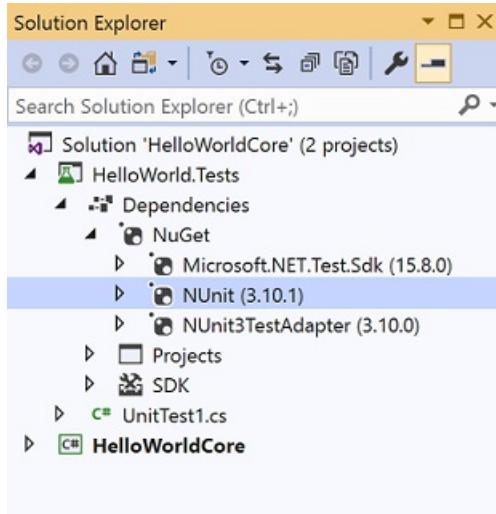
1. Open the solution that contains the code you want to test.
2. Right-click on the solution in **Solution Explorer** and choose **Add > New Project**.
3. Select the **NUnit Test Project** project template.



Click **Next**, name the project, and then click **Create**.

Name the project, and then click **OK** to create it.

The project template includes NuGet references to NUnit and NUnit3TestAdapter.



4. Add a reference from the test project to the project that contains the code you want to test.

Right-click on the project in **Solution Explorer**, and then select **Add > Reference**. (You can also add a reference from the right-click menu of the **References** or **Dependencies** node.)

5. Add code to your test method.

```
1  using Microsoft.VisualStudio.TestTools.UnitTesting;
2  using System;
3  using System.IO;
4
5  namespace HelloWorldTests
6  {
7      [TestClass]
8      public class UnitTest1
9      {
10         private const string Expected = "Hello World!";
11
12         [TestMethod]
13         public void TestMethod1()
14         {
15             using (var sw = new StringWriter())
16             {
17                 Console.SetOut(sw);
18
19                 HelloWorldCore.Program.Main();
20
21                 var result = sw.ToString().Trim();
22                 Assert.AreEqual(Expected, result);
23             }
24         }
25     }
26 }
```

6. Run the test from **Test Explorer** or by right-clicking on the test code and choosing **Run Test(s)**.

See also

- [Walkthrough: Create and run unit tests for managed code](#)
- [Create Unit Tests command](#)
- [Generate tests with IntelliTest](#)
- [Run tests with Test Explorer](#)
- [Analyze code coverage](#)

Unit test basics

1/10/2020 • 14 minutes to read • [Edit Online](#)

Check that your code is working as expected by creating and running unit tests. It's called unit testing because you break down the functionality of your program into discrete testable behaviors that you can test as individual *units*. Visual Studio Test Explorer provides a flexible and efficient way to run your unit tests and view their results in Visual Studio. Visual Studio installs the Microsoft unit testing frameworks for managed and native code. Use a *unit testing framework* to create unit tests, run them, and report the results of these tests. Rerun unit tests when you make changes to test that your code is still working correctly. Visual Studio Enterprise can do this automatically with [Live Unit Testing](#), which detects tests affected by your code changes and runs them in the background as you type.

Unit testing has the greatest effect on the quality of your code when it's an integral part of your software development workflow. As soon as you write a function or other block of application code, create unit tests that verify the behavior of the code in response to standard, boundary, and incorrect cases of input data, and that check any explicit or implicit assumptions made by the code. With *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.

You can quickly generate test projects and test methods from your code, or manually create the tests as you need them. When you use IntelliTest to explore your .NET code, you can generate test data and a suite of unit tests. For every statement in the code, a test input is generated that will execute that statement. Find out how to [generate unit tests for your code](#).

Test Explorer can also run third-party and open source unit test frameworks that have implemented Test Explorer add-on interfaces. You can add many of these frameworks through the Visual Studio Extension Manager and the Visual Studio gallery. For more information, see [Install third-party unit test frameworks](#).

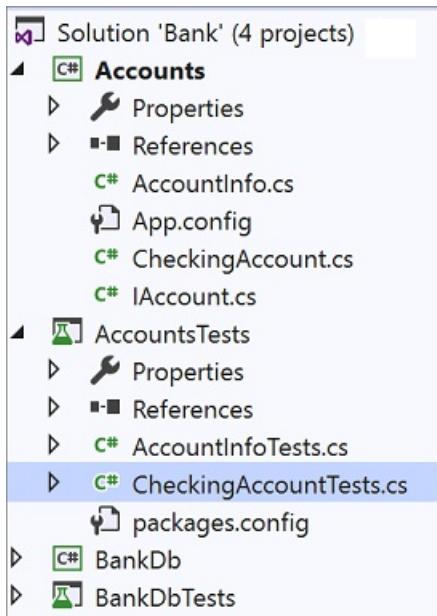
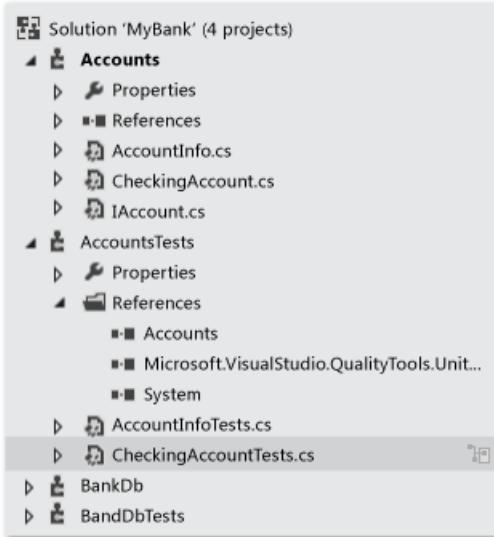
Get started

For an introduction to unit testing that takes you directly into coding, see one of these topics:

- [Walkthrough: Create and run unit tests for managed code](#)
- [Quickstart: Test driven development with Test Explorer](#)
- [Write unit tests for C/C++ in Visual Studio](#)

The MyBank solution example

In this article, we use the development of a fictional application called `MyBank` as an example. You don't need the actual code to follow the explanations in this topic. Test methods are written in C# and presented by using the Microsoft Unit Testing Framework for Managed Code. However, the concepts are easily transferred to other languages and frameworks.



Our first attempt at a design for the `MyBank` application includes an accounts component that represents an individual account and its transactions with the bank, and a database component that represents the functionality to aggregate and manage the individual accounts.

We create a `MyBank` solution that contains two projects:

- `Accounts`
- `BankDb`

Our first attempt at designing the `Accounts` project contains a class to hold basic information about an account, an interface that specifies the common functionality of any type of account, like depositing and withdrawing assets from the account, and a class derived from the interface that represents a checking account. We begin the Accounts projects by creating the following source files:

- `AccountInfo.cs` defines the basic information for an account.
- `IAccount.cs` defines a standard `IAccount` interface for an account, including methods to deposit and withdraw assets from an account and to retrieve the account balance.
- `CheckingAccount.cs` contains the `CheckingAccount` class that implements the `IAccount` interface for a checking account.

We know from experience that one thing a withdrawal from a checking account must do is to make sure that the

amount withdrawn is less than the account balance. So we override the `IAccount.Withdraw` method in `CheckingAccount` with a method that checks for this condition. The method might look like this:

```
public void Withdraw(double amount)
{
    if(m_balance >= amount)
    {
        m_balance -= amount;
    }
    else
    {
        throw new ArgumentException(nameof(amount), "Withdrawal exceeds balance!");
    }
}
```

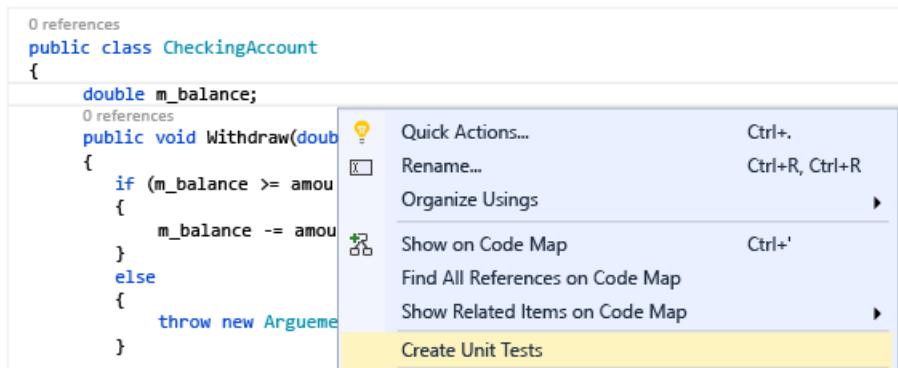
Now that we have some code, it's time for testing.

Create unit test projects and test methods

It is often quicker to generate the unit test project and unit test stubs from your code. Or you can choose to create the unit test project and tests manually depending on your requirements. If you want to create unit tests with a 3rd party framework you will need one of these extensions installed: [NUnit](#) or [xUnit](#).

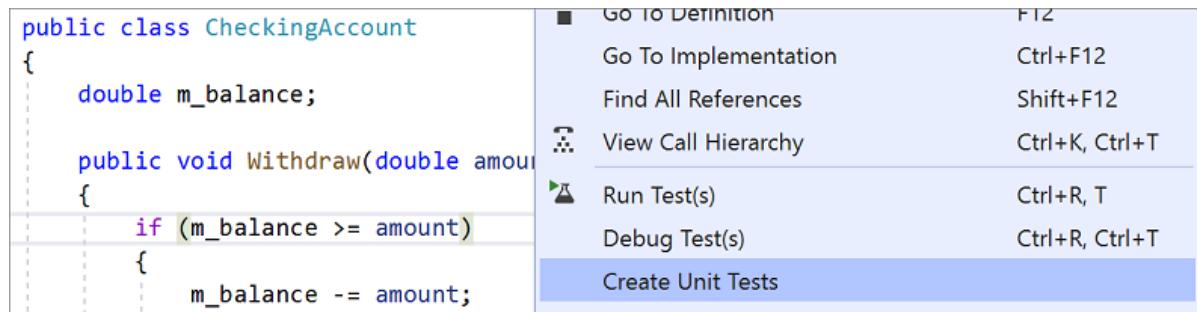
Generate unit test project and unit test stubs

- From the code editor window, right-click and choose **Create Unit Tests** from the right-click menu.



NOTE

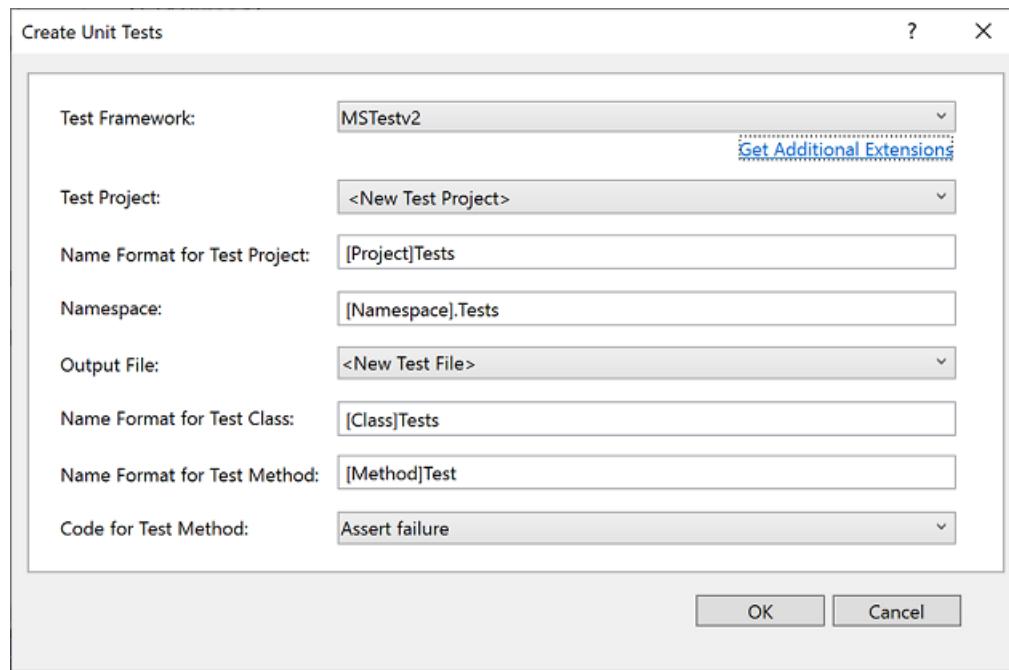
The **Create Unit Tests** menu command is only available for managed code that targets the .NET Framework (but not .NET Core).



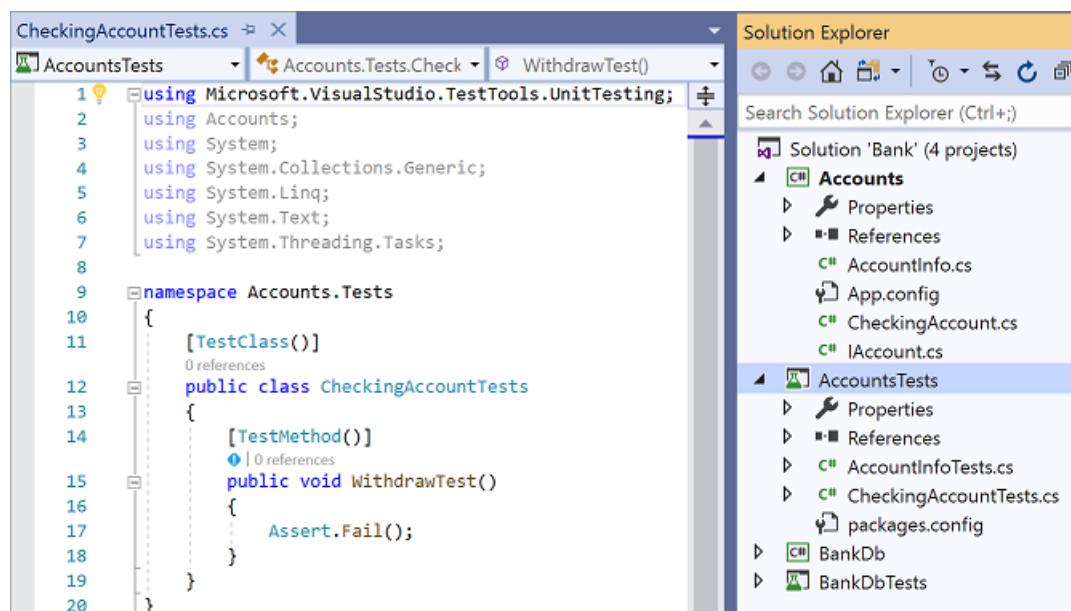
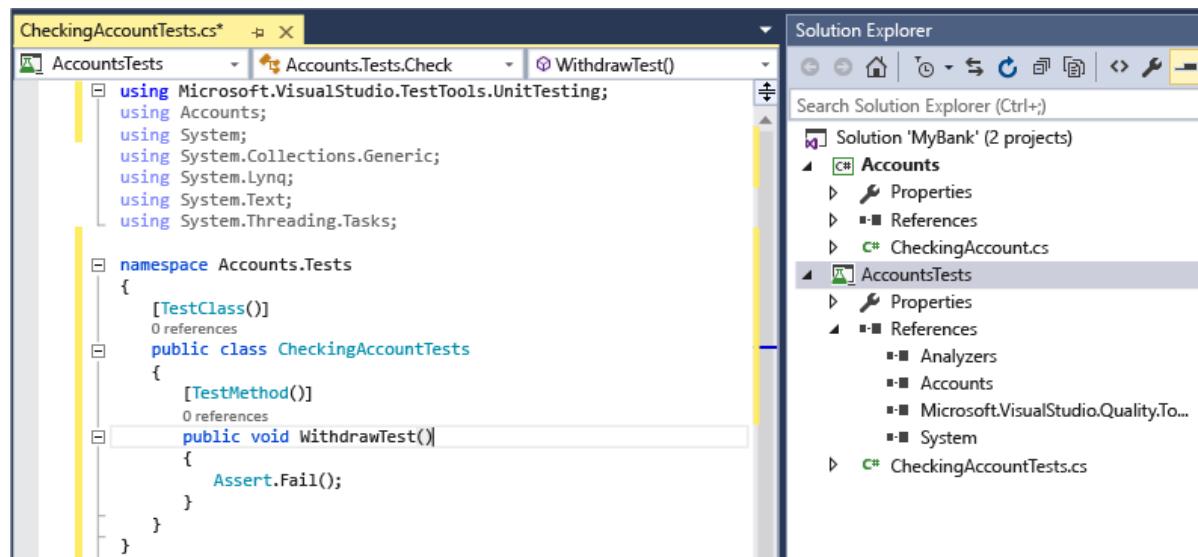
NOTE

The **Create Unit Tests** menu command is only available for managed code.

2. Click **OK** to accept the defaults to create your unit tests, or change the values used to create and name the unit test project and the unit tests. You can select the code that is added by default to the unit test methods.



3. The unit test stubs are created in a new unit test project for all the methods in the class.



4. Now jump ahead to learn how to [add code to the unit test methods](#) to make your unit test meaningful, and any extra unit tests that you might want to add to thoroughly test your code.

Create the unit test project and unit tests manually

A unit test project usually mirrors the structure of a single code project. In the MyBank example, you add two unit test projects named `AccountsTests` and `BankDbTests` to the `MyBanks` solution. The test project names are arbitrary, but adopting a standard naming convention is a good idea.

To add a unit test project to a solution:

1. In **Solution Explorer**, right-click on the solution and choose **Add > New Project**.
2. In the **New Project** dialog box, expand the **Installed** node, choose the language that you want to use for your test project, and then choose **Test**.
3. To use one of the Microsoft unit test frameworks, choose **Unit Test Project** from the list of project templates. Otherwise, choose the project template of the unit test framework that you want to use. To test the `Accounts` project of our example, you would name the project `AccountsTests`.

NOTE

Not all third-party and open source unit test frameworks provide a Visual Studio project template. Consult the framework document for information about creating a project.

2. Use the project template search box to find a unit test project template for the test framework that you want to use.
3. On the next page, name the project. To test the `Accounts` project of our example, you could name the project `AccountsTests`.
4. In your unit test project, add a reference to the code project under test, in our example to the `Accounts` project.

To create the reference to the code project:

- a. Select the project in **Solution Explorer**.
- b. On the **Project** menu, choose **Add Reference**.
- c. On the **Reference Manager** dialog box, open the **Solution** node and choose **Projects**. Select the code project name and close the dialog box.

Each unit test project contains classes that mirror the names of the classes in the code project. In our example, the `AccountsTests` project would contain the following classes:

- `AccountInfoTests` class contains the unit test methods for the `AccountInfo` class in the `Accounts` project
- `CheckingAccountTests` class contains the unit test methods for `CheckingAccount` class.

Write your tests

The unit test framework that you use and Visual Studio IntelliSense will guide you through writing the code for your unit tests for a code project. To run in **Test Explorer**, most frameworks require that you add specific attributes to identify unit test methods. The frameworks also provide a way—usually through assert statements or method attributes—to indicate whether the test method has passed or failed. Other attributes identify optional setup methods that are at class initialization and before each test method and teardown methods that are run after each test method and before the class is destroyed.

The AAA (Arrange, Act, Assert) pattern is a common way of writing unit tests for a method under test.

- The **Arrange** section of a unit test method initializes objects and sets the value of the data that is passed to the method under test.
- The **Act** section invokes the method under test with the arranged parameters.
- The **Assert** section verifies that the action of the method under test behaves as expected.

To test the `CheckingAccount.Withdraw` method of our example, we can write two tests: one that verifies the standard behavior of the method, and one that verifies that a withdrawal of more than the balance will fail. In the `CheckingAccountTests` class, we add the following methods:

```
[TestMethod]
public void Withdraw_ValidAmount_ChangesBalance()
{
    // arrange
    double currentBalance = 10.0;
    double withdrawal = 1.0;
    double expected = 9.0;
    var account = new CheckingAccount("JohnDoe", currentBalance);

    // act
    account.Withdraw(withdrawal);

    // assert
    Assert.AreEqual(expected, account.Balance);
}

[TestMethod]
public void Withdraw_AmountMoreThanBalance_Throws()
{
    // arrange
    var account = new CheckingAccount("John Doe", 10.0);

    // act and assert
    Assert.ThrowsException<System.ArgumentException>(() => account.Withdraw(20.0));
}
```

For more information about the Microsoft unit testing frameworks, see one of the following topics:

- [Unit test your code](#)
- [Writing unit tests for C/C++](#)
- [Use the MSTest framework in unit tests](#)

Set timeouts for unit tests

If you're using the MSTest framework, you can use the [TimeoutAttribute](#) to set a timeout on an individual test method:

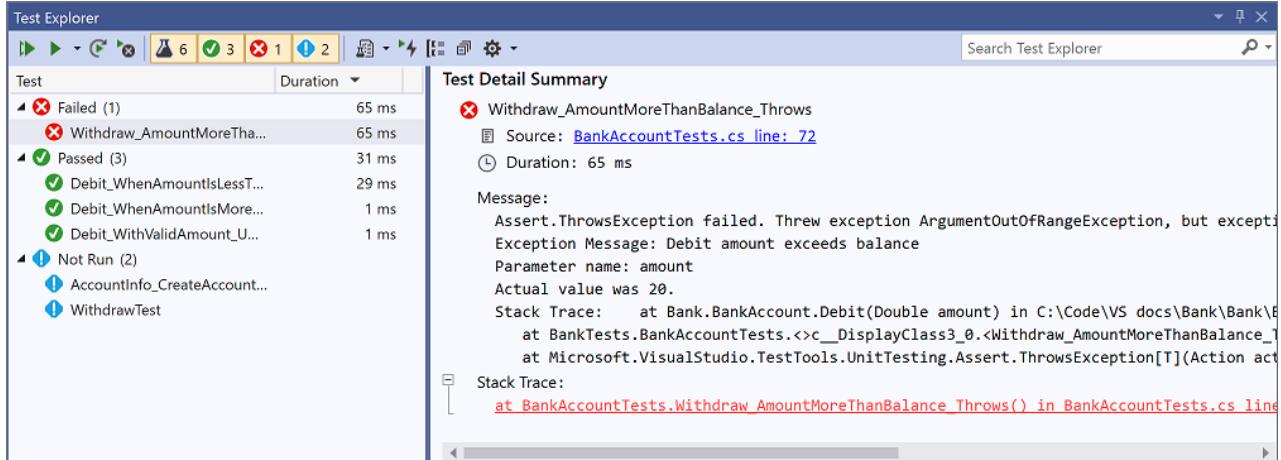
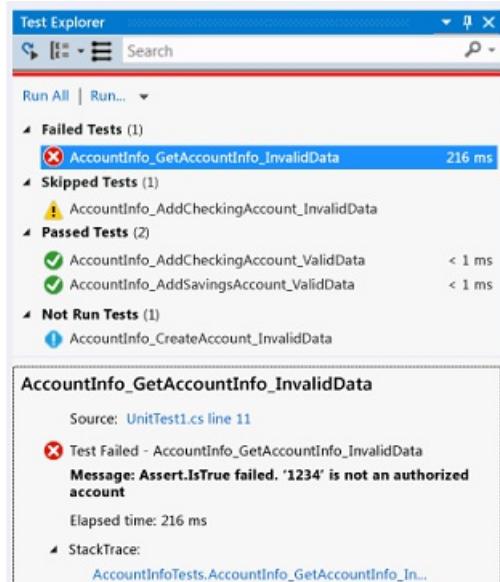
```
[TestMethod]
[Timeout(2000)] // Milliseconds
public void My_Test()
{ ... }
```

To set the timeout to the maximum allowed:

```
[TestMethod]
[Timeout(TestTimeout.Infinite)] // Milliseconds
public void My_Test ()
{
    ...
}
```

Run tests in Test Explorer

When you build the test project, the tests appear in **Test Explorer**. If **Test Explorer** is not visible, choose **Test** on the Visual Studio menu, choose **Windows**, and then choose **Test Explorer**.

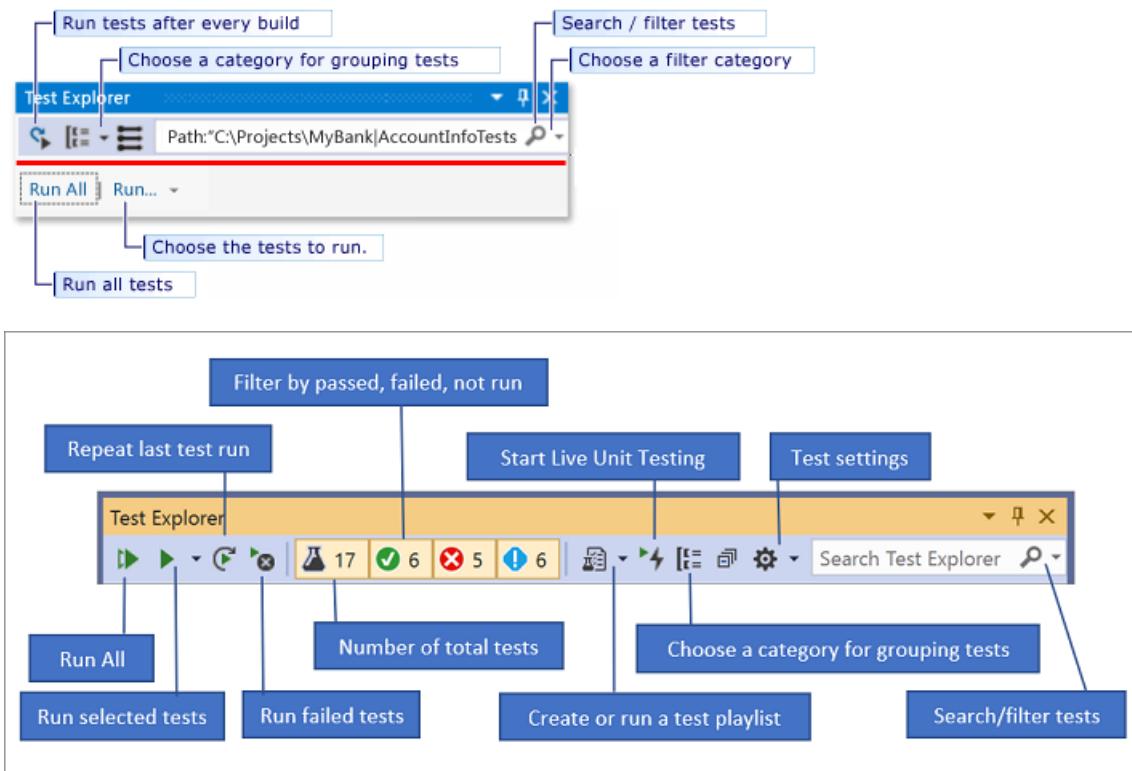


As you run, write, and rerun your tests, the **Test Explorer** can display the results in groups of **Failed Tests**, **Passed Tests**, **Skipped Tests** and **Not Run Tests**. You can choose different group by options in the toolbar.

You can also filter the tests in any view by matching text in the search box at the global level or by selecting one of the pre-defined filters. You can run any selection of the tests at any time. The results of a test run are immediately apparent in the pass/fail bar at the top of the explorer window. Details of a test method result are displayed when you select the test.

Run and view tests

The **Test Explorer** toolbar helps you discover, organize, and run the tests that you are interested in.



You can choose **Run All** to run all your tests, or choose **Run** to choose a subset of tests to run. Select a test to view the details of that test in the test details pane. Choose **Open Test** from the right-click menu (Keyboard: **F12**) to display the source code for the selected test.

If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution with the toggle button on the toolbar. This can noticeably reduce the time taken to run all the tests.

If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution in the settings menu of the toolbar. This can noticeably reduce the time taken to run all the tests.

Run tests after every build

BUTTON	DESCRIPTION
	To run your unit tests after each local build, choose Test on the standard menu, and then choose Run Tests After Build on the Test Explorer toolbar.

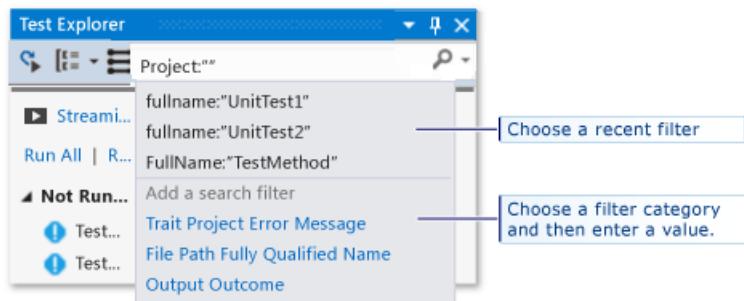
NOTE

Running unit tests after each build requires Visual Studio 2017 Enterprise edition or Visual Studio 2019. In Visual Studio 2019, the feature is available in Community and Professional edition, in addition to Enterprise edition.

To run your unit tests after each local build, open the settings icon in the Test Explorer toolbar and select **Run Tests After Build**.

Filter and group the test list

When you have a large number of tests, you can type in the **Test Explorer** search box to filter the list by the specified string. You can restrict your filter even more by choosing from the filter list.



BUTTON	DESCRIPTION
	To group your tests by category, choose the Group By button.

For more information, see [Run unit tests with Test Explorer](#).

Q&A

Q: How do I debug unit tests?

A: Use **Test Explorer** to start a debugging session for your tests. Stepping through your code with the Visual Studio debugger seamlessly takes you back and forth between the unit tests and the project under test. To start debugging:

1. In the Visual Studio editor, set a breakpoint in one or more test methods that you want to debug.

NOTE

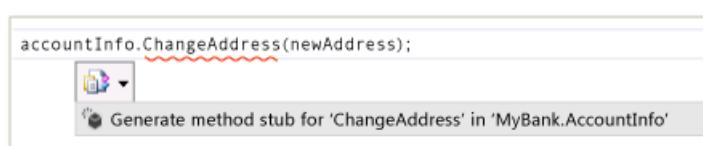
Because test methods can run in any order, set breakpoints in all the test methods that you want to debug.

2. In **Test Explorer**, select the test methods and then choose **Debug Selected Tests** from the shortcut menu.

Learn more details about [debugging unit tests](#).

Q: If I'm using TDD, how do I generate code from my tests?

A: Use Quick Actions to generate classes and methods in your project code. Write a statement in a test method that calls the class or method that you want to generate, then open the lightbulb that appears under the error. If the call is to a constructor of the new class, choose **Generate type** from the menu and follow the wizard to insert the class in your code project. If the call is to a method, choose **Generate method** from the IntelliSense menu.



A screenshot of the Visual Studio IDE showing a code editor with C# code. A tooltip is displayed over the 'NewMethod' call in the 'Assert.AreEqual' line, indicating a CS1061 error: 'BankAccount' does not contain a definition for 'NewMethod' and no accessible extension method 'NewMethod' accepting a first argument of type 'BankAccount' could be found. Below the tooltip, a code completion dropdown shows the definition of 'NewMethod':

```
public void NewMethod()
{
    throw new NotImplementedException();
}
```

Q: Can I create unit tests that take multiple sets of data as input to run the test?

A: Yes. *Data-driven test methods* let you test a range of values with a single unit test method. Use a `DataSource` attribute for the test method that specifies the data source and table that contains the variable values that you want to test. In the method body, you assign the row values to variables using the `TestContext.DataRow[ColumnName]` indexer.

NOTE

These procedures apply only to test methods that you write by using the Microsoft unit test framework for managed code. If you're using a different framework, consult the framework documentation for equivalent functionality.

For example, assume we add an unnecessary method to the `CheckingAccount` class that is named `AddIntegerHelper`. `AddIntegerHelper` adds two integers.

To create a data-driven test for the `AddIntegerHelper` method, we first create an Access database named `AccountsTest.accdb` and a table named `AddIntegerHelperData`. The `AddIntegerHelperData` table defines columns to specify the first and second operands of the addition and a column to specify the expected result. We fill a number of rows with appropriate values.

```
[DataSource(
    @"Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\Projects\MyBank\TestData\AccountsTest.accdb",
    "AddIntegerHelperData"
)]
[TestMethod()]
public void AddIntegerHelper_DataDrivenValues_AllShouldPass()
{
    var target = new CheckingAccount();
    int x = Convert.ToInt32(TestContext.DataRow["FirstNumber"]);
    int y = Convert.ToInt32(TestContext.DataRow["SecondNumber"]);
    int expected = Convert.ToInt32(TestContext.DataRow["Sum"]);
    int actual = target.AddIntegerHelper(x, y);
    Assert.AreEqual(expected, actual);
}
```

The attributed method runs once for each row in the table. **Test Explorer** reports a test failure for the method if any of the iterations fail. The test results detail pane for the method shows you the pass/fail status method for each row of data.

Learn more about [data-driven unit tests](#).

Q: Can I view how much of my code is tested by my unit tests?

A: Yes. You can determine the amount of your code that is actually being tested by your unit tests by using the Visual Studio code coverage tool in Visual Studio Enterprise. Native and managed languages and all unit test frameworks that can be run by the Unit Test Framework are supported.

You can run code coverage on selected tests or on all tests in a solution. The **Code Coverage Results** window

displays the percentage of the blocks of product code that were exercised by line, function, class, namespace and module.

To run code coverage for test methods in a solution, choose **Test > Analyze Code Coverage for All Tests**.

Coverage results appear in the **Code Coverage Results** window.

The screenshot shows the 'Code Coverage Results' window with the title bar 'Code Coverage Results'. Below it is a dropdown menu showing 'user1_FABLAB-0529 2012-06-04 10_40...'. The main area is a grid table with three columns: 'Hierarchy', 'Not Covered (Blocks)', and 'Not Covered (% Blocks)'. The first row under 'Hierarchy' is 'AccountInfoTests', with '3' in the 'Not Covered (Blocks)' column and '25.00%' in the 'Not Covered (% Blocks)' column. The second row is 'AccountInfo_GetAccountInfo_InvalidData()', which is highlighted with a blue background. It has '1' in the 'Not Covered (Blocks)' column and '50.00%' in the 'Not Covered (% Blocks)' column. The other five rows under 'Hierarchy' have '0' in the 'Not Covered (Blocks)' column and '0.00%' in the 'Not Covered (% Blocks)' column. The rows for 'AccountInfo_GetAccount_ValidateData()', 'AccountInfo_GetAccount_InvalidData()', 'AccountInfo_AddSavingsAccount_ValidateData()', and 'AccountInfo_AddCheckingAccount_ValidateDa...' are partially visible.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)
AccountInfoTests	3	25.00%
AccountInfo_GetAccountInfo_InvalidData()	1	50.00%
AccountInfo_GetAccount_ValidateData()	0	0.00%
AccountInfo_GetAccount_InvalidData()	0	0.00%
AccountInfo_AddSavingsAccount_ValidateData()	0	0.00%
AccountInfo_AddCheckingAccount_ValidateDa...	0	0.00%

Learn more about [code coverage](#).

Q: Can I test methods in my code that have external dependencies?

A: Yes. If you have Visual Studio Enterprise, Microsoft Fakes can be used with test methods that you write by using unit test frameworks for managed code.

Microsoft Fakes uses two approaches to create substitute classes for external dependencies:

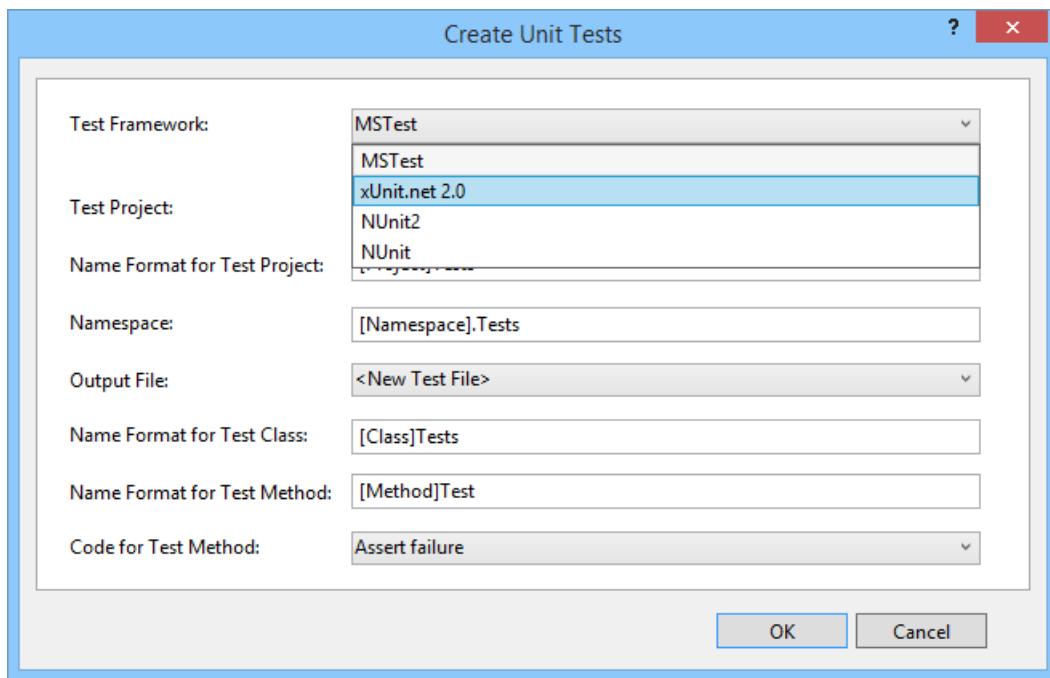
1. *Stubs* generate substitute classes derived from the parent interface of the target dependency class. Stub methods can be substituted for public virtual methods of the target class.
2. *Shims* use runtime instrumentation to divert calls to a target method to a substitute shim method for non-virtual methods.

In both approaches, you use the generated delegates of calls to the dependency method to specify the behavior that you want in the test method.

Learn more about [isolating unit test methods with Microsoft Fakes](#).

Q: Can I use other unit test frameworks to create unit tests?

A: Yes, follow these steps to [find and install other frameworks](#). After you restart Visual Studio, reopen your solution to create your unit tests, and then select your installed frameworks here:



Your unit test stubs will be created using the selected framework.

Create a unit test project

1/1/2020 • 2 minutes to read • [Edit Online](#)

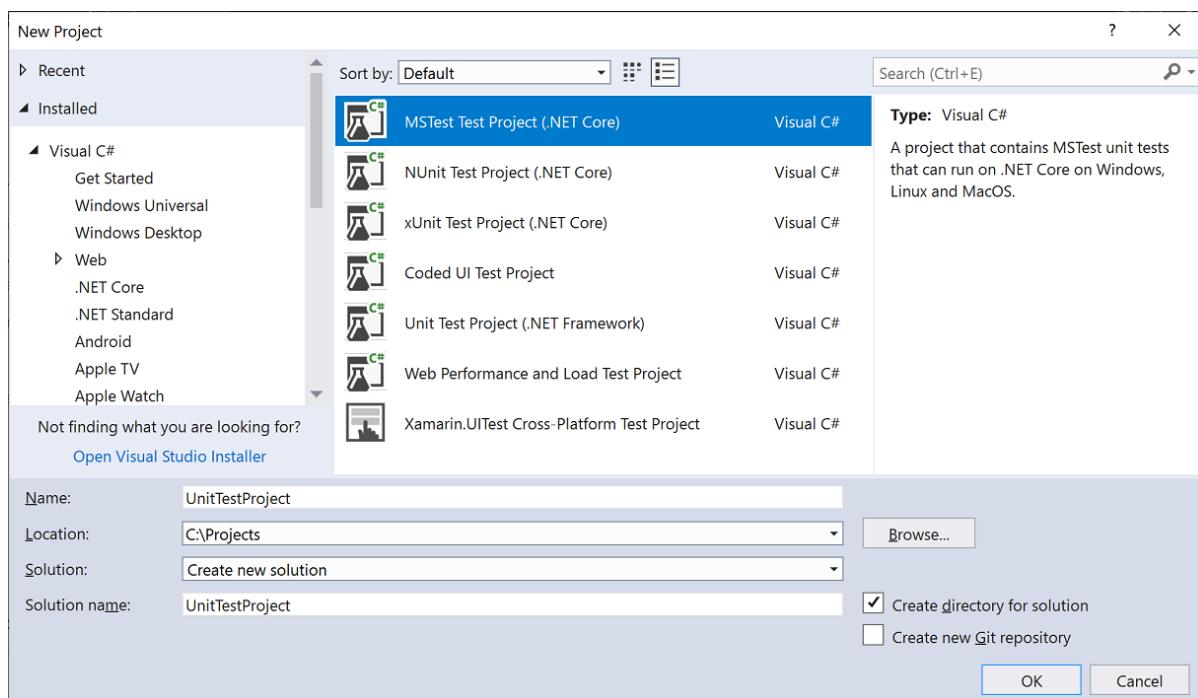
Unit tests often mirror the structure of the code under test. For example, a unit test project would be created for each code project in the product. The test project can be in the same solution as the production code, or it can be in a separate solution. You can have multiple unit test projects in a solution.

NOTE

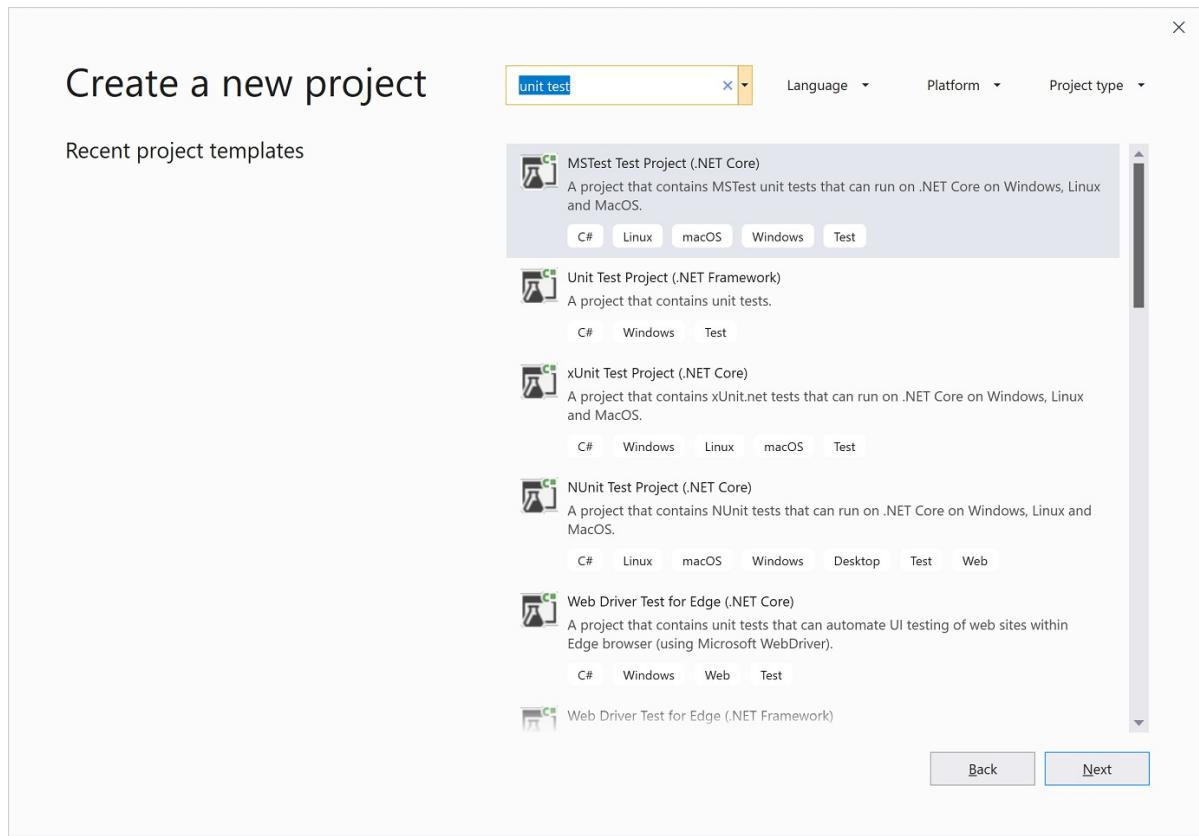
The location of unit tests for native code and the test project structure can be different than the structure that is described in this article. For more information, see [Writing unit tests for C/C++](#).

To create a unit test project

1. On the **File** menu, choose **New > Project**, or press **Ctrl+Shift+N**.
2. In the **New Project** dialog box, expand the **Installed** node, choose the language that you want to use for your test project, and then choose **Test**.
3. Select the project template for the test framework that you want to use, for example **MSTest Test Project** or **NUnit Test Project**. Name the project, and then choose **OK**.



2. On the **Create a new project** page, type **unit test** into the search box. Select the project template for the test framework that you want to use, for example **MSTest Test Project** or **NUnit Test Project**, and then choose **Next**.



3. On the **Configure your new project** page, enter a name for your project, and then choose **Create**.
4. In your unit test project, add a reference to the code under test. To add a reference to a code project in the same solution:
 - a. Select the test project in **Solution Explorer**.
 - b. On the **Project** menu, choose **Add Reference**.
 - c. In **Reference Manager**, select the **Solution** node under **Projects**. Select the code project you want to test, and then select **OK**.

If the code that you want to test is in another location, see [Managing references in a project](#) for information about adding a reference.

Next steps

See one of the following sections:

Writing unit tests

- [Unit test your code](#)
- [Writing unit tests for C/C++](#)
- [Use the MSTest framework in unit tests](#)

Running unit tests

- [Run unit tests with Test Explorer](#)

Create unit test method stubs with the Create Unit Tests command

1/10/2020 • 2 minutes to read • [Edit Online](#)

The **Create Unit Tests** command creates unit test method stubs. This feature allows easy configuration of a test project, the test class, and the test method stub within it.

NOTE

The **Create Unit Tests** menu command is only available for managed code that targets .NET Framework (but not .NET Core).

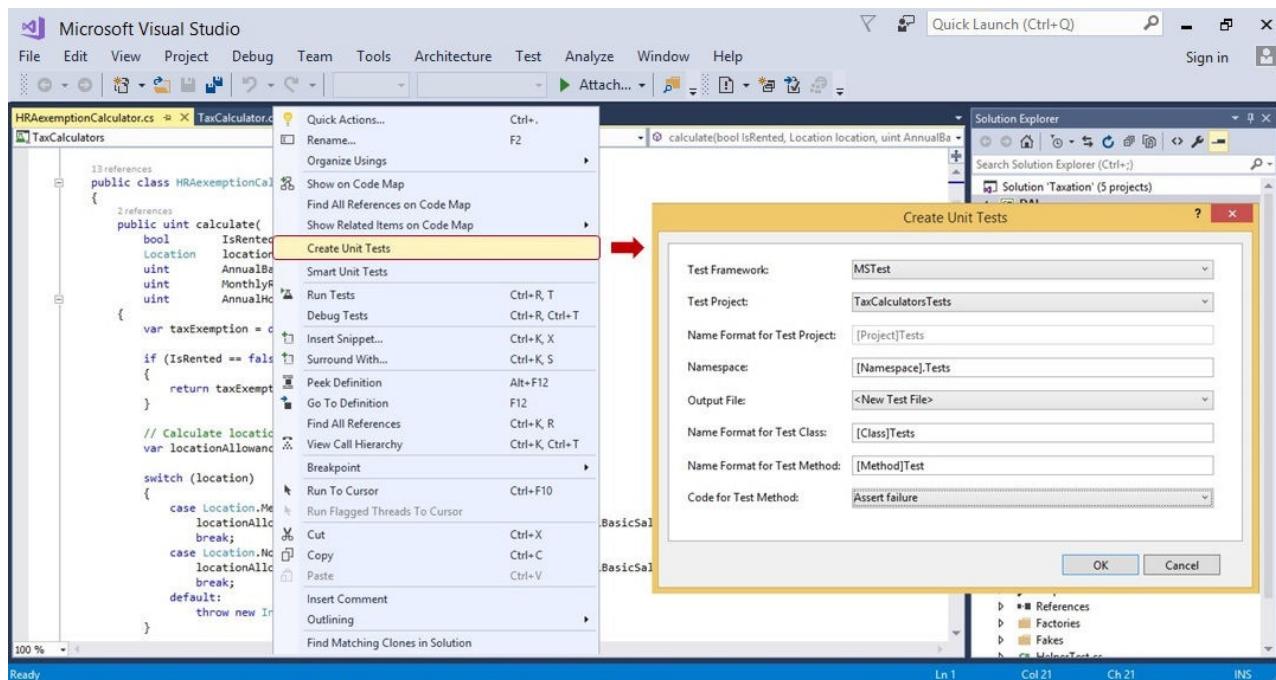
NOTE

The **Create Unit Tests** menu command is only available for managed code.

The **Create Unit Tests** menu command is extensible and can be used to generate tests for MSTest, MSTest V2, NUnit, and xUnit.

Get started

To get started, select a method, a type, or a namespace in the code editor in the project you want to test, right-click, and then choose **Create Unit Tests**. The **Create Unit Tests** dialog opens where you can configure how you want the tests to be created.



Set unit test traits

If you plan to run these tests as part of the test automation process, you might consider having the test created in another test project (the second option in the dialog above) and setting unit test traits for the unit test. This enables you to more easily include or exclude these specific tests as part of a continuous integration or continuous deployment pipeline. The traits are set by adding metadata to the unit test directly, as shown below.

```
partsTests.cs* X | Source Control Explorer - Disconnected CodedUI.cs Startup.cs
CalcTaxTests CarpartsTax.Carp...
using Microsoft.VisualStudio.TestTools.UnitTesting;
using CarpartsTax.CarpartsTax;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CarpartsTax.CarpartsTax.Tests
{
    [TestClass()]
    0 references | 0 changes | 0 authors, 0 changes
    public class partsTests
    {
        [TestMethod()]
        [Owner("Charles")]
        [TestCategory("Stubbed")]
        [Priority(9)]
        0 references | 0 changes | 0 authors, 0 changes
        public void getWashingtonTaxTest()
        {
            Assert.Fail();
        }
    }
}
```

Use third-party unit test frameworks

To automatically generate unit tests for NUnit or xUnit, install one of these test framework extensions from Visual Studio Marketplace:

- [NUnit extension for test generators](#)
- [xUnit.net extension for test generators](#)

When should I use this feature?

Use this feature whenever you need to create unit tests, but specifically when you are testing existing code that has little or no test coverage and no documentation. In other words, where there is limited or non-existent code specification. It effectively implements an approach similar to [Smart unit tests](#) that characterizes the observed behavior of the code.

However, this feature is equally applicable when a developer starts by writing some code and then uses that to bootstrap unit tests. Within the flow of coding, the developer might want to quickly create a unit test method stub (with a suitable test class and a suitable test project) for a particular piece of code.

See also

- [Creating unit test method stubs with "Create Unit Tests"](#)
- [Unit testing blog posts](#)

How to: Generate unit tests by using IntelliTest

1/1/2020 • 5 minutes to read • [Edit Online](#)

IntelliTest explores your .NET code to generate test data and a suite of unit tests. For every statement in the code, a test input is generated that will execute that statement. A case analysis is performed for every conditional branch in the code. For example, `if` statements, assertions, and all operations that can throw exceptions are analyzed. This analysis is used to generate test data for a parameterized unit test for each of your methods, creating unit tests with high code coverage.

When you run IntelliTest, you can easily see which tests are failing and add any necessary code to fix them. You can select which of the generated tests to save into a test project to provide a regression suite. As you change your code, rerun IntelliTest to keep the generated tests in sync with your code changes.

Availability and extensions

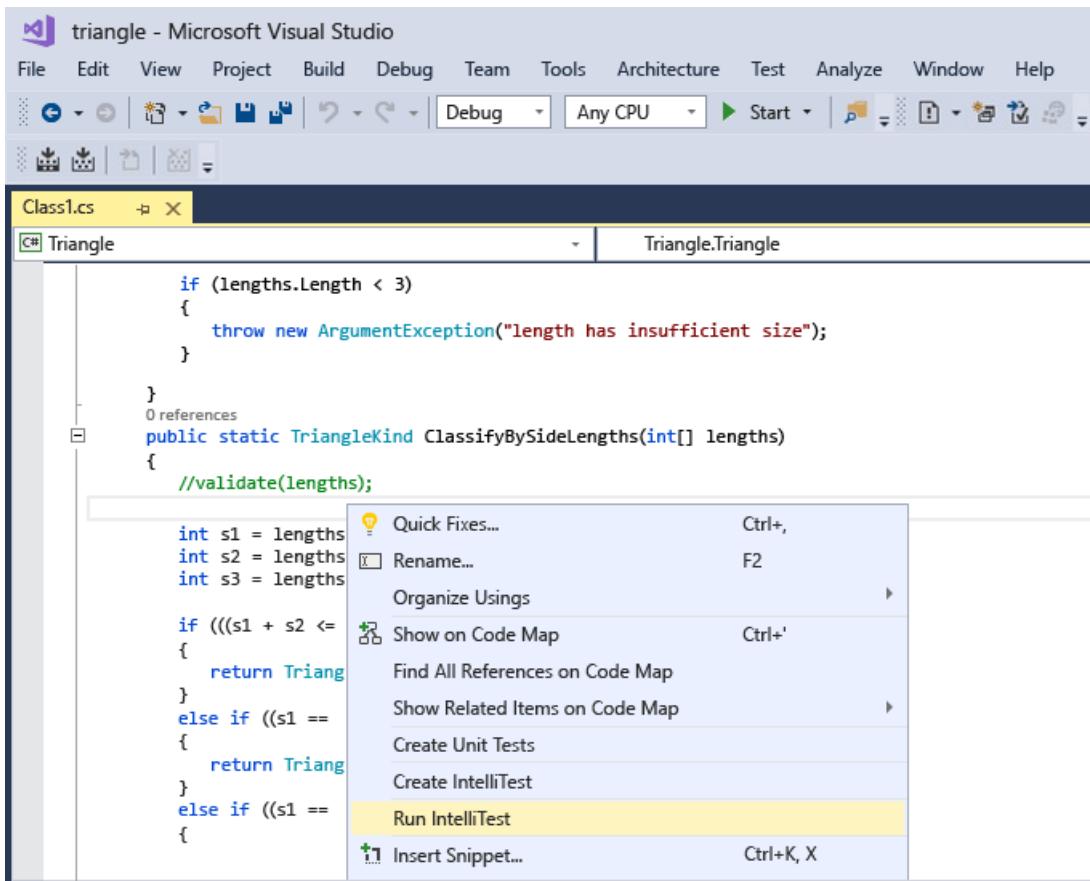
The **Create IntelliTest** and **Run IntelliTest** menu commands:

- Are available only in the Enterprise Edition of Visual Studio.
- Support only C# code that targets the .NET Framework.
- Are [extensible](#) and support emitting tests in MSTest, MSTest V2, NUnit, and xUnit format.
- Do not support x64 configuration.

Explore: Use IntelliTest to explore your code and generate unit tests

To generate unit tests, your types must be public.

1. Open your solution in Visual Studio and then open the class file that has methods you want to test.
2. Right-click on a method and choose **Run IntelliTest** to generate unit tests for the code in your method.



IntelliTest runs your code many times with different inputs. Each run is represented in the table showing the input test data and the resulting output or exception.

IntelliTest Exploration Results - stopped				
	lengths	result	Summary/Exception	Error Message
✗ 1	null		NullReferenceException	Object refer...
✗ 2	{}		IndexOutOfRangeException	Index was out...
✗ 3	{0}		IndexOutOfRangeException	Index was out...
✗ 4	{0, 0}		IndexOutOfRangeException	Index was out...
✓ 5	{0, 0, 0}	Invalid		
✓ 6	{5, 538, 0}	Invalid		
✓ 7	{67, 0, 0}	Invalid		
✓ 8	{422, 536, 654}	Scalene		
✓ 9	{528, 413, 512}	Isosceles		
✓ 10	{2, 2, 3}	Isosceles		
✓ 11	{1, 512, 512}	Isosceles		
✓ 12	{512, 512, 512}	Equilateral		

16/16 blocks, 0/0 asserts, 12 runs

Details:
Stack trace:

```

System.NullReferenceException...
at Triangle.ClassifyBySideLengths...
at TriangleTest.ClassifyBySideLengths...

```

To generate unit tests for all the public methods in a class, simply right-click in the class rather than a specific method, and then choose **Run IntelliTest**. Use the drop-down list in the **Exploration Results** window to display the unit tests and the input data for each method in the class.

IntelliTest Exploration Results - stopped		
Triangle.ClassifyBySideLengths(int)	Run	0 Warnings
(Global Events)	16/16 blocks, 0/0 asserts, 12 runs	
Triangle.ClassifyByInternalAngles(Int32[])	mary/Exception	Error Message
Triangle.ClassifyBySideLengths(Int32[])		
✗ 1 null	NullReferenceException	Object reference not set to an instance o...
✗ 2 {}	IndexOutOfRangeException	Index was outside the bounds of array.
✗ 3 {0}	IndexOutOfRangeException	Index was outside the bounds of array.
✗ 4 {0, 0}	IndexOutOfRangeException	Index was outside the bounds of array.
✓ 5 {0, 0, 0}	Invalid	
✓ 6 {5, 538, 0}	Invalid	
✓ 7 {67, 0, 0}	Invalid	
✓ 8 {422, 536, 6...	Scalene	
✓ 9 {528, 413, 5...	Isosceles	
✓ 10 {2, 2, 3}	Isosceles	
✓ 11 {1, 512, 512}	Isosceles	
✓ 12 {512, 512, 5...	Equilateral	

For tests that pass, check that the reported results in the result column match your expectations for your code. For tests that fail, fix your code as appropriate. Then rerun IntelliTest to validate the fixes.

Persist: Save the unit tests as a regression suite

1. Select the data rows that you want to save with the parameterized unit test into a test project.

IntelliTest Exploration Results - stopped		
Triangle.ClassifyBySideLengths(int)	Run	0 Warnings
(Global Events)	16/16 blocks, 0/0 asserts, 12 runs	
✓ 8 ✗ 4	Save	
✗ 1 null	NullReferenceException	Object reference not set to an instance o...
✗ 2 {}	IndexOutOfRangeException	Index was outside the bounds of array.
✗ 3 {0}	IndexOutOfRangeException	Index was outside the bounds of array.
✗ 4 {0, 0}	IndexOutOfRangeException	Index was outside the bounds of array.
✓ 5 {0, 0, 0}	Invalid	
✓ 6 {5, 538, 0}	Invalid	
✓ 7 {67, 0, 0}	Invalid	
✓ 8 {422, 536, 6...	Scalene	
✓ 9 {528, 413, 5...	Isosceles	
✓ 10 {2, 2, 3}	Isosceles	
✓ 11 {1, 512, 512}	Isosceles	
✓ 12 {512, 512, 5...	Equilateral	

You can view the test project and the parameterized unit test that has been created - the individual unit tests, corresponding to each of the rows, are saved in the .g.cs file in the test project, and a parameterized unit test is saved in its corresponding .cs file. You can run the unit tests and view the results from Test Explorer just as you would for any unit tests that you created manually.

The screenshot shows the Visual Studio interface with the Solution Explorer and the code editor. The Solution Explorer displays two projects: 'Triangle' and 'Triangle.Tests'. The 'Triangle.Tests' project is expanded, showing its files: AssemblyInfo.cs, PexAssemblyInfo.cs, References, and the file 'TriangleTest.cs' which is currently selected and highlighted with an orange border. The code editor shows the content of 'TriangleTest.cs'.

```

using System;
using Microsoft.Pex.Framework;
using Microsoft.Pex.Framework.Validation;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Triangle;

namespace Triangle
{
    [TestClass]
    [PexClass(typeof(global::Triangle.Triangle))]
    [PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentException))]
    [PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperationException))]
    public partial class TriangleTest
    {
        [PexMethod]
        public TriangleKind ClassifyBySideLengths(int[] lengths)
        {
            TriangleKind result = global::Triangle.Triangle.Clas...
            return result;
            // TODO: add assertions to method TriangleTest.Clas...
        }
    }
}

```

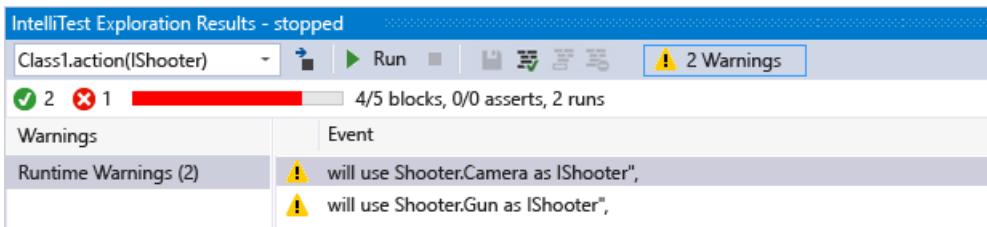
Any necessary references are also added to the test project.

If the method code changes, rerun IntelliTest to keep the unit tests in sync with the changes.

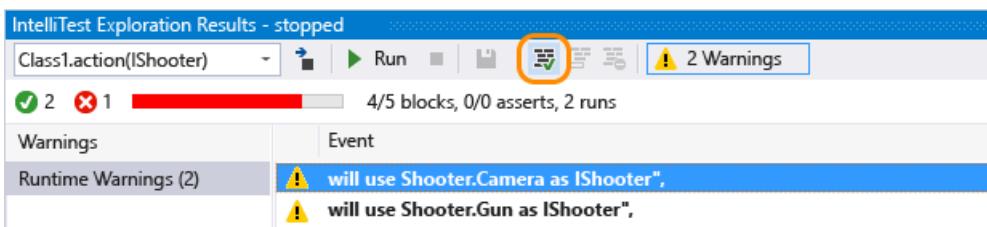
Assist: Use IntelliTest to focus code exploration

1. If you have more complex code, IntelliTest assists you with focusing exploration of your code. For example, if you have a method that has an interface as a parameter, and there is more than one class that implements that interface, IntelliTest discovers those classes and reports a warning.

View the warnings to decide what you want to do.



2. After you investigate the code and understand what you want to test, you can fix the warning to choose which classes to use to test the interface.



This choice is added into the `PexAssemblyInfo.cs` file.

```
[assembly: PexUseType(typeof(Camera))]
```

3. Now you can rerun IntelliTest to generate a parameterized unit test and test data just using the class that you fixed.

The screenshot shows the 'IntelliTest Exploration Results - stopped' window. At the top, it displays 'Class1.action(Ishooter)' with a dropdown arrow, followed by standard Windows-style buttons for Run, Stop, and Refresh. To the right, it shows '0 Warnings'. Below this is a summary bar with two green checkmarks and one red X, indicating '5/6 blocks, 0/0 asserts, 2 runs'. A detailed table follows, with columns for target, result, Summary / Exception, and Error Message. The first row has a red X and says 'new Class1() null' with 'NullReferenceException' in the exception column. The second row has a green checkmark and says 'new Class1() new Came... new Class1() "Click"' with 'Object refer...' in the error message column. To the right of the table is a 'Details:' section containing C# code. The code includes annotations like [TestMethod] and [PexGeneratedBy], and defines a method action63 that creates a Class1 object, a Camera object, and performs assertions on them.

Specify: Use IntelliTest to validate correctness properties that you specify in code

Specify the general relationship between inputs and outputs that you want the generated unit tests to validate. This specification is encapsulated in a method that looks like a test method but is universally quantified. This is the parameterized unit test method, and any assertions you make must hold for all possible input values that IntelliTest can generate.

Q & A

Q: Can you use IntelliTest for unmanaged code?

A: No, IntelliTest only works with managed code.

Q: When does a generated test pass or fail?

A: It passes like any other unit test if no exceptions occur. It fails if any assertion fails, or if the code under test throws an unhandled exception.

If you have a test that can pass if certain exceptions are thrown, you can set one of the following attributes based on your requirements at the test method, test class or assembly level:

- **PexAllowedExceptionAttribute**
- **PexAllowedExceptionFromTypeAttribute**
- **PexAllowedExceptionFromTypeUnderTestAttribute**
- **PexAllowedExceptionFromAssemblyAttribute**

Q: Can I add assumptions to the parameterized unit test?

A: Yes, use assumptions to specify which test data is not required for the unit test for a specific method. Use the [PexAssume](#) class to add assumptions. For example, you can add an assumption that the `lengths` variable is not null like this:

```
PexAssume.IsNotNull(lengths);
```

If you add an assumption and rerun IntelliTest, the test data that is no longer relevant will be removed.

Q: Can I add assertions to the parameterized unit test?

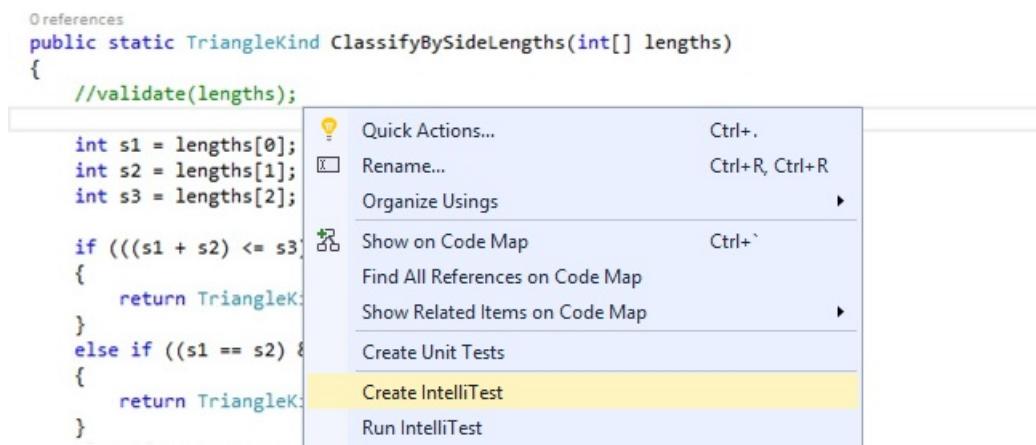
A: Yes, IntelliTest will check that what you are asserting in your statement is in fact correct when it runs the unit tests. Use the [PexAssert](#) class or the assertion API that comes with the test framework to add assertions. For example, you can add an assertion that two variables are equal.

```
PexAssert.AreEqual(a, b);
```

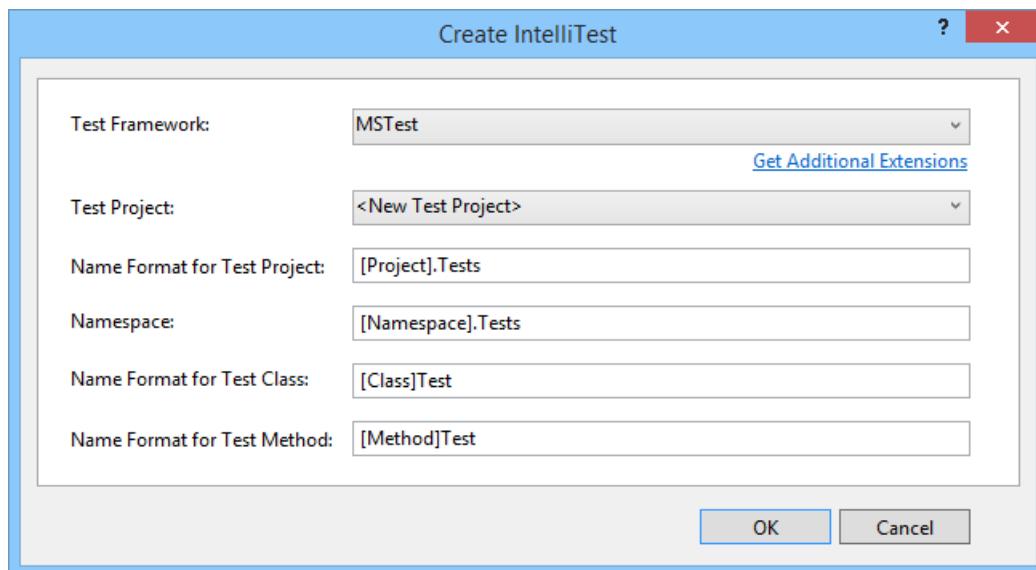
If you add an assertion and rerun IntelliTest, it will check that your assertion is valid and the test fails if it's not.

Q: Can I generate parameterized unit tests without running IntelliTest first?

A: Yes, right-click in the class or method, then choose **Create IntelliTest**.



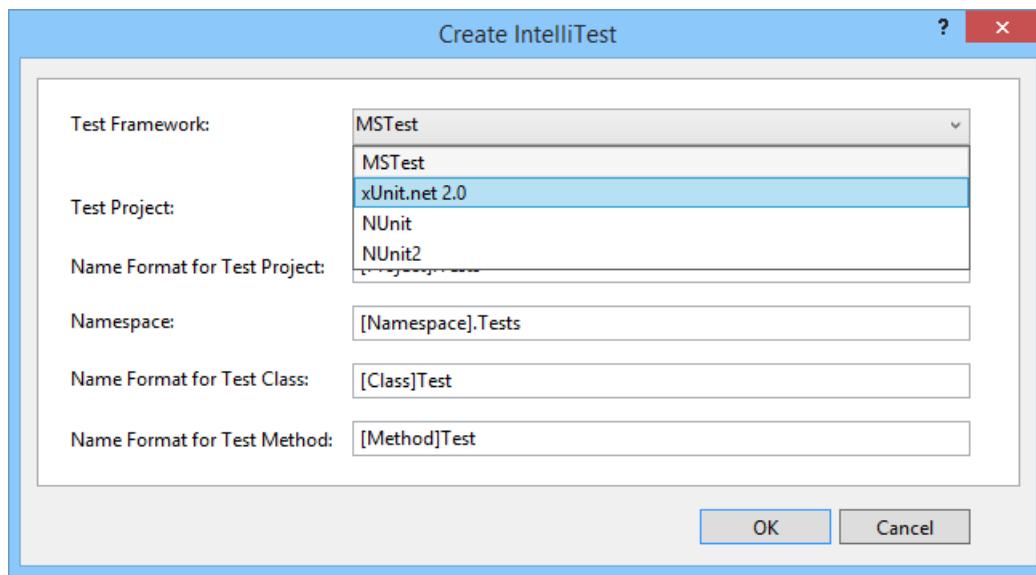
Accept the default format to generate your tests, or change how your project and tests are named. You can create a new test project or save your tests to an existing project.



Q: Can I use other unit test frameworks with IntelliTest?

A: Yes, follow these steps to [find and install other frameworks](#). Test framework extensions are also available in Visual Studio Marketplace, for example, [NUnit Test Generator](#).

After you restart Visual Studio and reopen your solution, right-click in the class or method, then choose **Create IntelliTest**. Select your installed framework here:



Then run IntelliTest to generate individual unit tests in their corresponding .g.cs files.

Q: Can I learn more about how the tests are generated?

A: Yes, to get a high-level overview, read this [blog post](#).

Overview of Microsoft IntelliTest

1/1/2020 • 5 minutes to read • [Edit Online](#)

IntelliTest enables you to find bugs early, and reduces test maintenance costs. Using an automated and transparent testing approach, IntelliTest can generate a candidate suite of tests for your .NET code. Test suite generation can be further guided by *correctness properties* you specify. IntelliTest will even evolve the test suite automatically as the code under test evolves.

Characterization tests IntelliTest enables you to determine the behavior of code in terms of a suite of traditional unit tests. Such a test suite can be used as a regression suite, forming the basis for tackling the complexity associated with refactoring legacy or unfamiliar code.

Guided test input generation IntelliTest uses an open code analysis and constraint solving approach to automatically generate precise test input values; usually without the need for any user intervention. For complex object types, it automatically generates factories. You can guide test input generation by extending and configuring the factories to suit your requirements. Correctness properties specified as assertions in code will also be used automatically to further guide test input generation.

IDE integration IntelliTest is fully integrated into the Visual Studio IDE. All of the information gathered during test suite generation (such as the automatically generated inputs, the output from your code, the generated test cases, and their pass or fail status) appears within the Visual Studio IDE. You can easily iterate between fixing your code and rerunning IntelliTest, without leaving the Visual Studio IDE. The tests can be saved into the solution as a Unit Test Project, and will be automatically detected afterwards by Visual Studio Test Explorer.

Complement existing testing practices Use IntelliTest to complement any existing testing practices that you may already follow.

If you want to test:

- Algorithms over primitive data, or arrays of primitive data:
 - write [parameterized unit tests](#)
- Algorithms over complex data, such as compiler:
 - let IntelliTest first generate an abstract representation of the data, and then feed it to the algorithm
 - let IntelliTest build instances using [custom object creation](#) and data invariants, and then invoke the algorithm
- Data containers:
 - write [parameterized unit tests](#)
 - let IntelliTest build instances using [custom object creation](#) and data invariants, and then invoke a method of the container and recheck invariants afterwards
 - write [parameterized unit tests](#) that call different methods of the implementation, depending on the parameter values
- An existing code base:
 - use Visual Studio's IntelliTest Wizard to get started by generating a set of [parameterized unit tests \(PUTs\)](#)

The Hello World of IntelliTest

IntelliTest finds inputs relevant to the tested program, which means you can use it to generate the famous **Hello World!** string. This assumes that you have created a C# MSTest-based test project and added a reference to **Microsoft.Pex.Framework**. If you are using a different test framework, create a C# class library and refer to the test framework documentation on how to set up the project.

The following example creates two constraints on the parameter named **value** so that IntelliTest will generate the required string:

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

[TestClass]
public partial class HelloWorldTest {
    [PexMethod]
    public void HelloWorld([PexAssumeNotNull]string value) {
        if (value.StartsWith("Hello")
            && value.EndsWith("World!")
            && value.Contains(" "))
            throw new Exception("found it!");
    }
}
```

Once compiled and executed, IntelliTest generates a set of tests such as the following set:

1. ""
2. "\0\0\0\0\0"
3. "Hello"
4. "\0\0\0\0\0\0"
5. "Hello\0"
6. "Hello\0\0"
7. "Hello\0World!"
8. "Hello World!"

NOTE

For build issues, try replacing Microsoft.VisualStudio.TestTools.TestFramework and Microsoft.VisualStudio.TestTools.TestFramework.Extensions references with a reference to Microsoft.VisualStudio.QualityTools.UnitTesting.

Read [Generate unit tests with IntelliTest](#) to understand where the generated tests are saved. The generated test code should include a test such as the following code:

```
[TestMethod]
[PexGeneratedBy(typeof(global::HelloWorldTest))]
[PexRaisedException(typeof(Exception))]
public void HelloWorldThrowsException167()
{
    this.HelloWorld("Hello World!");
}
```

It's that easy!

Limitations

This section describes the limitations of IntelliTest:

- [Nondeterminism](#)
- [Concurrency](#)
- [Native .NET code](#)

- [Platform](#)
- [Language](#)
- [Symbolic reasoning](#)
- [Stack traces](#)

Nondeterminism

IntelliTest assumes that the analyzed program is deterministic. If it is not, IntelliTest will cycle until it reaches an exploration bound.

IntelliTest considers a program to be non-deterministic if it relies on inputs that IntelliTest cannot control.

IntelliTest controls inputs provided to [parameterized unit tests](#) and obtained from the [PexChoose](#). In that sense, results of calls to unmanaged or uninstrumented code are also considered as "inputs" to the instrumented program, but IntelliTest cannot control them. If the control flow of the program depends on specific values coming from these external sources, IntelliTest cannot "steer" the program towards previously uncovered areas.

In addition, the program is considered to be non-deterministic if the values from external sources change when rerunning the program. In such cases IntelliTest loses control over the execution of the program and its search becomes inefficient.

Sometimes it is not obvious when this happens. Consider the following examples:

- The result of the **GetHashCode()** method is provided by unmanaged code, and is not predictable.
- The **System.Random** class uses the current system time to deliver truly random values.
- The **System.DateTime** class provides the current time, which is not under the control of IntelliTest.

Concurrency

IntelliTest does not handle multithreaded programs.

Native code

IntelliTest does not understand native code, such as x86 instructions called through **P/Invoke**. It does not know how to translate such calls into constraints that can be passed to the [constraint solver](#). Even for .NET code, it can only analyze code it instruments. IntelliTest cannot instrument certain parts of **mscorlib**, including the reflection library. **DynamicMethod** cannot be instrumented.

The suggested workaround is to have a test mode where such methods are located in types in a dynamic assembly. However, even if some methods are uninstrumented, IntelliTest will try to cover as much of the instrumented code as possible.

Platform

IntelliTest is supported only on the X86, 32-bit .NET framework.

Language

In principle, IntelliTest can analyze arbitrary .NET programs, written in any .NET language. However, in Visual Studio it supports only C#.

Symbolic reasoning

IntelliTest uses an automatic [constraint solver](#) to determine which values are relevant for the test and the program under test. However, the abilities of the constraint solver are, and always will be, limited.

Incorrect stack traces

Because IntelliTest catches and "rethrows" exceptions in each instrumented method, the line numbers in stack traces will not be correct. This is a limitation by design of the "rethrow" instruction.

Further reading

- [Introductory blog post.](#)
- [Generate unit tests for your code with IntelliTest](#)

Get started with Microsoft IntelliTest

1/1/2020 • 2 minutes to read • [Edit Online](#)

- If this is your first time with IntelliTest:
 - watch the [Channel 9 video](#)
 - read this [overview on MSDN Magazine](#)
 - read our [documentation](#)
- Ask your questions on [Stack Overflow](#)
- Read the rest of this reference manual
- Print this page for quick reference

Important attributes

- [PexClass](#) marks a type containing **PUT**
- [PexMethod](#) marks a **PUT**
- [PexAssumeNotNull](#) marks a non-null parameter

```
using Microsoft.Pex.Framework;

[..., PexClass(typeof(Foo))]
public partial class FooTest {
    [PexMethod]
    public void Bar([PexAssumeNotNull]Foo target, int i) {
        target.Bar(i);
    }
}
```

- [PexAssemblyUnderTest](#) binds a test project to a project
- [PexInstrumentAssembly](#) specifies an assembly to instrument

```
[assembly: PexAssemblyUnderTest("MyAssembly")] // also instruments "MyAssembly"
[assembly: PexInstrumentAssembly("Lib")]
```

Important static helper classes

- [PexAssume](#) evaluates assumptions (input filtering)
- [PexAssert](#) evaluates assertions
- [PexChoose](#) generates new choices (additional inputs)
- [PexObserve](#) logs live values to the generated tests

```
[PexMethod]
void StaticHelpers(Foo target) {
    PexAssume.IsNotNull(target);

    int i = PexChoose.Value<int>("i");
    string result = target.Bar(i);

    PexObserve.ValueForViewing<string>("result", result);
    PexAssert.IsNotNull(result);
}
```

Got feedback?

Post your ideas and feature requests on [Developer Community](#).

IntelliTest Reference Manual

1/1/2020 • 2 minutes to read • [Edit Online](#)

Contents

- [Overview of IntelliTest](#)

- [The Hello World of IntelliTest](#)
- [Limitations](#)
 - [Nondeterminism](#)
 - [Concurrency](#)
 - [Native code](#)
 - [Platform](#)
 - [Language](#)
 - [Symbolic reasoning](#)
 - [Incorrect stack traces](#)
- [Further reading](#)

- [Get started with IntelliTest](#)

- [Important attributes](#)
- [Important static helper classes](#)

- [Test Generation](#)

- [Test generators](#)
- [Parameterized unit testing](#)
- [Generic parameterized unit testing](#)
- [Allowing exceptions](#)
- [Testing internal types](#)
- [Assumptions and assertions](#)
- [Precondition](#)
- [Postcondition](#)
- [Test failures](#)
- [Setup and tear down](#)
- [Further reading](#)

- [Input Generation](#)

- [Constraint solver](#)
- [Dynamic code coverage](#)
- [Integers and floats](#)
- [Objects](#)
- [Instantiating existing classes](#)
- [Visibility](#)
- [Parameterized mocks](#)
- [Structs](#)
- [Arrays and strings](#)
- [Obtaining additional inputs](#)

- Further reading

- **Exploration Bounds**

- [MaxConstraintSolverTime](#)
- [MaxConstraintSolverMemory](#)
- [MaxBranches](#)
- [MaxCalls](#)
- [MaxStack](#)
- [MaxConditions](#)
- [MaxRuns](#)
- [MaxRunsWithoutNewTests](#)
- [MaxRunsWithUniquePaths](#)
- [MaxExceptions](#)
- [TestExcludePathBoundsExceeded](#)
- [TestEmissionFilter](#)
- [TestEmissionBranchHits](#)

- **Attribute Glossary**

- [PexAssumeNotNull](#)
- [PexClass](#)
- [PexGenericArguments](#)
- [PexMethod](#)
- [PexExplorationAttributeBase](#)
- [PexAssemblySettings](#)
- [PexAssemblyUnderTest](#)
- [PexInstrumentAssemblyAttribute](#)
- [PexUseType](#)
- [PexAllowedException](#)
- [PexAllowedExceptionFromAssembly](#)
- [PexAllowedExceptionFromType](#)
- [PexAllowedExceptionFromTypeUnderTest](#)

- **Settings Waterfall**

- **Static Helper Classes**

- [PexAssume](#)
- [PexAssert](#)
- [PexChoose](#)
- [PexObserve](#)
- [PexSymbolicValue](#)

- **Warnings and Errors**

- [MaxBranches exceeded](#)
- [MaxConstraintSolverTime exceeded](#)
- [MaxConditions exceeded](#)
- [MaxCalls exceeded](#)
- [MaxStack exceeded](#)
- [MaxRuns exceeded](#)
- [MaxRunsWithoutNewTests exceeded](#)

- Cannot concretize solution
- Need help to construct object
- Need help to find types
- Usable type guessed
- Unexpected failure during exploration
- TargetInvocationException
- Uninstrumented method called
- External method called
- Uninstrumentable method called
- Testability issue
- Limitation
- Observed call mismatch
- Value stored in static field

Got feedback?

Post your ideas and feature requests on [Developer Community](#).

Test generation

1/1/2020 • 4 minutes to read • [Edit Online](#)

In traditional unit testing, a test consists of several things:

- A [sequence of method calls](#)
- The arguments with which the methods are called; the arguments are the [test inputs](#)
- Validation of the intended behavior of the tested application by stating a set of [assertions](#)

Following is an example test structure:

```
[Test]
void MyTest() {
    // data
    ArrayList a = new ArrayList();

    // method sequence
    a.Add(5);

    // assertions
    Assert.IsTrue(a.Count==1);
    Assert.AreEqual(a[0], 5);
}
```

IntelliTest can often automatically determine relevant argument values for more general [Parameterized Unit Tests](#), which provide the sequence of method calls and assertions.

Test generators

IntelliTest generates test cases by selecting a sequence of methods of the implementation under test to execute, and then generating inputs for the methods while checking assertions over the derived data.

A [parameterized unit test](#) directly states a sequence of method calls in its body.

When IntelliTest needs to construct objects, calls to constructors and factory methods will be added automatically to the sequence as required.

Parameterized unit testing

Parameterized Unit Tests (PUTs) are tests that take parameters. Unlike traditional unit tests, which are usually closed methods, PUTs take any set of parameters. Is it that simple? Yes - from there, IntelliTest will try to [generate the \(minimal\) set of inputs](#) that [fully cover](#) the code reachable from the test.

PUTs are defined using the [PexMethod](#) custom attribute in a similar fashion to MSTest (or NUnit, xUnit). PUTs are instance methods logically grouped in classes tagged with [PexClass](#). The following example shows a simple PUT stored in the **MyPexTest** class:

```
[PexMethod]
void ReplaceFirstChar(string target, char c) {

    string result = StringHelper.ReplaceFirstChar(target, c);

    Assert.AreEqual(result[0], c);
}
```

where **ReplaceFirstChar** is a method that replaces the first character of a string:

```
class StringHelper {
    static string ReplaceFirstChar(string target, char c) {
        if (target == null) throw new ArgumentNullException();
        if (target.Length == 0) throw new ArgumentOutOfRangeException();
        return c + target.Substring(1);
    }
}
```

From this test, IntelliTest can automatically [generate inputs](#) for a PUT that covers many execution paths of the tested code. Each input that covers a different execution path gets "serialized" as a unit test:

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
void ReplaceFirstChar0() {
    this.ReplaceFirstChar(null, 0);
}

...
[TestMethod]
void ReplaceFirstChar10() {
    this.ReplaceFirstChar("a", 'c');
}
```

Generic parameterized unit testing

Parameterized unit tests can be generic methods. In this case, the user must specify the types used to instantiate the method by using [PexGenericArguments](#).

```
[PexClass]
public partial class ListTest {
    [PexMethod]
    [PexGenericArguments(typeof(int))]
    [PexGenericArguments(typeof(object))]
    public void AddItem<T>(List<T> list, T value)
    { ... }
}
```

Allowing exceptions

IntelliTest provides numerous validation attributes to help triage exceptions into expected exceptions and unexpected exceptions.

Expected exceptions generate negative test cases with the appropriate annotation such as **ExpectedException(typeof(xxx))**, while unexpected exceptions generate failing test cases.

```
[PexMethod, PexAllowedException(typeof(ArgumentNullException))]
void SomeTest() {...}
```

The validators are:

- [PexAllowedException](#): allows a particular exception type from anywhere
- [PexAllowedExceptionFromAssembly](#): allows a particular exception type from a specified assembly
- [PexAllowedExceptionFromType](#): allows a particular exception type from a specified type
- [PexAllowedExceptionFromTypeUnderTest](#): allows a particular exception type from the type under test

Testing internal types

IntelliTest can "test" internal types, as long as it can see them. For IntelliTest to see the types, the following attribute is added to your product or test project by the Visual Studio IntelliTest wizards:

```
[assembly: InternalsVisibleTo("Microsoft.Pex,
PublicKey=0024000004800000940000000602000002400005253413100040000100010007d1fa57c4aed9f0a32e84aa0faef0de9e8
fd6aec8f87fb03766c834c99921eb23be79ad9d5dcc1dd9ad236132102900b723cf980957fc4e177108fc607774f29e8320e92ea05ece4
e821c0a5efe8f1645c4c0c93c1ab99285d622caa652c1dfad63d745d6f2de5f17e5eaf0fc4963d261c8a12436518206dc093344d5ad293
")]
```

Assumptions and assertions

Users can use assumptions and assertions to express [preconditions](#) (assumptions) and [postconditions](#) (assertions) about their tests. When IntelliTest generates a set of parameter values and "explores" the code, it might violate an assumption of the test. When that happens, it will not generate a test for that path but will silently ignore it.

Assertions are a well known concept in regular unit test frameworks, so IntelliTest already "understands" the built-in **Assert** classes provided by each supported test framework. However, most frameworks do not provide an **Assume** class. In that case, IntelliTest provides the [PexAssume](#) class. If you do not want to use an existing test framework, IntelliTest also has the [PexAssert](#) class.

```
[PexMethod]
public void Test1(object o) {
    // precondition: o should not be null
    PexAssume.IsNotNull(o);

    ...
}
```

In particular, the non-nullness assumption can be encoded as a custom attribute:

```
[PexMethod]
public void Test2([PexAssumeNotNull] object o)
// precondition: o should not be null
{
    ...
}
```

Precondition

A precondition of a method expresses the conditions under which the method will succeed.

Usually, the precondition is enforced by checking the parameters and the object state, and throwing an **ArgumentException** or **InvalidOperationException** if it is violated.

In IntelliTest, a precondition of a [parameterized unit test](#) is expressed with [PexAssume](#).

Postcondition

A postcondition of a method expresses the conditions which should hold during and after execution of the method, assuming that its preconditions were initially valid.

Usually, the postcondition is enforced by calls to **Assert** methods.

With IntelliTest, a postcondition of a [parameterized unit test](#) is expressed with **PexAssert**.

Test failures

When does a generated test case fail?

1. If it does not terminate within the [configured path bounds](#), it is considered as a failure unless the [TestExcludePathBoundsExceeded](#) option is set
2. If the test throws a **PexAssumeFailedException**, it succeeds. However, it is usually filtered out unless [TestEmissionFilter](#) is set to **All**
3. If the test violates an [assertion](#); for example, by throwing an assertion violation exception of a unit testing framework, it fails

If none of the above produce a decision, a test succeeds if and only if it does not throw an exception. Assertion violations are treated in the same way as exceptions.

Setup and tear down

As part of the integration with test frameworks, IntelliTest supports detecting and running setup and tear down methods.

Example

```
using Microsoft.Pex.Framework;
using NUnit.Framework;

namespace MyTests
{
    [PexClass]
    [TestFixture]
    public partial class MyTestClass
    {
        [SetUp]
        public void Init()
        {
            // monitored
        }

        [PexMethod]
        public void MyTest(int i)
        {
        }

        [TearDown]
        public void Dispose()
        {
            // monitored
        }
    }
}
```

Further reading

- [Test to code binding](#)
- [One test to rule them all](#)

Got feedback?

Post your ideas and feature requests on [Developer Community](#).

Input generation using dynamic symbolic execution

1/1/2020 • 6 minutes to read • [Edit Online](#)

IntelliTest generates inputs for [parameterized unit tests](#) by analyzing the branch conditions in the program. Test inputs are chosen based on whether they can trigger new branching behaviors of the program. The analysis is an incremental process. It refines a predicate $q: I \rightarrow \{\text{true}, \text{false}\}$ over the formal test input parameters I . q represents the set of behaviors that IntelliTest has already observed. Initially, $q := \text{false}$, since nothing has yet been observed.

The steps of the loop are:

1. IntelliTest determines inputs i such that $q(i) = \text{false}$ using a [constraint solver](#). By construction, the input i will take an execution path not seen before. Initially, this means that i can be any input, because no execution path has yet been discovered.
2. IntelliTest executes the test with the chosen input i , and monitors the execution of the test and the program under test.
3. During the execution, the program takes a particular path that is determined by all the conditional branches of the program. The set of all conditions that determine the execution is called the *path condition*, written as the predicate $p: I \rightarrow \{\text{true}, \text{false}\}$ over the formal input parameters. IntelliTest computes a representation of this predicate.
4. IntelliTest sets $q := (q \text{ or } p)$. In other words, it records the fact that it has seen the path represented by p .
5. Go to step 1.

IntelliTest's [constraint solver](#) can deal with values of all types that may appear in .NET programs:

- [Integers](#) and [Floats](#)
- [Objects](#)
- [Structs](#)
- [Arrays](#) and [Strings](#)

IntelliTest filters out inputs that violate stated assumptions.

Besides the immediate inputs (arguments to [parameterized unit tests](#)), a test can draw further input values from the [PexChoose](#) static class. The choices also determine the behavior of [parameterized mocks](#).

Constraint solver

IntelliTest uses a constraint solver to determine the relevant input values of a test and the program under test.

IntelliTest uses the [Z3](#) constraint solver.

Dynamic code coverage

As a side-effect of the runtime monitoring, IntelliTest collects dynamic code coverage data. This is called *dynamic* because IntelliTest knows only about code that has been executed, therefore it cannot give absolute values for coverage in the same way as other coverage tool usually do.

For example, when IntelliTest reports the dynamic coverage as 5/10 basic blocks, this means that five blocks

out of ten were covered, where the total number of blocks in all methods that have been reached so far by the analysis (as opposed to all methods that exist in the assembly under test) is ten. Later in the analysis, as more reachable methods are discovered, both the numerator (5 in this example) and the denominator (10) may increase.

Integers and floats

IntelliTest's [constraint solver](#) determines test input values of primitive types such as **byte**, **int**, **float**, and others in order to trigger different execution paths for the test and the program under test.

Objects

IntelliTest can either [create instances of existing .NET classes](#), or you can use IntelliTest to automatically [create mock objects](#) that implement a specific interface and behave in different ways depending on usage.

Instantiate existing classes

What's the problem?

IntelliTest monitors the executed instructions when it runs a test and the program under test. In particular, it monitors all access to fields. It then uses a [constraint solver](#) to determine new test inputs, including objects and their field values, such that the test and the program under test will behave in other interesting ways.

This means that IntelliTest must create objects of certain types and set their field values. If the class is [visible](#) and has a [visible](#) default constructor, IntelliTest can create an instance of the class. If all the fields of the class are [visible](#), IntelliTest can set the fields automatically.

If the type is not visible, or the fields are not [visible](#), IntelliTest needs help to create objects and bring them into interesting states in order to achieve maximal code coverage. IntelliTest could use reflection to create and initialize instances in arbitrary ways, but this is not usually desirable because it might bring the object into a state that can never occur during normal program execution. Instead, IntelliTest relies on hints from the user.

Visibility

.NET has an elaborate visibility model: types, methods, fields, and other members can be **private**, **public**, **internal**, and more.

When IntelliTest generates tests, it will attempt to perform only actions (such as calling constructors, methods, and setting fields) that are legal with regard to .NET visibility rules from within the context of the generated tests.

The rules are as follows:

- **Visibility of internal members**

- IntelliTest assumes that the generated tests will have access to internal members that were visible to the enclosing [PexClass](#). .NET has the **InternalsVisibleToAttribute** to extend the visibility of internal members to other assemblies.

- **Visibility of private and family (protected in C#) members of the PexClass**

- IntelliTest always places the generated tests directly in the [PexClass](#) or into a subclass. Therefore, IntelliTest assumes that it may use all visible family members (**protected** in C#).
- If the generated tests are placed directly into the [PexClass](#) (usually by using partial classes), IntelliTest assumes that it may also use all private members of the [PexClass](#).

- **Visibility of public members**

- IntelliTest assumes that it may use all exported members visible in the context of the [PexClass](#).

Parameterized mocks

How to test a method that has a parameter of an interface type? Or of a non-sealed class? IntelliTest does not know which implementations will later be used when this method is called. And perhaps there isn't even a real implementation available at test time.

The conventional answer is to use *mock objects* with explicit behavior.

A mock object implements an interface (or extends a non-sealed class). It does not represent a real implementation, but just a shortcut that allows the execution of tests using the mock object. Its behavior is defined manually as part of each test case where it is used. Many tools exist that make it easy to define mock objects and their expected behavior, but this behavior must still be manually defined.

Instead of hard-coded values in mock objects, IntelliTest can generate the values. Just as it enables [parameterized unit testing](#), IntelliTest also enables parameterized mocks.

Parameterized mocks have two different execution modes:

- **choosing**: when exploring code, parameterized mocks are a source of additional test inputs, and IntelliTest will attempt to choose interesting values
- **replay**: when executing a previously generated test, parameterized mocks behave like stubs with behavior (in other words, predefined behavior).

Use [PexChoose](#) to obtain values for parameterized mocks.

Structs

IntelliTest's reasoning about **struct** values is similar to the way it deals with [objects](#).

Arrays and strings

IntelliTest monitors the executed instructions as it runs a test and the program under test. In particular, it observes when the program depends on the length of a string or an array (and the lower bounds and lengths of a multi-dimensional array). It also observes how the program uses the different elements of a string or array. It then uses a [constraint solver](#) to determine which lengths and element values might cause the test and the program under test to behave in interesting ways.

IntelliTest attempts to minimize the size of the arrays and strings required to trigger interesting program behaviors.

Obtain additional inputs

The [PexChoose](#) static class can be used to obtain additional inputs to a test, and can be used to implement [parameterized mocks](#).

Got feedback?

Post your ideas and feature requests on [Developer Community](#).

Further reading

- [How does it work?](#)

Exploration bounds

10/18/2019 • 7 minutes to read • [Edit Online](#)

PexSettingsAttributeBase is the abstract base class for settings bounds as attributes. See [Settings Waterfall](#) for an overview of settings in IntelliTest.

You can modify the settings by using named properties of this and its derived attributes:

```
[PexClass(MaxRuns = 10)]
public partial class FooTest {....}
```

- **Constraint solving bounds**

- [MaxConstraintSolverTime](#) - The number of seconds the [constraint solver](#) has to discover inputs that will cause a new and different execution path to be followed.
- [MaxConstraintSolverMemory](#) - The size in Megabytes that the [constraint solver](#) may use to discover inputs.

- **Exploration Path Bounds**

- [MaxBranches](#) - The maximum number of branches that may be taken along a single execution path.
- [MaxCalls](#) - The maximum number of calls that may be made during a single execution path.
- [MaxStack](#) - The maximum size of the stack at any time during a single execution path, measured as the number of active call frames.
- [MaxConditions](#) - The maximum number of conditions over the inputs that may be checked during a single execution path.

- **Exploration Bounds**

- [MaxRuns](#) - The maximum number of runs that will be attempted during an exploration.
- [MaxRunsWithoutNewTests](#) - The maximum number of consecutive runs without a new test being emitted.
- [MaxRunsWithUniquePaths](#) - The maximum number of runs with unique execution paths that will be attempted during an exploration.
- [MaxExceptions](#) - The maximum number of exceptions that may be found for a combination of all discovered execution paths.

- **Test Suite Code Generation Settings**

- [TestExcludePathBoundsExceeded](#) - When true, execution paths which exceed any of the path bounds ([MaxCalls](#), [MaxBranches](#), [MaxStack](#), [MaxConditions](#)) are ignored.
- [TestEmissionFilter](#) - Indicates under which circumstances IntelliTest should emit tests.
- [TestEmissionBranchHits](#) - Controls how many tests IntelliTest emits.

MaxConstraintSolverTime

The number of seconds the [constraint solver](#) has to calculate inputs that will cause a new and different execution path to be taken. This is an option of the **PexSettingsAttributeBase** and its derived types.

The deeper that IntelliTest explores the execution paths of a program, the more complex the constraint systems that IntelliTest builds from the control-flow and data-flow of the program become. Depending on your time limitation, you can set this value to allow IntelliTest to take more or less time discovering new execution paths.

Typically, the reason for a timeout is that IntelliTest is trying to find a solution for a constraint system that does not have a solution, but it is not aware of this fact. Since this is the most common case for a timeout, it may not make

sense to increase the bound.

MaxConstraintSolverMemory

The number of Megabytes that the [constraint solver](#) has to calculate inputs that will cause a new and different execution path to be taken. This is an option of the *PexSettingsAttributeBase** and its derived types.

The deeper IntelliTest explores the execution paths of a program, the more complex the constraint systems that IntelliTest builds from the control-flow and data-flow of the program become. Depending on the available memory of your computer, you can set this value to allow IntelliTest to tackle more complex constraint systems.

Typically, the reason for a timeout is that IntelliTest is trying to find a solution for a constraint system that does not have a solution, but it is not aware of this fact. Since this is the most common cause of an out-of-memory situation, it may not make sense to increase the bound.

MaxBranches

The maximum number of branches that may be taken along a single execution path.

The motivation behind this exploration bound is to limit the length of any execution path that IntelliTest explores during [input generation](#). In particular, it prevents IntelliTest from becoming unresponsive if the program goes into an infinite loop.

Each conditional and unconditional branch of the executed and monitored code is counted towards this limit, including branches which do not depend on the inputs of the parameterized test.

For example, the following code consumes branches in of the order 100:

```
for (int i=0; i<100; i++) { }
```

MaxCalls

The maximum number of calls that may be made during a single execution path.

The motivation behind this exploration bound is to limit the length of any execution path that IntelliTest explores during [input generation](#). In particular, it prevents IntelliTest from becoming unresponsive if the program calls a method recursively an infinite number of times, which would cause a stack overflow that IntelliTest cannot recover from.

Each call (direct, indirect, virtual, jump) of the executed and monitored code is counted towards this limit.

MaxStack

The maximum size of the stack at any time during a single execution path, measured by the number of active call frames.

The motivation behind this exploration bound is to limit the size of the stack of any execution path that IntelliTest explores during [input generation](#). In particular, it prevents IntelliTest from using all available stack space, which would cause a stack overflow that IntelliTest cannot recover from.

MaxConditions

The emaximum number of conditions over the inputs that may be checked during a single execution path.

The motivation behind this exploration bound is to limit the complexity of any execution path that IntelliTest explores during [input generation](#). Each conditional branch that depends on the inputs of the parameterized test is

counted towards this limit.

For example, each path in the following code consumes $n+1$ conditions:

```
[PexMethod]
void ParameterizedTest(int n)
{
    for (int i=0; i<n; i++) { // conditions are "0<n", "1<n", ..., "!(n<n)"
        ...
    }
    for (int i=0; i<100; i++) { // irrelevant for MaxConditions, since conditions do not depend on input
        ...
    }
}
```

MaxRuns

The maximum number of runs that IntelliTest will try during the exploration of a test.

The motivation behind this exploration bound is that any code which contains loops or recursion might have an infinite number of execution paths, and thus IntelliTest needs to be limited during [input generation](#).

The two settings **MaxRuns** and **MaxRunsWithUniquePaths** are related as follows:

- IntelliTest will call a parameterized test method up to **MaxRuns** times with different test inputs.
- If the executed code is deterministic, IntelliTest will take a different execution path each time. However, under some conditions the executed code might follow an execution path it has already taken before, with different inputs.
- IntelliTest counts how many unique execution paths it finds; this number is limited by the **MaxRunsWithUniquePaths** option.

MaxRunsWithoutNewTests

The maximum number of consecutive runs without a new test being emitted.

While IntelliTest can often find many interesting test inputs within a short time, after a while it will not find any more new test inputs and will emit no more unit tests. This configuration option places a bound on the number of consecutive attempts IntelliTest may perform without emitting a new test. When reached, it will halt the exploration.

MaxRunsWithUniquePaths

The maximum number of unique paths that IntelliTest will consider during an exploration.

The motivation behind this exploration bound is that any code containing loops or recursion might have an infinite number of execution paths, and so IntelliTest must be limited during [input generation](#).

The two settings **MaxRuns** and **MaxRunsWithUniquePaths** are related as follows:

- IntelliTest will call a parameterized test method up to **MaxRuns** times with different test inputs.
- If the executed code is deterministic, IntelliTest will take a different execution path each time. However, under some conditions the executed code might follow an execution path it has already taken before, with different inputs.
- IntelliTest counts how many unique execution paths it finds; this number is limited by the **MaxRunsWithUniquePaths** option.

MaxExceptions

The maximum number of exceptions that can be encountered before exploration is halted.

The motivation behind this exploration bound is to stop the exploration of code that contains many bugs. If IntelliTest finds too many errors in the code, exploration is stopped.

TestExcludePathBoundsExceeded

Execution paths that exceed the configured path bounds [MaxCalls](#), [MaxBranches](#), [MaxStack](#), and [MaxConditions](#) are ignored.

The motivation behind this exploration bound is to deal with (most likely) non-terminating tests. When IntelliTest reaches an exploration bound such as [MaxCalls](#), [MaxBranches](#), [MaxStack](#), or [MaxConditions](#), it assumes that the test will not be a non-terminating process, and will not cause a stack overflow later on. Such test cases may pose problems to other test frameworks, and this attribute provides a way to prevent IntelliTest from emitting test cases for potentially non-terminating processes or test cases that will cause a stack overflow.

TestEmissionFilter

Indicates the types of tests that IntelliTest should emit. The possible values are:

- **All** - Emit tests for everything, including assumption violations.
- **FailuresAndIncreasedBranchHits** (the default) - Emit tests for all unique failures, and whenever a test case increases coverage as controlled by [TestEmissionBranchHits](#).
- **FailuresAndUniquePaths** - Emit tests for all failures IntelliTest finds, and also for each test input that causes a unique execution path.
- **Failures** - Emit tests for failures only.

TestEmissionBranchHits

Depending on the current [TestEmissionFilter](#) setting, IntelliTest emits new test cases when they cover a branch in the program that was not covered before.

The **TestEmissionBranchHits** setting determines if IntelliTest should just consider whether a branch was covered at all (**TestEmissionBranchHits=1**), if a test covered it either once or twice (**TestEmissionBranchHits=2**), and so on.

TestEmissionBranchHits=1 will produce a very small test suite that will cover all branches IntelliTest could reach. In particular, this test suite will also cover all basic blocks and statements it reached.

The default for this option is **TestEmissionBranchHits=2**, which generates a more expressive test suite that is also better suited to detecting future regression errors.

Got feedback?

Post your ideas and feature requests on [Developer Community](#).

Attribute glossary

1/1/2020 • 3 minutes to read • [Edit Online](#)

Attributes by namespace

- **Microsoft.Pex.Framework**
 - [PexAssumeNotNull](#)
 - [PexClass](#)
 - [PexGenericArguments](#)
 - [PexMethod](#)
 - [PexExplorationAttributeBase](#)
- **Microsoft.Pex.Framework.Settings**
 - [PexAssemblySettings](#)
- **Microsoft.Pex.Framework.Instrumentation**
 - [PexAssemblyUnderTest](#)
 - [PexInstrumentAssembly](#)
- **Microsoft.Pex.Framework.Using**
 - [PexUseType](#)
- **Microsoft.Pex.Framework.Validation**
 - [PexAllowedException](#)
 - [PexAllowedExceptionFromAssembly](#)
 - [PexAllowedExceptionFromType](#)
 - [PexAllowedExceptionFromTypeUnderTest](#)

PexAssumeNotNull

This attribute asserts that the governed value cannot be **null**. It can be attached to:

- a **parameter** of a parameterized test method

```
// assume foo is not null
[PexMethod]
public void SomeTest([PexAssumeNotNull]IFoo foo, ...)
```

- a **field**

```
public class Foo {
    // this field should not be null
    [PexAssumeNotNull]
    public object Bar;
}
```

- a **type**

```
// never consider null for Foo types
[PexAssumeNotNull]
public class Foo {}
```

It can also be attached to a test assembly, test fixture or test method; in this case the first arguments must indicate to which field or type the assumptions apply. When the attribute applies to a type, it applies to all fields with this formal type.

PexClass

This attribute marks a class that contains *explorations*. It is the equivalent of the MSTest **TestClassAttribute** (or the NUnit **TestFixtureAttribute**). This attribute is optional.

The classes marked with **PexClass** must be *default constructible*:

- publicly exported type
- default constructor
- not abstract

If the class does not meet those requirements, an error is reported and the exploration fails.

It is also strongly advised to make those classes **partial** so that IntelliTest can generate new tests that are part of the class, but in a separate file. This approach solves many problems due to **visibility** and is a typical technique in C#.

Additional suite and categories:

```
[TestClass] // MSTest test fixture attribute
[PexClass(Suite = "checkin")] // fixture attribute
public partial class MyTests { ... }
```

Specifying the type under test:

```
[PexClass(typeof(Foo))] // this is a test for Foo
public partial class FooTest { ... }
```

The class may contain methods annotated with **PexMethod**. IntelliTest also understands [set up and tear down methods](#).

PexGenericArguments

This attribute provides a type tuple for instantiating a [generic parameterized unit test](#).

PexMethod

This attribute marks a method as a [parameterized unit test](#). The method must reside within a class marked with the **PexClass** attribute.

IntelliTest will generate traditional, parameterless tests, which call the [parameterized unit test](#) with different parameters.

The parameterized unit test:

- must be an instance method
- must be [visible](#) to the test class into which the generated tests are placed according to the [Settings Waterfall](#)

- may take any number of parameters
- may be generic

Example

```
[PexClass]
public partial class MyTests {
    [PexMethod]
    public void MyTest(int i)
    { ... }
}
```

PexExplorationAttributeBase

[More information](#)

PexAssemblySettings

This attribute can be set at the assembly level to override default setting values for all explorations.

```
using Microsoft.Pex.Framework;
// overriding the test framework selection
[assembly: PexAssemblySettings(TestFramework = "MSTestv2")]
```

PexAssemblyUnderTest

This attribute specifies an assembly that is being tested by the current test project.

```
[assembly: PexAssemblyUnderTest("MyAssembly")]
```

PexInstrumentAssemblyAttribute

This attribute is used to specify an assembly to be instrumented.

Example

```
using Microsoft.Pex.Framework;

// the assembly containing ATypeFromTheAssemblyToInstrument should be instrumented
[assembly: PexInstrumentAssembly(typeof(ATypeFromTheAssemblyToInstrument))]

// the assembly name can be used as well
[assembly: PexInstrumentAssembly("MyAssemblyName")]
```

PexUseType

This attribute tells IntelliTest that it can use a particular type to instantiate (abstract) base types or interfaces.

Example

```
[PexMethod]
[PexUseType(typeof(A))]
[PexUseType(typeof(B))]
public void MyTest(object testParameter)
{
    ... // IntelliTest will consider types A and B to instantiate 'testParameter'
}
```

PexAllowedException

If this attribute is attached to a [PexMethod](#) (or to a [PexClass](#)), it changes the default IntelliTest logic that indicates when tests fails. The test will not be considered as failed, even if it throws the specified exception.

Example

The following test specifies that the constructor of **Stack** may throw an **ArgumentOutOfRangeException**:

```
class Stack {
    int[] _elements;
    int _count;
    public Stack(int capacity) {
        if (capacity<0) throw new ArgumentOutOfRangeException();
        _elements = new int[capacity];
        _count = 0;
    }
    ...
}
```

The filter is attached to a fixture as follows (it can also be defined at the assembly or test level):

```
[PexMethod]
[PexAllowedException(typeof(ArgumentOutOfRangeException))]
class CtorTest(int capacity) {
    Stack s = new Stack(capacity); // may throw ArgumentOutOfRangeException
}
```

PexAllowedExceptionFromAssembly

[More information](#)

PexAllowedExceptionFromType

[More information](#)

PexAllowedExceptionFromTypeUnderTest

[More information](#)

Got feedback?

Post your ideas and feature requests on [Developer Community](#).

Settings waterfall

1/1/2020 • 2 minutes to read • [Edit Online](#)

The concept of the settings waterfall means that the user can specify settings at the **Assembly**, **Fixture**, and **Exploration** level:

- Assembly - [PexAssemblySettings](#)
- Fixture - [PexClass](#)
- Exploration - [PexExplorationAttributeBase](#)

Settings specified at the **Assembly** level affect all fixtures and exploration under that assembly. Settings specified at the **Fixture** level affect all explorations under that fixture. Child settings win—if a setting is defined at the **Assembly** and the **Fixture** levels, the **Fixture** settings are used.

Note that some settings are specific to the **Assembly** level or **Fixture** level.

Example

```
using Microsoft.Pex.Framework;

[assembly: PexAssemblySettings(MaxBranches = 1000)] // we override the default value of maxbranches

namespace MyTests
{
    [PexClass(MaxBranches = 500)] // we override the 1000 value and set maxbranches to 500
    public partial class MyTests
    {
        [PexMethod(MaxBranches = 100)] // we override again, maxbranches = 100
        public void MyTest(...) { ... }
    }
}
```

Got feedback?

Post your ideas and feature requests on [Developer Community](#).

Static helper classes

1/1/2020 • 3 minutes to read • [Edit Online](#)

IntelliTest provides a set of static helper class that can be used when authoring [parameterized unit tests](#):

- [PexAssume](#): used to define assumptions on inputs, and is useful for filtering undesirable inputs
- [PexAssert](#): a simple assertion class for use if your test framework does not provide one
- [PexChoose](#): a stream of additional test inputs that IntelliTest manages
- [PexObserve](#): logs concrete values and, optionally, validates them in the generated code

Some classes allow you to interact with the IntelliTest reasoning engine at a low-level:

- [PexSymbolicValue](#): utilities to inspect or modify symbolic constraints on variables

PexAssume

A static class used to express assumptions, such as [preconditions](#), in [parameterized unit tests](#). The methods of this class can be used to filter out undesirable test inputs.

If the assumed condition does not hold for some test input, a **PexAssumeFailedException** is thrown. This will cause the test to be silently ignored.

Example

The following parameterized test will not consider **j=0**:

```
public void TestSomething(int i, int j) {  
    PexAssume.AreNotEqual(j, 0);  
    int k = i/j;  
    ...  
}
```

Remarks

The code above is almost equivalent to:

```
if (j==0)  
    return;
```

except that a failing **PexAssume** results in no test cases. In the case of an **if** statement, IntelliTest generates a separate test case to cover the **then** branch of the **if** statement.

PexAssume also contains specialized nested classes for assumptions on string, arrays, and collections.

PexAssert

A static class used to express assertions, such as [postconditions](#), in [parameterized unit tests](#).

If the asserted condition does not hold for some test input, a **PexAssertFailedException** is thrown, which causes the test to fail.

Example

The following asserts that the absolute value of an integer is positive:

```
public void TestSomething(int i) {
    int j = Maths.Abs(i);
    PexAssert.IsTrue(j >= 0);
    ...
}
```

PexChoose

A static class that supplies auxiliary input values to a test, which can be used to implement [Parameterized Mocks](#).

The **PexChoose** class does not help in determining whether a test passes or fails for particular input values.

PexChoose simply provides input values, which are also referred to as *choices*. It is still up to the user to restrict the input values, and to write assertions that define when a test passes or fails.

Modes of operation

The **PexChoose** class can operate in two modes:

- While IntelliTest is performing a symbolic analysis of the test and the tested code during [input generation](#), the chooser returns arbitrary values and IntelliTest tracks how each value is used in the test and the tested code. IntelliTest will generate relevant values to trigger different execution paths in the test and the tested code.
- The generated code for particular test cases sets up the choice provider in a specific way, so that the re-execution of such a test case will make specific choices to trigger a particular execution path.

Usage

- Simple call **PexChoose.Value** to generate a new value:

```
public int Foo() {
    return PexChoose.Value<int>("foo");
}
```

PexObserve

A static class to log named values.

When IntelliTest explores the code, **PexObserve** is used to record computed values using their formatted string representations. The values are associated with unique names.

```
PexObserve.Value<string>("result", result);
```

Example

```

// product code
public static class MathEx {
    public static int Square(int value) { return value * value; }
}

// fixture
[TestClass]
public partial class MathExTests {
    [PexMethod]
    public int SquareTest(int a) {
        int result = MathEx.Square(a);
        // storing result
        return result;
    }
}

```

PexSymbolicValue

A static class used to ignore constraints on parameters, and to print the symbolic information associated with values.

Usage

Normally, IntelliTest tries to cover all execution paths of the code during execution. However, especially when computing assumption and assertion conditions, it should not explore all possible cases.

Example

This example shows the implementation of the **PexAssume.Arrays.ElementsAreNotNull** method. In the method, you ignore the constraints on the length of the array value to avoid IntelliTest trying to generate different sizes of array. The constraints are ignored only here. If the tested code behaves differently for different array lengths, IntelliTest cannot generate different sized arrays from the constraints of the tested code.

```

public static void AreElementsNotNull<T>(T[] value)
    where T : class
{
    PexAssume.NotNull(value);
    // the followings prevents the exploration of all array lengths
    int len = PexSymbolicValue.Ignore<int>(value.Length);

    // building up a boolean value as follows prevents exploration
    // of all combinations of non-null (instead, there are just two cases)
    bool anyNull = false;
    for (int i = 0; i < len; ++i)
        anyNull |= value[i] == null;

    // was any element null?
    if (anyNull)
        PexAssume.Fail("some element of array is a null reference");
}

```

Got feedback?

Post your ideas and feature requests on [Developer Community](#).

Warnings and errors

1/16/2020 • 9 minutes to read • [Edit Online](#)

Warnings and errors by category

- **Boundaries**

- [MaxBranches exceeded](#)
- [MaxConstraintSolverTime exceeded](#)
- [MaxConditions exceeded](#)
- [MaxCalls exceeded](#)
- [MaxStack exceeded](#)
- [MaxRuns exceeded](#)
- [MaxRunsWithoutNewTests exceeded](#)

- **Constraint Solving**

- [Cannot Concretize Solution](#)

- **Domains**

- [Need Help To Construct Object](#)
- [Need Help To Find Types](#)
- [Usable Type Guessed](#)

- **Execution**

- [Unexpected Failure During Exploration](#)
- [TargetInvocationException](#)

- **Instrumentation**

- [Uninstrumented Method Called](#)
- [External Method Called](#)
- [Uninstrumentable Method Called](#)
- [Testability Issue](#)
- [Limitation](#)

- **Interpreter**

- [Observed Call Mismatch](#)
- [Value Stored In Static Field](#)

MaxBranches exceeded

IntelliTest limits the length of any execution path that it explores during [input generation](#). This feature prevents IntelliTest from becoming unresponsive when the program goes into an infinite loop.

Every conditional and unconditional branch of the executed and monitored code is counted towards this limit, including branches that do not depend on the inputs of the [parameterized unit test](#).

For example, the following code consumes branches in the order of 100:

```
for (int i=0; i<100; i++) { }
```

You can edit the **MaxBranches** option of an attribute derived from **PexSettingsAttributeBase**, such as [PexClass](#) or [PexMethod](#). The following example effectively removes this bound:

```
[PexMethod(MaxBranches=int.MaxValue)]
public void MyTest(...) {
    // ...
}
```

You can also set the **TestExcludePathBoundsExceeded** option to inform IntelliTest how generally to deal with these issues.

In the test code, you can use [PexSymbolicValue](#) to ignore constraints generated by the loop condition:

```
for (int i=0;
    PexSymbolicValue.Ignore(i<100); // IntelliTest will 'forget' about this path condition
    i++)
{ }
```

MaxConstraintSolverTime exceeded

IntelliTest uses a [constraint solver](#) to compute new test inputs. Constraint solving can be a very time consuming process, so IntelliTest allows you to configure bounds - in particular, **MaxConstraintSolverTime**.

For many applications, significantly increasing the timeout will not result in better coverage. The reason for this is that most timeouts are caused by constraint systems that have no solutions. However, IntelliTest might not be able to determine that it is inconsistent without trying all possible solutions, which will result in a timeout.

MaxConditions exceeded

IntelliTest limits the length of any execution path that it explores during [input generation](#). This feature prevents IntelliTest from becoming unresponsive when the program enters an infinite loop.

Each conditional branch that depends on the inputs of the [parameterized unit test](#) is counted towards this limit.

For example, each path in the following code consumes **n+1** conditions:

```
[PexMethod]
void ParameterizedTest(int n) {
    // conditions are "0<n", "1<n", ..., !(n<n)"
    for (int i=0; i<n; i++)
    { ... }

    // irrelevant for MaxConditions, since conditions do not depend on input
    for (int i=0; i<100; i++)
    { ... }
}
```

You can edit the **MaxConditions** option of an attribute derived from **PexSettingsAttributeBase**, such as [PexClass](#) or [PexMethod](#). For example:

```
[PexMethod(MaxConditions=10000)]
void ParameterizedTest(int n) {
    // ...
}
```

You can also set the **TestExcludePathBoundsExceeded** option to inform IntelliTest how to generally deal with

these issues.

You can use [PexSymbolicValue](#) to ignore constraints generated by the loop condition:

```
[PexMethod]
void ParameterizedTest(int n) {
    int nshadow = PexSymbolicValue.Ignore(n); // IntelliTest loses track of 'n'

    // irrelevant for MaxConditions, since nshadow is not related to input
    for (int i=0; i<nshadow; i++)
    {...}
}
```

MaxCalls exceeded

IntelliTest limits the length of any execution path that it explores during [input generation](#). This feature prevents IntelliTest from becoming unresponsive when the program enters an infinite loop.

Each call (direct, indirect, virtual, or jump) of the executed and monitored code is counted towards this limit.

You can edit the **MaxCalls** option of an attribute derived from [PexSettingsAttributeBase](#), such as [PexClass](#) or [PexMethod](#). The following example effectively removes this bound:

```
[PexMethod(MaxCalls=int.MaxValue)]
public void MyTest(...) {
    // ...
}
```

You can also set the **TestExcludePathBoundsExceeded** option to inform IntelliTest how to generally deal with these issues.

MaxStack exceeded

IntelliTest limits the size of the call stack of any execution path that it explores during [input generation](#). This feature prevents IntelliTest from terminating when a stack overflow occurs.

You can edit the **MaxStack** option of an attribute derived from [PexSettingsAttributeBase](#), such as [PexClass](#) or [PexMethod](#). The following example effectively removes this bound (not recommended):

```
[PexMethod(MaxStack=int.MaxValue)]
public void MyTest(...) {
    // ...
}
```

You can also set the **TestExcludePathBoundsExceeded** option to inform IntelliTest how to generally deal with these issues.

MaxRuns exceeded

IntelliTest limits the number of execution paths that it explores during [input generation](#). This feature ensures that IntelliTest terminates when the program has loops or recursion.

It may not be the case that, every time IntelliTest runs the parameterized test with particular inputs, it emits a new test case. See [TestEmissionFilter](#) for more information.

You can edit the **MaxRuns** option of an attribute derived from [PexSettingsAttributeBase](#), such as [PexClass](#) or [PexMethod](#). The following example effectively removes this bound (not recommended):

```
[PexMethod(MaxRuns=2000)]
public void MyTest(...) {
    // ...
}
```

MaxRunsWithoutNewTests exceeded

IntelliTest limits the number of execution paths that it explores during [input generation](#). This feature ensures that IntelliTest terminates when the program has loops or recursion.

It may not be the case that, every time IntelliTest runs the parameterized test with particular inputs, it emits a new test case. See [TestEmissionFilter](#) for more information.

While IntelliTest often finds many interesting test inputs initially, it might not - after a while - emit any more tests. This option governs how long IntelliTest may keep trying to find another relevant test input.

You can edit the **MaxRunsWithoutNewTests** option of an attribute derived from **PexSettingsAttributeBase**, such as [PexClass](#) or [PexMethod](#). The following example effectively removes this bound (not recommended):

```
[PexMethod(MaxRunsWithoutNewTests=2000)]
public void MyTest(...) {
    // ...
}
```

Cannot concretize solution

This error is often the consequence of an earlier error. IntelliTest uses a [constraint solver](#) to determine new test inputs. Sometimes, test inputs proposed by the [constraint solver](#) are invalid. This can happen when:

- certain constraints are not known
- if values are created in a user-defined way, causing errors to occur in user code
- some of the types involved have initialization logic not controlled by IntelliTest (for example, COM classes)

Need help to construct object

IntelliTest [generates test inputs](#), and some of the inputs may be objects with fields. Here, IntelliTest tries to generate an instance of a class that has a private field, and it assumes that an interesting program behavior will occur when this private field has a particular value.

However, while this is possible with Reflection, IntelliTest does not manufacture objects with arbitrary field values. Instead, in these cases, it relies on the user to provide hints about how to use the public methods of a class to create an object, and bring it into a state where its private field has the desired value.

Read [Instantiating existing classes](#) to learn how you can help IntelliTest construct interesting objects.

Need help to find types

IntelliTest [generates test inputs](#) for any .NET type. Here, IntelliTest tries to create an instance that derives from an abstract class or implements an abstract interface, and IntelliTest does not know of any type that fulfills the constraints.

You can help IntelliTest by pointing to one or more types that match the constraints. Usually, one of the following attributes will help:

- **PexUseTypeAttribute**, which points to a particular type.

For example, if IntelliTest reports that it "does not know of any types assignable to **System.Collections.IDictionary**", you can help it by attaching the following **PexUseTypeAttribute** to the test (or to the fixture class):

```
[PexMethod]
[PexUseType(typeof(System.Collections.Hashtable))]
public void MyTest(IDictionary[] dictionaries) { ... }
```

- **An assembly-level attribute**

```
[assembly: PexUseType(typeof(System.Collections.Hashtable))]
```

Usable type guessed

IntelliTest [generates test inputs](#) for any .NET types. When a type is abstract or an interface, IntelliTest must choose a particular implementation of that type. To make that choice, it needs to know which types exist.

When this warning is shown, it indicates that IntelliTest looked at some of referenced assemblies and found an implementation type, but it is not sure if it should use that type, or if there are more appropriate types available elsewhere. IntelliTest simply chose a type that looked promising.

In order to avoid this warning, you can either accept IntelliTest's type choice, or assist IntelliTest in using other types by adding a corresponding [PexUseType](#).

Unexpected failure during exploration

An unexpected exception was caught while exploring a test.

Please [report this as a bug](#).

TargetInvocationException

An exception occurred in user code. Inspect the stack trace, and remove the bug in your code.

Uninstrumented method called

IntelliTest [generates test inputs](#) by monitoring program execution. It is essential that the relevant code is properly instrumented so that IntelliTest can monitor its behavior.

This warning appears when the instrumented code calls methods in another, uninstrumented assembly. If you want IntelliTest to explore the interaction of both, you must also instrument the other assembly (or parts of it).

External method called

IntelliTest [generates test inputs](#) by monitoring execution of .NET applications. IntelliTest cannot generate meaningful test inputs for code that is not written in a .NET language.

This warning appears when the instrumented code calls an unmanaged, native method that IntelliTest cannot analyze. If you want IntelliTest to explore the interaction of both, you must mock the unmanaged method.

Uninstrumentable method called

IntelliTest [generates test inputs](#) by monitoring execution of .NET applications. However, there are some methods that, for technical reasons, IntelliTest cannot monitor. For example, IntelliTest cannot monitor a static constructor.

This warning appears when the instrumented code calls a method that IntelliTest cannot monitor.

Testability issue

IntelliTest [generates test inputs](#) by monitoring the program execution. It can only generate relevant test inputs when the program is deterministic, and when the relevant behavior is controlled by the test inputs.

This warning appears because, during execution of your test case, a method was called that either behaves non-deterministically, or interacts with the environment. Examples are methods of **System.Random** and **System.IO.File**. If you want IntelliTest to create meaningful test inputs, you must mock the methods that IntelliTest flags as testability issues.

Limitation

IntelliTest [generates test inputs](#) by using a [constraint solver](#). However, there are some operations that are beyond the scope of the [constraint solver](#). This currently includes:

- most floating point operations (only some linear arithmetic is supported on floating point numbers)
- conversions between floating point numbers and integers
- all operations on the **System.Decimal** type

This warning appears when the executed code performs an operation or calls a method that IntelliTest cannot interpret.

Observed call mismatch

IntelliTest [generates test inputs](#) by monitoring program execution. However, IntelliTest might not be able to monitor all instructions. For example, it cannot monitor native code, and it cannot monitor code that is not instrumented.

When IntelliTest cannot monitor code, it cannot generate test inputs that are relevant to that code. Often, IntelliTest is not aware of the fact that it cannot monitor a method until a call to that method returns. However, the cause of this warning is:

- IntelliTest monitored some code, which initiated a call to an uninstrumented method
- The uninstrumented method called a method that is instrumented
- IntelliTest monitors the instrumented method that was called

IntelliTest does not know what the uninstrumented intermediate method did, so it might not be able to generate test inputs that are relevant to the nested instrumented call.

Value stored in static field

IntelliTest can systematically determine [relevant test inputs](#) only when the unit test is deterministic; in other words, it always behaves the same way for the same test inputs. In particular, this means that the test should leave the system in a state that allows to re-execute that test. Ideally, the unit test should not change any global state, but all interactions with globals should be mocked.

This warning indicates that a static field was changed; this might make the test behave non-deterministically.

In some situations, changing a static field is acceptable:

- when the test inputs causes setup or cleanup code to undo the change
- when the field is initiated only once, and the value does not change afterwards

Got feedback?

Post your ideas and feature requests on [Developer Community](#).

Install unit test frameworks

1/1/2020 • 2 minutes to read • [Edit Online](#)

Visual Studio Test Explorer can run tests from any unit test framework that has developed an adapter interface for it. Installing the framework copies the binaries and adds Visual Studio project templates for the languages it supports. When you create a project with the template, the framework is registered with Test Explorer.

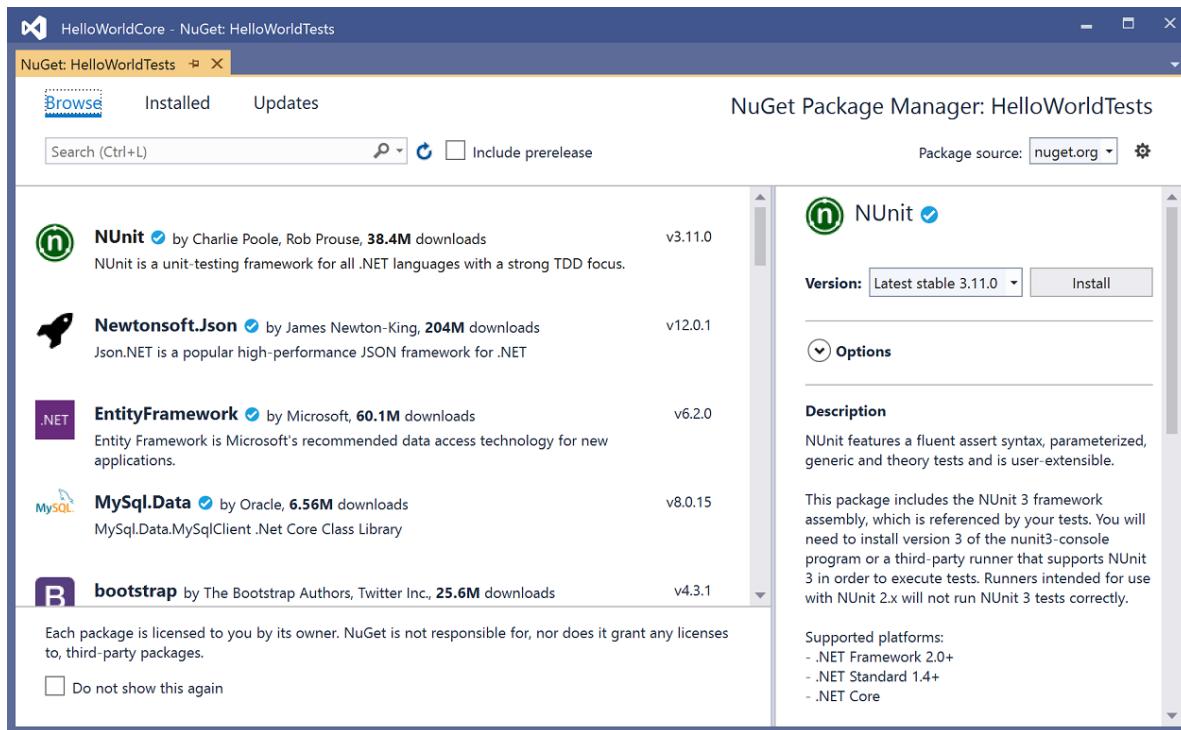
A Visual Studio solution can contain unit test projects that use different frameworks and that are targeted at different languages.

[MSTest](#) is the test framework provided by Visual Studio and is installed by default.

Acquire frameworks

Install third-party unit test frameworks by using **NuGet Package Manager**.

1. Right-click on the project that will contain your test code and select **Manage NuGet Packages**.
2. In **NuGet Package Manager**, search for the test framework you want to install, and then click **Install**.



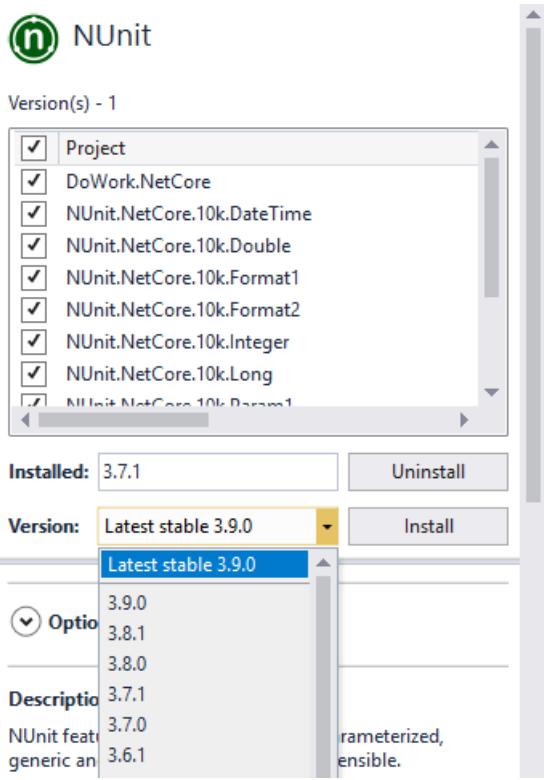
Update to the latest test adapters

Update to the latest stable test adapter to experience better test discovery and execution. For more information about updates to MSTest, NUNIT, and xUnit test adapters, see the [Visual Studio blog](#).

To update to the latest stable test adapter version

1. Open the Nuget Package Manager for your solution by navigating to **Tools > NuGet Package Manager > Manage NuGet Packages for Solution**.
2. Click on the **Updates** tab and search for MSTest, NUNIT, or xUnit test adapters that are installed.
3. Select each test adapter, and then select the latest stable version in the drop-down menu.

4. Choose the **Install** button.



See also

- [Unit test your code](#)

Run unit tests with Test Explorer

1/1/2020 • 13 minutes to read • [Edit Online](#)

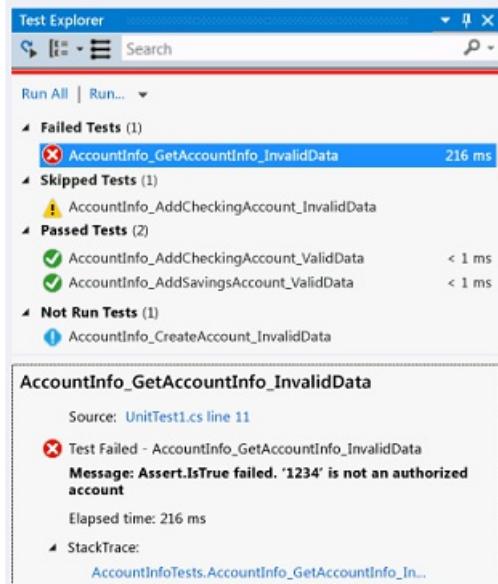
Use Test Explorer to run unit tests from Visual Studio or third-party unit test projects. You can also use Test Explorer to group tests into categories, filter the test list, and create, save, and run playlists of tests. You can debug tests and analyze test performance and code coverage.

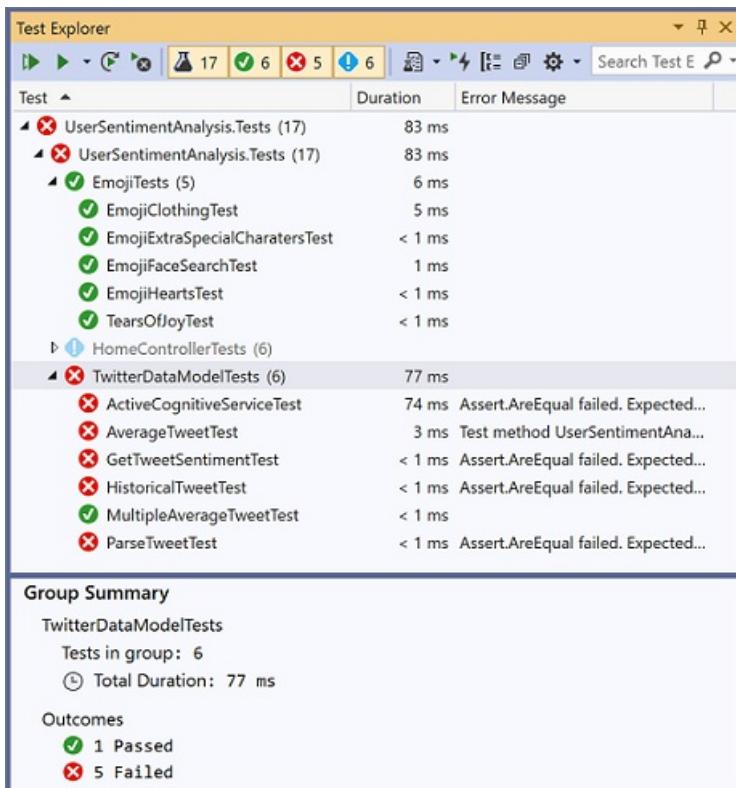
Visual Studio includes the Microsoft unit testing frameworks for both managed and native code. However, Test Explorer can also run any unit test framework that has implemented a Test Explorer adapter. For more information about installing third-party unit test frameworks, see [Install third-party unit test frameworks](#)

Test Explorer can run tests from multiple test projects in a solution and from test classes that are part of the production code projects. Test projects can use different unit test frameworks. When the code under test is written for .NET, the test project can be written in any language that also targets .NET, regardless of the language of the target code. Native C/C++ code projects must be tested by using a C++ unit test framework. For more information, see [Write unit tests for C/C++](#).

Run tests in Test Explorer

When you build the test project, the tests appear in Test Explorer. If Test Explorer is not visible, choose **Test** on the Visual Studio menu, choose **Windows**, and then choose **Test Explorer**.

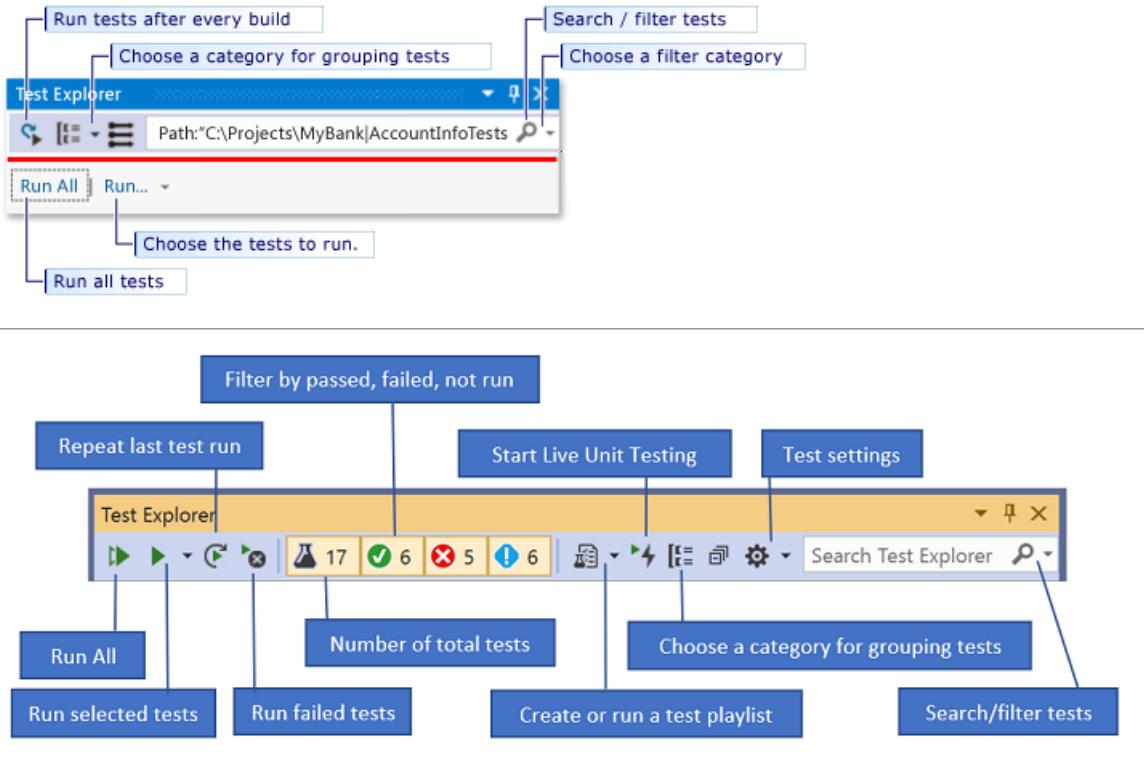




As you run, write, and rerun your tests, Test Explorer displays the results in default groups of **Failed Tests**, **Passed Tests**, **Skipped Tests** and **Not Run Tests**. You can change the way Test Explorer groups your tests.

As you run, write, and rerun your tests, the Test Explorer displays the results in a default grouping of **Project**, **Namespace**, and **Class**. You can change the way the Test Explorer groups your tests.

You can perform much of the work of finding, organizing and running tests from the **Test Explorer** toolbar.



Run tests

You can run all the tests in the solution, all the tests in a group, or a set of tests that you select. Do one of the following:

- To run all the tests in a solution, choose **Run All**.

- To run all the tests in a default group, choose **Run** and then choose the group on the menu.
- Select the individual tests that you want to run, open the right-click menu for a selected test and then choose **Run Selected Tests**.
- If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution with the  toggle button on the toolbar. This can noticeably reduce the time taken to run all the tests.

The **pass/fail bar** at the top of the **Test Explorer** window is animated as the tests run. At the conclusion of the test run, the **pass/fail bar** turns green if all tests passed or turns red if any test failed.

You can run all the tests in the solution, all the tests in a group, or a set of tests that you select. Do one of the following:

- To run all the tests in a solution, choose the **Run All** icon.
- To run all the tests in a default group, choose the **Run** icon and then choose the group on the menu.
- Select the individual tests that you want to run, open the right-click menu for a selected test and then choose **Run Selected Tests**.
- If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution in the settings menu of the toolbar. This can noticeably reduce the time taken to run all the tests.

Run tests after every build

BUTTON	DESCRIPTION
	To run your unit tests after each local build, choose Test on the standard menu, and then choose Run Tests After Build on the Test Explorer toolbar.

NOTE

Running unit tests after each build requires Visual Studio 2017 Enterprise or Visual Studio 2019. In Visual Studio 2019 it is included in Community and Professional as well as Enterprise.

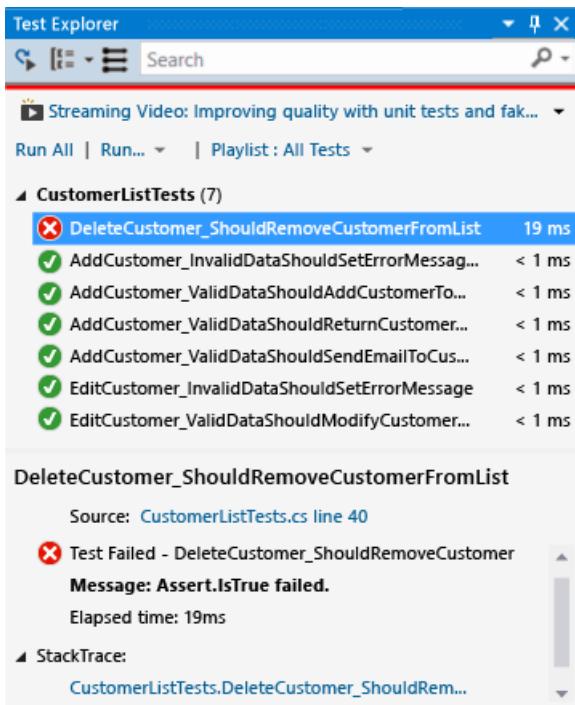
To run your unit tests after each local build, open the settings icon in the Test Explorer toolbar and select **Run Tests After Build**.

View test results

As you run, write, and rerun your tests, Test Explorer displays the results in groups of **Failed Tests**, **Passed Tests**, **Skipped Tests** and **Not Run Tests**. The details pane at the bottom or side of the Test Explorer displays a summary of the test run.

View test details

To view the details of an individual test, select the test.



The test details pane displays the following information:

- The source file name and the line number of the test method.
- The status of the test.
- The elapsed time that the test method took to run.

If the test fails, the details pane also displays:

- The message returned by the unit test framework for the test.
- The stack trace at the time the test failed.

View the source code of a test method

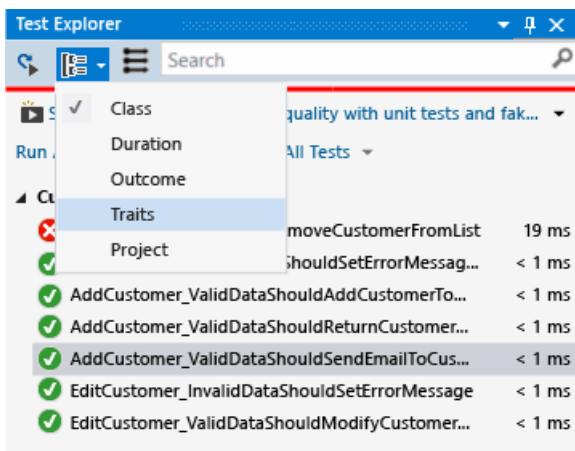
To display the source code for a test method in the Visual Studio editor, select the test and then choose **Open Test** on the right-click menu (Keyboard: **F12**).

Group and filter the test list

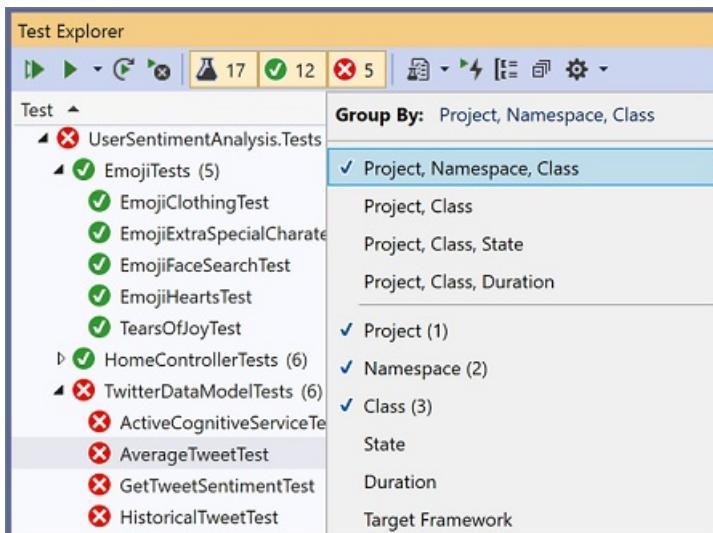
Test Explorer lets you group your tests into predefined categories. Most unit test frameworks that run in Test Explorer let you define your own categories and category/value pairs to group your tests. You can also filter the list of tests by matching strings against test properties.

Group tests in the test list

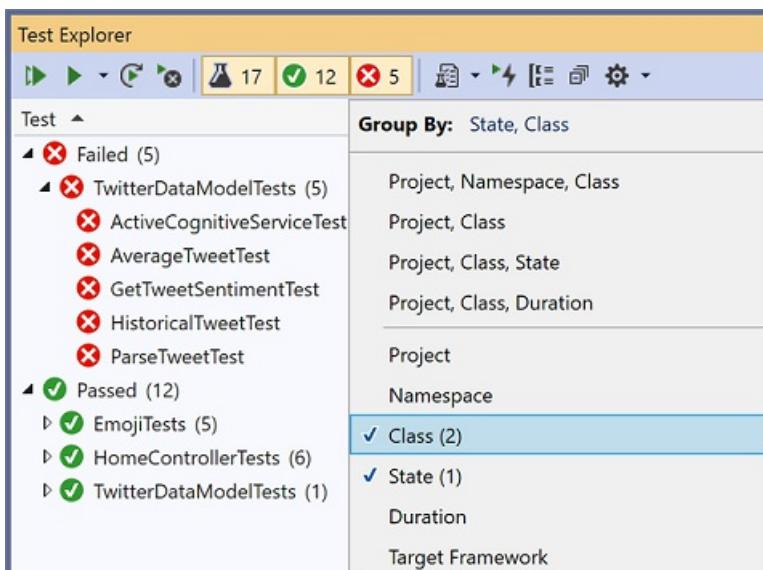
To change the way that tests are organized, choose the down arrow next to the **Group By** button and select a new grouping criteria.



Test Explorer lets you group your tests into a hierarchy. The default hierarchy grouping is **Project, Namespace, Class**, and then **Class**. To change the way that tests are organized, choose the **Group By** button and select a new grouping criteria.



You can define your own levels of the hierarchy and group by **State** and then **Class** for example by selecting Group By options in your preferred order.



Test Explorer groups

GROUP	DESCRIPTION
Duration	Groups test by execution time: Fast , Medium , and Slow .

GROUP	DESCRIPTION
Outcome	Groups tests by execution results: Failed Tests, Skipped Tests, Passed Tests.
Traits	Groups test by category/value pairs that you define. The syntax to specify trait categories and values is defined by the unit test framework.
Project	Groups test by the name of the projects.
GROUP	DESCRIPTION
Duration	Groups tests by execution time: Fast, Medium, and Slow.
State	Groups tests by execution results: Failed Tests, Skipped Tests, Passed Tests, Not Run
Target Framework	Groups tests by the framework their projects target
Namespace	Groups tests by the containing namespace.
Project	Groups tests by the containing project.
Class	Groups tests by the containing class.

Traits

A trait is usually a category name/value pair, but it can also be a single category. Traits can be assigned to methods that are identified as a test method by the unit test framework. A unit test framework can define trait categories. You can add values to the trait categories to define your own category name/value pairs. The syntax to specify trait categories and values is defined by the unit test framework.

Traits in the Microsoft Unit Testing Framework for Managed Code

In the Microsoft unit test framework for managed apps, you define a trait name/ value pair in a [TestPropertyAttribute](#) attribute. The test framework also contains these predefined traits:

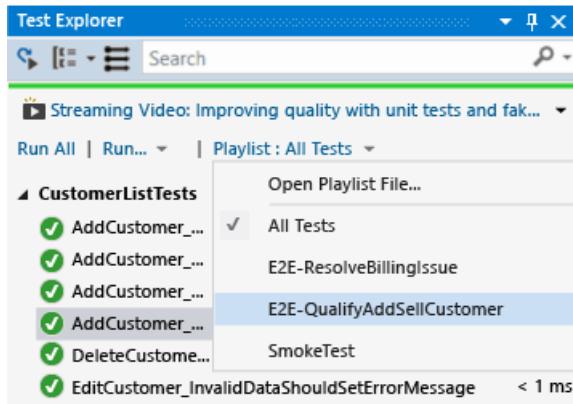
TRAIT	DESCRIPTION
OwnerAttribute	The Owner category is defined by the unit test framework and requires you to provide a string value of the owner.
PriorityAttribute	The Priority category is defined by the unit test framework and requires you to provide an integer value of the priority.
TestCategoryAttribute	The TestCategory attribute enables you to provide a category without a value.
TestPropertyAttribute	The TestProperty attribute enables you to define trait category/value pair.

Traits in the Microsoft Unit Testing Framework for C++

See [How to use the Microsoft Unit Testing Framework for C++](#).

Create custom playlists

You can create and save a list of tests that you want to run or view as a group. When you select a playlist, the tests in the list are displayed in Test Explorer. You can add a test to more than one playlist, and all tests in your project are available when you choose the default **All Tests** playlist.



To create a playlist, choose one or more tests in Test Explorer. On the right-click menu, choose **Add to Playlist** > **New Playlist**. Save the file with the name and location that you specify in the **Create New Playlist** dialog box.

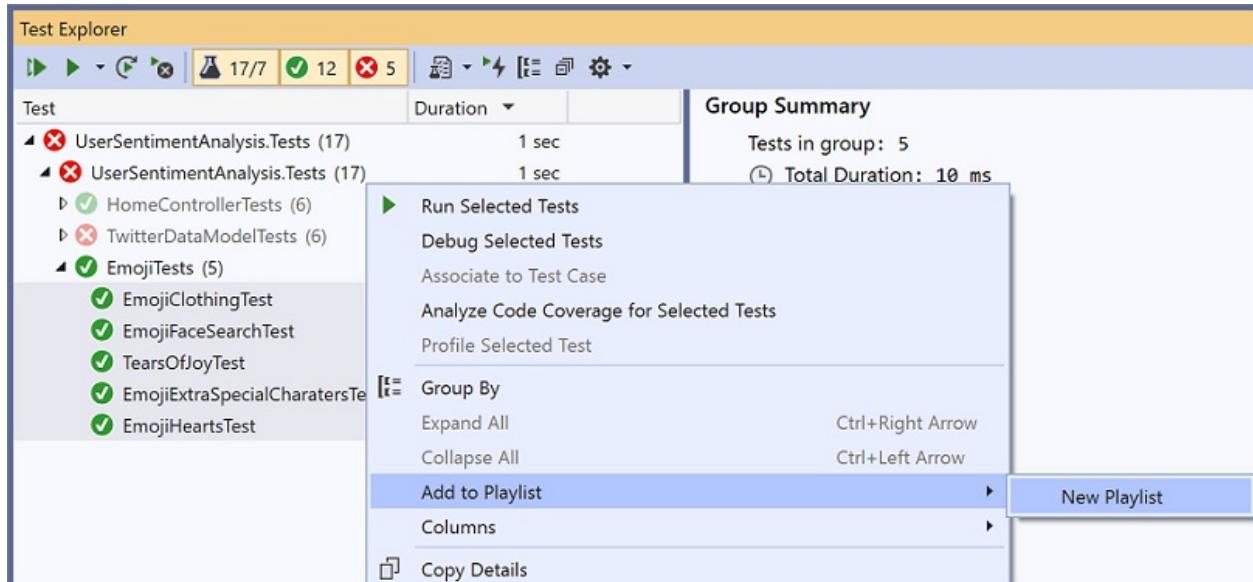
To add tests to a playlist, choose one or more tests in Test Explorer. On the right-click menu, choose **Add to Playlist**, and then choose the playlist that you want to add the tests to.

To open a playlist, choose **Test** > **Playlist** from the Visual Studio menu, and either choose from the list of recently used playlists, or choose **Open Playlist** to specify the name and location of the playlist.

If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution with the toggle button on the toolbar. This can noticeably reduce the time taken to run all the tests.

You can create and save a list of tests that you want to run or view as a group. When you select a playlist, the tests in the list are displayed in a new Test Explorer tab. You can add a test to more than one playlist.

To create a playlist, choose one or more tests in Test Explorer. On the right-click menu, choose **Add to Playlist** > **New Playlist**.



The playlist opens in a new Test Explorer tab. You can use this playlist once and then discard it, or you can click the **Save** button in the playlist window's toolbar, and then select a name and location to save the playlist.

The screenshot shows the Test Explorer interface with a title bar 'Tests: Playlist #1'. The toolbar includes icons for running tests, filtering, and saving. The main area displays a tree view of tests under 'Test' and a 'Duration' column. A specific test, 'EmojiClothingTest', is selected and highlighted with a dotted border. To the right, a 'Test Detail Summary' pane shows the selected test's details: 'Source: EmojiTests.cs line: 26' and 'Duration: 8 ms'. Other tests listed include 'UserSentimentAnalysis.Tests' (5), 'EmojiTests' (5), 'EmojiFaceSearchTest', 'TearsOfJoyTest', 'EmojiExtraSpecialCharater...', and 'EmojiHeartsTest'.

To create a playlist, choose one or more tests in Test Explorer. Right-click and choose **Add to Playlist > New playlist**.

To open a playlist, choose the playlist icon in the Visual Studio toolbar and select a previously saved playlist file from the menu.

Test Explorer columns

The **groups** are also available as columns in Test Explorer, along with Trait, Stack Trace, Error Message, and Fully Qualified Name. Most columns are not visible by default, and you can customize which columns you see and the order in which they appear.

The screenshot shows the Test Explorer interface with a context menu open over a selected test case, 'EmojiHeartsTest'. The menu includes options like 'Run Selected Tests', 'Debug Selected Tests', 'Associate to Test Case', 'Analyze Code Coverage for Selected Tests', 'Profile Selected Test', 'Group By', 'Expand All', 'Collapse All', 'Add to Playlist', 'Columns', and 'Copy Details'. The 'Columns' option is currently selected. The 'Test Detail Summary' pane on the right shows the selected test's details: 'Source: EmojiTests.cs line: 46' and 'Duration: 8 ms'.

Filter, sort, and rearrange test columns

Columns can be filtered, sorted, and rearranged.

- To filter to specific traits, click the filter icon at the top of the Traits column.

The screenshot shows the Test Explorer interface with a 'Clear Filter' dialog open over the 'Traits' column header. The dialog lists available traits: '(Select All)', 'Integration (3)' (which is checked), 'SearchTest (2)', and 'SkipWhenLiveUnitTesting (2)'. The main Test Explorer window shows a list of tests with their durations and traits. The 'Traits' column header has a small filter icon next to it.

- To change the order of the columns, click on a column header and drag it left or right.
- To sort a column, click on the column header. Not all columns can be sorted. You can also sort by a

secondary column by holding the **Shift** key and clicking on an additional column header.

Duration	▼
276 ms	
276 ms	
206 ms	
61 ms	
9 ms	
8 ms	
1 ms	
< 1 ms	

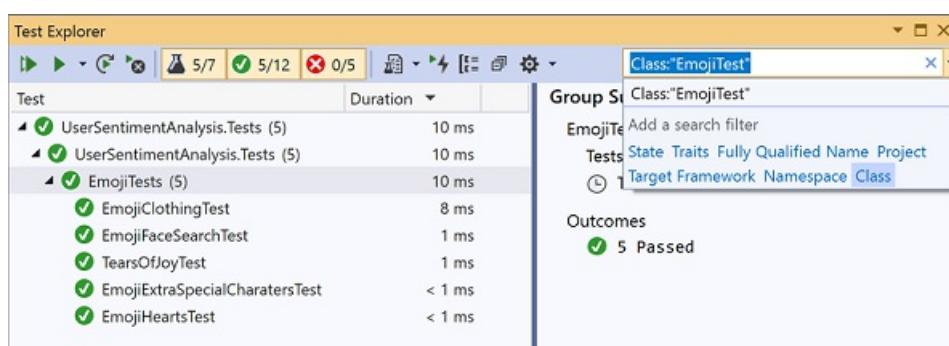
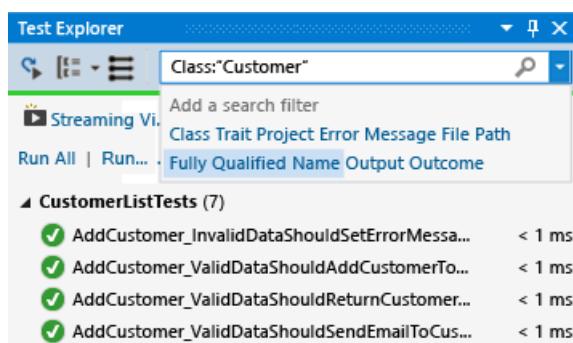
Search and filter the test list

You can also use Test Explorer search filters to limit the test methods in your projects that you view and run.

When you type a string in the **Test Explorer** search box and choose **Enter**, the test list is filtered to display only those tests whose fully qualified names contain the string.

To filter by a different criteria:

1. Open the drop-down list to the right of the search box.
2. Choose a new criteria.
3. Enter the filter value between the quotation marks. If you want to search for an exact match on the string instead of a containing match use an equals sign (=) instead of the colon (:).



NOTE

Searches are case insensitive and match the specified string to any part of the criteria value.

QUALIFIER	DESCRIPTION
Trait	Searches both trait category and value for matches. The syntax to specify trait categories and values are defined by the unit test framework.

QUALIFIER	DESCRIPTION
Project	Searches the test project names for matches.
Error Message	Searches the user-defined error messages returned by failed asserts for matches.
File Path	Searches the fully qualified file name of test source files for matches.
Fully Qualified Name	Searches the fully qualified name of test namespaces, classes, and methods for matches.
Output	Searches the user-defined error messages that are written to standard output (stdout) or standard error (stderr). The syntax to specify output messages are defined by the unit test framework.
Outcome	Searches the Test Explorer category names for matches: Failed Tests, Skipped Tests, Passed Tests .

QUALIFIER	DESCRIPTION
State	Searches the Test Explorer category names for matches: Failed Tests, Skipped Tests, Passed Tests .
Traits	Searches both trait category and value for matches. The syntax to specify trait categories and values are defined by the unit test framework.
Fully Qualified Name	Searches the fully qualified name of test namespaces, classes, and methods for matches.
Project	Searches the test project names for matches.
Target Framework	Searches the Test Explorer category names for matches: Failed Tests, Skipped Tests, Passed Tests .
Namespace	Searches the test namespaces for matches.
Class	Searches the test classes names for matches.

To exclude a subset of the results of a filter, use the following syntax:

```
FilterName:"Criteria" -FilterName:"SubsetCriteria"
```

For example, `FullName:"MyClass" - FullName:"PerfTest"` returns all tests that include "MyClass" in their name, except tests that also include "PerfTest" in their name.

Debug and analyze unit tests

You can use Test Explorer to start a debugging session for your tests. Stepping through your code with the Visual Studio debugger seamlessly takes you back and forth between the unit tests and the project under test. To start debugging:

1. In the Visual Studio editor, set a breakpoint in one or more test methods that you want to debug.

NOTE

Because test methods can run in any order, set breakpoints in all the test methods that you want to debug.

2. In Test Explorer, select the test methods and then choose **Debug Selected Tests** on the right-click menu.

For more information, about the debugger, see [Debug in Visual Studio](#).

Diagnose test method performance issues

To diagnose why a test method is taking too much time, select the method in Test Explorer and then choose **Profile Selected Test** on the right-click menu. See [Performance Explorer](#).

Analyze unit test code coverage

You can determine the amount of product code that is actually being tested by your unit tests by using the Visual Studio code coverage tool that's available in Visual Studio Enterprise edition. You can run code coverage on selected tests or on all tests in a solution.

To run code coverage for test methods in a solution:

1. Choose **Test** on the top menu bar and then choose **Analyze code coverage**.
2. Choose one of the following commands from the sub-menu:
 - **Selected tests** runs the test methods that you have selected in Test Explorer.
 - **All tests** runs all the test methods in the solution.
 - Right-click in Test Explorer and select **Analyze Code Coverage for Selected tests**

The **Code Coverage Results** window displays the percentage of the blocks of product code that were exercised by line, function, class, namespace and module.

For more information, see [Use code coverage to determine how much code is being tested](#).

Test shortcuts

Tests can be run from Test Explorer by right-clicking in the code editor on a test and selecting **Run test** or by using the default [Test Explorer shortcuts](#) in Visual Studio. Some of the shortcuts are context-based. This means that they run or debug tests based on where your cursor is in the code editor. If your cursor is inside a test method, then that test method runs. If your cursor is at the class level, then all the tests in that class run. This is the same for the namespace level as well.

FREQUENT COMMANDS	KEYBOARD SHORTCUTS
TestExplorer.DebugAllTestsInContext	Ctrl+R, Ctrl+T
TestExplorer.RunAllTestsInContext	Ctrl+R, T
TestExplorer.RunAllTests	Ctrl+R, A
TestExplorer.RepeatLastRun	Ctrl+R, L

NOTE

You can't run a test in an abstract class, because tests are only defined in abstract classes and not instantiated. To run tests in abstract classes, create a class that derives from the abstract class.

See also

- [Unit test your code](#)
- [Run a unit test as a 64-bit process](#)
- [Test Explorer FAQ](#)

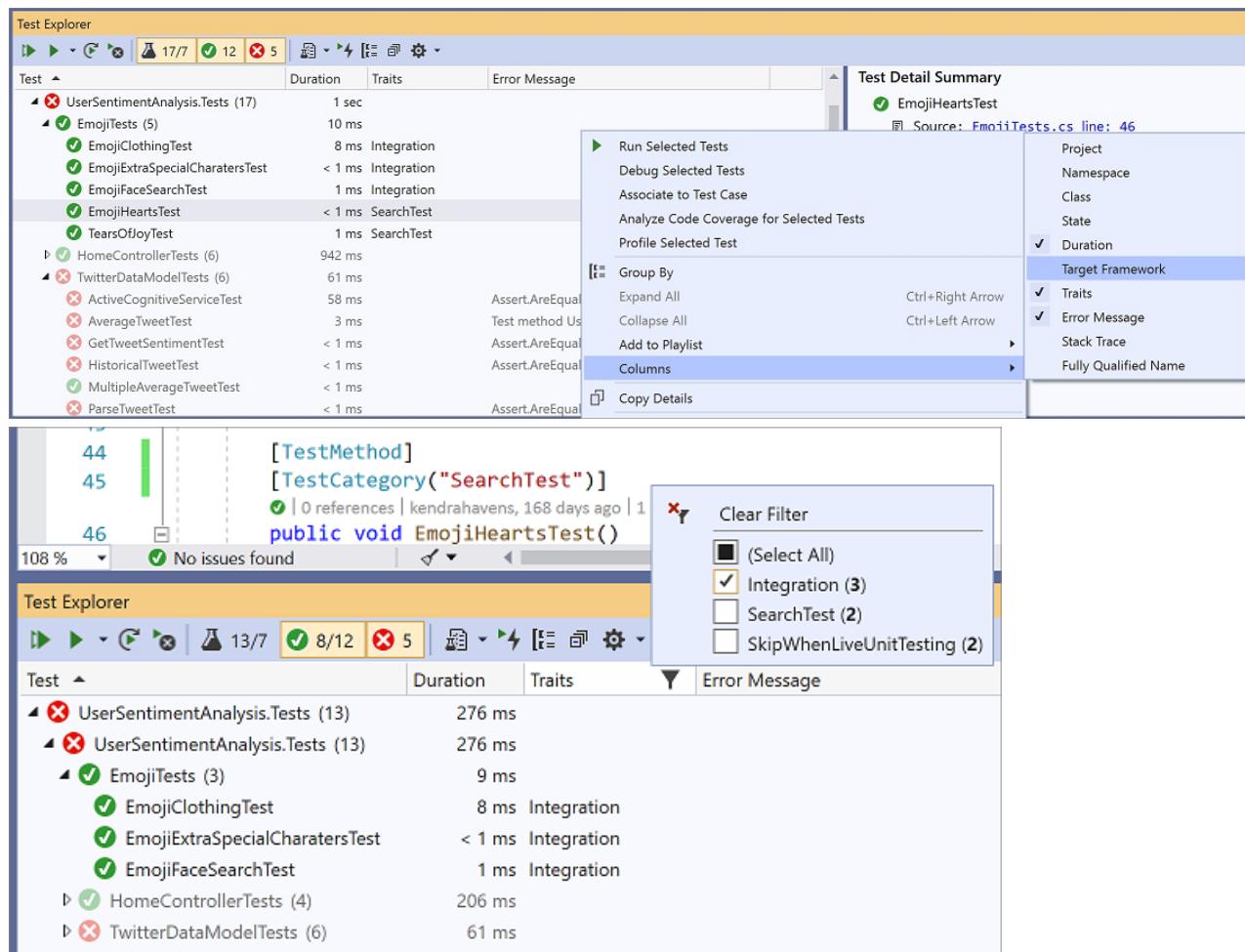
Visual Studio Test Explorer FAQ

1/10/2020 • 6 minutes to read • [Edit Online](#)

Where is group by Traits in Visual Studio 2019?

This Trait grouping was moved to be a column. With the multi-tiered and customizable hierarchy in Visual Studio 2019 version 16.2, we thought including traits as a grouping created unneeded visual complexity. We are definitely listening to feedback on this design! <https://developercommunity.visualstudio.com/content/problem/588029/no-longer-able-to-group-by-trait-in-test-explorer.html>

For now, you can right click on the column in the Test Explorer and select Columns. Check the Trait column and it will appear in the Test Explorer. You can now filter this column by what traits you are interested in.



Dynamic test discovery

Test Explorer is not discovering my tests that are dynamically defined. (For example, theories, custom adapters, custom traits, #ifdefs, etc.) How can I discover these tests?

Build your project to run assembly-based discovery.

Build your project and make sure assembly-based discovery is turned on in **Tools > Options > Test**.

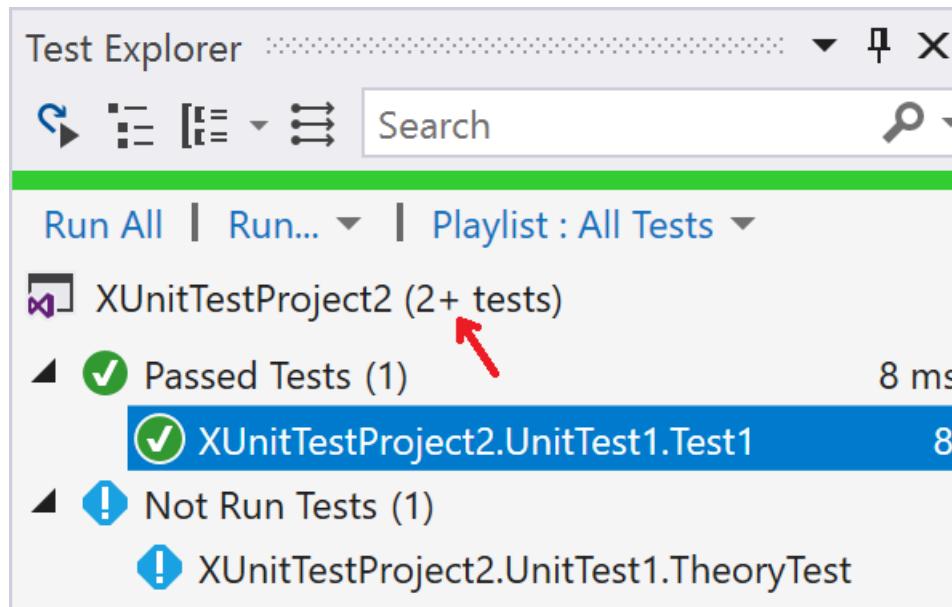
Real-time test discovery is source-based test discovery. It can't discover tests that use theories, custom adapters, custom traits, `#ifdef` statements, and more because they're defined at run time. A build is required for those tests to be accurately found. In Visual Studio 2017 version 15.6 and later, assembly-based discovery (the traditional

discoverer) runs only after builds. This setting means real-time test discovery finds as many tests as it can while you're editing, and assembly-based discovery allows dynamically defined tests to appear after a build. Real-time test discovery improves responsiveness, but still allows you to get complete and precise results after a build.

Test Explorer '+' (plus) symbol

What does the '+' (plus) symbol that appears in the top line of Test Explorer mean?

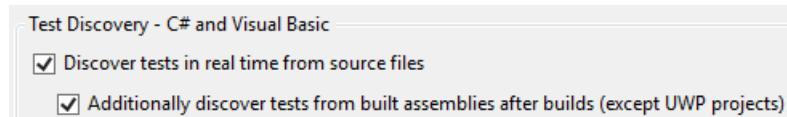
The '+' (plus) symbol indicates that more tests may be discovered after a build when assembly-based discovery runs. This symbol appears if dynamically defined tests are detected in your project.



Assembly-based discovery

Assembly-based discovery is no longer working for my project. How do I turn it back on?

Go to **Tools > Options > Test** and check the box for **Additionally discover tests from built assemblies after builds**.



Real-time test discovery

Tests now appear in Test Explorer while I type, without having to build my project. What changed?

This feature is called [Real-time test discovery](#). It uses a Roslyn analyzer to find tests and populate Test Explorer in real time, without requiring you to build your project. For more information about test discovery behavior for dynamically defined tests such as theories or custom traits, see [Dynamic test discovery](#).

Real-time test discovery compatibility

What languages and test frameworks can use Real Time Test Discovery?

[Real-time test discovery](#) only works for the managed languages (C# and Visual Basic), since it's built using the Roslyn compiler. For now, real-time test discovery only works for the xUnit, NUnit, and MSTest frameworks.

Test Explorer logs

How can I turn on logs for the Test Explorer?

Navigate to **Tools > Options > Test** and find the Logging section there.

UWP test discovery

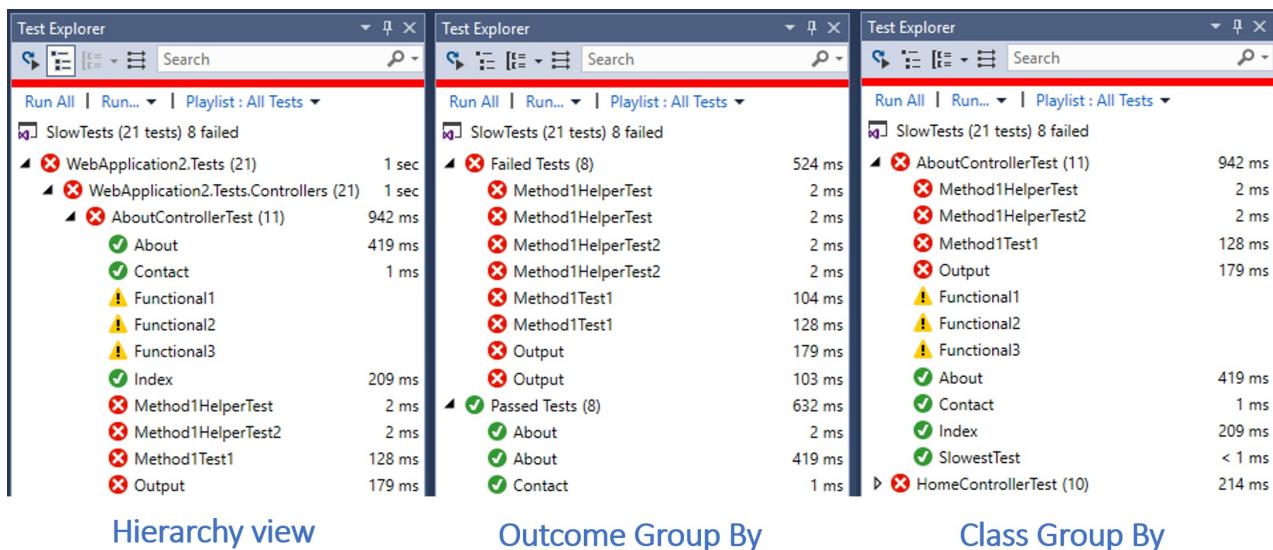
Why are my tests in UWP projects not discovered until I deploy my app?

UWP tests target a different runtime when the app is deployed. This means that to find tests accurately for UWP projects you not only need to build your project, but also deploy.

Test Explorer sorting

How does sorting test results work in the hierarchy view?

The hierarchy view sorts tests alphabetically as opposed to by outcome. The other group by settings normally sort test results by outcome and then alphabetically. See the different group by options in the following image for comparison. You can provide feedback about the design [in this GitHub issue](#).



Test Explorer hierarchy view

In the hierarchy view, there are passed, failed, skipped, and not run icons next to parent-node groupings. What do these icons mean?

The icons next to the Project, Namespace, and Class groupings show the state of the tests within that grouping. See the following table.

	if group contains failed tests
	if group contains passed tests and NO failed tests
	If group contains skipped tests and NO failed or passed tests
	if group contains not run tests and NO failed, passed, or skipped tests

Search by file path

There is no longer a "File Path" filter in the Test Explorer search box.

The file path filter in the **Test Explorer** search box was removed in Visual Studio 2017 version 15.7. This feature had low usage, and Test Explorer can retrieve test methods faster by leaving out this feature. If this change interrupts your development flow, let us know by submitting feedback on [Developer Community](#).

Remove undocumented interfaces

Some test-related APIs are no longer present in Visual Studio 2019. What changed?

In Visual Studio 2019, some test window APIs that were previously marked public but were never officially documented will be removed. They were marked as "deprecated" in Visual Studio 2017 to give extension maintainers an early warning. To our knowledge, very few extensions had found these APIs and taken a dependency on them. These include `IGroupByProvider`, `IGroupByProvider<T>`, `KeyComparer`, `ISearchFilter`, `ISearchFilterToken`, `ISearchToken`, and `SearchFilterTokenType`. If this change affects your extension, let us know by filing a bug on [Developer Community](#).

Test adapter NuGet reference

In Visual Studio 2017 version 15.8 my tests are discovered, but don't execute.

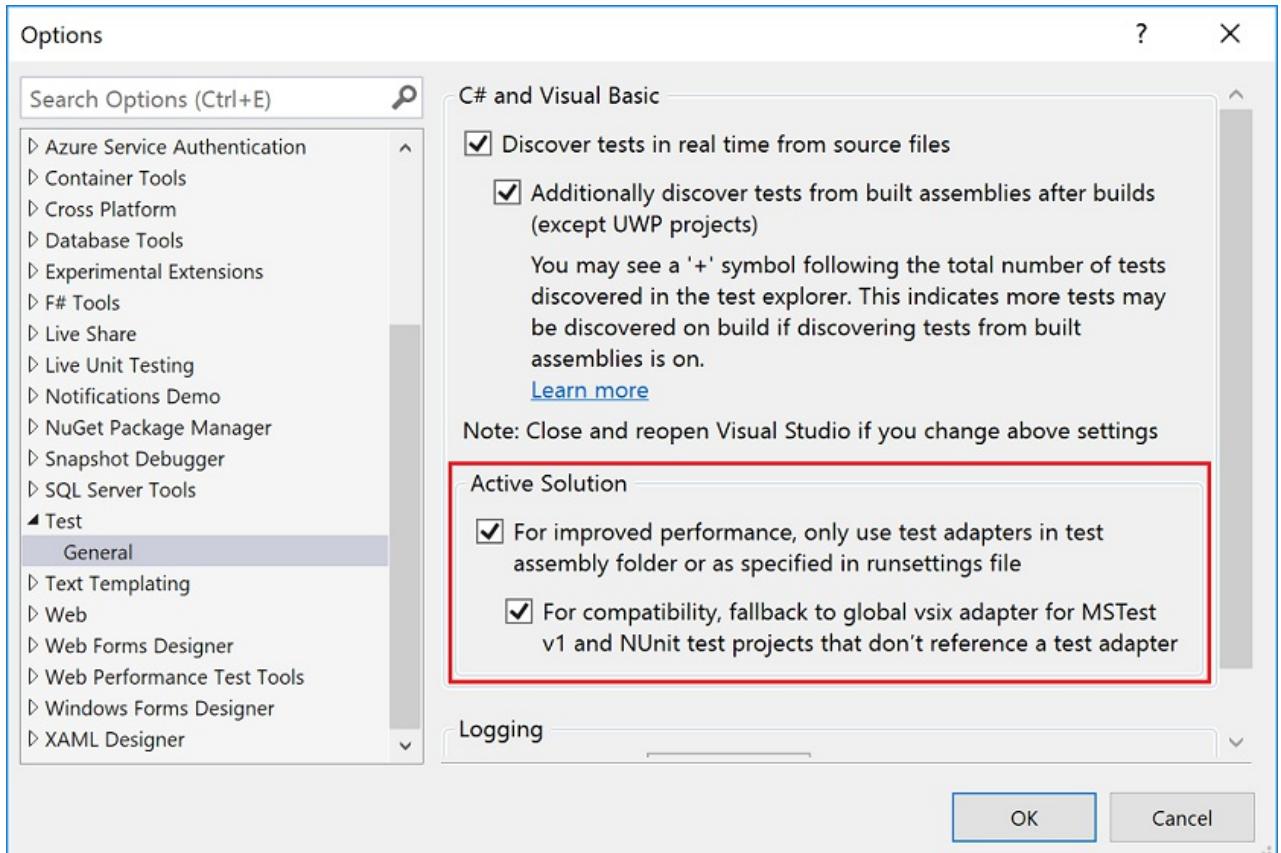
All test projects must include their .NET test adapter NuGet reference in their .csproj file. If they don't, the following test output appears on the project if discovery by a test adapter extension is kicked off after a build, or if the user tries to run the selected tests:

Test project {} does not reference any .NET NuGet adapter. Test discovery or execution might not work for this project. It is recommended to reference NuGet test adapters in each .NET test project in the solution.

Instead of using test adapter extensions, projects are required to use test adapter NuGet packages. This requirement greatly improves performance and causes fewer issues with continuous integration. Read more about .NET Test Adapter Extension deprecation in the [release notes](#).

NOTE

If you are using the NUnit 2 Test Adapter and are unable to migrate to the NUnit 3 test adapter, you can turn off this new discovery behavior in Visual Studio version 15.8 in **Tools > Options > Test**.



UWP TestContainer was not found

My UWP tests are no longer being executed in Visual Studio 2017 version 15.7 and later.

Recent UWP test projects specify a test platform build property that allows better performance for identifying test apps. If you have a UWP test project that was initialized before Visual Studio version 15.7, you may see this error in **Output > Tests**:

```
System.AggregateException: One or more errors occurred. ---> System.InvalidOperationException: The
following TestContainer was not found {} at
Microsoft.VisualStudio.TestTools.Controller.TestContainerProvider
<GetTestContainerAsync>d__61.MoveNext()
```

To fix this error:

- Update your test project build property using the following code:

```
<UnitTestPlatformVersion Condition="'$(UnitTestPlatformVersion)' == ''">$(VisualStudioVersion)
</UnitTestPlatformVersion>
```

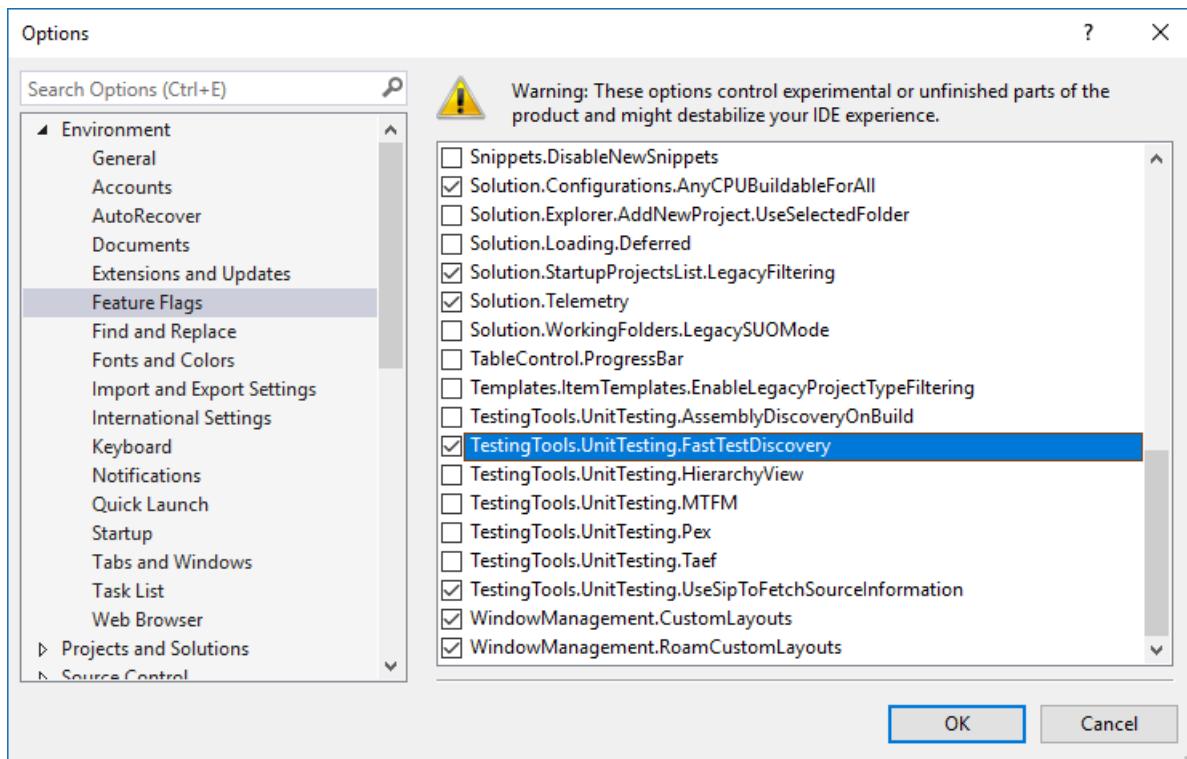
- Update the TestPlatform SDK version using the following code:

```
<SDKReference Include="TestPlatform.Universal, Version=$(UnitTestPlatformVersion)" />
```

Using feature flags

How can I turn on feature flags to try out new testing features?

Feature flags are used to ship experimental or unfinished parts of the product to avid users who would like to give feedback before the features ship officially. They may destabilize your IDE experience. Use them only in safe development environments, such as virtual machines. Feature flags are always use-at-your-own-risk settings. You can turn on experimental features with the [feature flags extension](#), or through the developer command prompt.



To turn on a feature flag through the Visual Studio developer command prompt, use the following command.

Change the path to where Visual Studio is installed on your machine, and change the registry key to the feature flag that you want.

```
vsregedit set "C:\Program Files (x86)\Microsoft Visual Studio\Preview\Enterprise" HKLM  
FeatureFlags\TestingTools\UnitTesting\HierarchyView Value dword 1
```

NOTE

You can turn off the flag with the same command, by using a value of 0 instead of 1 after dword.

See also

- [Microsoft.VisualStudio.TestTools.UnitTesting](#)
- [Create and run unit tests for existing code](#)
- [Unit test your code](#)
- [Live unit testing FAQ](#)

VSTest.Console.exe command-line options

1/13/2020 • 3 minutes to read • [Edit Online](#)

VSTest.Console.exe is the command-line tool to run tests. You can specify several options in any order on the command line. These options are listed in [General command-line options](#).

NOTE

The MSTest adapter in Visual Studio also works in legacy mode (equivalent to running tests with *mstest.exe*) for compatibility. In legacy mode, it can't take advantage of the *TestCaseFilter* feature. The adapter can switch to legacy mode when a *testsettings* file is specified, **forcelegacymode** is set to **true** in a *runsettings* file, or by using attributes such as **HostType**.

To run automated tests on an ARM architecture-based machine, you must use *VSTest.Console.exe*.

Open a [Developer Command Prompt](#) to use the command-line tool, or you can find the tool in `%Program Files(x86)%\Microsoft Visual Studio\<version>\<edition>\common7\ide\CommonExtensions\<Platform | Microsoft>`.

General command-line options

The following table lists all the options for *VSTest.Console.exe* and short descriptions of them. You can see a similar summary by typing `VSTest.Console/?` at a command line.

OPTION	DESCRIPTION
[test file names]	Run tests from the specified files. Separate multiple test file names with spaces. Examples: <code>mytestproject.dll</code> , <code>mytestproject.dll myothertestproject.exe</code>
/Settings:[file name]	Run tests with additional settings such as data collectors. Example: <code>/Settings:Local.RunSettings</code>
/Tests:[test name]	Run tests with names that contain the provided values. To provide multiple values, separate them by commas. Example: <code>/Tests:TestMethod1,TestMethod2</code> The /Tests command-line option cannot be used with the /TestCaseFilter command-line option.
/Parallel	Specifies that the tests be executed in parallel. By default, up to all available cores on the machine can be used. You can configure the number of cores to use in a settings file.
/Enablecodecoverage	Enables data diagnostic adapter CodeCoverage in the test run. Default settings are used if not specified using settings file.
/InIsolation	Runs the tests in an isolated process. This isolation makes the <i>vstest.console.exe</i> process less likely to be stopped on an error in the tests, but tests might run slower.
/UseVsixExtensions	This option makes the <i>vstest.console.exe</i> process use or skip the VSIX extensions installed (if any) in the test run. This option is deprecated. Starting from the next major release of Visual Studio this option may be removed. Move to consuming extensions made available as a NuGet package. Example: <code>/UseVsixExtensions:true</code>
/TestAdapterPath:[path]	Forces the <i>vstest.console.exe</i> process to use custom test adapters from a specified path (if any) in the test run. Example: <code>/TestAdapterPath:[pathToCustomAdapters]</code>
/Platform:[platform type]	Target platform architecture to be used for test execution. Valid values are x86, x64, and ARM.

OPTION	DESCRIPTION
/Framework: [framework version]	Target .NET version to be used for test execution. Example values are <code>Framework35</code> , <code>Framework40</code> , <code>Framework45</code> , <code>FrameworkUap10</code> , <code>.NETCoreApp, Version=v1.1</code> . If the target framework is specified as <code>Framework35</code> , the tests run in CLR 4.0 "compatibility mode". Example: <code>/Framework:framework40</code>
/TestCaseFilter:[expression]	Run tests that match the given expression. <Expression> is of the format <property>=<value> <Expression>. Example: <code>/TestCaseFilter:"Priority=1"</code> Example: <code>/TestCaseFilter:"TestCategory=Nightly FullyQualifiedName=Namespace.ClassName"</code> The <code>/TestCaseFilter</code> command-line option cannot be used with the <code>/Tests</code> command-line option. For information about creating and using expressions, see TestCase filter .
/?	Displays usage information.
/Logger:[uri/friendlyname]	Specify a logger for test results. Example: To log results into a Visual Studio Test Results File (TRX), use /Logger:trx [;LogFileName=<Defaults to unique file name>] Example: To publish test results to Team Foundation Server, use TfsPublisher: /logger:TfsPublisher; Collection=<project url>; BuildName=<build name>; TeamProject=<project name>; [;Platform=<Defaults to "Any CPU">] [;Flavor=<Defaults to "Debug">] [;RunTitle=<title>] Note: The TfsPublisher logger is deprecated in Visual Studio 2017 and is not supported in later versions of Visual Studio. For these scenarios, use a custom logger instead. This logger switches the logger to legacy mode.
/ListTests:[file name]	Lists discovered tests from the given test container.
/ListDiscoverers	Lists installed test discoverers.
/ListExecutors	Lists installed test executors.
/ListLoggers	Lists installed test loggers.
/ListSettingsProviders	Lists installed test settings providers.
/Blame	Tracks the tests as they're executing and, if the test host process crashes, emits the test names in their sequence of execution up to and including the specific test that was running at the time of the crash. This output makes it easier to isolate the offending test and diagnose further. More information .
/Diag:[file name]	Writes diagnostic trace logs to the specified file.
/ResultsDirectory:[path]	Test results directory will be created in specified path if not exists. Example: <code>/ResultsDirectory:<pathToResultsDirectory></code>
/ParentProcessId:[parentProcessId]	Process ID of the Parent Process responsible for launching current process.
/Port:[port]	The Port for socket connection and receiving the event messages.
/Collect:[dataCollector friendlyName]	Enables data collector for the test run. More information .

TIP

The options and values are not case-sensitive.

Examples

The syntax for running `VS Test.Console.exe` is:

```
Vstest.console.exe [TestFileNames] [Options]
```

The following command runs *VSTest.Console.exe* for the test library **myTestProject.dll**:

```
vstest.console.exe myTestProject.dll
```

The following command runs *VSTest.Console.exe* with multiple test files. Separate test file names with spaces:

```
Vstest.console.exe myTestFile.dll myOtherTestFile.dll
```

The following command runs *VSTest.Console.exe* with several options. It runs the tests in the *myTestFile.dll* file in an isolated process and uses settings specified in the *Local.RunSettings* file. Additionally, it only runs tests marked "Priority=1", and logs the results to a *.trx* file.

```
vstest.console.exe myTestFile.dll /Settings:Local.RunSettings /InIsolation /TestCaseFilter:"Priority=1" /Logger:trx
```

Run a unit test as a 64-bit process

1/1/2020 • 2 minutes to read • [Edit Online](#)

If you have a 64-bit machine, you can run unit tests and capture code coverage information as a 64-bit process.

To run a unit test as a 64-bit process

1. If your code or tests were compiled as 32-bit/x86, but you now want to run them as a 64-bit process, recompile them as **Any CPU**, or optionally as **64-bit**.

TIP

For maximum flexibility, compile your test projects with the **Any CPU** configuration. Then you can run on both 32-bit and 64-bit agents. There's no advantage to compiling test projects with the **64-bit** configuration.

2. From the Visual Studio menu, choose **Test**, then choose **Settings**, and then choose **Processor Architecture**. Choose **x64** to run the tests as a 64-bit process.

- or -

Specify `<TargetPlatform>x64</TargetPlatform>` in a `.runsettings` file. An advantage of this method is that you can specify groups of settings in different files and quickly switch between different settings. You can also copy settings between solutions. For more information, see [Configure unit tests by using a .runsettings file](#).

See also

- [Run unit tests with Test Explorer](#)
- [Unit test your code](#)

Configure unit tests by using a *.runsettings* file

1/1/2020 • 9 minutes to read • [Edit Online](#)

Unit tests in Visual Studio can be configured by using a *.runsettings* file. For example, you can change the .NET version on which the tests are run, the directory for the test results, or the data that's collected during a test run.

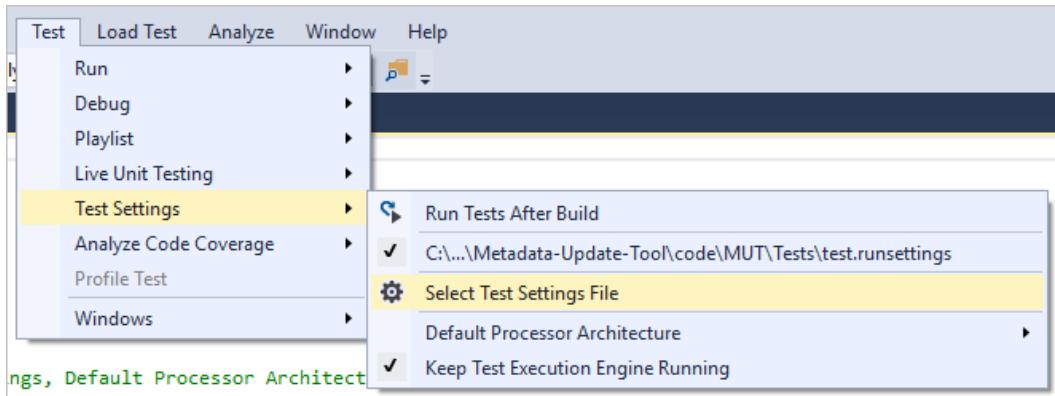
Run settings files are optional. If you don't require any special configuration, you don't need a *.runsettings* file. A common use of a *.runsettings* file is to customize [code coverage analysis](#).

Specify a run settings file

Run settings files can be used to configure tests that are run from the [command line](#), in the IDE, or in a [build workflow](#) using Azure Test Plans or Team Foundation Server (TFS).

IDE

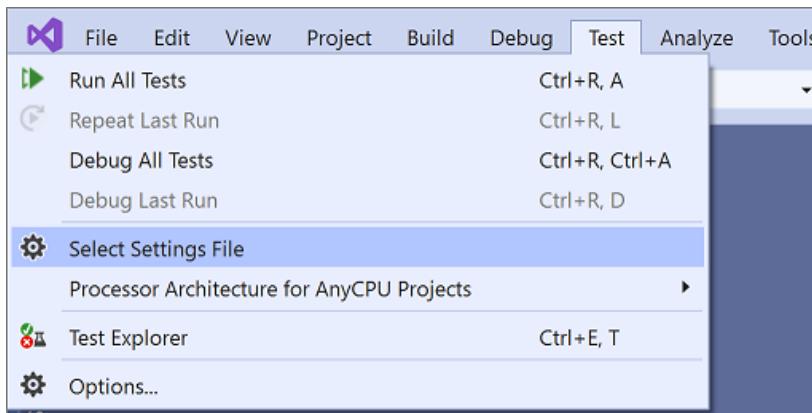
To specify a run settings file in the IDE, select **Test > Test Settings > Select Test Settings File**, and then select the *.runsettings* file.



The file appears on the Test Settings menu, and you can select or deselect it. While selected, the run settings file applies whenever you select **Analyze Code Coverage**.

Visual Studio 2019 version 16.3 and earlier

To specify a run settings file in the IDE, select **Test > Select Settings File**. Browse to and select the *.runsettings* file.



The file appears on the Test menu, and you can select or deselect it. While selected, the run settings file applies whenever you select **Analyze Code Coverage**.

Visual Studio 2019 version 16.4 and later

There are three ways of specifying a run settings file in Visual Studio 2019 version 16.4 and later:

- Add a build property to a project through either the project file or a Directory.Build.props file. The run settings file for a project is specified by the property **RunSettingsFilePath**.

- Project-level run settings is currently supported in C#, VB, C++, and F# projects.
- A file specified for a project overrides any other run settings file specified in the solution.

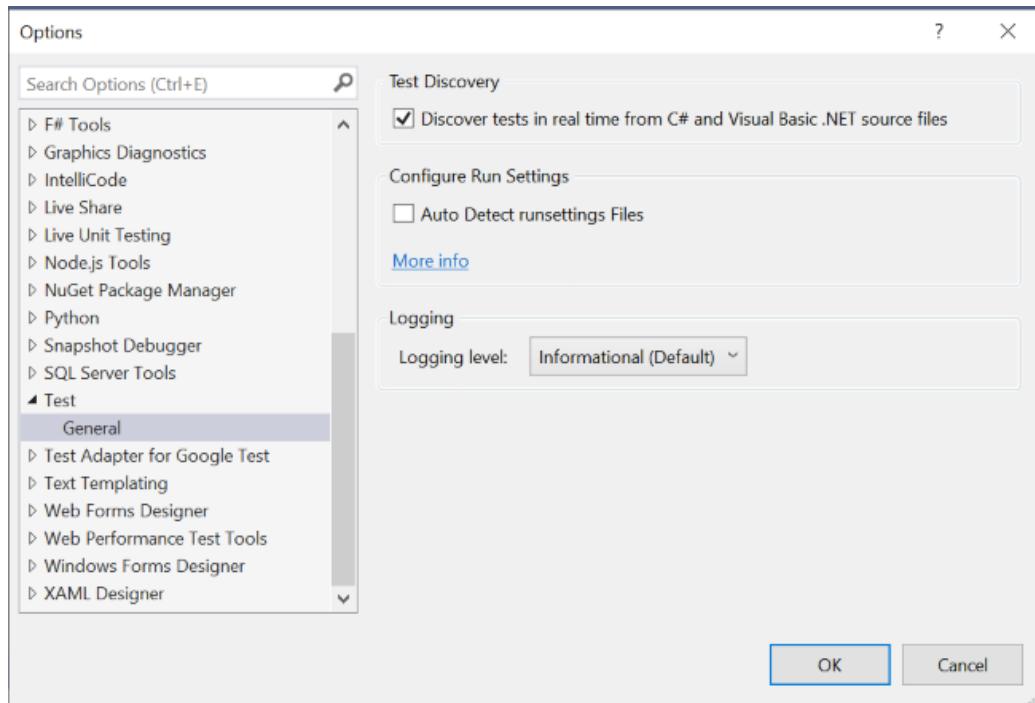
Example of specifying a *.runsettings* file for a project:

```
<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
    <RunSettingsFilePath>$(SolutionDir)\example.runsettings</RunSettingsFilePath>
</PropertyGroup>
...
</Project>
```

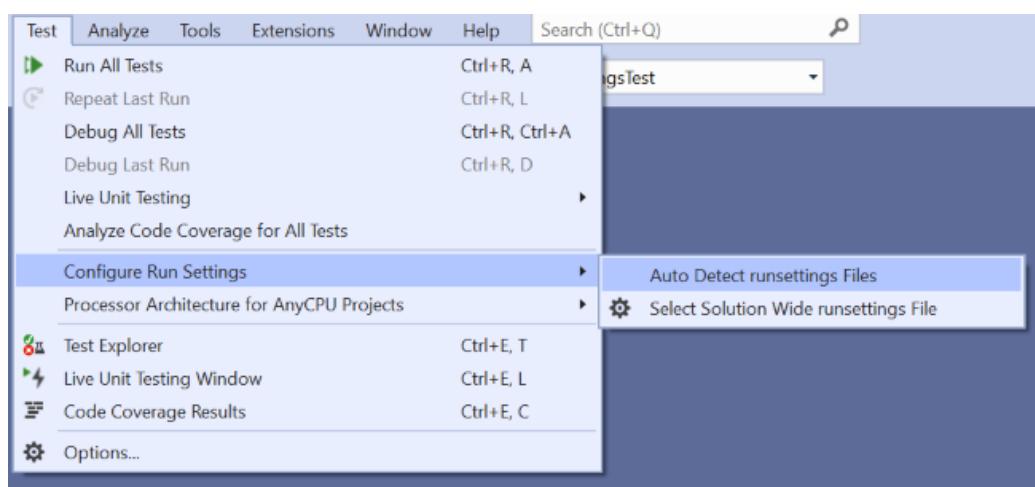
- Place a run settings file named ".runsettings" at the root of your solution.

If auto detection of run settings files is enabled, the settings in this file are applied across all tests run. You can turn on auto detection of runsettings files from two places:

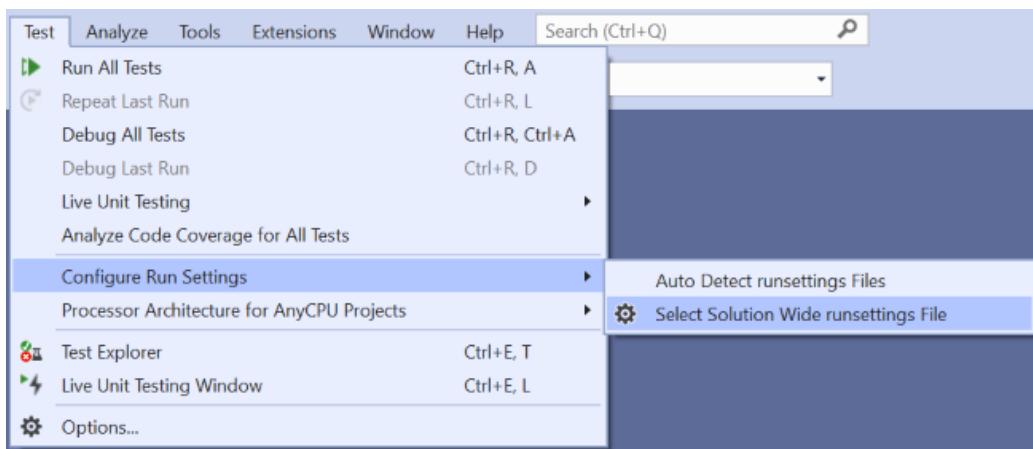
- **Tools > Options > Test > Auto Detect runsettings Files**



- **Test > Configure Run Settings > Auto Detect runsettings Files**



- In the IDE, select **Test > Configure Run Settings > Select Solution Wide runsettings File**, and then select the *.runsettings* file.



- This file overrides the ".runsettings" file at the root of the solution, if it exists, and is applied across all tests run.
- This file selection only persists locally.

Command line

To run tests from the command line, use `vstest.console.exe`, and specify the settings file by using the `/Settings` parameter.

- Launch the Visual Studio Developer Command Prompt:

On the Windows **Start** menu, choose **Visual Studio 2017 > Developer Command Prompt for VS 2017**.

On the Windows **Start** menu, choose **Visual Studio 2019 > Developer Command Prompt for VS 2019**.

- Enter a command similar to:

```
vstest.console.exe MyTestAssembly.dll /EnableCodeCoverage /Settings:CodeCoverage.runsettings
```

or

```
vstest.console.exe --settings:test.runsettings test.dll
```

For more information, see [VSTest.Console.exe command-line options](#).

Customize tests

To customize your tests using a *.runsettings* file, follow these steps:

- Add an XML file to your Visual Studio solution and save it as *test.runsettings*.

TIP

The file name doesn't matter, as long as you use the extension *.runsettings*.

- Replace the file contents with the XML from the example that follows, and customize it as needed.
- On the **Test** menu, choose **Test Settings > Select Test Settings File**. Browse to the *.runsettings* file you created, and then select **OK**.
- To select the run settings file, choose **Test > Select Settings File**. Browse to the *.runsettings* file you created,

and then select **OK**.

TIP

You can create more than one *.runsettings* file in your solution and select one as the active test settings file as needed.

Example *.runsettings* file

The following XML shows the contents of a typical *.runsettings* file. Each element of the file is optional because it has a default value.

```
<?xml version="1.0" encoding="utf-8"?>
<RunSettings>
    <!-- Configurations that affect the Test Framework -->
    <RunConfiguration>
        <MaxCpuCount>1</MaxCpuCount>
        <!-- Path relative to directory that contains .runsettings file-->
        <ResultsDirectory>.\TestResults</ResultsDirectory>

        <!-- x86 or x64 -->
        <!-- You can also change it from the Test menu; choose "Processor Architecture for AnyCPU Projects" -->
        <TargetPlatform>x86</TargetPlatform>

        <!-- Framework35 | [Framework40] | Framework45 -->
        <TargetFrameworkVersion>Framework40</TargetFrameworkVersion>

        <!-- Path to Test Adapters -->
        <TestAdaptersPaths>%SystemDrive%\Temp\foo;%SystemDrive%\Temp\bar</TestAdaptersPaths>

        <!-- TestSessionTimeout was introduced in Visual Studio 2017 version 15.5 -->
        <!-- Specify timeout in milliseconds. A valid value should be greater than 0 -->
        <TestSessionTimeout>10000</TestSessionTimeout>
    </RunConfiguration>

    <!-- Configurations for data collectors -->
    <DataCollectionRunSettings>
        <DataCollectors>
            <DataCollector friendlyName="Code Coverage" uri="datacollector://Microsoft/CodeCoverage/2.0"
                assemblyQualifiedName="Microsoft.VisualStudio.Coverage.DynamicCoverageDataCollector,
                Microsoft.VisualStudio.TraceCollector, Version=11.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a">
                <Configuration>
                    <CodeCoverage>
                        <ModulePaths>
                            <Exclude>
                                <ModulePath>.*CPPUnitTestFramework.*</ModulePath>
                            </Exclude>
                        </ModulePaths>

                        <!-- We recommend you do not change the following values: -->
                        <UseVerifiableInstrumentation>True</UseVerifiableInstrumentation>
                        <AllowLowIntegrityProcesses>True</AllowLowIntegrityProcesses>
                        <CollectFromChildProcesses>True</CollectFromChildProcesses>
                        <CollectAspDotNet>False</CollectAspDotNet>

                    </CodeCoverage>
                </Configuration>
            </DataCollector>
        </DataCollectors>
    </DataCollectionRunSettings>

```

```

only -->
    <MediaRecorder sendRecordedMediaForPassedTestCase="true" xmlns="">
        <ScreenCaptureVideo bitRate="512" frameRate="2" quality="20" />
    </MediaRecorder>
</Configuration>
</DataCollector>
</DataCollectors>
</DataCollectionRunSettings>

<!-- Parameters used by tests at run time -->
<TestRunParameters>
    <Parameter name="webAppUrl" value="http://localhost" />
    <Parameter name="webAppUserName" value="Admin" />
    <Parameter name="webAppPassword" value="Password" />
</TestRunParameters>

<!-- Adapter Specific sections -->

<!-- MSTest adapter -->
<MSTest>
    <MapInconclusiveToFailed>True</MapInconclusiveToFailed>
    <CaptureTraceOutput>false</CaptureTraceOutput>
    <DeleteDeploymentDirectoryAfterTestRunIsComplete>False</DeleteDeploymentDirectoryAfterTestRunIsComplete>
    <DeploymentEnabled>False</DeploymentEnabled>
    <AssemblyResolution>
        <Directory path="D:\myfolder\bin\" includeSubDirectories="false"/>
    </AssemblyResolution>
</MSTest>

</RunSettings>

```

Elements of a *.runsettings* file

The sections that follow detail the elements of a *.runsettings* file.

Run configuration

```

<RunConfiguration>
    <MaxCpuCount>1</MaxCpuCount>
    <ResultsDirectory>.\TestResults</ResultsDirectory>
    <TargetPlatform>x86</TargetPlatform>
    <TargetFrameworkVersion>Framework40</TargetFrameworkVersion>
    <TestAdaptersPaths>%SystemDrive%\Temp\foo;%SystemDrive%\Temp\bar</TestAdaptersPaths>
    <TestSessionTimeout>10000</TestSessionTimeout>
</RunConfiguration>

```

The **RunConfiguration** element can include the following elements:

NODE	DEFAULT	VALUES
ResultsDirectory		The directory where test results are placed.

NODE	DEFAULT	VALUES
TargetFrameworkVersion	Framework40	<p><code>FrameworkCore10</code> for .NET Core sources, <code>FrameworkUap10</code> for UWP-based sources, <code>Framework45</code> for .NET Framework 4.5 and higher, <code>Framework40</code> for .NET Framework 4.0, and <code>Framework35</code> for .NET Framework 3.5.</p> <p>This setting specifies the version of the unit test framework used to discover and execute the tests. It can be different from the version of the .NET platform that you specify in the build properties of the unit test project.</p> <p>If you omit the <code>TargetFrameworkVersion</code> element from the <code>.runsettings</code> file, the platform automatically determines the framework version based on the built binaries.</p>
TargetPlatform	x86	x86, x64
TreatTestAdapterErrorsAsWarnings	false	false, true
TestAdaptersPaths		One or more paths to the directory where the TestAdapters are located
MaxCpuCount	1	<p>This setting controls the degree of parallel test execution when running unit tests using available cores on the machine. The test execution engine starts as a distinct process on each available core, and gives each core a container with tests to run. A container can be an assembly, DLL, or relevant artifact. The test container is the scheduling unit. In each container, the tests are run according to the test framework. If there are many containers, then as processes finish executing the tests in a container, they're given the next available container.</p> <p>MaxCpuCount can be:</p> <p>n, where $1 \leq n \leq$ number of cores: up to n processes are launched</p> <p>n, where n = any other value: the number of processes launched can be up to the number of available cores</p>

Node	Default	Values
TestSessionTimeout		Allows users to terminate a test session when it exceeds a given timeout. Setting a timeout ensures that resources are well consumed and test sessions are constrained to a set time. The setting is available in Visual Studio 2017 version 15.5 and later.

Diagnostic data adapters (data collectors)

The **DataCollectors** element specifies settings of diagnostic data adapters. Diagnostic data adapters gather additional information about the environment and the application under test. Each adapter has default settings, and you only have to provide settings if you don't want to use the defaults.

Code coverage adapter

```
<CodeCoverage>
  <ModulePaths>
    <Exclude>
      <ModulePath>.*CPPUnitTestFramework.*</ModulePath>
    </Exclude>
  </ModulePaths>

  <UseVerifiableInstrumentation>True</UseVerifiableInstrumentation>
  <AllowLowIntegrityProcesses>True</AllowLowIntegrityProcesses>
  <CollectFromChildProcesses>True</CollectFromChildProcesses>
  <CollectAspNet>False</CollectAspNet>
</CodeCoverage>
```

The code coverage data collector creates a log of which parts of the application code have been exercised in the test. For more information about customizing the settings for code coverage, see [Customize code coverage analysis](#).

Video data collector

The video data collector captures a screen recording when tests are run. This recording is useful for troubleshooting UI tests. The video data collector is available in **Visual Studio 2017 version 15.5** and later.

To customize any other type of diagnostic data adapters, use a [test settings file](#).

TestRunParameters

```
<TestRunParameters>
  <Parameter name="webAppUrl" value="http://localhost" />
  <Parameter name="docsUrl" value="https://docs.microsoft.com" />
</TestRunParameters>
```

Test run parameters provide a way to define variables and values that are available to the tests at run time. Access the parameters using the [TestContext.Properties](#) property:

```
[TestMethod]
public void HomePageTest()
{
    string appURL = TestContext.Properties["webAppUrl"];
```

To use test run parameters, add a private [TestContext](#) field and a public [TestContext](#) property to your test class.

MSTest run settings

```

<MSTest>
  <MapInconclusiveToFailed>True</MapInconclusiveToFailed>
  <CaptureTraceOutput>false</CaptureTraceOutput>
  <DeleteDeploymentDirectoryAfterTestRunIsComplete>False</DeleteDeploymentDirectoryAfterTestRunIsComplete>
  <DeploymentEnabled>False</DeploymentEnabled>
  <AssemblyResolution>
    <Directory Path="D:\myfolder\bin\" includeSubDirectories="false"/>
  </AssemblyResolution>
</MSTest>

```

These settings are specific to the test adapter that runs test methods that have the [TestMethodAttribute](#) attribute.

CONFIGURATION	DEFAULT	VALUES
ForcedLegacyMode	false	<p>In Visual Studio 2012, the MSTest adapter was optimized to make it faster and more scalable. Some behavior, such as the order in which tests are run, might not be exactly as it was in previous editions of Visual Studio. Set this value to true to use the older test adapter.</p> <p>For example, you might use this setting if you have an <i>app.config</i> file specified for a unit test.</p> <p>We recommend that you consider refactoring your tests to allow you to use the newer adapter.</p>
IgnoreTestImpact	false	<p>The test impact feature prioritizes tests that are affected by recent changes, when run in MSTest or from Microsoft Test Manager. This setting deactivates the feature. For more information, see Which tests should be run since a previous build.</p>
SettingsFile		<p>You can specify a test settings file to use with the MSTest adapter here. You can also specify a test settings file from the settings menu.</p> <p>If you specify this value, you must also set the ForcedLegacyMode to true.</p> <div style="border: 1px solid black; padding: 2px; margin-top: 10px;"> <ForcedLegacyMode>true</ForcedLegacyMode> </div>
KeepExecutorAliveAfterLegacyRun	false	<p>After a test run is completed, MSTest is shut down. Any process that is launched as part of the test is also killed. If you want to keep the test executor alive, set the value to true. For example, you could use this setting to keep the browser running between coded UI tests.</p>

Configuration	Default	Values
DeploymentEnabled	true	If you set the value to false , deployment items that you've specified in your test method aren't copied to the deployment directory.
CaptureTraceOutput	true	You can write to the debug trace from your test method using Trace.WriteLine .
DeleteDeploymentDirectoryAfterTestRunsComplete	true	To retain the deployment directory after a test run, set this value to false .
MapInconclusiveToFailed	false	If a test completes with an inconclusive status, it is mapped to the skipped status in Test Explorer . If you want inconclusive tests to be shown as failed, set the value to true .
InProcMode	false	If you want your tests to be run in the same process as the MSTest adapter, set this value to true . This setting provides a minor performance gain. But if a test exits with an exception, the remaining tests don't run.
AssemblyResolution	false	<p>You can specify paths to additional assemblies when finding and running unit tests. For example, use these paths for dependency assemblies that aren't in the same directory as the test assembly. To specify a path, use a Directory Path element. Paths can include environment variables.</p> <pre><AssemblyResolution> <Directory Path="D:\myfolder\bin\" includeSubDirectories="false"/> </AssemblyResolution></pre>

See also

- [Configure a test run](#)
- [Customize code coverage analysis](#)
- [Visual Studio test task \(Azure Test Plans\)](#)

Walkthrough: Create and run unit tests for managed code

1/1/2020 • 10 minutes to read • [Edit Online](#)

This article steps you through creating, running, and customizing a series of unit tests using the Microsoft unit test framework for managed code and Visual Studio **Test Explorer**. You start with a C# project that is under development, create tests that exercise its code, run the tests, and examine the results. Then you change the project code and rerun the tests.

Create a project to test

1. Open Visual Studio.
 2. On the **File** menu, select **New > Project**.
- The **New Project** dialog box appears.
3. Under the **Visual C# > .NET Core** category, choose the **Console App (.NET Core)** project template.
 4. Name the project **Bank**, and then click **OK**.

The Bank project is created and displayed in **Solution Explorer** with the *Program.cs* file open in the code editor.

NOTE

If *Program.cs* is not open in the editor, double-click the file *Program.cs* in **Solution Explorer** to open it.

1. Open Visual Studio.
2. On the start window, choose **Create a new project**.
3. Search for and select the C# **Console App (.NET Core)** project template, and then click **Next**.
4. Name the project **Bank**, and then click **Create**.

The Bank project is created and displayed in **Solution Explorer** with the *Program.cs* file open in the code editor.

NOTE

If *Program.cs* is not open in the editor, double-click the file *Program.cs* in **Solution Explorer** to open it.

5. Replace the contents of *Program.cs* with the following C# code that defines a class, *BankAccount*:

```
using System;

namespace BankAccountNS
{
    /// <summary>
    /// Bank account demo class.
    /// </summary>
    public class BankAccount
    {
        private readonly string m_customerName;
        private double m_balance;

        private BankAccount() { }

        public BankAccount(string customerName, double balance)
        {
            m_customerName = customerName;
            m_balance = balance;
        }

        public string CustomerName
        {
            get { return m_customerName; }
        }

        public double Balance
        {
            get { return m_balance; }
        }

        public void Debit(double amount)
        {
            if (amount > m_balance)
            {
                throw new ArgumentOutOfRangeException("amount");
            }

            if (amount < 0)
            {
                throw new ArgumentOutOfRangeException("amount");
            }

            m_balance += amount; // intentionally incorrect code
        }

        public void Credit(double amount)
        {
            if (amount < 0)
            {
                throw new ArgumentOutOfRangeException("amount");
            }

            m_balance += amount;
        }

        public static void Main()
        {
            BankAccount ba = new BankAccount("Mr. Bryan Walton", 11.99);

            ba.Credit(5.77);
            ba.Debit(11.22);
            Console.WriteLine("Current balance is ${0}", ba.Balance);
        }
    }
}
```

6. Rename the file to `BankAccount.cs` by right-clicking and choosing **Rename** in **Solution Explorer**.

7. On the **Build** menu, click **Build Solution**.

You now have a project with methods you can test. In this article, the tests focus on the `Debit` method. The `Debit` method is called when money is withdrawn from an account.

Create a unit test project

1. On the **File** menu, select **Add > New Project**.

TIP

You can also right-click on the solution in **Solution Explorer** and choose **Add > New Project**.

2. In the **New Project** dialog box, expand **Installed**, expand **Visual C#**, and then choose **Test**.

3. From the list of templates, select **MSTest Test Project (.NET Core)**.

4. In the **Name** box, enter `BankTests`, and then select **OK**.

The **BankTests** project is added to the **Bank** solution.

2. Search for and select the C# **MSTest Test Project (.NET Core)** project template, and then click **Next**.

3. Name the project **BankTests**.

4. Click **Create**.

The **BankTests** project is added to the **Bank** solution.

5. In the **BankTests** project, add a reference to the **Bank** project.

In **Solution Explorer**, select **Dependencies** under the **BankTests** project and then choose **Add Reference** from the right-click menu.

6. In the **Reference Manager** dialog box, expand **Projects**, select **Solution**, and then check the **Bank** item.

7. Choose **OK**.

Create the test class

Create a test class to verify the `BankAccount` class. You can use the `UnitTest1.cs` file that was generated by the project template, but give the file and class more descriptive names.

Rename a file and class

1. To rename the file, in **Solution Explorer**, select the `UnitTest1.cs` file in the **BankTests** project. From the right-click menu, choose **Rename**, and then rename the file to `BankAccountTests.cs`.

2. To rename the class, choose **Yes** in the dialog box that pops up and asks whether you want to also rename references to the code element.

2. To rename the class, position the cursor on `UnitTest1` in the code editor, right-click, and then choose **Rename**. Type in **BankAccountTests** and then press **Enter**.

The `BankAccountTests.cs` file now contains the following code:

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

namespace BankTests
{
    [TestClass]
    public class BankAccountTests
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

Add a using statement

Add a `using` statement to the test class to be able to call into the project under test without using fully qualified names. At the top of the class file, add:

```
using BankAccountNS;
```

Test class requirements

The minimum requirements for a test class are:

- The `[TestClass]` attribute is required on any class that contains unit test methods that you want to run in Test Explorer.
- Each test method that you want Test Explorer to recognize must have the `[TestMethod]` attribute.

You can have other classes in a unit test project that do not have the `[TestClass]` attribute, and you can have other methods in test classes that do not have the `[TestMethod]` attribute. You can call these other classes and methods from your test methods.

Create the first test method

In this procedure, you'll write unit test methods to verify the behavior of the `Debit` method of the `BankAccount` class.

There are at least three behaviors that need to be checked:

- The method throws an `ArgumentOutOfRangeException` if the debit amount is greater than the balance.
- The method throws an `ArgumentOutOfRangeException` if the debit amount is less than zero.
- If the debit amount is valid, the method subtracts the debit amount from the account balance.

TIP

You can delete the default `TestMethod1` method, because you won't use it in this walkthrough.

To create a test method

The first test verifies that a valid amount (that is, one that is less than the account balance and greater than zero) withdraws the correct amount from the account. Add the following method to that `BankAccountTests` class:

```
[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = 4.55;
    double expected = 7.44;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // Act
    account.Debit(debitAmount);

    // Assert
    double actual = account.Balance;
    Assert.AreEqual(expected, actual, 0.001, "Account not debited correctly");
}
```

The method is straightforward: it sets up a new `BankAccount` object with a beginning balance and then withdraws a valid amount. It uses the `.AreEqual` method to verify that the ending balance is as expected.

Test method requirements

A test method must meet the following requirements:

- It's decorated with the `[TestMethod]` attribute.
- It returns `void`.
- It cannot have parameters.

Build and run the test

1. On the **Build** menu, choose **Build Solution**.
2. If **Test Explorer** is not open, open it by choosing **Test > Windows > Test Explorer** from the top menu bar.
3. Choose **Run All** to run the test.

While the test is running, the status bar at the top of the **Test Explorer** window is animated. At the end of the test run, the bar turns green if all the test methods pass, or red if any of the tests fail.

In this case, the test fails.

4. Select the method in **Test Explorer** to view the details at the bottom of the window.

Fix your code and rerun your tests

The test result contains a message that describes the failure. For the `.AreEqual` method, the message displays what was expected and what was actually received. You expected the balance to decrease, but instead it increased by the amount of the withdrawal.

The unit test has uncovered a bug: the amount of the withdrawal is *added* to the account balance when it should be *subtracted*.

Correct the bug

To correct the error, in the `BankAccount.cs` file, replace the line:

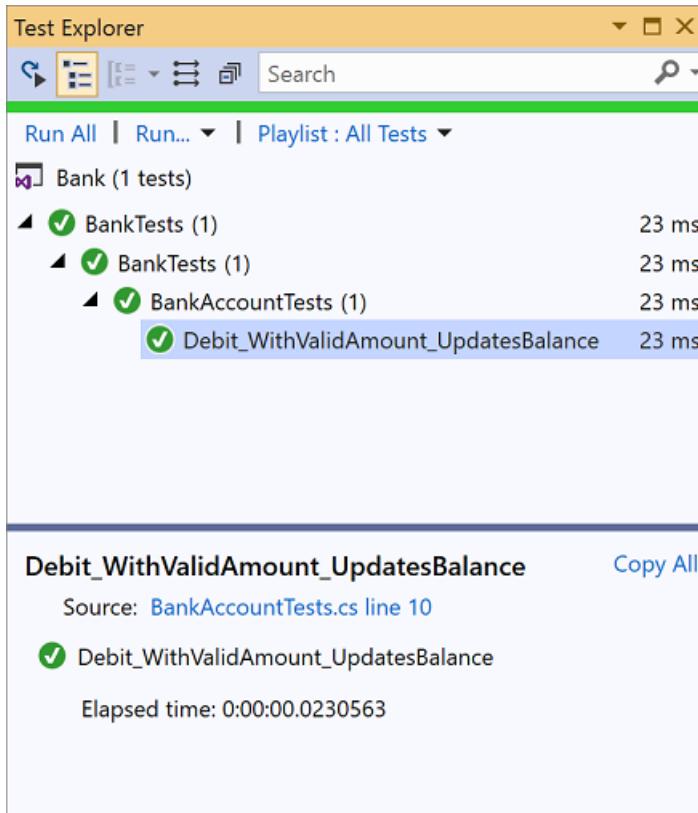
```
m_balance += amount;
```

with:

```
m_balance -= amount;
```

Rerun the test

In **Test Explorer**, choose **Run All** to rerun the test. The red/green bar turns green to indicate that the test passed.



Use unit tests to improve your code

This section describes how an iterative process of analysis, unit test development, and refactoring can help you make your production code more robust and effective.

Analyze the issues

You've created a test method to confirm that a valid amount is correctly deducted in the `Debit` method. Now, verify that the method throws an `ArgumentOutOfRangeException` if the debit amount is either:

- greater than the balance, or
- less than zero.

Create and run new test methods

Create a test method to verify correct behavior when the debit amount is less than zero:

```
[TestMethod]
public void Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRangeException()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = -100.00;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // Act and assert
    Assert.ThrowsException<System.ArgumentOutOfRangeException>(() => account.Debit(debitAmount));
}
```

Use the `ThrowsException` method to assert that the correct exception has been thrown. This method causes the

test to fail unless an [ArgumentOutOfRangeException](#) is thrown. If you temporarily modify the method under test to throw a more generic [ApplicationException](#) when the debit amount is less than zero, the test behaves correctly—that is, it fails.

To test the case when the amount withdrawn is greater than the balance, do the following steps:

1. Create a new test method named `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException`.
2. Copy the method body from `Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRangeException` to the new method.
3. Set the `debitAmount` to a number greater than the balance.

Running the two tests and verify that they pass.

Continue the analysis

The method being tested can be improved further. With the current implementation, we have no way to know which condition (`amount > m_balance` or `amount < 0`) led to the exception being thrown during the test. We just know that an [ArgumentOutOfRangeException](#) was thrown somewhere in the method. It would be better if we could tell which condition in `BankAccount.Debit` caused the exception to be thrown (`amount > m_balance` or `amount < 0`) so we can be confident that our method is sanity-checking its arguments correctly.

Look at the method being tested (`BankAccount.Debit`) again, and notice that both conditional statements use an [ArgumentOutOfRangeException](#) constructor that just takes name of the argument as a parameter:

```
throw new ArgumentOutOfRangeException("amount");
```

There is a constructor you can use that reports far richer information: [ArgumentOutOfRangeException\(String, Object, String\)](#) includes the name of the argument, the argument value, and a user-defined message. You can refactor the method under test to use this constructor. Even better, you can use publicly available type members to specify the errors.

Refactor the code under test

First, define two constants for the error messages at class scope. Put these in the class under test, `BankAccount`:

```
public const string DebitAmountExceedsBalanceMessage = "Debit amount exceeds balance";
public const string DebitAmountLessThanZeroMessage = "Debit amount is less than zero";
```

Then, modify the two conditional statements in the `Debit` method:

```
if (amount > m_balance)
{
    throw new System.ArgumentOutOfRangeException("amount", amount, DebitAmountExceedsBalanceMessage);
}

if (amount < 0)
{
    throw new System.ArgumentOutOfRangeException("amount", amount, DebitAmountLessThanZeroMessage);
}
```

Refactor the test methods

Refactor the test methods by removing the call to [Assert.ThrowsException](#). Wrap the call to `Debit()` in a `try/catch` block, catch the specific exception that's expected, and verify its associated message. The [StringAssert.Contains](#) method provides the ability to compare two strings.

Now, the `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException` might look like this:

```

[TestMethod]
public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = 20.0;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // Act
    try
    {
        account.Debit(debitAmount);
    }
    catch (System.ArgumentOutOfRangeException e)
    {
        // Assert
        StringAssert.Contains(e.Message, BankAccount.DebitAmountExceedsBalanceMessage);
    }
}

```

Retest, rewrite, and reanalyze

Assume there's a bug in the method under test and the `Debit` method doesn't even throw an `ArgumentOutOfRangeException` never mind output the correct message with the exception. Currently, the test method doesn't handle this case. If the `debitAmount` value is valid (that is, less than the balance and greater than zero), no exception is caught, so the assert never fires. Yet, the test method passes. This is not good, because you want the test method to fail if no exception is thrown.

This is a bug in the test method. To resolve the issue, add an `Fail` assert at the end of the test method to handle the case where no exception is thrown.

Rerunning the test shows that the test now *fails* if the correct exception is caught. The `catch` block catches the exception, but the method continues to execute and it fails at the new `Fail` assert. To resolve this problem, add a `return` statement after the `StringAssert` in the `catch` block. Rerunning the test confirms that you've fixed this problem. The final version of the `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException` looks like this:

```

[TestMethod]
public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = 20.0;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // Act
    try
    {
        account.Debit(debitAmount);
    }
    catch (System.ArgumentOutOfRangeException e)
    {
        // Assert
        StringAssert.Contains(e.Message, BankAccount.DebitAmountExceedsBalanceMessage);
        return;
    }

    Assert.Fail("The expected exception was not thrown.");
}

```

Conclusion

The improvements to the test code led to more robust and informative test methods. But more importantly, they

also improved the code under test.

TIP

This walkthrough uses the Microsoft unit test framework for managed code. **Test Explorer** can also run tests from third-party unit test frameworks that have adapters for **Test Explorer**. For more information, see [Install third-party unit test frameworks](#).

See also

For information about how to run tests from a command line, see [VSTest.Console.exe command-line options](#).

Walkthrough: Test-driven development using Test Explorer

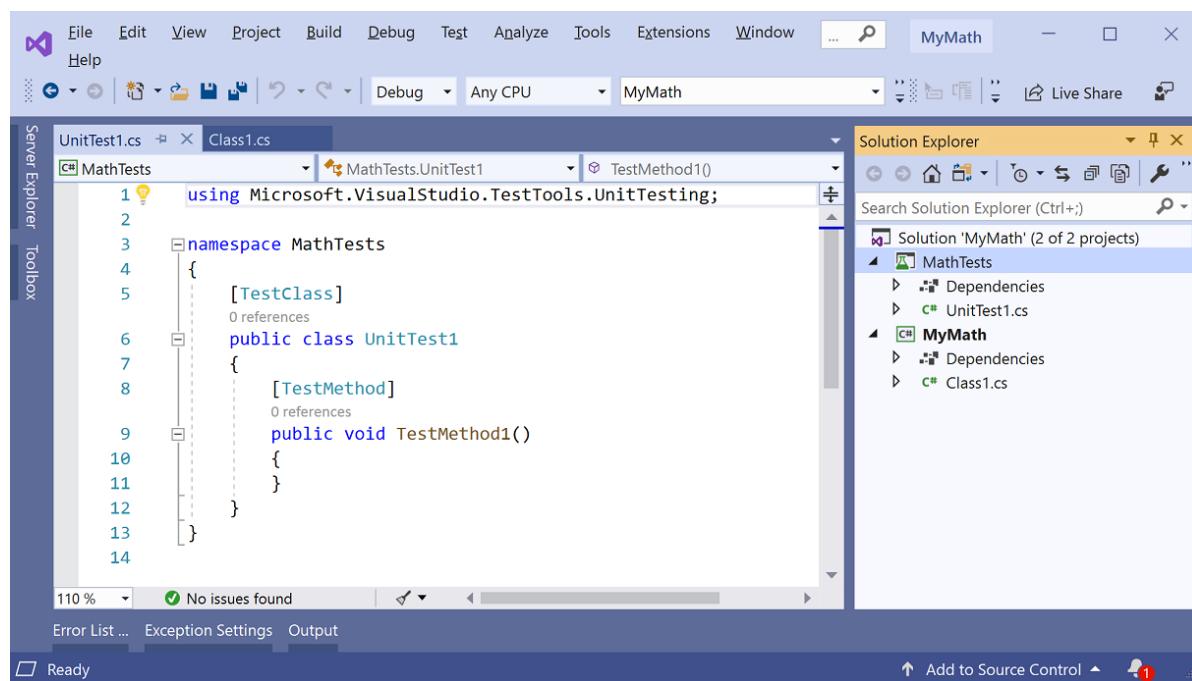
1/1/2020 • 4 minutes to read • [Edit Online](#)

Create unit tests to help keep your code working correctly through incremental code changes. There are several frameworks that you can use to write unit tests, including some developed by third parties. Some test frameworks are specialized for testing in different languages or platforms. Test Explorer provides a single interface for unit tests in any of these frameworks. For more information about **Test Explorer**, see [Run unit tests with Test Explorer](#) and [Test Explorer FAQ](#).

This walkthrough demonstrates how to develop a tested method in C# using Microsoft Test Framework (MS Test). You can easily adapt it for other languages or other test frameworks, such as NUnit. For more information, see [Install third-party unit test frameworks](#).

Create a test and generate code

1. Create a C# **Class Library (.NET Standard)** project. This project will contain the code that we want to test. Name the project **MyMath**.
2. In the same solution, add a new **MSTest Test Project (.NET Core)** project. Name the test project **MathTests**.



3. Write a simple test method that verifies the result obtained for a specific input. Add the following code to the `UnitTest1` class:

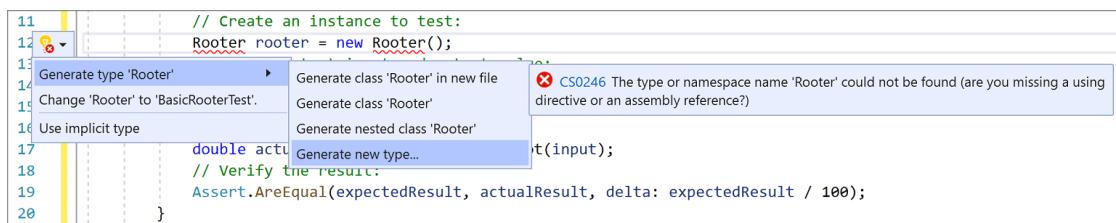
```

[TestMethod]
public void BasicRooterTest()
{
    // Create an instance to test:
    Rooter rooter = new Rooter();
    // Define a test input and output value:
    double expectedResult = 2.0;
    double input = expectedResult * expectedResult;
    // Run the method under test:
    double actualResult = rooter.SquareRoot(input);
    // Verify the result:
    Assert.AreEqual(expectedResult, actualResult, delta: expectedResult / 100);
}

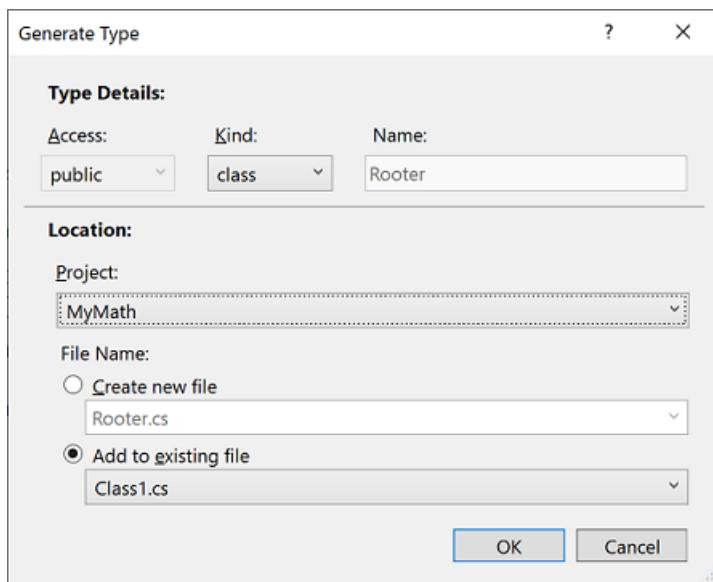
```

4. Generate a type from the test code.

- Place the cursor on `Rooter`, and then from the light bulb menu, choose **Generate type 'Rooter' > Generate new type**.



- In the **Generate Type** dialog box, set **Project** to **MyMath**, the class library project, and then choose **OK**.



5. Generate a method from the test code. Place the cursor on `SquareRoot`, and then from the light bulb menu, choose **Generate method 'Rooter.SquareRoot'**.

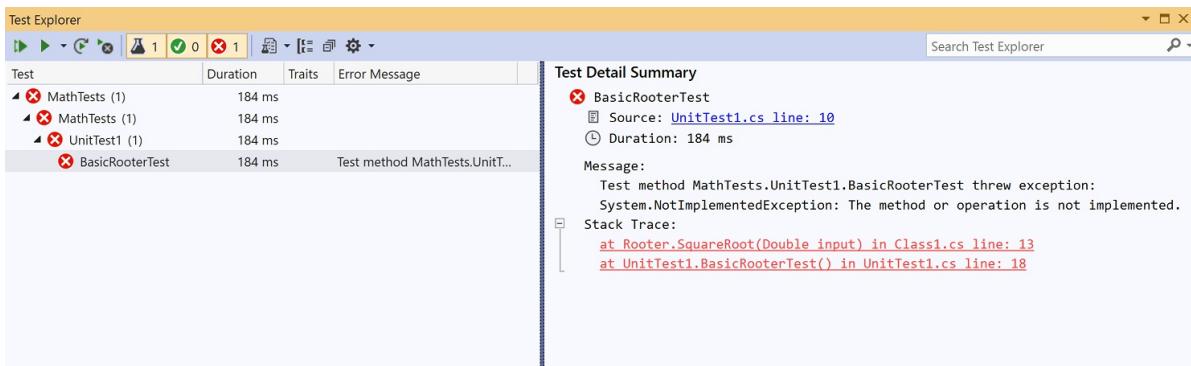
6. Run the unit test.

- To open **Test Explorer**, on the **Test** menu, choose **Windows > Test Explorer**.
- In **Test Explorer**, choose the **Run All** button to run the test.

The solution builds, and the test runs and fails.

7. Select the name of the test.

The details of the test appear in the **Test Detail Summary** pane.



8. Select the top link under **Stack Trace** to jump to the location where the test failed.

At this point, you've created a test and a stub that you can modify so that the test passes.

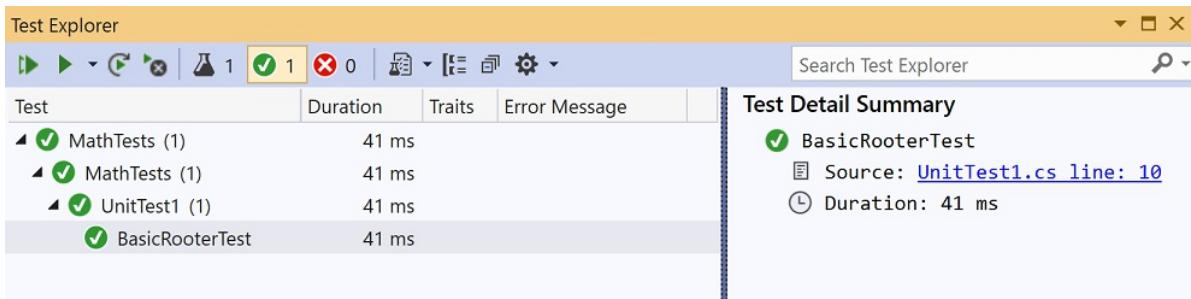
Verify a code change

1. In the *Class1.cs* file, improve the code of `SquareRoot`:

```
public double SquareRoot(double input)
{
    return input / 2;
}
```

2. In **Test Explorer**, choose **Run All**.

The solution builds, and the test runs and passes.



Extend the range of inputs

To improve our confidence that the code works in all cases, add tests that try a broader range of input values.

TIP

Avoid altering existing tests that pass. Instead, add new tests. Change existing tests only when the user requirements change. This policy helps to make sure that you don't lose existing functionality as you work to extend the code.

1. In the test class, add the following test, which tries a range of input values:

```

[TestMethod]
public void RooterValueRange()
{
    // Create an instance to test.
    Rooter rooter = new Rooter();

    // Try a range of values.
    for (double expected = 1e-8; expected < 1e+8; expected *= 3.2)
    {
        RooterOneValue(rooter, expected);
    }
}

private void RooterOneValue(Rooter rooter, double expectedResult)
{
    double input = expectedResult * expectedResult;
    double actualResult = rooter.SquareRoot(input);
    Assert.AreEqual(expectedResult, actualResult, delta: expectedResult / 1000);
}

```

2. In **Test Explorer**, choose **Run All**.

The new test fails (although the first test still passes). To find the point of failure, select the failing test, and then look at the details in the **Test Detail Summary** pane.

3. Inspect the method under test to see what might be wrong. Alter the `SquareRoot` code as follows:

```

public double SquareRoot(double input)
{
    double result = input;
    double previousResult = -input;
    while (Math.Abs(previousResult - result) > result / 1000)
    {
        previousResult = result;
        result = result - (result * result - input) / (2 * result);
    }
    return result;
}

```

4. In **Test Explorer**, choose **Run All**.

Both tests now pass.

Add tests for exceptional cases

1. Add a new test for negative inputs:

```

[TestMethod]
public void RooterTestNegativeInputx()
{
    Rooter rooter = new Rooter();
    try
    {
        rooter.SquareRoot(-10);
    }
    catch (System.ArgumentOutOfRangeException)
    {
        return;
    }
    Assert.Fail();
}

```

2. In **Test Explorer**, choose **Run All**.

The method under test loops and must be canceled manually.

3. Choose **Cancel** on the toolbar of **Test Explorer**.

The test stops executing.

4. Fix the `SquareRoot` code by adding the following `if` statement at the beginning of the method:

```
public double SquareRoot(double input)
{
    if (input <= 0.0)
    {
        throw new ArgumentOutOfRangeException();
    }
    ...
}
```

5. In **Test Explorer**, choose **Run All**.

All the tests pass.

Refactor the code under test

Refactor the code, but do not change the tests.

TIP

A *refactoring* is a change that is intended to make the code perform better or easier to understand. It is not intended to alter the behavior of the code, and therefore the tests are not changed.

We recommend that you perform refactoring steps separately from steps that extend functionality. Keeping the tests unchanged gives you confidence that you have not accidentally introduced bugs while refactoring.

1. Change the line that calculates `result` in the `SquareRoot` method as follows:

```
public double SquareRoot(double input)
{
    if (input <= 0.0)
    {
        throw new ArgumentOutOfRangeException();
    }

    double result = input;
    double previousResult = -input;
    while (Math.Abs(previousResult - result) > result / 1000)
    {
        previousResult = result;
        result = (result + input / result) / 2;
        //was: result = result - (result * result - input) / (2*result);
    }
    return result;
}
```

2. Choose **Run All**, and verify that all the tests still pass.

Test Explorer					
Test	Duration	Traits	Error Message		
▲ ✓ MathTests (3)	36 ms				
▲ ✓ MathTests (3)	36 ms				
▲ ✓ UnitTest1 (3)	36 ms				
✓ BasicRooterTest	36 ms				
✓ RooterTestNegativ...	< 1 ms				
✓ RooterValueRange	< 1 ms				

Use the MSTest framework in unit tests

1/1/2020 • 2 minutes to read • [Edit Online](#)

The [MSTest](#) framework supports unit testing in Visual Studio. Use the classes and members in the [Microsoft.VisualStudio.TestTools.UnitTesting](#) namespace when you are coding unit tests. You can also use them when you are refining a unit test that was generated from code.

Framework members

To help provide a clearer overview of the unit testing framework, this section organizes the members of the [Microsoft.VisualStudio.TestTools.UnitTesting](#) namespace into groups of related functionality.

NOTE

Attribute elements, whose names end with "Attribute", can be used either with or without "Attribute" on the end. For example, the following two code examples function identically:

```
[TestClass()]
```

```
[TestClassAttribute()]
```

Members used for data-driven testing

Use the following elements to set up data-driven unit tests. For more information, see [Create a data-driven unit test](#) and [Use a configuration file to define a data source](#).

- [DataAccessMethod](#)
- [DataSourceAttribute](#)
- [DataSourceElement](#)
- [DataSourceElementCollection](#)

Attributes used to establish a calling order

A code element decorated with one of the following attributes is called at the moment you specify. For more information, see [Anatomy of a unit test](#).

Attributes for assemblies

AssemblyInitialize and AssemblyCleanup are called right after your assembly is loaded and right before your assembly is unloaded.

- [AssemblyInitializeAttribute](#)
- [AssemblyCleanupAttribute](#)

Attributes for classes

ClassInitialize and ClassCleanup are called right after your class is loaded and right before your class is unloaded.

- [ClassInitializeAttribute](#)
- [ClassCleanupAttribute](#)

Attributes for test methods

- [TestInitializeAttribute](#)
- [TestCleanupAttribute](#)

Attributes used to identify test classes and methods

Every test class must have the `TestClass` attribute, and every test method must have the `TestMethod` attribute. For more information, see [Anatomy of a unit test](#).

- [TestClassAttribute](#)
- [TestMethodAttribute](#)

Assert classes and related exceptions

Unit tests can verify specific application behavior by their use of various kinds of assertions, exceptions, and attributes. For more information, see [Using the assert classes](#).

- [Assert](#)
- [Assert.ThrowsException](#)
- [CollectionAssert](#)
- [StringAssert](#)
- [AssertFailedException](#)
- [AssertInconclusiveException](#)
- [UnitTestAssertException](#)

The TestContext class

The following attributes and the values assigned to them appear in the Visual Studio Properties window for a particular test method. These attributes are not meant to be accessed through the code of the unit test. Instead, they affect the ways the unit test is used or run, either by you through the IDE of Visual Studio, or by the Visual Studio test engine. For example, some of these attributes appear as columns in the **Test Manager** window and **Test Results** window, which means that you can use them to group and sort tests and test results. One such attribute is [TestPropertyAttribute](#), which you use to add arbitrary metadata to unit tests. For example, you could use it to store the name of a test pass that this test covers, by marking the unit test with

`[TestProperty("TestPass", "Accessibility")]`. Or, you could use it to store an indicator of the kind of test it is with `[TestProperty("TestKind", "Localization")]`. The property you create by using this attribute, and the property value you assign, are both displayed in the Visual Studio **Properties** window under the heading **Test specific**.

- [OwnerAttribute](#)
- [DeploymentItemAttribute](#)
- [DescriptionAttribute](#)
- [IgnoreAttribute](#)
- [PriorityAttribute](#)
- [TestPropertyAttribute](#)
- [WorkItemAttribute](#)

Test configuration classes

- [ObjectTypes](#)
- [TestConfigurationSection](#)

Attributes used to generate reports

The attributes in this section relate the test method that they decorate to entities in the project hierarchy of a Team Foundation Server team project.

- [CssIterationAttribute](#)
- [CssProjectStructureAttribute](#)

Classes used with private accessors

You can generate a unit test for a private method. This generation creates a private accessor class, which instantiates an object of the [PrivateObject](#) class. The [PrivateObject](#) class is a wrapper class that uses reflection as part of the private accessor process. The [PrivateType](#) class is similar, but is used for calling private static methods instead of calling private instance methods.

- [PrivateObject](#)
- [PrivateType](#)

See also

- [Microsoft.VisualStudio.TestTools.UnitTesting](#) reference documentation

Use Assert classes for unit testing

1/1/2020 • 2 minutes to read • [Edit Online](#)

Use the Assert classes of the [Microsoft.VisualStudio.TestTools.UnitTesting](#) namespace to verify specific functionality. A unit test method exercises the code of a method in your application's code, but it reports the correctness of the code's behavior only if you include Assert statements.

Kinds of asserts

The [Microsoft.VisualStudio.TestTools.UnitTesting](#) namespace provides several kinds of Assert classes.

In your test method, you can call any methods of the [Microsoft.VisualStudio.TestTools.UnitTesting.Assert](#) class, such as [AreEqual](#). The [Assert](#) class has many methods to choose from, and many of the methods have several overloads.

Compare strings and collections

Use the [CollectionAssert](#) class to compare collections of objects, or to verify the state of a collection.

Use the [StringAssert](#) class to compare and examine strings. This class contains a variety of useful methods, such as [Contains](#), [StringAssert.Matches](#), and [StringAssert.StartsWith](#).

Exceptions

The [AssertFailedException](#) exception is thrown whenever a test fails. A test fails if it times out, throws an unexpected exception, or contains an assert statement that produces a **Failed** result.

The [AssertInconclusiveException](#) is thrown whenever a test produces a result of **Inconclusive**. Typically, you add an [Assert.Inconclusive](#) statement to a test that you are still working on, to indicate it's not yet ready to be run.

NOTE

An alternative strategy is to mark a test that is not ready to run with the [IgnoreAttribute](#) attribute. However, this has the disadvantage that you can't easily generate a report on the number of tests that aren't implemented.

If you write a new assert exception class, inherit from the base class [UnitTestAssertException](#) to make it easier to identify the exception as an assertion failure instead of an unexpected exception thrown from your test or production code.

To verify that an exception you expect to be thrown by a method in your application code is actually thrown, use the [Assert.ThrowsException](#) method.

See also

- [Unit test your code](#)

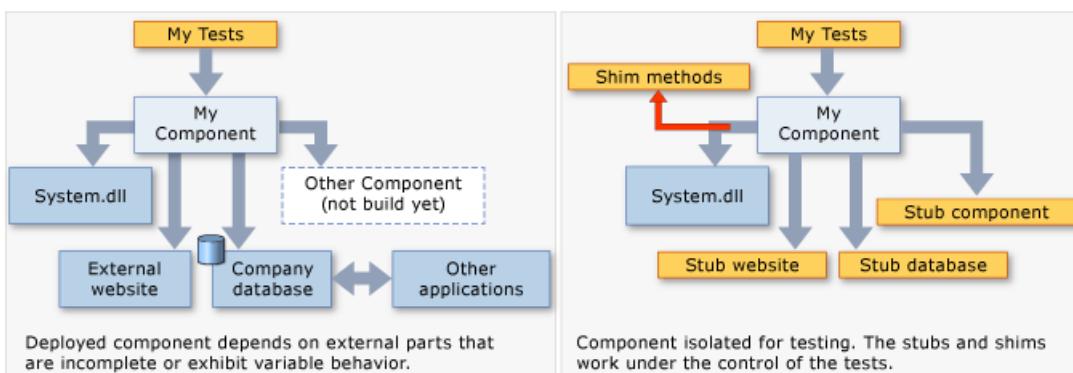
Isolate code under test with Microsoft Fakes

1/1/2020 • 7 minutes to read • [Edit Online](#)

Microsoft Fakes helps 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.

Fakes come in two flavors:

- A **stub** replaces a class with a small substitute that implements the same interface. To use stubs, you have to design your application so that each component depends only on interfaces, and not on other components. (By "component" we mean a class or group of classes that are designed and updated together and typically contained in an assembly.)
- A **shim** modifies the compiled code of your application at run time so that instead of making a specified method call, it runs the shim code that your test provides. Shims can be used to replace calls to assemblies that you cannot modify, such as .NET assemblies.



Requirements

- Visual Studio Enterprise
- A .NET Framework project

NOTE

- .NET Standard projects are not supported.
- Profiling with Visual Studio is not available for tests that use Microsoft Fakes.

Choose between stub and shim types

Typically, you would consider a Visual Studio project to be a component, because you develop and update those classes at the same time. You would consider using stubs and shims for calls that the project makes to other projects in your solution, or to other assemblies that the project references.

As a general guide, use stubs for calls within your Visual Studio solution, and shims for calls to other referenced assemblies. This is because within your own solution it is good practice to decouple the components by defining interfaces in the way that stubbing requires. But external assemblies such as *System.dll* typically are not provided with separate interface definitions, so you must use shims instead.

Other considerations are:

Performance. Shims run slower because they rewrite your code at run time. Stubs do not have this performance overhead and are as fast as virtual methods can go.

Static methods, sealed types. You can only use stubs to implement interfaces. Therefore, stub types cannot be used for static methods, non-virtual methods, sealed virtual methods, methods in sealed types, and so on.

Internal types. Both stubs and shims can be used with internal types that are made accessible by using the assembly attribute [InternalsVisibleToAttribute](#).

Private methods. Shims can replace calls to private methods if all the types on the method signature are visible. Stubs can only replace visible methods.

Interfaces and abstract methods. Stubs provide implementations of interfaces and abstract methods that can be used in testing. Shims can't instrument interfaces and abstract methods, because they don't have method bodies.

In general, we recommend that you use stub types to isolate from dependencies within your codebase. You can do this by hiding the components behind interfaces. Shim types can be used to isolate from third-party components that do not provide a testable API.

Get started with stubs

For a more detailed description, see [Use stubs to isolate parts of your application from each other for unit testing](#).

1. Inject interfaces

To use stubs, you have to write the code you want to test in such a way that it does not explicitly mention classes in another component of your application. By "component" we mean a class or classes that are developed and updated together, and typically contained in one Visual Studio project. Variables and parameters should be declared by using interfaces and instances of other components should be passed in or created by using a factory. For example, if StockFeed is a class in another component of the application, then this would be considered bad:

```
return (new StockFeed()).GetSharePrice("C000"); // Bad
```

Instead, define an interface that can be implemented by the other component, and which can also be implemented by a stub for test purposes:

```
public int GetContosoPrice(IStockFeed feed) => feed.GetSharePrice("C000");
```

```
Public Function GetContosoPrice(feed As IStockFeed) As Integer
    Return feed.GetSharePrice("C000")
End Function
```

2. Add Fakes Assembly

- a. In **Solution Explorer**, expand the test project's reference list. If you are working in Visual Basic, you must choose **Show All Files** in order to see the reference list.
 - b. Select the reference to the assembly in which the interface (for example `IStockFeed`) is defined. On the shortcut menu of this reference, choose **Add Fakes Assembly**.
 - c. Rebuild the solution.
3. In your tests, construct instances of the stub and provide code for its methods:

```

[TestClass]
class TestStockAnalyzer
{
    [TestMethod]
    public void TestContosoStockPrice()
    {
        // Arrange:

        // Create the fake stockFeed:
        IStockFeed stockFeed =
            new StockAnalysis.Fakes.StubIStockFeed() // Generated by Fakes.
        {
            // Define each method:
            // Name is original name + parameter types:
            GetSharePriceString = (company) => { return 1234; }
        };

        // In the completed application, stockFeed would be a real one:
        var componentUnderTest = new StockAnalyzer(stockFeed);

        // Act:
        int actualValue = componentUnderTest.GetContosoPrice();

        // Assert:
        Assert.AreEqual(1234, actualValue);
    }
    ...
}

```

```

<TestClass()> _
Class TestStockAnalyzer

<TestMethod()> _
Public Sub TestContosoStockPrice()
    ' Arrange:
    ' Create the fake stockFeed:
    Dim stockFeed As New StockAnalysis.Fakes.StubIStockFeed
    With stockFeed
        .GetSharePriceString = Function(company)
            Return 1234
        End Function
    End With
    ' In the completed application, stockFeed would be a real one:
    Dim componentUnderTest As New StockAnalyzer(stockFeed)
    ' Act:
    Dim actualValue As Integer = componentUnderTest.GetContosoPrice
    ' Assert:
    Assert.AreEqual(1234, actualValue)
End Sub
End Class

```

The special piece of magic here is the class `StubIStockFeed`. For every interface in the referenced assembly, the Microsoft Fakes mechanism generates a stub class. The name of the stub class is derived from the name of the interface, with "`Fakes.Stub`" as a prefix, and the parameter type names appended.

Stubs are also generated for the getters and setters of properties, for events, and for generic methods. For more information, see [Use stubs to isolate parts of your application from each other for unit testing](#).

Get started with shims

(For a more detailed description, see [Use shims to isolate your application from other assemblies for unit testing](#).)

Suppose your component contains calls to `DateTime.Now`:

```
// Code under test:  
public int GetTheCurrentYear()  
{  
    return DateTime.Now.Year;  
}
```

During testing, you would like to shim the `Now` property, because the real version inconveniently returns a different value at every call.

To use shims, you don't have to modify the application code or write it a particular way.

1. Add Fakes Assembly

In **Solution Explorer**, open your unit test project's references and select the reference to the assembly that contains the method you want to fake. In this example, the `DateTime` class is in *System.dll*. To see the references in a Visual Basic project, choose **Show All Files**.

Choose **Add Fakes Assembly**.

2. Insert a shim in a ShimsContext

```
[TestClass]  
public class TestClass1  
{  
    [TestMethod]  
    public void TestCurrentYear()  
    {  
        int fixedYear = 2000;  
  
        // Shims can be used only in a ShimsContext:  
        using (ShimsContext.Create())  
        {  
            // Arrange:  
            // Shim DateTime.Now to return a fixed date:  
            System.Fakes.ShimDateTime.NowGet =  
                () =>  
                { return new DateTime(fixedYear, 1, 1); };  
  
            // Instantiate the component under test:  
            var componentUnderTest = new MyComponent();  
  
            // Act:  
            int year = componentUnderTest.GetTheCurrentYear();  
  
            // Assert:  
            // This will always be true if the component is working:  
            Assert.AreEqual(fixedYear, year);  
        }  
    }  
}
```

```

<TestClass()> _
Public Class TestClass1
    <TestMethod()> _
    Public Sub TestCurrentYear()
        Using s = Microsoft.QualityTools.Testing.Fakes.ShimsContext.Create()
            Dim fixedYear As Integer = 2000
            ' Arrange:
            ' Detour DateTime.Now to return a fixed date:
            System.Fakes.ShimDateTime.NowGet = _
                Function() As DateTime
                    Return New DateTime(fixedYear, 1, 1)
                End Function

            ' Instantiate the component under test:
            Dim componentUnderTest = New MyComponent()
            ' Act:
            Dim year As Integer = componentUnderTest.GetTheCurrentYear
            ' Assert:
            ' This will always be true if the component is working:
            Assert.AreEqual(fixedYear, year)
        End Using
    End Sub
End Class

```

Shim class names are made up by prefixing `Fakes.Shim` to the original type name. Parameter names are appended to the method name. (You don't have to add any assembly reference to `System.Fakes`.)

The previous example uses a shim for a static method. To use a shim for an instance method, write `AllInstances` between the type name and the method name:

```
System.IO.Fakes.ShimFile.AllInstances.ReadToEnd = ...
```

(There is no '`System.IO.Fakes`' assembly to reference. The namespace is generated by the shim creation process. But you can use 'using' or 'Import' in the usual way.)

You can also create shims for specific instances, for constructors, and for properties. For more information, see [Use shims to isolate your application from other assemblies for unit testing](#).

In this section

[Use stubs to isolate parts of your application from each other for unit testing](#)

[Use shims to isolate your application from other assemblies for unit testing](#)

[Code generation, compilation, and naming conventions in Microsoft Fakes](#)

Use stubs to isolate parts of your application from each other for unit testing

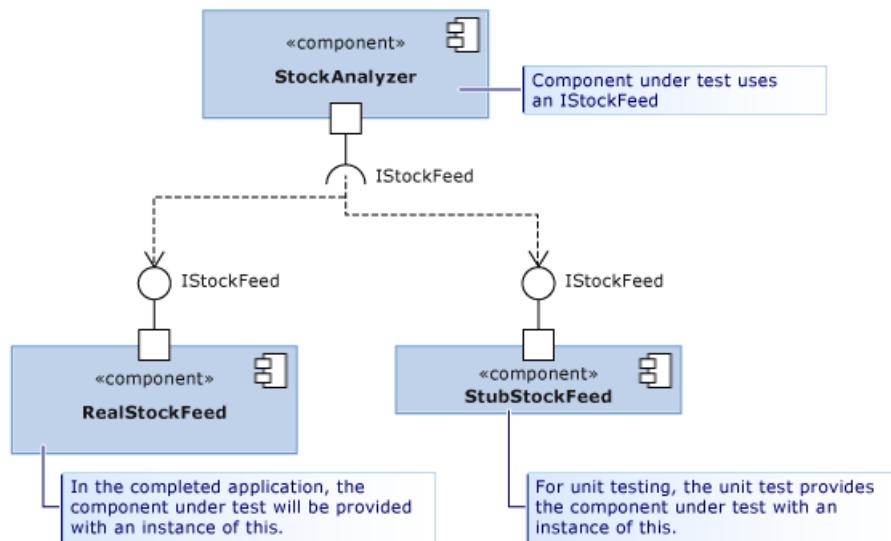
1/1/2020 • 11 minutes to read • [Edit Online](#)

Stub types are one of two technologies that the Microsoft Fakes framework provides to let you easily isolate a component you are testing from other components that it calls. A stub is a small piece of code that takes the place of another component during testing. The benefit of using a stub is that it returns consistent results, making the test easier to write. And you can run tests even if the other components are not working yet.

For an overview and quick start guide to Fakes, see [Isolate code under test with Microsoft Fakes](#).

To use stubs, you have to write your component so that it uses only interfaces, not classes, to refer to other parts of the application. This is a good design practice because it makes changes in one part less likely to require changes in another. For testing, it allows you to substitute a stub for a real component.

In the diagram, the component StockAnalyzer is the one we want to test. It normally uses another component, RealStockFeed. But RealStockFeed returns different results every time its methods are called, making it difficult to test StockAnalyzer. During testing, we replace it with a different class, StubStockFeed.



Because stubs rely on your being able to structure your code in this way, you typically use stubs to isolate one part of your application from another. To isolate it from other assemblies that are not under your control, such as `System.dll`, you would normally use shims. See [Use shims to isolate your application from other assemblies for unit testing](#).

How to use stubs

Design for dependency injection

To use stubs, your application has to be designed so that the different components are not dependent on each other, but only dependent on interface definitions. Instead of being coupled at compile time, components are connected at run time. This pattern helps to make software that is robust and easy to update, because changes tend not to propagate across component boundaries. We recommend following it even if you don't use stubs. If you are writing new code, it's easy to follow the [dependency injection](#) pattern. If you are writing tests for existing software, you might have to refactor it. If that would be impractical, you could consider using shims instead.

Let's start this discussion with a motivating example, the one in the diagram. The class StockAnalyzer reads share

prices and generates some interesting results. It has some public methods, which we want to test. To keep things simple, let's just look at one of those methods, a very simple one that reports the current price of a particular share. We want to write a unit test of that method. Here's the first draft of a test:

```
[TestMethod]
public void TestMethod1()
{
    // Arrange:
    var analyzer = new StockAnalyzer();
    // Act:
    var result = analyzer.GetContosoPrice();
    // Assert:
    Assert.AreEqual(123, result); // Why 123?
}
```

```
<TestMethod()> Public Sub TestMethod1()
    ' Arrange:
    Dim analyzer = New StockAnalyzer()
    ' Act:
    Dim result = analyzer.GetContosoPrice()
    ' Assert:
    Assert.AreEqual(123, result) ' Why 123?
End Sub
```

One problem with this test is immediately obvious: share prices vary, and so the assertion will usually fail.

Another problem might be that the StockFeed component, which is used by the StockAnalyzer, is still under development. Here's the first draft of the code of the method under test:

```
public int GetContosoPrice()
{
    var stockFeed = new StockFeed(); // NOT RECOMMENDED
    return stockFeed.GetSharePrice("C000");
}
```

```
Public Function GetContosoPrice()
    Dim stockFeed = New StockFeed() ' NOT RECOMMENDED
    Return stockFeed.GetSharePrice("C000")
End Function
```

As it stands, this method might not compile or might throw an exception because work on the StockFeed class is not yet complete. Interface injection addresses both of these problems. Interface injection applies the following rule:

The code of any component of your application should never explicitly refer to a class in another component, either in a declaration or in a `new` statement. Instead, variables and parameters should be declared with interfaces. Component instances should be created only by the component's container.

- By "component", we mean a class, or a group of classes that you develop and update together. Typically, a component is the code in one Visual Studio project. It's less important to decouple classes within one component, because they are updated at the same time.
- It is also not so important to decouple your components from the classes of a relatively stable platform such as `System.dll`. Writing interfaces for all these classes would clutter your code.

You can decouple the StockAnalyzer code from the StockFeed by using an interface like this:

```

public interface IStockFeed
{
    int GetSharePrice(string company);
}

public class StockAnalyzer
{
    private IStockFeed stockFeed;
    public StockAnalyzer(IStockFeed feed)
    {
        stockFeed = feed;
    }
    public int GetContosoPrice()
    {
        return stockFeed.GetSharePrice("C000");
    }
}

```

```

Public Interface IStockFeed
    Function GetSharePrice(company As String) As Integer
End Interface

Public Class StockAnalyzer
    ' StockAnalyzer can be connected to any IStockFeed:
    Private stockFeed As IStockFeed
    Public Sub New(feed As IStockFeed)
        stockFeed = feed
    End Sub
    Public Function GetContosoPrice()
        Return stockFeed.GetSharePrice("C000")
    End Function
End Class

```

In this example, StockAnalyzer is passed an implementation of an IStockFeed when it is constructed. In the completed application, the initialization code would perform the connection:

```
analyzer = new StockAnalyzer(new StockFeed());
```

There are more flexible ways of performing this connection. For example, StockAnalyzer could accept a factory object that can instantiate different implementations of IStockFeed in different conditions.

Generate stubs

You've decoupled the class you want to test from the other components that it uses. As well as making the application more robust and flexible, the decoupling allows you to connect the component under test to stub implementations of the interfaces for test purposes.

You could simply write the stubs as classes in the usual way. But Microsoft Fakes provides you with a more dynamic way to create the most appropriate stub for every test.

To use stubs, you must first generate stub types from the interface definitions.

Add a Fakes Assembly

1. In **Solution Explorer**, expand your unit test project's **References**.

If you're working in Visual Basic, select **Show All Files** in the **Solution Explorer** toolbar in order to see the **References** node.

2. Select the assembly that contains the interface definitions for which you want to create stubs.
3. On the shortcut menu, choose **Add Fakes Assembly**.

Write your test with stubs

```
[TestClass]
class TestStockAnalyzer
{
    [TestMethod]
    public void TestContosoStockPrice()
    {
        // Arrange:

        // Create the fake stockFeed:
        IStockFeed stockFeed =
            new StockAnalysis.Fakes.StubIStockFeed() // Generated by Fakes.
            {
                // Define each method:
                // Name is original name + parameter types:
                GetSharePriceString = (company) => { return 1234; }
            };

        // In the completed application, stockFeed would be a real one:
        var componentUnderTest = new StockAnalyzer(stockFeed);

        // Act:
        int actualValue = componentUnderTest.GetContosoPrice();

        // Assert:
        Assert.AreEqual(1234, actualValue);
    }

    ...
}
```

```
<TestClass()> _
Class TestStockAnalyzer

    <TestMethod()> _
    Public Sub TestContosoStockPrice()
        ' Arrange:
        ' Create the fake stockFeed:
        Dim stockFeed As New StockAnalysis.Fakes.StubIStockFeed
        With stockFeed
            .GetSharePriceString = Function(company)
                Return 1234
            End Function
        End With
        ' In the completed application, stockFeed would be a real one:
        Dim componentUnderTest As New StockAnalyzer(stockFeed)
        ' Act:
        Dim actualValue As Integer = componentUnderTest.GetContosoPrice
        ' Assert:
        Assert.AreEqual(1234, actualValue)
    End Sub
End Class
```

The special piece of magic here is the class `StubIStockFeed`. For every public type in the referenced assembly, the Microsoft Fakes mechanism generates a stub class. The name of the stub class is derived from the name of the interface, with "`Fakes.Stub`" as a prefix, and the parameter type names appended.

Stubs are also generated for the getters and setters of properties, for events, and for generic methods.

Verify parameter values

You can verify that when your component makes a call to another component, it passes the correct values. You can either place an assertion in the stub, or you can store the value and verify it in the main body of the test. For example:

```

[TestClass]
class TestMyComponent
{
    [TestMethod]
    public void TestVariableContosoPrice()
    {
        // Arrange:
        int priceToReturn = 345;
        string companyCodeUsed = "";
        var componentUnderTest = new StockAnalyzer(new StockAnalysis.Fakes.StubIStockFeed()
        {
            GetSharePriceString = (company) =>
            {
                // Store the parameter value:
                companyCodeUsed = company;
                // Return the value prescribed by this test:
                return priceToReturn;
            };
        });

        // Act:
        int actualResult = componentUnderTest.GetContosoPrice();

        // Assert:
        // Verify the correct result in the usual way:
        Assert.AreEqual(priceToReturn, actualResult);

        // Verify that the component made the correct call:
        Assert.AreEqual("C000", companyCodeUsed);
    }
}

...
}

```

```

<TestClass()> _
Class TestMyComponent
    <TestMethod()> _
    Public Sub TestVariableContosoPrice()
        ' Arrange:
        Dim priceToReturn As Integer = 345
        Dim companyCodeUsed As String = ""
        Dim stockFeed As New StockAnalysis.Fakes.StubIStockFeed()
        With stockFeed
            ' Implement the interface's method:
            .GetSharePriceString = _
                Function(company)
                    ' Store the parameter value:
                    companyCodeUsed = company
                    ' Return a fixed result:
                    Return priceToReturn
                End Function
        End With
        ' Create an object to test:
        Dim componentUnderTest As New StockAnalyzer(stockFeed)

        ' Act:
        Dim actualResult As Integer = componentUnderTest.GetContosoPrice()

        ' Assert:
        ' Verify the correct result in the usual way:
        Assert.AreEqual(priceToReturn, actualResult)
        ' Verify that the component made the correct call:
        Assert.AreEqual("C000", companyCodeUsed)
    End Sub

    ...
End Class

```

Stubs for different kinds of type members

Methods

As described in the example, methods can be stubbed by attaching a delegate to an instance of the stub class. The name of the stub type is derived from the names of the method and parameters. For example, given the following `IMyInterface` interface and method `MyMethod`:

```
// application under test
interface IMyInterface
{
    int MyMethod(string value);
}
```

We attach a stub to `MyMethod` that always returns 1:

```
// unit test code
var stub = new StubIMyInterface();
stub.MyMethodString = (value) => 1;
```

If you do not provide a stub for a function, Fakes generates a function that returns the default value of the return type. For numbers, the default value is 0 and for class types it is `null` (C#) or `Nothing` (Visual Basic).

Properties

Property getters and setters are exposed as separate delegates and can be stubbed separately. For example, consider the `Value` property of `IMyInterface`:

```
// code under test
interface IMyInterface
{
    int Value { get; set; }
}
```

We attach delegates to the getter and setter of `Value` to simulate an auto-property:

```
// unit test code
int i = 5;
var stub = new StubIMyInterface();
stub.ValueGet = () => i;
stub.ValueSet = (value) => i = value;
```

If you do not provide stub methods for either the setter or the getter of a property, Fakes generates a stub that stores values so that the stub property works like a simple variable.

Events

Events are exposed as delegate fields. As a result, any stubbed event can be raised simply by invoking the event backing field. Let's consider the following interface to stub:

```
// code under test
interface IWithEvents
{
    event EventHandler Changed;
}
```

To raise the `Changed` event, we simply invoke the backing delegate:

```
// unit test code
var withEvents = new StubIWithEvents();
// raising Changed
withEvents.ChangedEvent(withEvents, EventArgs.Empty);
```

Generic methods

It's possible to stub generic methods by providing a delegate for each desired instantiation of the method. For example, given the following interface containing a generic method:

```
// code under test
interface IGenericMethod
{
    T GetValue<T>();
}
```

You could write a test that stubs the `GetValue<int>` instantiation:

```
// unit test code
[TestMethod]
public void TestGetValue()
{
    var stub = new StubIGenericMethod();
    stub.GetValueOf1<int>(() => 5);

    IGenericMethod target = stub;
    Assert.AreEqual(5, target.GetValue<int>());
}
```

If the code were to call `GetValue<T>` with any other instantiation, the stub would simply call the behavior.

Stubs of virtual classes

In the previous examples, the stubs have been generated from interfaces. You can also generate stubs from a class that has virtual or abstract members. For example:

```
// Base class in application under test
public abstract class MyClass
{
    public abstract void DoAbstract(string x);
    public virtual int DoVirtual(int n)
    { return n + 42; }
    public int DoConcrete()
    { return 1; }
}
```

In the stub generated from this class, you can set delegate methods for `DoAbstract()` and `DoVirtual()`, but not `DoConcrete()`.

```
// unit test
var stub = new Fakes.MyClass();
stub.DoAbstractString = (x) => { Assert.IsTrue(x>0); };
stub.DoVirtualInt32 = (n) => 10 ;
```

If you do not provide a delegate for a virtual method, Fakes can either provide the default behavior, or it can call the method in the base class. To have the base method called, set the `CallBase` property:

```
// unit test code
var stub = new Fakes.MyClass();
stub.CallBase = false;
// No delegate set - default delegate:
Assert.AreEqual(0, stub.DoVirtual(1));

stub.CallBase = true;
// No delegate set - calls the base:
Assert.AreEqual(43, stub.DoVirtual(1));
```

Debug stubs

The stub types are designed to provide a smooth debugging experience. By default, the debugger is instructed to step over any generated code, so it should step directly into the custom member implementations that were attached to the stub.

Stub limitations

- Method signatures with pointers aren't supported.
- Sealed classes or static methods can't be stubbed because stub types rely on virtual method dispatch. For such cases, use shim types as described in [Use shims to isolate your application from other assemblies for unit testing](#)

Change the default behavior of stubs

Each generated stub type holds an instance of the `IStubBehavior` interface (through the `IStub.InstanceBehavior` property). The behavior is called whenever a client calls a member with no attached custom delegate. If the behavior has not been set, it uses the instance returned by the `StubsBehaviors.Current` property. By default, this property returns a behavior that throws a `NotImplementedException` exception.

The behavior can be changed at any time by setting the `InstanceBehavior` property on any stub instance. For example, the following snippet changes a behavior that does nothing or returns the default value of the return type: `default(T)`:

```
// unit test code
var stub = new StubIFileSystem();
// return default(T) or do nothing
stub.InstanceBehavior = StubsBehaviors.DefaultValue;
```

The behavior can also be changed globally for all stub objects for which the behavior has not been set by setting the `StubsBehaviors.Current` property:

```
// Change default behavior for all stub instances
// where the behavior has not been set.
StubBehaviors.Current = BehavedBehaviors.DefaultValue;
```

See also

- [Isolate code under test with Microsoft Fakes](#)

Use shims to isolate your app for unit testing

1/1/2020 • 12 minutes to read • [Edit Online](#)

Shim types are one of two technologies that the Microsoft Fakes Framework uses to let you isolate components under test from the environment. Shims divert calls to specific methods to code that you write as part of your test. Many methods return different results dependent on external conditions, but a shim is under the control of your test and can return consistent results at every call. This makes it easier to write the tests.

Use *shims* to isolate your code from assemblies that are not part of your solution. To isolate components of your solution from each other, use *stubs*.

For an overview and "quick start" guidance, see [Isolate code under test with Microsoft Fakes](#).

Requirements

- Visual Studio Enterprise
- A .NET Framework project

NOTE

.NET Standard projects are not supported.

Example: The Y2K bug

Consider a method that throws an exception on January 1st of 2000:

```
// code under test
public static class Y2KChecker {
    public static void Check() {
        if (DateTime.Now == new DateTime(2000, 1, 1))
            throw new ApplicationException("y2kbug!");
    }
}
```

Testing this method is problematic because the program depends on `DateTime.Now`, a method that depends on the computer's clock, an environment-dependent, non-deterministic method. Furthermore, the `DateTime.Now` is a static property so a stub type can't be used here. This problem is symptomatic of the isolation issue in unit testing: programs that directly call into database APIs, communicate with web services, and so on, are hard to unit test because their logic depends on the environment.

This is where shim types should be used. Shim types provide a mechanism to detour any .NET method to a user-defined delegate. Shim types are code-generated by the Fakes generator, and they use delegates, which we call shim types, to specify the new method implementations.

The following test shows how to use the shim type, `ShimDateTime`, to provide a custom implementation of `DateTime.Now`:

```
//unit test code
// create a ShimsContext cleans up shims
using (ShimsContext.Create()) {
    // hook delegate to the shim method to redirect DateTime.Now
    // to return January 1st of 2000
    ShimDateTime.NowGet = () => new DateTime(2000, 1, 1);
    Y2KChecker.Check();
}
```

How to use shims

First, add a Fakes assembly:

1. In **Solution Explorer**, expand your unit test project's **References** node.
 - If you're working in Visual Basic, select **Show All Files** in the **Solution Explorer** toolbar in order to see the **References** node.
2. Select the assembly that contains the class definitions for which you want to create shims. For example, if you want to shim **DateTime**, select **System.dll**.
3. On the shortcut menu, choose **Add Fakes Assembly**.

Use ShimsContext

When using shim types in a unit test framework, wrap the test code in a `ShimsContext` to control the lifetime of your shims. Otherwise, the shims would last until the AppDomain shut down. The easiest way to create a `ShimsContext` is by using the static `Create()` method as shown in the following code:

```
//unit test code
[Test]
public void Y2kCheckerTest() {
    using(ShimsContext.Create()) {
        ...
    } // clear all shims
}
```

It's critical to properly dispose each shim context. As a rule of thumb, call the `ShimsContext.Create` inside of a `using` statement to ensure proper clearing of the registered shims. For example, you might register a shim for a test method that replaces the `DateTime.Now` method with a delegate that always returns the first of January 2000. If you forget to clear the registered shim in the test method, the rest of the test run would always return the first of January 2000 as the `DateTime.Now` value. This might be surprising and confusing.

Write a test with shims

In your test code, insert a *detour* for the method you want to fake. For example:

```

[TestClass]
public class TestClass1
{
    [TestMethod]
    public void TestCurrentYear()
    {
        int fixedYear = 2000;

        using (ShimsContext.Create())
        {
            // Arrange:
            // Detour DateTime.Now to return a fixed date:
            System.Fakes.ShimDateTime.NowGet =
                () =>
                { return new DateTime(fixedYear, 1, 1); };

            // Instantiate the component under test:
            var componentUnderTest = new MyComponent();

            // Act:
            int year = componentUnderTest.GetTheCurrentYear();

            // Assert:
            // This will always be true if the component is working:
            Assert.AreEqual(fixedYear, year);
        }
    }
}

```

```

<TestClass()> _
Public Class TestClass1
    <TestMethod()> _
    Public Sub TestCurrentYear()
        Using s = Microsoft.QualityTools.Testing.Fakes.ShimsContext.Create()
            Dim fixedYear As Integer = 2000
            ' Arrange:
            ' Detour DateTime.Now to return a fixed date:
            System.Fakes.ShimDateTime.NowGet = _
                Function() As DateTime
                    Return New DateTime(fixedYear, 1, 1)
                End Function

            ' Instantiate the component under test:
            Dim componentUnderTest = New MyComponent()
            ' Act:
            Dim year As Integer = componentUnderTest.GetTheCurrentYear
            ' Assert:
            ' This will always be true if the component is working:
            Assert.AreEqual(fixedYear, year)
        End Using
    End Sub
End Class

```

Shim class names are made up by prefixing `Fakes.Shim` to the original type name.

Shims work by inserting *detours* into the code of the application under test. Wherever a call to the original method occurs, the Fakes system performs a detour, so that instead of calling the real method, your shim code is called.

Notice that detours are created and deleted at run time. You must always create a detour within the life of a `ShimsContext`. When it is disposed, any shims you created while it was active are removed. The best way to do this is inside a `using` statement.

You might see a build error stating that the Fakes namespace does not exist. This error sometimes appears when there are other compilation errors. Fix the other errors and it will vanish.

Shims for different kinds of methods

Shim types allow you to replace any .NET method, including static methods or non-virtual methods, with your own delegates.

Static methods

The properties to attach shims to static methods are placed in a shim type. Each property has only a setter that can be used to attach a delegate to the target method. For example, given a class `MyClass` with a static method `MyMethod`:

```
//code under test
public static class MyClass {
    public static int MyMethod() {
        ...
    }
}
```

We can attach a shim to `MyMethod` that always returns 5:

```
// unit test code
ShimMyClass.MyMethod = () => 5;
```

Instance methods (for all instances)

Similarly to static methods, instance methods can be shimmed for all instances. The properties to attach those shims are placed in a nested type named `AllInstances` to avoid confusion. For example, given a class `MyClass` with an instance method `MyMethod`:

```
// code under test
public class MyClass {
    public int MyMethod() {
        ...
    }
}
```

You can attach a shim to `MyMethod` that always returns 5, regardless of the instance:

```
// unit test code
ShimMyClass.AllInstances.MyMethod = () => 5;
```

The generated type structure of `ShimMyClass` looks like the following code:

```
// Fakes generated code
public class ShimMyClass : ShimBase<MyClass> {
    public static class AllInstances {
        public static Func<MyClass, int> MyMethod {
            set {
                ...
            }
        }
    }
}
```

Notice that Fakes passes the runtime instance as the first argument of the delegate in this case.

Instance methods (for one runtime instance)

Instance methods can also be shimmed by different delegates, based on the receiver of the call. This enables the same instance method to have different behaviors per instance of the type. The properties to set up those shims are instance methods of the shim type itself. Each instantiated shim type is also associated with a raw instance of a shimmed type.

For example, given a class `MyClass` with an instance method `MyMethod`:

```
// code under test
public class MyClass {
    public int MyMethod() {
        ...
    }
}
```

We can set up two shim types of `MyMethod` such that the first one always returns 5 and the second always returns 10:

```
// unit test code
var myClass1 = new ShimMyClass()
{
    MyMethod = () => 5
};
var myClass2 = new ShimMyClass { MyMethod = () => 10 };
```

The generated type structure of `ShimMyClass` looks like the following code:

```
// Fakes generated code
public class ShimMyClass : ShimBase<MyClass> {
    public Func<int> MyMethod {
        set {
            ...
        }
    }
    public MyClass Instance {
        get {
            ...
        }
    }
}
```

The actual shimmed type instance can be accessed through the `Instance` property:

```
// unit test code
var shim = new ShimMyClass();
var instance = shim.Instance;
```

The shim type also has an implicit conversion to the shimmed type, so you can usually simply use the shim type as is:

```
// unit test code
var shim = new ShimMyClass();
MyClass instance = shim; // implicit cast retrieves the runtime instance
```

Constructors

Constructors can also be shimmed in order to attach shim types to future objects. Each constructor is exposed as a static method `Constructor` in the shim type. For example, given a class `MyClass` with a constructor taking an integer:

```
// code under test
public class MyClass {
    public MyClass(int value) {
        this.Value = value;
    }
    ...
}
```

We set up the shim type of the constructor so that every future instance returns -5 when the `Value` getter is invoked, regardless of the value in the constructor:

```
// unit test code
ShimMyClass.ConstructorInt32 = (@this, value) => {
    var shim = new ShimMyClass(@this) {
        ValueGet = () => -5
    };
}
```

Each shim type exposes two constructors. The default constructor should be used when a fresh instance is needed, while the constructor taking a shimmed instance as argument should be used in constructor shims only:

```
// unit test code
public ShimMyClass() { }
public ShimMyClass(MyClass instance) : base(instance) { }
```

The generated type structure of `ShimMyClass` resembles the following code:

```
// Fakes generated code
public class ShimMyClass : ShimBase<MyClass>
{
    public static Action<MyClass, int> ConstructorInt32 {
        set {
            ...
        }
    }

    public ShimMyClass() { }
    public ShimMyClass(MyClass instance) : base(instance) { }
    ...
}
```

Base members

The shim properties of base members can be accessed by creating a shim for the base type and passing the child instance as a parameter to the constructor of the base shim class.

For example, given a class `MyBase` with an instance method `MyMethod` and a subtype `MyChild`:

```

public abstract class MyBase {
    public int MyMethod() {
        ...
    }
}

public class MyChild : MyBase {
}

```

We can set up a shim of `MyBase` by creating a new `ShimMyBase` shim:

```

// unit test code
var child = new ShimMyChild();
new Shim MyBase(child) { MyMethod = () => 5 };

```

Note that the child shim type is implicitly converted to the child instance when passed as a parameter to the base shim constructor.

The generated type structure of `ShimMyChild` and `Shim MyBase` resembles the following code:

```

// Fakes generated code
public class ShimMyChild : ShimBase<MyChild> {
    public ShimMyChild() { }
    public ShimMyChild(Child child)
        : base(child) { }
}

public class Shim MyBase : ShimBase<MyBase> {
    public Shim MyBase(Base target) { }
    public Func<int> MyMethod
    { set { ... } }
}

```

Static constructors

Shim types expose a static method `StaticConstructor` to shim the static constructor of a type. Since static constructors are executed once only, you need to make sure that the shim is configured before any member of the type is accessed.

Finalizers

Finalizers are not supported in Fakes.

Private methods

The Fakes code generator creates shim properties for private methods that only have visible types in the signature, that is, parameter types and return type visible.

Binding interfaces

When a shimmed type implements an interface, the code generator emits a method that allows it to bind all the members from that interface at once.

For example, given a class `MyClass` that implements `IEnumerable<int>`:

```

public class MyClass : IEnumerable<int> {
    public IEnumerator<int> GetEnumerator() {
        ...
    }
    ...
}

```

You can shim the implementations of `IEnumerable<int>` in `MyClass` by calling the `Bind` method:

```
// unit test code
var shimMyClass = new Shim MyClass();
shimMyClass.Bind(new List<int> { 1, 2, 3 });
```

The generated type structure of `Shim MyClass` resembles the following code:

```
// Fakes generated code
public class Shim MyClass : ShimBase<MyClass> {
    public Shim MyClass Bind(IEnumerable<int> target) {
        ...
    }
}
```

Change the default behavior

Each generated shim type holds an instance of the `IShimBehavior` interface, through the `ShimBase<T>.InstanceBehavior` property. The behavior is used whenever a client calls an instance member that was not explicitly shimmed.

If the behavior has not been explicitly set, it uses the instance returned by the static `ShimsBehaviors.Current` property. By default, this property returns a behavior that throws a `NotImplementedException` exception.

This behavior can be changed at any time by setting the `InstanceBehavior` property on any shim instance. For example, the following snippet changes the shim to a behavior that does nothing or returns the default value of the return type—that is, `default(T)`:

```
// unit test code
var shim = new Shim MyClass();
//return default(T) or do nothing
shim.InstanceBehavior = ShimsBehaviors.DefaultValue;
```

The behavior can also be changed globally for all shimmed instances for which the `InstanceBehavior` property was not explicitly set by setting the static `ShimsBehaviors.Current` property:

```
// unit test code
// change default shim for all shim instances
// where the behavior has not been set
ShimsBehaviors.Current = ShimsBehaviors.DefaultValue;
```

Detect environment accesses

It is possible to attach a behavior to all the members, including static methods, of a particular type by assigning the `ShimsBehaviors.NotImplemented` behavior to the static property `Behavior` of the corresponding shim type:

```
// unit test code
// assigning the not implemented behavior
Shim MyClass.Behavior = ShimsBehaviors.NotImplemented;
// shorthand
Shim MyClass.BehaveAsNotImplemented();
```

Concurrency

Shim types apply to all threads in the AppDomain and don't have thread affinity. This is an important fact if you plan to use a test runner that supports concurrency. Tests involving shim types cannot run concurrently. This property is not enforced by the Fakes runtime.

Call the original method from the shim method

Imagine that you want to write the text to the file system after validating the file name passed to the method. In that case, you'd call the original method in the middle of the shim method.

The first approach to solve this problem is to wrap a call to the original method using a delegate and `ShimsContext.ExecuteWithoutShims()`, as in the following code:

```
// unit test code
ShimFile.WriteAllTextStringString = (fileName, content) => {
    ShimsContext.ExecuteWithoutShims(() => {

        Console.WriteLine("enter");
        File.WriteAllText(fileName, content);
        Console.WriteLine("leave");
    });
};
```

Another approach is to set the shim to null, call the original method, and restore the shim.

```
// unit test code
ShimsDelegates.Action<string, string> shim = null;
shim = (fileName, content) => {
    try {
        Console.WriteLine("enter");
        // remove shim in order to call original method
        ShimFile.WriteAllTextStringString = null;
        File.WriteAllText(fileName, content);
    }
    finally
    {
        // restore shim
        ShimFile.WriteAllTextStringString = shim;
        Console.WriteLine("leave");
    }
};
// initialize the shim
ShimFile.WriteAllTextStringString = shim;
```

System.Environment

To shim [System.Environment](#), add the following content to the mscorlib.fakes file after the **Assembly** element:

```
<ShimGeneration>
    <Add FullName="System.Environment"/>
</ShimGeneration>
```

After you rebuild the solution, the methods and properties in the [System.Environment](#) class are available to be shimmed, for example:

```
System.Fakes.ShimEnvironment.GetCommandLineArgsGet = ...
```

Limitations

Shims cannot be used on all types from the .NET base class library **mscorlib** and **System**.

See also

- [Isolate code under test with Microsoft Fakes](#)
- [Peter Provost's blog: Visual Studio 2012 shims](#)
- [Video \(1h16\): Testing untestable code with fakes in Visual Studio 2012](#)

Code generation, compilation, and naming conventions in Microsoft Fakes

1/1/2020 • 7 minutes to read • [Edit Online](#)

This article discusses options and issues in Fakes code generation and compilation, and describes the naming conventions for Fakes generated types, members, and parameters.

Requirements

- Visual Studio Enterprise
- A .NET Framework project

NOTE

.NET Standard projects are not supported.

Code generation and compilation

Configure code generation of stubs

The generation of stub types is configured in an XML file that has the *.fakes* file extension. The Fakes framework integrates in the build process through custom MSBuild tasks and detects those files at build time. The Fakes code generator compiles the stub types into an assembly and adds the reference to the project.

The following example illustrates stub types defined in *FileSystem.dll*:

```
<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">
    <Assembly Name="FileSystem"/>
</Fakes>
```

Type filtering

Filters can be set in the *.fakes* file to restrict which types should be stubbed. You can add an unbounded number of Clear, Add, Remove elements under the StubGeneration element to build the list of selected types.

For example, the following *.fakes* file generates stubs for types under the System and System.IO namespaces, but excludes any type containing "Handle" in System:

```
<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">
    <Assembly Name="mscorlib" />
    <!-- user code -->
    <StubGeneration>
        <Clear />
        <Add Namespace="System!" />
        <Add Namespace="System.IO!" />
        <Remove TypeName="Handle" />
    </StubGeneration>
    <!-- /user code -->
</Fakes>
```

The filter strings use a simple grammar to define how the matching should be done:

- Filters are case-insensitive by default; filters perform a substring matching:

```
el matches "hello"
```

- Adding `!` to the end of the filter makes it a precise case-sensitive match:

```
el! does not match "hello"
```

```
hello! matches "hello"
```

- Adding `*` to the end of the filter makes it match the prefix of the string:

```
el* does not match "hello"
```

```
he* matches "hello"
```

- Multiple filters in a semicolon-separated list are combined as a disjunction:

```
el;wo matches "hello" and "world"
```

Stub concrete classes and virtual methods

By default, stub types are generated for all non-sealed classes. It is possible to restrict the stub types to abstract classes through the `.fakes` configuration file:

```
<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">
  <Assembly Name="mscorlib" />
  <!-- user code -->
  <StubGeneration>
    <Types>
      <Clear />
      <Add AbstractClasses="true"/>
    </Types>
  </StubGeneration>
  <!-- /user code -->
</Fakes>
```

Internal types

The Fakes code generator generates shim types and stub types for types that are visible to the generated Fakes assembly. To make internal types of a shimmed assembly visible to Fakes and your test assembly, add `InternalsVisibleToAttribute` attributes to the shimmed assembly code that gives visibility to the generated Fakes assembly and to the test assembly. Here's an example:

```
// FileSystem\AssemblyInfo.cs
[assembly: InternalsVisibleTo("FileSystem.Fakes")]
[assembly: InternalsVisibleTo("FileSystem.Tests")]
```

Internal types in strongly named assemblies

If the shimmed assembly is strongly named, and you want to access internal types of the assembly:

- Both your test assembly and the Fakes assembly must be strongly named.
- Add the public keys of the test and Fakes assembly to the `InternalsVisibleToAttribute` attributes in the shimmed assemblies. Here's how the example attributes in the shimmed assembly code would look when the shimmed assembly is strongly named:

```
// FileSystem\AssemblyInfo.cs
[assembly: InternalsVisibleTo("FileSystem.Fakes",
    PublicKey=<Fakes_assembly_public_key>)]
[assembly: InternalsVisibleTo("FileSystem.Tests",
    PublicKey=<Test_assembly_public_key>)]
```

If the shimmed assembly is strongly named, the Fakes framework automatically strongly signs the generated Fakes assembly. You have to strong sign the test assembly. See [Strong-Named assemblies](#).

The Fakes framework uses the same key to sign all generated assemblies, so you can use this snippet as a starting point to add the **InternalsVisibleTo** attribute for the fakes assembly to your shimmed assembly code.

```
[assembly: InternalsVisibleTo("FileSystem.Fakes",
    PublicKey=0024000048000009400000006020000024000052534131000400001000100e92decb949446f688ab9f6973436c535bf50
    acd1fd580495aae3f875aa4e4f663ca77908c63b7f0996977cb98fcfdb35e05aa2c842002703cad835473caac5ef14107e3a7fae01120a
    96558785f48319f66daabc862872b2c53f5ac11fa335c0165e202b4c011334c7bc8f4c4e570cf255190f4e3e2cbc9137ca57cb687947bc
    ")]
```

You can specify a different public key for the Fakes assembly, such as a key you have created for the shimmed assembly, by specifying the full path to the *.snk* file that contains the alternate key as the **KeyFile** attribute value in the **Fakes \ Compilation** element of the *.fakes* file. For example:

```
<-- FileSystem.Fakes.fakes -->
<Fakes ...>
  <Compilation KeyFile="full_path_to_the_alternate_snk_file" />
</Fakes>
```

You then have to use the public key of the alternate *.snk* file as the second parameter of the **InternalVisibleTo** attribute for the Fakes assembly in the shimmed assembly code:

```
// FileSystem\AssemblyInfo.cs
[assembly: InternalsVisibleTo("FileSystem.Fakes",
    PublicKey=<Alternate_public_key>)]
[assembly: InternalsVisibleTo("FileSystem.Tests",
    PublicKey=<Test_assembly_public_key>)]
```

In the example above, the values **Alternate_public_key** and the **Test_assembly_public_key** can be the same.

Optimize build times

The compilation of Fakes assemblies can significantly increase your build time. You can minimize the build time by generating the Fakes assemblies for .NET System assemblies and third-party assemblies in a separate centralized project. Because such assemblies rarely change on your machine, you can reuse the generated Fakes assemblies in other projects.

From your unit test projects, add a reference to the compiled Fakes assemblies that are placed under **FakesAssemblies** in the project folder.

1. Create a new Class Library with the .NET runtime version matching your test projects. Let's call it **Fakes.Prebuild**. Remove the *class1.cs* file from the project, not needed.
2. Add reference to all the System and third-party assemblies you need Fakes for.
3. Add a *.fakes* file for each of the assemblies and build.
4. From your test project
 - Make sure that you have a reference to the Fakes runtime DLL:

%ProgramFiles(x86)%\Microsoft Visual Studio\2017\Enterprise\Common7\IDE\PublicAssemblies\Microsoft.QualityTools.Testing.Fakes.dll

- For each assembly that you have created Fakes for, add a reference to the corresponding DLL file in the *Fakes.Prebuild\FakesAssemblies* folder of your project.

Avoid assembly name clashing

In a Team Build environment, all build outputs are merged into a single directory. If multiple projects use Fakes, it might happen that Fakes assemblies from different versions override each other. For example, TestProject1 fakes *mscorlib.dll* from the .NET Framework 2.0 and TestProject2 fakes *mscorlib.dll* for the .NET Framework 4 would both yield to a *mscorlib.Fakes.dll* Fakes assembly.

To avoid this issue, Fakes should automatically create version qualified Fakes assembly names for non-project references when adding the *.fakes* files. A version-qualified Fakes assembly name embeds a version number when you create the Fakes assembly name:

Given an assembly MyAssembly and a version 1.2.3.4, the Fakes assembly name is MyAssembly.1.2.3.4.Fakes.

You can change or remove this version by the editing the Version attribute of the Assembly element in the *.fakes*:

```
attribute of the Assembly element in the .fakes:  
<Fakes ...>  
  <Assembly Name="MyAssembly" Version="1.2.3.4" />  
  ...  
</Fakes>
```

Fakes naming conventions

Shim type and stub type naming conventions

Namespaces

- .Fakes suffix is added to the namespace.

For example, `System.Fakes` namespace contains the shim types of System namespace.

- Global.Fakes contains the shim type of the empty namespace.

Type names

- Shim prefix is added to the type name to build the shim type name.

For example, ShimExample is the shim type of the Example type.

- Stub prefix is added to the type name to build the stub type name.

For example, StubIExample is the stub type of the IExample type.

Type Arguments and Nested Type Structures

- Generic type arguments are copied.
- Nested type structure is copied for shim types.

Shim delegate property or stub delegate field naming conventions

Basic rules for field naming, starting from an empty name:

- The method name is appended.
- If the method name is an explicit interface implementation, the dots are removed.

- If the method is generic, `[of] n` is appended where `n` is the number of generic method arguments.

Special method names such as property getter or setters are treated as described in the following table:

IF METHOD IS...	EXAMPLE	METHOD NAME APPENDED
A constructor	<code>.ctor</code>	<code>Constructor</code>
A static constructor	<code>.cctor</code>	<code>StaticConstructor</code>
An accessor with method name composed of two parts separated by <code>_</code> (such as property getters)	<code>kind_name</code> (common case, but not enforced by ECMA)	<code>NameKind</code> , where both parts have been capitalized and swapped
	Getter of property <code>Prop</code>	<code>PropGet</code>
	Setter of property <code>Prop</code>	<code>PropSet</code>
	Event adder	<code>Add</code>
	Event remover	<code>Remove</code>
An operator composed of two parts	<code>op_name</code>	<code>NameOp</code>
For example: + operator	<code>op_Add</code>	<code>AddOp</code>
For a conversion operator , the return type is appended.	<code>T op_Implicit</code>	<code>ImplicitOpT</code>

NOTE

- Getters and setters of indexers** are treated similarly to the property. The default name for an indexer is `Item`.
- Parameter type** names are transformed and concatenated.
- Return type** is ignored unless there's an overload ambiguity. If there's an overload ambiguity, the return type is appended at the end of the name.

Parameter type naming conventions

GIVEN	APPENDED STRING IS...
A type <code>T</code>	<code>T</code>
	The namespace, nested structure, and generic tics are dropped.
An out parameter <code>out T</code>	<code>TOut</code>
A ref parameter <code>ref T</code>	<code>TRef</code>
An array type <code>T[]</code>	<code>TArray</code>
A multi-dimensional array type <code>T[, ,]</code>	<code>T3</code>

GIVEN	APPENDED STRING IS...
A pointer type <code>T*</code>	<code>TPtr</code>
A generic type <code>T<R1, ...></code>	<code>TOFR1</code>
A generic type argument <code>!i</code> of type <code>C<TType></code>	<code>Ti</code>
A generic method argument <code>!!i</code> of method <code>M<MMethod></code>	<code>Mi</code>
A nested type <code>N.T</code>	<code>N</code> is appended, then <code>T</code>

Recursive rules

The following rules are applied recursively:

- Because Fakes uses C# to generate the Fakes assemblies, any character that would produce an invalid C# token is escaped to "_" (underscore).
- If a resulting name clashes with any member of the declaring type, a numbering scheme is used by appending a two-digit counter, starting at 01.

See also

- [Isolating code under test with Microsoft Fakes](#)

How to: Create a data-driven unit test

1/1/2020 • 5 minutes to read • [Edit Online](#)

You can use the Microsoft unit test framework for managed code to set up a unit test method to retrieve values from a data source. The method is run successively for each row in the data source, which makes it easy to test a variety of input by using a single method.

Creating a data-driven unit test involves the following steps:

1. Create a data source that contains the values that you use in the test method. The data source can be any type that is registered on the machine that runs the test.
2. Add a private `TestContext` field and a public `TestContext` property to the test class.
3. Create a unit test method and add a `DataSourceAttribute` attribute to it.
4. Use the `DataRow` indexer property to retrieve the values that you use in a test.

The method under test

As an example, let's assume that you have:

1. A solution called `MyBank` that accepts and processes transactions for different types of accounts.
2. A project in `MyBank` called `BankDb` that manages the transactions for accounts.
3. A class called `Maths` in the `BankDb` project that performs the mathematical functions to ensure that any transaction is advantageous to the bank.
4. A unit test project called `BankDbTests` to test the behavior of the `BankDb` component.
5. A unit test class called `MathsTests` to verify the behavior of the `Maths` class.

We'll test a method in `Maths` that adds two integers using a loop:

```
public int AddIntegers(int first, int second)
{
    int sum = first;
    for( int i = 0; i < second; i++)
    {
        sum += 1;
    }
    return sum;
}
```

Create a data source

To test the `AddIntegers` method, create a data source that specifies a range of values for the parameters and the sum that you expect to be returned. In this example, we'll create a Sql Compact database named `MathsData` and a table named `AddIntegersData` that contains the following column names and values

FIRSTNUMBER	SECONDNUMBER	SUM
0	1	1

FIRSTNUMBER	SECONDNUMBER	SUM
1	1	2
2	-3	-1

Add a TestContext to the test class

The unit test framework creates a `TestContext` object to store the data source information for a data-driven test. The framework then sets this object as the value of the `TestContext` property that you create.

```
private TestContext testContextInstance;
public TestContext TestContext
{
    get { return testContextInstance; }
    set { testContextInstance = value; }
}
```

In your test method, you access the data through the `DataRow` indexer property of the `TestContext`.

NOTE

.NET Core does not support the `DataSource` attribute. If you try to access test data in this way in a .NET Core or UWP unit test project, you'll see an error similar to **"'TestContext' does not contain a definition for 'DataRow' and no accessible extension method 'DataRow' accepting a first argument of type 'TestContext' could be found (are you missing a using directive or an assembly reference?)"**.

Write the test method

The test method for `AddIntegers` is fairly simple. For each row in the data source, call `AddIntegers` with the **FirstNumber** and **SecondNumber** column values as parameters, and verify the return value against **Sum** column value:

```
[DataSource(@"Provider=Microsoft.SqlServerCe.Client.4.0; Data Source=C:\Data\MathsData.sdf;", "Numbers")]
[TestMethod()]
public void AddIntegers_FromDataSourceTest()
{
    var target = new Maths();

    // Access the data
    int x = Convert.ToInt32(TestContext.DataRow["FirstNumber"]);
    int y = Convert.ToInt32(TestContext.DataRow["SecondNumber"]);
    int expected = Convert.ToInt32(TestContext.DataRow["Sum"]);
    int actual = target.IntegerMethod(x, y);
    Assert.AreEqual(expected, actual,
        "x:{0} y:{1}",
        new object[] {x, y});
}
```

The `Assert` method includes a message that displays the `x` and `y` values of a failed iteration. By default, the asserted values - `expected` and `actual` - are already included in failed test details.

Specify the `DataSourceAttribute`

The `DataSource` attribute specifies the connection string for the data source and the name of the table that you use in the test method. The exact information in the connection string differs, depending on what kind of data source

you are using. In this example, we used a SqlServerCe database.

```
[DataSource(@"Provider=Microsoft.SqlServerCe.Client.4.0;Data Source=C:\Data\MathsData.sdf",
"AddIntegersData")]
```

The `DataSource` attribute has three constructors.

```
[DataSource(dataSourceSettingName)]
```

A constructor with one parameter uses connection information that is stored in the `app.config` file for the solution. The `dataSourceSettingsName` is the name of the `Xml` element in the config file that specifies the connection information.

Using an `app.config` file allows you to change the location of the data source without making changes to the unit test itself. For information about how to create and use an `app.config` file, see [Walkthrough: Using a Configuration File to Define a Data Source](#)

```
[DataSource(connectionString, tableName)]
```

The `DataSource` constructor with two parameters specifies the connection string for the data source and the name of the table that contains the data for the test method.

The connection strings depend on the type of the type of data source, but it should contain a `Provider` element that specifies the invariant name of the data provider.

```
[DataSource(
    dataProvider,
    connectionString,
    tableName,
    dataAccessMethod
)]
```

Use `TestContext.DataRow` to access the data

To access the data in the `AddIntegersData` table, use the `TestContext.DataRow` indexer. `DataRow` is a `DataRow` object, so retrieve column values by index or column names. Because the values are returned as objects, convert them to the appropriate type:

```
int x = Convert.ToInt32(TestContext.DataRow["FirstNumber"]);
```

Run the test and view results

When you've finished writing a test method, build the test project. The test method appears in **Test Explorer** in the **Not Run Tests** group. As you run, write, and rerun your tests, **Test Explorer** displays the results in groups of **Failed Tests**, **Passed Tests**, and **Not Run Tests**. You can choose **Run All** to run all your tests, or choose **Run** to choose a subset of tests to run.

The test results bar at the top of **Test Explorer** is animated as your test runs. At the end of the test run, the bar will be green if all of the tests have passed or red if any of the tests have failed. A summary of the test run appears in the details pane at the bottom of the **Test Explorer** window. Select a test to view the details of that test in the bottom pane.

NOTE

There's a result for each row of data and also one summary result. If the test passed on each row of data, the summary run shows as **Passed**. If the test failed on any data row, the summary run shows as **Failed**.

If you ran the `AddIntegers_FromDataSourceTest` method in our example, the results bar turns red and the test method is moved to the **Failed Tests**. A data-driven test fails if any of the iterated methods from the data source fails. When you choose a failed data-driven test in the **Test Explorer** window, the details pane displays the results of each iteration that is identified by the data row index. In our example, it appears that the `AddIntegers` algorithm does not handle negative values correctly.

When the method under test is corrected and the test rerun, the results bar turns green and the test method is moved to the **Passed Test** group.

See also

- [Microsoft.VisualStudio.TestTools.UnitTesting.DataSourceAttribute](#)
- [Microsoft.VisualStudio.TestTools.UnitTesting.TestContext](#)
- [TestContext.RowStyle](#)
- [Microsoft.VisualStudio.TestTools.UnitTesting.Assert](#)
- [Unit test your code](#)
- [Run unit tests with Test Explorer](#)
- [Write unit tests for .NET with the Microsoft unit test framework](#)

Walkthrough: Using a configuration file to define a data source

1/9/2020 • 5 minutes to read • [Edit Online](#)

This walkthrough illustrates how to use a data source defined in an *app.config* file for unit testing. You will learn how to create an *app.config* file that defines a data source that can be used by the [DataSourceAttribute](#) class. Tasks presented in this walkthrough include the following:

- Creating an *app.config* file.
- Defining a custom configuration section.
- Defining connection strings.
- Defining the data sources.
- Accessing the data sources using the [DataSourceAttribute](#) class.

Prerequisites

To complete this walkthrough, you need:

- Visual Studio Enterprise
- Either Microsoft Access or Microsoft Excel to provide data for at least one of the test methods.
- A Visual Studio solution that contains a test project.

Add an *app.config* file to the project

1. If your test project already has an *app.config* file, go to [Define a custom configuration section](#).
2. Right-click your test project in **Solution Explorer**, and then select **Add > New Item**.

The **Add New Item** window opens.

3. Select the **Application Configuration File** template and click **Add**.

Define a custom configuration section

Examine the *app.config* file. It contains at least the XML declaration and a root element.

To add the custom configuration section to the *app.config* file

1. The root element of *app.config* should be the **configuration** element. Create a **configSections** element within the **configuration** element. The **configSections** should be the first element in the *app.config* file.
2. Within the **configSections** element, create a **section** element.
3. In the **section** element, add an attribute called `name` and assign it a value of `microsoft.visualstudio.testtools`. Add another attribute called `type` and assign it a value of `Microsoft.VisualStudio.TestTools.UnitTesting.TestConfigurationSection`, `Microsoft.VisualStudio.TestTools.UnitTesting.TestPlatform.Extensions`

The **section** element should look similar to this:

```
<section name="microsoft.visualstudio.testtools"
type="Microsoft.VisualStudio.TestTools.UnitTesting.TestConfigurationSection,
Microsoft.VisualStudio.TestTools.UnitTesting.Extensions" />
```

NOTE

The assembly name must match the version that you are using.

Define connection strings

The connection strings define provider-specific information for accessing data sources. Connection strings defined in configuration files provide reusable data provider information across an application. In this section, you create two connection strings that will be used by data sources that are defined in the Custom Configuration Section.

To define connection strings

1. After the **configSections** element, create a **connectionStrings** element.
2. Within the **connectionStrings** element, create two **add** elements.
3. In the first **add** element, create the following attributes and values for a connection to a Microsoft Access database:

ATTRIBUTE	VALUES
name	"MyJetConn"
connectionString	"Provider=Microsoft.Jet.OLEDB.4.0; Data Source=C:\testdatasource.accdb; Persist Security Info=False;"
providerName	"System.Data.OleDb"

In the second **add** element, create the following attributes and values for a connection to a Microsoft Excel spreadsheet:

ATTRIBUTE	VALUES
name	"MyExcelConn"
connectionString	"Dsn=Excel Files;dbq=data.xlsx;defaultdir=.\;; driverid=790;maxbuffersize=2048;pagetimeout=5"
providerName	"System.Data.Odbc"

The **connectionStrings** element should look similar to this:

```
<connectionStrings>
  <add name="MyJetConn" connectionString="Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=C:\testdatasource.accdb; Persist Security Info=False;" providerName="System.Data.OleDb" />
  <add name="MyExcelConn" connectionString="Dsn=Excel Files;dbq=data.xlsx;defaultdir=.\;;
driverid=790;maxbuffersize=2048;pagetimeout=5" providerName="System.Data.Odbc" />
</connectionStrings>
```

Define data sources

The data sources section contains four attributes that are used by the test engine to retrieve data from a data source.

- `name` defines the identity used by the [DataSourceAttribute](#) to specify which data source to use.
- `connectionString` identifies the connection string created in the previous Define Connection Strings section.
- `dataTableNames` defines the table or sheet that holds the data to use in the test.
- `dataAccessMethod` defines the technique for accessing data values in the data source.

In this section, you'll define two data sources to use in a unit test.

To define data sources

1. After the `connectionStrings` element, create a **microsoft.visualstudio.testtools** element. This section was created in Define a Custom Configuration Section.
2. Within the **microsoft.visualstudio.testtools** element, create a **dataSources** element.
3. Within the **dataSources** element, create two **add** elements.
4. In the first **add** element, create the following attributes and values for a Microsoft Access data source:

ATTRIBUTE	VALUES
<code>name</code>	"MyJetDataSource"
<code>connectionString</code>	"MyJetConn"
<code>dataTableNames</code>	"MyDataTable"
<code>dataAccessMethod</code>	"Sequential"

In the second **add** element, create the following attributes and values for a Microsoft Excel data source:

ATTRIBUTE	VALUES
<code>Name</code>	"MyExcelDataSource"
<code>connectionString</code>	"MyExcelConn"
<code>dataTableNames</code>	"Sheet1\$"
<code>dataAccessMethod</code>	"Sequential"

The **microsoft.visualstudio.testtools** element should look similar to this:

```

<microsoft.visualstudio.testtools>
  <dataSources>
    <add name="MyJetDataSource" connectionString="MyJetConn" dataTableName="MyDataTable"
      dataAccessMethod="Sequential"/>
    <add name="MyExcelDataSource" connectionString="MyExcelConn" dataTableName="Sheet1$"
      dataAccessMethod="Sequential"/>
  </dataSources>
</microsoft.visualstudio.testtools>

```

The final *app.config* file should look similar to this:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="microsoft.visualstudio.testtools"
      type="Microsoft.VisualStudio.TestTools.UnitTesting.TestConfigurationSection,
      Microsoft.VisualStudio.TestTools.UnitTesting.Extensions" />
  </configSections>
  <connectionStrings>
    <add name="MyJetConn" connectionString="Provider=Microsoft.Jet.OLEDB.4.0; Data
      Source=C:\testdatasource.accdb; Persist Security Info=False;" providerName="System.Data.OleDb" />
    <add name="MyExcelConn" connectionString="Dsn=Excel Files;dbq=data.xlsx;defaultdir=.\;
      driverid=790;maxbuffersize=2048;pagetimeout=5" providerName="System.Data.Odbc" />
  </connectionStrings>
  <microsoft.visualstudio.testtools>
    <dataSources>
      <add name="MyJetDataSource" connectionString="MyJetConn" dataTableName="MyDataTable"
        dataAccessMethod="Sequential"/>
      <add name="MyExcelDataSource" connectionString="MyExcelConn" dataTableName="Sheet1$"
        dataAccessMethod="Sequential"/>
    </dataSources>
  </microsoft.visualstudio.testtools>
</configuration>

```

Create a unit test that uses data sources defined in *app.config*

Now that an *app.config* file has been defined, you will create a unit test that uses data located in the data sources that are defined in the *app.config* file. In this section, we will:

- Create the data sources found in the *app.config* file.
- Use the data sources in two test methods that compare the values in each data source.

To create a Microsoft Access data source

1. Create a Microsoft Access database named *testdatasource.accdb*.
2. Create a table and name it `MyDataTable` in *testdatasource.accdb*.
3. Create two fields in `MyDataTable` named `Arg1` and `Arg2` using the `Number` data type.
4. Add five entities to `MyDataTable` with the following values for `Arg1` and `Arg2`, respectively: (10,50), (3,2), (6,0), (0,8) and (12312,1000).
5. Save and close the database.
6. Change the connection string to point to the location of the database. Change the value of `Data Source` to reflect the location of the database.

To create a Microsoft Excel data source

1. Create a Microsoft Excel spreadsheet named *data.xlsx*.

2. Create a sheet named `Sheet1` if it does not already exist in `data.xlsx`.
 3. Create two column headers and name them `Val1` and `Val2` in `Sheet1`.
 4. Add five entities to `Sheet1` with the following values for `Val1` and `Val2`, respectively: (1,1), (2,2), (3,3), (4,4) and (5,0).
 5. Save and close the spreadsheet.
 6. Change the connection string to point to the location of the spreadsheet. Change the value of `dbq` to reflect the location of the spreadsheet.
- To create a unit test using the app.config data sources**
1. Add a unit test to the test project.
 2. Replace the auto-generated contents of the unit test with the following code:

```

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

namespace TestProject1
{
    [TestClass]
    public class UnitTest1
    {
        private TestContext context;

        public TestContext TestContext
        {
            get { return context; }
            set { context = value; }
        }

        [TestMethod()]
        [DeploymentItem("MyTestProject\\testdataresource.accdb")]
        [DataSource("MyJetDataSource")]
        public void MyTestMethod()
        {
            int a = Int32.Parse(context.DataRow["Arg1"].ToString());
            int b = Int32.Parse(context.DataRow["Arg2"].ToString());
            Assert.AreNotEqual(a, b, "A value was equal.");
        }

        [TestMethod()]
        [DeploymentItem("MyTestProject\\data.xlsx")]
        [DataSource("MyExcelDataSource")]
        public void MyTestMethod2()
        {
            Assert.AreEqual(context.DataRow["Val1"], context.DataRow["Val2"]);
        }
    }
}

```

3. Examine the `DataSource` attributes. Notice the setting names from the `app.config` file.
4. Build your solution and run `MyTestMethod` and `MyTestMethod2` tests.

IMPORTANT

Deploy items like data sources so that they are accessible to the test in the deployment directory.

See also

- [Unit test your code](#)
- [How To: Create a data-driven unit test](#)

Unit tests for generic methods

1/1/2020 • 5 minutes to read • [Edit Online](#)

You can generate unit tests for generic methods exactly as you do for other methods. The following sections provide information about and examples of creating unit tests for generic methods.

Type arguments and type constraints

When Visual Studio generates a unit test for a generic class, such as `MyList<T>`, it generates two methods: a generic helper and a test method. If `MyList<T>` has one or more type constraints, the type argument must satisfy all the type constraints. To make sure that the generic code under test works as expected for all permissible inputs, the test method calls the generic helper method with all the constraints that you want to test.

Examples

The following examples illustrate unit tests for generics:

- [Edit generated test code](#). This example has two sections, Generated Test Code and Edited Test Code. It shows how to edit the raw test code that is generated from a generic method into a useful test method.
- [Use a type constraint](#). This example shows a unit test for a generic method that uses a type constraint. In this example, the type constraint is not satisfied.

Example 1: Editing generated test code

The test code in this section tests a code-under-test method named `SizeOfLinkedList()`. This method returns an integer that specifies the number of nodes in the linked list.

The first code sample, in the section Generated Test Code, shows the unedited test code as it was generated by Visual Studio Enterprise. The second sample, in the section Edited Test Code, shows how you could make it test the functioning of the `SizeOfLinkedList` method for two different data types, `int` and `char`.

This code illustrates two methods:

- a test helper method, `SizeOfLinkedListTestHelper<T>()`. By default, a test helper method has "TestHelper" in its name.
- a test method, `SizeOfLinkedListTest()`. Every test method is marked with the `TestMethod` attribute.

Generated test code

The following test code was generated from the `SizeOfLinkedList()` method. Because this is the unedited generated test, it must be modified to correctly test the `SizeOfLinkedList` method.

```

public void SizeOfLinkedListTestHelper<T>()
{
    T val = default(T); // TODO: Initialize to an appropriate value
    MyLinkedList<T> target = new MyLinkedList<T>(val); // TODO: Initialize to an appropriate value
    int expected = 0; // TODO: Initialize to an appropriate value
    int actual;
    actual = target.SizeOfLinkedList();
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Verify the correctness of this test method.");
}

[TestMethod()]
public void SizeOfLinkedListTest()
{
    SizeOfLinkedListTestHelper<GenericParameterHelper>();
}

```

In the preceding code, the generic type parameter is `GenericParameterHelper`. Whereas you can edit it to supply specific data types, as shown in the following example, you could run the test without editing this statement.

Edited test code

In the following code, the test method and the test helper method have been edited to make them successfully test the code-under-test method `SizeOfLinkedList()`.

Test helper method

The test helper method performs the following steps, which correspond to lines in the code labeled step 1 through step 5.

1. Create a generic linked list.
2. Append four nodes to the linked list. The data type of the contents of these nodes is unknown.
3. Assign the expected size of the linked list to the variable `expected`.
4. Compute the actual size of the linked list and assign it to the variable `actual`.
5. Compare `actual` with `expected` in an `Assert` statement. If the actual is not equal to the expected, the test fails.

Test method

The test method is compiled into the code that is called when you run the test named `SizeOfLinkedListTest`. It performs the following steps, which correspond to lines in the code labeled step 6 and step 7.

1. Specify `<int>` when you call the test helper method, to verify that the test works for `integer` variables.
2. Specify `<char>` when you call the test helper method, to verify that the test works for `char` variables.

```

public void SizeOfLinkedListTestHelper<T>()
{
    T val = default(T);
    MyLinkedList<T> target = new MyLinkedList<T>(val); // step 1
    for (int i = 0; i < 4; i++) // step 2
    {
        MyLinkedList<T> newNode = new MyLinkedList<T>(val);
        target.Append(newNode);
    }
    int expected = 5; // step 3
    int actual;
    actual = target.SizeOfLinkedList(); // step 4
    Assert.AreEqual(expected, actual); // step 5
}

[TestMethod()]
public void SizeOfLinkedListTest()
{
    SizeOfLinkedListTestHelper<int>(); // step 6
    SizeOfLinkedListTestHelper<char>(); // step 7
}

```

NOTE

Each time the `SizeOfLinkedListTest` test runs, its `TestHelper` method is called two times. The assert statement must evaluate to true every time for the test to pass. If the test fails, it might not be clear whether the call that specified `<int>` or the call that specified `<char>` caused it to fail. To find the answer, you could examine the call stack, or you could set breakpoints in your test method and then debug while running the test. For more information, see [How to: Debug while running a test in an ASP.NET solution](#).

Example 2: Using a type constraint

This example shows a unit test for a generic method that uses a type constraint that is not satisfied. The first section shows code from the code-under-test project. The type constraint is highlighted.

The second section shows code from the test project.

Code-under-test project

```

using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;

namespace ClassLibrary2
{
    public class Employee
    {
        public Employee(string s, int i)
        {
        }
    }

    public class GenericList<T> where T : Employee
    {
        private class Node
        {
            private T data;
            public T Data
            {
                get { return data; }
                set { data = value; }
            }
        }
    }
}

```

Test project

As with all newly generated unit tests, you must add non-inconclusive Assert statements to this unit test to make it return useful results. You do not add them to the method marked with the `TestMethod` attribute but to the "TestHelper" method, which for this test is named `DataTestHelper<T>()`.

In this example, the generic type parameter `T` has the constraint `where T : Employee`. This constraint is not satisfied in the test method. Therefore, the `DataTest()` method contains an Assert statement that alerts you to the requirement to supply the type constraint that has been placed on `T`. The message of this Assert statement reads as follows:

```
("No appropriate type parameter is found to satisfies the type constraint(s) of T. " + "Please call DataTestHelper<T>() with appropriate type parameters.");
```

In other words, when you call the `DataTestHelper<T>()` method from the test method, `DataTest()`, you must pass a parameter of type `Employee` or a class derived from `Employee`.

```
using ClassLibrary2;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TestProject1
{
    [TestClass()]
    public class GenericList_NodeTest
    {

        public void DataTestHelper<T>()
            where T : Employee
        {
            GenericList_Shadow<T>.Node target = new GenericList_Shadow<T>.Node(); // TODO: Initialize to an appropriate value
            T expected = default(T); // TODO: Initialize to an appropriate value
            T actual;
            target.Data = expected;
            actual = target.Data;
            Assert.AreEqual(expected, actual);
            Assert.Inconclusive("Verify the correctness of this test method.");
        }

        [TestMethod()]
        public void DataTest()
        {
            Assert.Inconclusive("No appropriate type parameter is found to satisfies the type constraint(s) of T. " +
                "Please call DataTestHelper<T>() with appropriate type parameters.");
        }
    }
}
```

See also

- [Unit test your code](#)

How to: Configure unit tests to target an earlier version of the .NET Framework

1/1/2020 • 3 minutes to read • [Edit Online](#)

When you create a test project in Microsoft Visual Studio, the most recent version of the .NET Framework is set as the target, by default. Additionally, if you upgrade test projects from previous versions of Visual Studio, they are upgraded to target the most recent version of the .NET Framework. By editing the project properties, you can explicitly re-target the project to earlier versions of the .NET Framework.

You can create unit test projects that target specific versions of the .NET Framework. The targeted version must be 3.5 or later, and cannot be a client version. Visual Studio enables the following basic support for unit tests that target specific versions:

- You can create unit test projects and target them to a specific version of the .NET Framework.
- You can run unit tests that target a specific version of the .NET Framework from Visual Studio on your local machine.
- You can run unit tests that target a specific version of the .NET Framework by using *MS Test.exe* from the command prompt.
- You can run unit tests on a build agent as part of a build.

Testing SharePoint Applications

The capabilities listed above also enable you to write unit tests and integration tests for SharePoint applications using Visual Studio. For more information about how to develop SharePoint applications using Visual Studio, see [Create SharePoint solutions](#), [Build and debug SharePoint solutions](#) and [Verify and debug SharePoint code](#).

Limitations

The following limitations apply when you re-target your test projects to use earlier versions of the .NET Framework:

- In the .NET Framework 3.5, multitargeting is supported for test projects that contain only unit tests. The .NET Framework 3.5 does not support any other test type, such as coded UI or load test. The re-targeting is blocked for test types other than unit tests.
- Execution of tests that are targeted at an earlier version of the .NET Framework is supported only in the default host adapter. It is not supported in the ASP.NET host adapter. ASP.NET applications that have to run in the ASP.NET Development Server context must be compatible with the current version of the .NET Framework.
- Data collection support is disabled when you run tests that support .NET Framework 3.5 multitargeting. You can run code coverage by using the Visual Studio command-line tools.
- Unit tests that use .NET Framework 3.5 cannot run on a remote machine.
- You cannot target unit tests to earlier client versions of the framework.

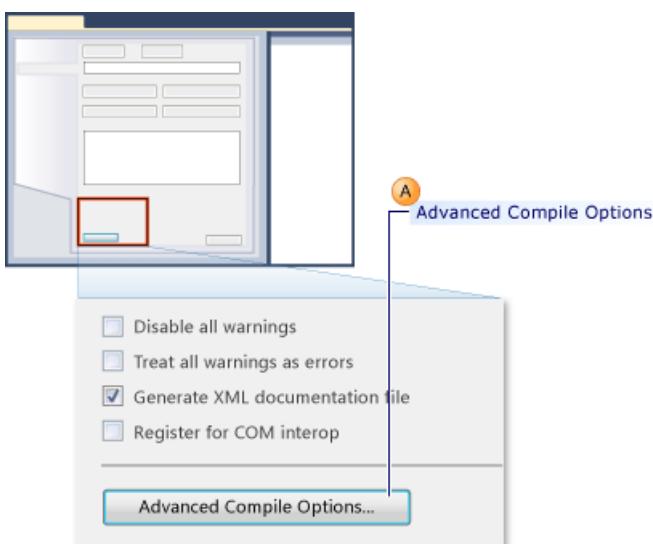
Retargeting for Visual Basic unit test projects

1. Create a new Visual Basic **Unit Test Project** project.

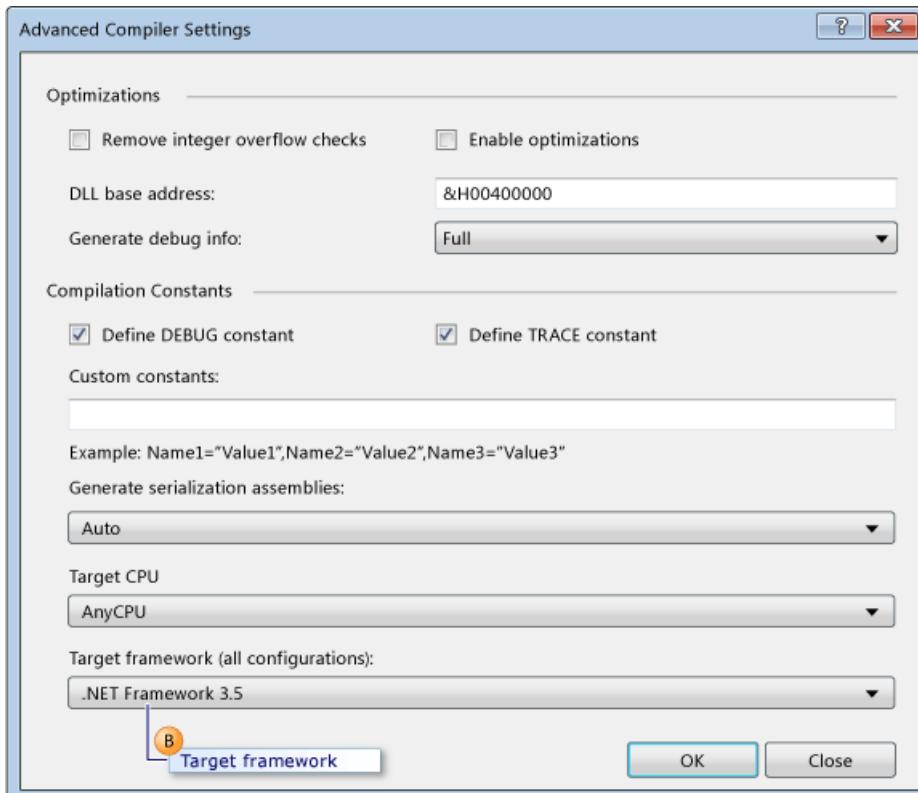
2. In **Solution Explorer**, choose **Properties** from the right-click menu of the new Visual Basic test project.

The properties for your Visual Basic test project are displayed.

3. On the **Compile** tab, choose **Advanced Compile Options** as shown in the following illustration.



4. Use the **Target framework (all configurations)** drop-down list to change the target framework to **.NET Framework 3.5** or a later version as shown in callout B in the following illustration. You should not specify a client version.



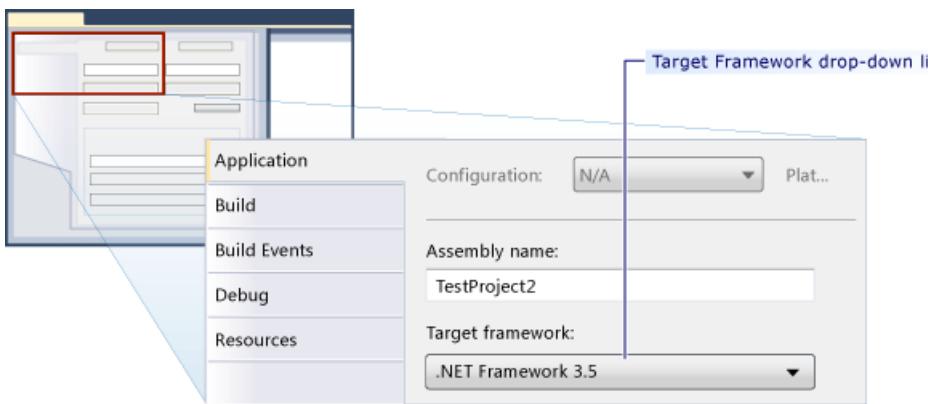
Retargeting for C# unit test projects

1. Create a new C# **Unit Test Project** project.

2. In **Solution Explorer**, choose **Properties** from the right-click menu of your new C# test project.

The properties for your C# test project are displayed.

3. On the **Application** tab, choose **Target framework**. From the drop-down list, choose **.NET Framework 3.5** or a later version, as shown in the following illustration. You should not specify a client version.



Retargeting for C++/CLI unit test projects

1. Create a new C++ **Unit Test Project** project.

WARNING

To build C++/CLI unit tests for a previous version of the .NET framework for Visual C++, you must use the corresponding version of Visual Studio.

2. In **Solution Explorer**, choose **Unload Project** from your new C++ test project.

3. In **Solution Explorer**, choose the unloaded C++ test project and then choose **Edit <project name>.vcxproj**.

The *.vcxproj* file opens in the editor.

4. Set the `TargetFrameworkVersion` to version 3.5 or a later version in the `PropertyGroup` labeled "Globals". You should not specify a client version:

```
<PropertyGroup Label="Globals">
  <TargetName>DefaultTest</TargetName>
  <ProjectTypes>{3AC096D0-A1C2-E12C-1390-A8335801FDAB};{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}</ProjectTypes>
  <ProjectGUID>{CE16D77A-E364-4ACD-948B-1EB6218B0EA3}</ProjectGUID>
  <TargetFrameworkVersion>3.5</TargetFrameworkVersion>
  <Keyword>ManagedCProj</Keyword>
  <RootNamespace>CPP_Test</RootNamespace>
</PropertyGroup>
```

5. Save and close the *.vcxproj* file.

6. In **Solution Explorer**, choose select **Reload Project** from the right-click menu of your new C++ test project.

See also

- [Create SharePoint solutions](#)
- [Build and debug SharePoint solutions](#)
- [Advanced compiler settings dialog box \(Visual Basic\)](#)

Write unit tests for C/C++ in Visual Studio

1/8/2020 • 6 minutes to read • [Edit Online](#)

You can write and run your C++ unit tests by using the **Test Explorer** window. It works just like it does for other languages. For more information about using **Test Explorer**, see [Run unit tests with Test Explorer](#).

NOTE

Some features such as Live Unit Testing, Coded UI Tests and IntelliTest are not supported for C++.

Visual Studio includes these C++ test frameworks with no additional downloads required:

- Microsoft Unit Testing Framework for C++
- Google Test
- Boost.Test
- CTest

Along with using the installed frameworks, you can write your own test adapter for whatever framework you would like to use within Visual Studio. A test adapter can integrate unit tests with the **Test Explorer** window. Several third-party adapters are available on the [Visual Studio Marketplace](#). For more information, see [Install third-party unit test frameworks](#).

Visual Studio 2017 and later (Professional and Enterprise)

C++ unit test projects support [CodeLens](#).

Visual Studio 2017 and later (all editions)

- **Google Test Adapter** is included as a default component of the **Desktop development with C++** workload. It has a project template that you can add to a solution. Use the **Add New Project** right-click menu on the solution node in **Solution Explorer** to add it. It also has options you can configure via **Tools > Options**. For more information, see [How to: Use Google Test in Visual Studio](#).
- **Boost.Test** is included as a default component of the **Desktop development with C++** workload. It's integrated with **Test Explorer**, but currently doesn't have a project template. It must be manually configured. For more information, see [How to: Use Boost.Test in Visual Studio](#).
- **CTest** support is included with the **C++ CMake tools** component, which is part of the **Desktop development with C++** workload. However, CTest isn't fully integrated with **Test Explorer** yet. For more information, see [How to: Use CTest in Visual Studio](#).

Visual Studio 2015 and earlier

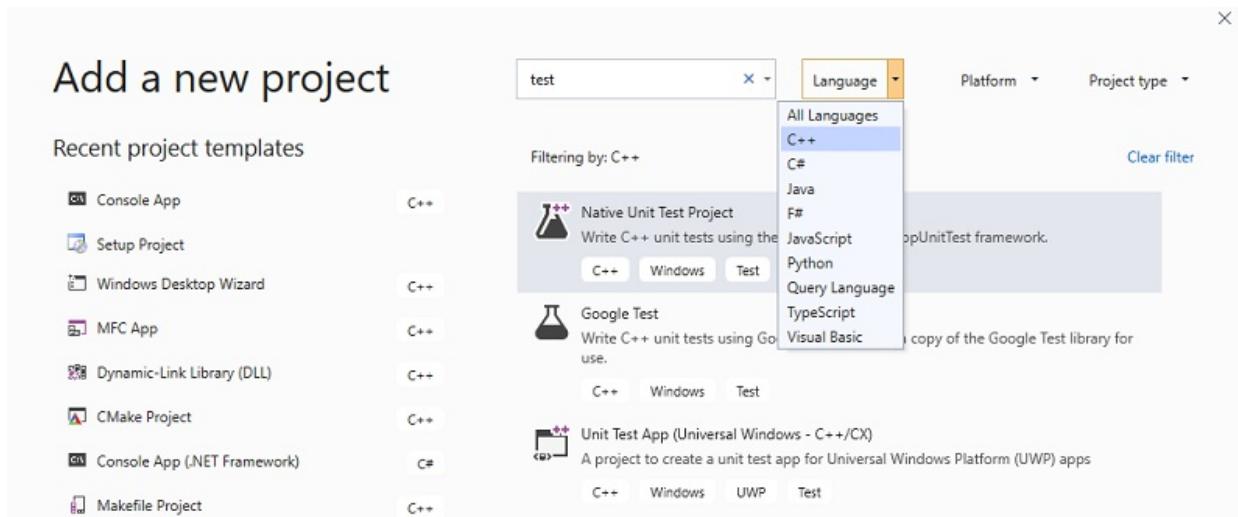
You can download the Google Test adapter and Boost.Test Adapter extensions on the Visual Studio Marketplace. Find them at [Test adapter for Boost.Test](#) and [Test adapter for Google Test](#).

Basic test workflow

The following sections show the basic steps to get you started with C++ unit testing. The basic configuration is similar for both the Microsoft and Google Test frameworks. Boost.Test requires that you manually create a test project.

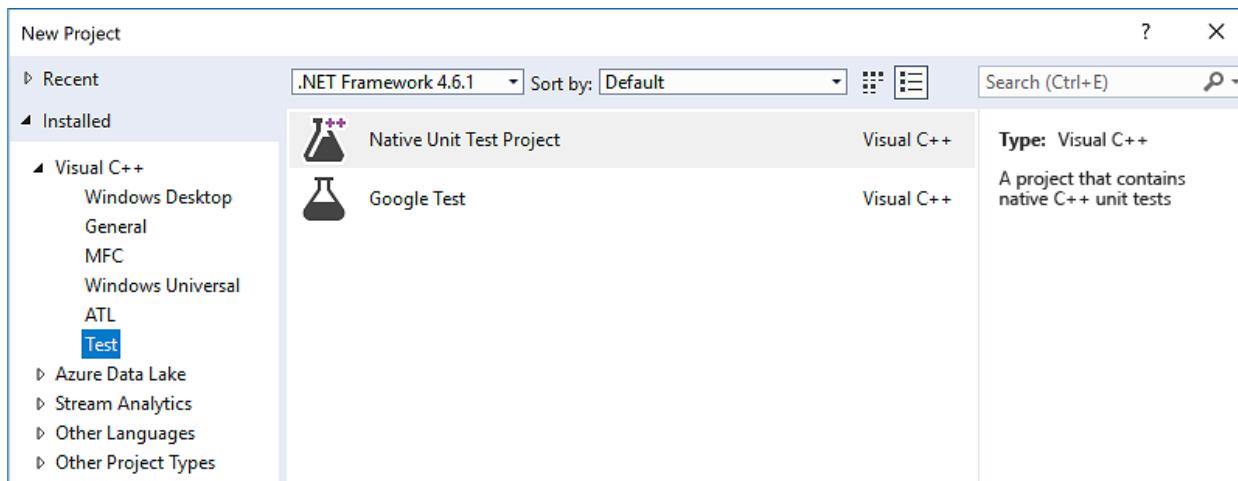
Create a test project in Visual Studio 2019

You define and run tests inside one or more test projects. You create the projects in the same solution as the code you want to test. To add a new test project to an existing solution, right-click on the Solution node in **Solution Explorer**. In the pop-up menu, choose **Add > New Project**. Set **Language** to C++ and type "test" into the search box. The following illustration shows the test projects that are available when the **Desktop Development with C++** and the **UWP Development** workload are installed:



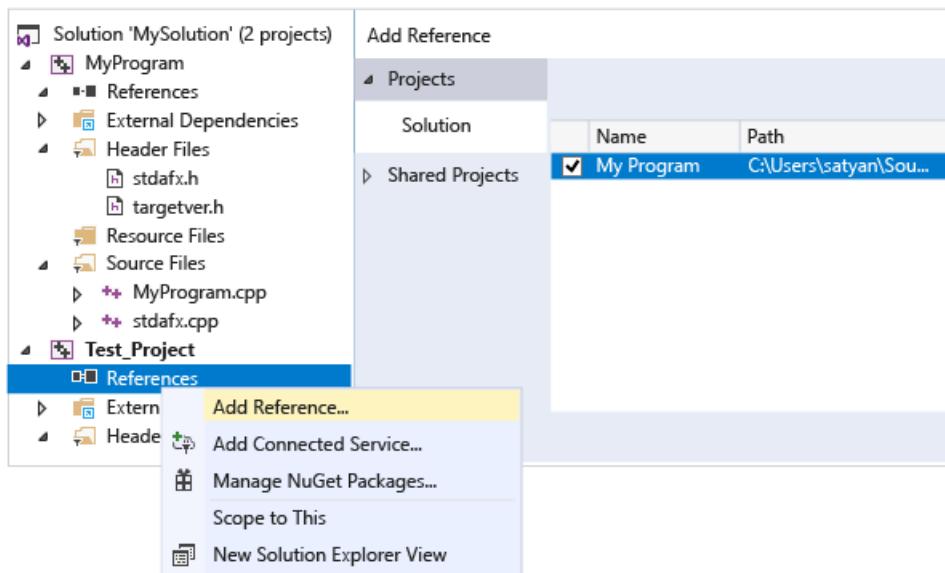
Create a test project in Visual Studio 2017

You define and run tests inside one or more test projects. You create the projects in the same solution as the code you want to test. To add a new test project, right-click on the Solution node in **Solution Explorer** and choose **Add > New Project**. In the left pane, choose **Visual C++ Test**. Then, choose one of the project types from the center pane. The following illustration shows the test projects that are available when the **Desktop Development with C++** workload is installed:



Create references to other projects in the solution

To enable access to the functions in the project under test, add a reference to the project in your test project. Right-click on the test project node in **Solution Explorer** for a pop-up menu. Choose **Add > Reference**. In the Add Reference dialog, choose the project(s) you want to test.

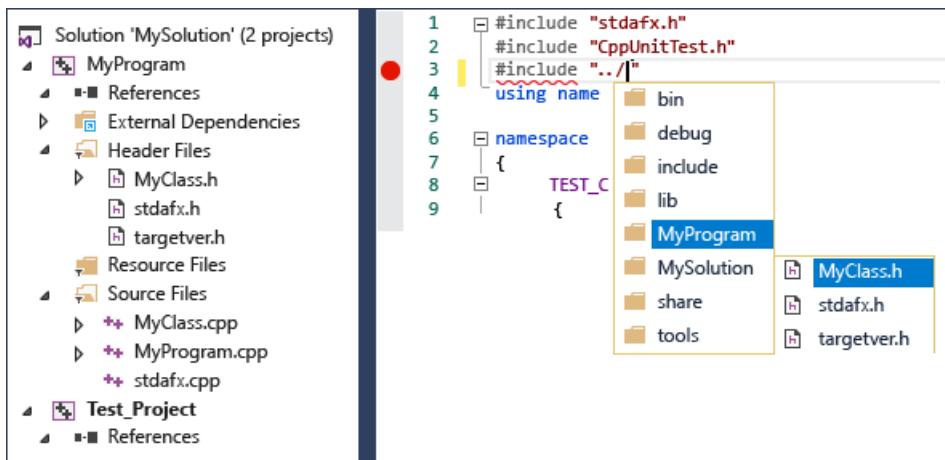


Link to object or library files

If the test code doesn't export the functions that you want to test, you can add the output .obj or .lib files to the dependencies of the test project. For more information, see [To link the tests to the object or library files](#).

Add #include directives for header files

Next, in your unit test .cpp file, add an `#include` directive for any header files that declare the types and functions you want to test. Type `#include "` and then IntelliSense will activate to help you choose. Repeat for any additional headers.



To avoid having to type the full path in each include statement in the source file, you can add the required folders in **Project > Properties > C/C++ > General > Additional Include Directories**.

Write test methods

NOTE

This section shows syntax for the Microsoft Unit Testing Framework for C/C++. It is documented here:

[Microsoft.VisualStudio.TestTools.CppUnitTestFramework API reference](#). For Google Test documentation, see [Google Test primer](#). For Boost.Test, see [Boost Test library: The unit test framework](#).

The .cpp file in your test project has a stub class and method defined for you. They show an example of how to write test code. The signatures use the TEST_CLASS and TEST_METHOD macros, which make the methods discoverable from the **Test Explorer** window.

```

1 #include "stdafx.h"
2 #include "CppUnitTest.h"
3 #include "../MyProgram/MyClass.h"
4 using namespace Microsoft::VisualStudio::CppUnitTestFramework;
5
6 namespace MySolution
7 {
8     TEST_CLASS(UnitTest1)
9     {
10        public:
11        TEST_METHOD(TestMethod1)
12        {
13            // TODO: Your test code here
14        }
15    }
16
17 }
18

```

TEST_CLASS and TEST_METHOD are part of the [Microsoft Native Test Framework](#). **Test Explorer** discovers test methods in other supported frameworks in a similar way.

A TEST_METHOD returns void. To produce a test result, use the static methods in the `Assert` class to test actual results against what is expected. In the following example, assume `MyClass` has a constructor that takes a `std::string`. We can test that the constructor initializes the class as expected like so:

```

TEST_METHOD(TestClassInit)
{
    std::string name = "Bill";
    MyClass mc(name);
    Assert::AreEqual(name, mc.GetName());
}

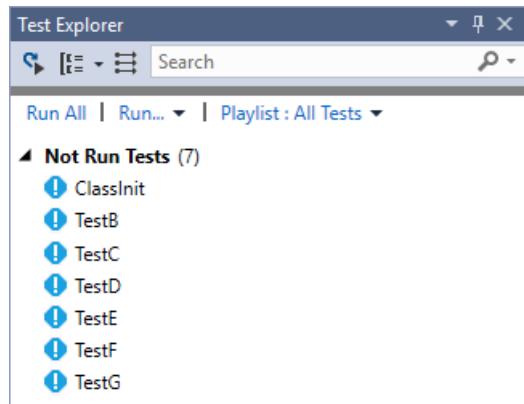
```

In the previous example, the result of the `Assert::AreEqual` call determines whether the test passes or fails. The `Assert` class contains many other methods for comparing expected vs. actual results.

You can add *traits* to test methods to specify test owners, priority, and other information. You can then use these values to sort and group tests in **Test Explorer**. For more information, see [Run unit tests with Test Explorer](#).

Run the tests

1. On the **Test** menu, choose **Windows > Test Explorer**. The following illustration shows a test project whose tests have not yet run.



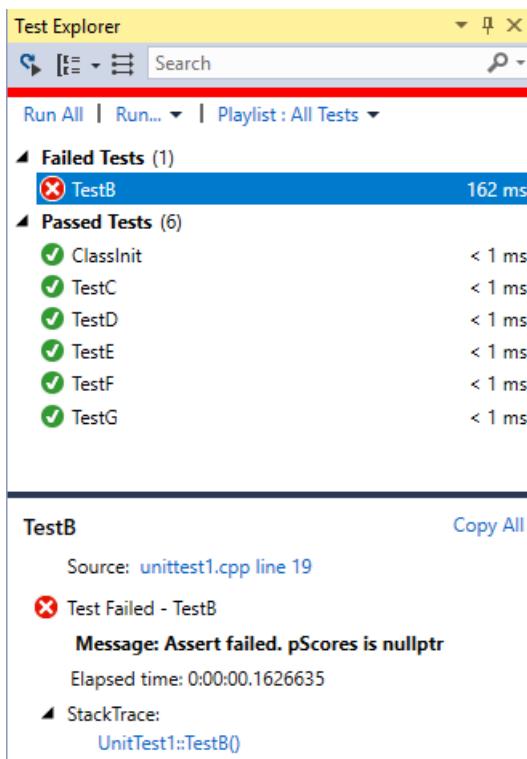
NOTE

CTest integration with **Test Explorer** is not yet available. Run CTest tests from the CMake main menu.

2. If not all your tests are visible in the window, build the test project by right-clicking its node in **Solution**

Explorer and choosing **Build** or **Rebuild**.

3. In **Test Explorer**, choose **Run All**, or select the specific tests you want to run. Right-click on a test for other options, including running it in debug mode with breakpoints enabled. After running all the tests, the window shows which tests passed and which ones failed:



For failed tests, the message offers details that help to diagnose the cause. Right-click on the failing test for a pop-up menu. Choose **Debug Selected Tests** to step through the function where the failure occurred.

For more information about using **Test Explorer**, see [Run unit tests with Test Explorer](#).

For more information related to unit testing, see [Unit test basics](#)

Use CodeLens

Visual Studio 2017 and later (Professional and Enterprise editions)

[CodeLens](#) lets you quickly see the status of a unit test without leaving the code editor.

You can initialize CodeLens for a C++ unit test project in any of these ways:

- Edit and build your test project or solution.
- Rebuild your project or solution.
- Run tests from the **Test Explorer** window.

After it's initialized, you can see test status icons above each unit test.

```
9 //Microsoft native unit test framework
10 {
11     TEST_CLASS(MultiplicationTests)
12     {
13         public:
14             ✓ TEST_METHOD(timesZero)
15             {
16                 int i = 12, j = 0;
17                 int actual = i * j;
18                 Assert::AreEqual(0, actual);
19             }
20             ✘ TEST_METHOD(timesOne)
21             {
22                 int i = 13, j = 1;
23                 int actual = i + j;
24                 Assert::AreEqual(13, actual);
25             }
26     };

```

Click on the icon for more information, or to run or debug the unit test:

```
10 {
11     {
12         T ✘ Test Failed - timesOne
13         P Message: Assert failed. Expected:<13> Actual:<14>
14             Elapsed time: 152 ms
15             ▲ StackTrace:
16                 MultiplicationTests::timesOne()
17
18             Run | Debug
19
20     {
21         ✓ TEST_METHOD(timesZero)
22             {
23                 int i = 12, j = 0;
24                 int actual = i * j;
25                 Assert::AreEqual(0, actual);
26     };

```

See also

- [Unit test your code](#)

Use the Microsoft Unit Testing Framework for C++ in Visual Studio

1/8/2020 • 3 minutes to read • [Edit Online](#)

The Microsoft Unit Testing Framework for C++ is included by default in the **Desktop Development with C++** workload.

To write unit tests in a separate project

Typically, you run your test code in its own project in the same solution as the code you want to test. To set up and configure a new test project, see [Writing unit tests for C/C++](#).

To write unit tests in the same project

In some cases, for example when testing non-exported functions in a DLL, you might need to create the tests in the same project as the program you're testing. To write unit tests in the same project:

1. Modify the project properties to include the headers and library files that are required for unit testing.
 - a. In Solution Explorer, on the shortcut menu of the project you're testing, choose **Properties**. The project properties window opens.
 - b. In the Property Pages dialog, select **Configuration Properties > VC++ Directories**.
 - c. Click on the down arrow in the following rows and choose <**Edit**>. Add these paths:

DIRECTORY	PROPERTY
Include Directories	<code>\$(VCInstallDir)Auxiliary\VS\UnitTest\include</code>
Library Directories	<code>\$(VCInstallDir)Auxiliary\VS\UnitTest\lib</code>

2. Add a C++ Unit Test file:

- Right-click on the project node in **Solution Explorer** and choose **Add > New Item > C++ File (.cpp)**.

To link the tests to the object or library files

If the code under test doesn't export the functions that you want to test, you can add the output **.obj** or **.lib** file to the dependencies of the test project. Modify the test project's properties to include the headers and library or object files that are required for unit testing.

1. In Solution Explorer, on the shortcut menu of the test project, choose **Properties**. The project properties window opens.

2. Select the **Configuration Properties > Linker > Input** page, then select **Additional Dependencies**.

Choose **Edit**, and add the names of the **.obj** or **.lib** files. Don't use the full path names.

3. Select the **Configuration Properties > Linker > General** page, then select **Additional Library Directories**.

Choose **Edit**, and add the directory path of the **.obj** or **.lib** files. The path is typically within the build folder

of the project under test.

4. Select the **Configuration Properties > VC++ Directories** page, then select **Include Directories**.

Choose **Edit**, and then add the header directory of the project under test.

Write the tests

Any .cpp file with test classes must include "CppUnitTest.h" and have a using statement for

```
using namespace Microsoft::VisualStudio::CppUnitTestFramework.
```

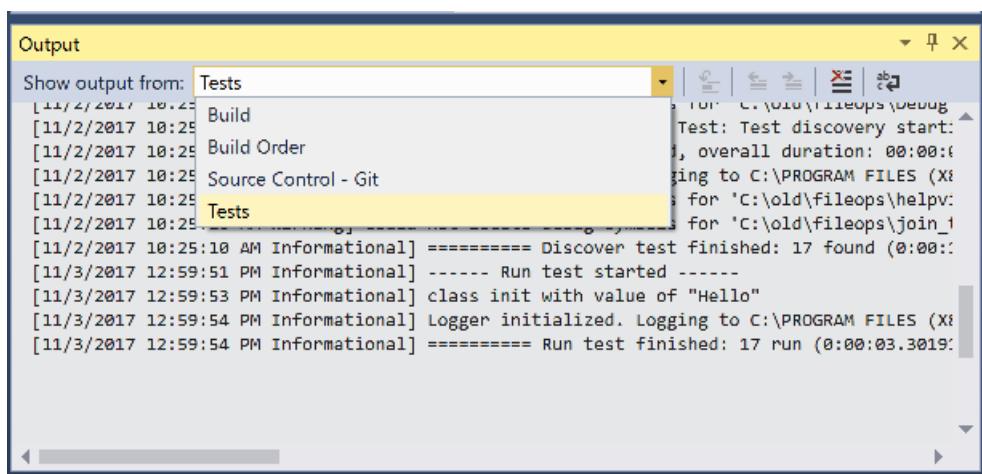
The test project is already configured for you. It also includes a namespace definition, and a TEST_CLASS with a TEST_METHOD to get you started. You can modify the namespace name and the names in parentheses in the class and method macros.

The test framework defines special macros for initializing test modules, classes, and methods, and for cleanup of resources after tests complete. These macros generate code to execute before a class or method is first accessed, and after the last test has run. For more information, see [Initialize and cleanup](#).

Use the static methods in the [Assert](#) class to define test conditions. Use the [Logger](#) class to write messages to the **Output Window**. Add attributes to test methods

Run the tests

1. On the **Test** menu, choose **Windows > Test Explorer**.
2. If not all your tests are visible in the window, build the test project by right-clicking its node in **Solution Explorer** and choosing **Build** or **Rebuild**.
3. In **Test Explorer**, choose **Run All**, or select the specific tests you want to run. Right-click on a test for other options, including running it in debug mode with breakpoints enabled.
4. In the **Output Window** choose **Tests** in the drop-down to view messages written out by the [Logger](#) class:



Define traits to enable grouping

You can define traits on test methods, which enable you to categorize and group tests in **Test Explorer**. To define a trait, use the [TEST_METHOD_ATTRIBUTE](#) macro. For example, to define a trait named [TEST_MY_TRAIT](#):

```
#define TEST_MY_TRAIT(traitValue) TEST_METHOD_ATTRIBUTE(L"MyTrait", traitValue)
```

To use the defined trait in your unit tests:

```

BEGIN_TEST_METHOD_ATTRIBUTE(Method1)
    TEST_OWNER(L"OwnerName")
    TEST_PRIORITY(1)
    TEST_MY_TRAIT(L"thisTraitValue")
END_TEST_METHOD_ATTRIBUTE()

TEST_METHOD(Method1)
{
    Logger::WriteMessage("In Method1");
    Assert::AreEqual(0, 0);
}

```

C++ trait attribute macros

The following pre-defined traits are found in `cppUnitTest.h`. For more information, see [The Microsoft Unit Testing Framework for C++ API reference](#).

MACRO	DESCRIPTION
<code>TEST_METHOD_ATTRIBUTE(attributeName, attributeName)</code>	Use the TEST_METHOD_ATTRIBUTE macro to define a trait.
<code>TEST_OWNER(ownerAlias)</code>	Use the predefined Owner trait to specify an owner of the test method.
<code>TEST_PRIORITY(priority)</code>	Use the predefined Priority trait to assign relative priorities to your test methods.

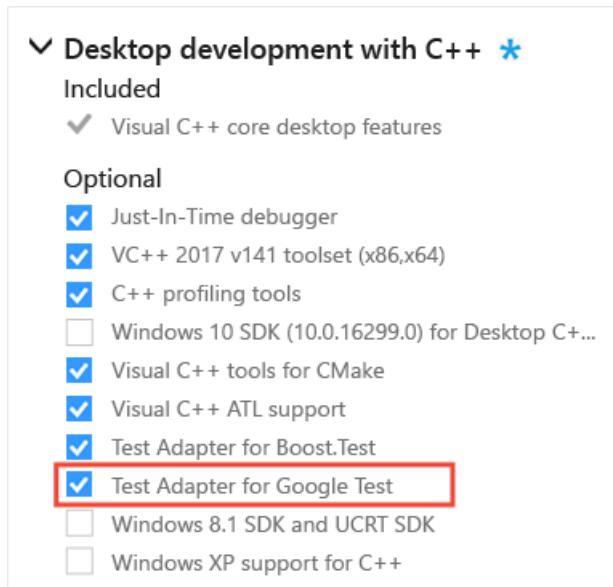
See also

- [Quickstart: Test driven development with Test Explorer](#)

How to use Google Test for C++ in Visual Studio

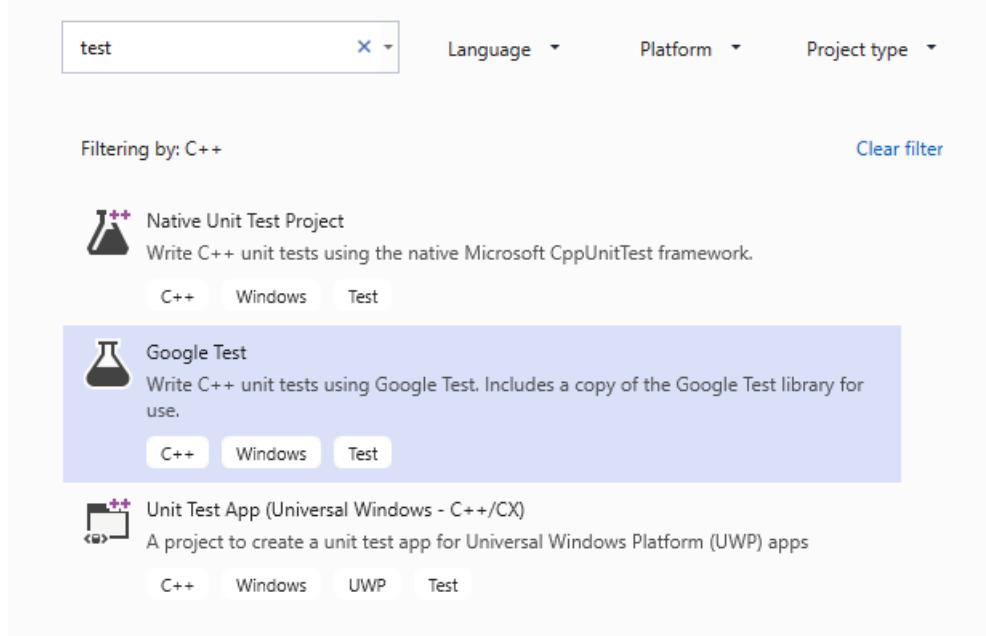
8/9/2019 • 2 minutes to read • [Edit Online](#)

In Visual Studio 2017 and later, Google Test is integrated into the Visual Studio IDE as a default component of the **Desktop Development with C++** workload. To verify that it is installed on your machine, open the Visual Studio Installer and find Google Test under the list of workload components:



Add a Google Test project in Visual Studio 2019

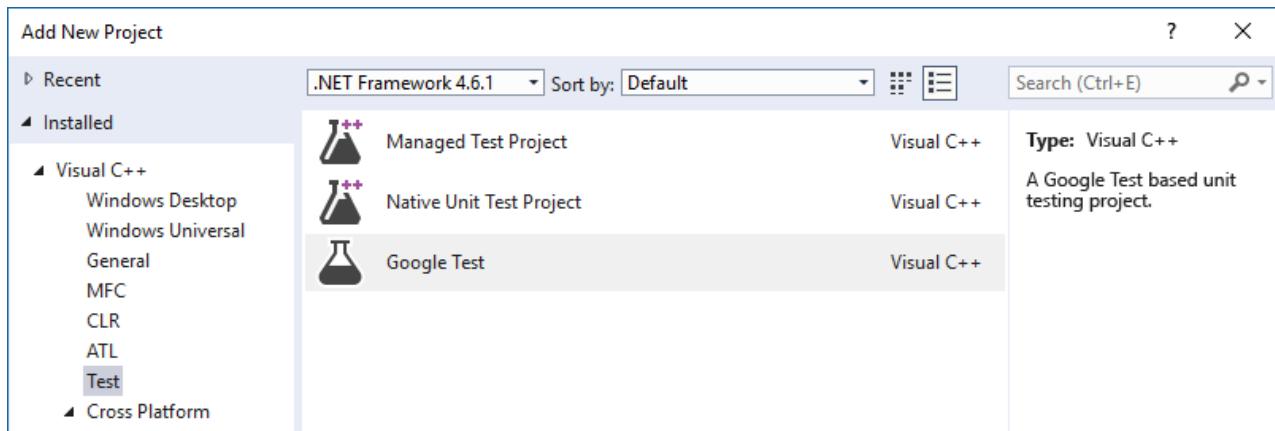
1. In **Solution Explorer**, right-click on the solution node and choose **Add > New Project**.
2. Set **Language** to **C++** and type **test** in the search box. From the results list, choose **Google Test Project**.
3. Give the test project a name and click **OK**.



Add a Google Test project in Visual Studio 2017

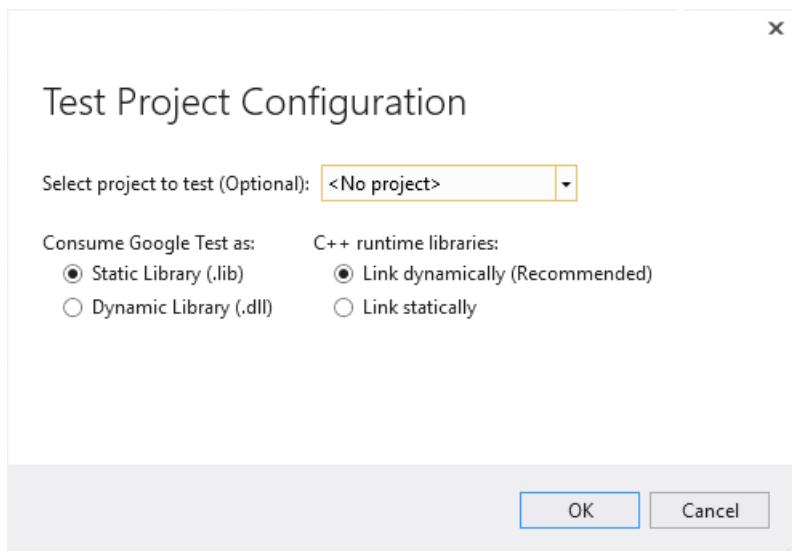
1. In **Solution Explorer**, right-click on the solution node and choose **Add > New Project**.

2. In the left pane, choose **Visual C++ > Test** and then choose **Google Test Project** in the center pane.
3. Give the test project a name and click **OK**.



Configure the test project

In the **Test Project Configuration** dialog that appears, you can choose the project you want to test. When you choose a project, Visual Studio adds a reference to the selected project. If you choose no project, then you need to manually add references to the project(s) you want to test. When choosing between static and dynamic linking to the Google Test binaries, the considerations are the same as for any C++ program. For more information, see [DLLs in Visual C++](#).



Set additional options

From the main menu, choose **Tools > Options > Test Adapter for Google Test** to set additional options. See the Google Test documentation for more information about these settings.

Options

Search Options (Ctrl+E) 🔎

- ▷ Performance Tools
- ▷ Azure Data Lake
- ▷ Container Tools
- ▷ Cookiecutter
- ▷ Cross Platform
- ▷ Database Tools
- ▷ F# Tools
- ▷ Graphics Diagnostics
- ▷ Live Unit Testing
- ▷ NuGet Package Manager
- ▷ Python Tools
- ▷ SQL Server Tools
- ▷ Test
- ◀ **Test Adapter for Google Test**
- General
- Google Test
- Parallelization
- ▷ Text Templating
- ▷ Web
- ▷ Web Forms Designer

Misc

Parse symbol information	True
Print debug info	False
Print test output	False
Test name separator	
Timestamp output	False
Use new test execution framework (experimental)	True

Regexes for trait assignment

After test discovery	
Before test discover	

Test execution

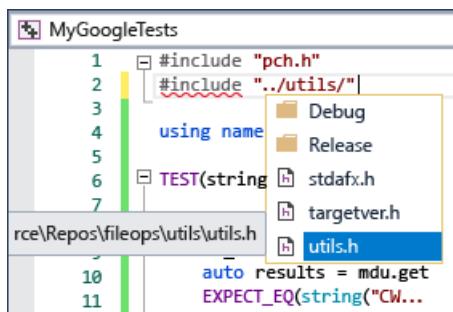
Additional test execution parameters	
PATH extension	
Regex for test discovery	
Terminate processes on cancel	False
Test discovery timeout in s	30

Additional test execution parameters

Additional parameters for Google Test executable. Placeholders:
\$(TestDir) - path of a directory which can be used by the tests \$(ThreadId) - id of the thread executing the curr...

Add include directives

In your test .cpp file, add any needed `#include` directives to make your program's types and functions visible to the test code. Typically, the program is up one level in the folder hierarchy. If you type `#include "../"` an IntelliSense window will appear and enable you to select the full path to the header file.



Write and run tests

You are now ready to write and run Google Tests. See the [Google Test primer](#) for information about the test macros. See [Run unit tests with Test Explorer](#) for information about discovering, running, and grouping your tests by using **Test Explorer**.

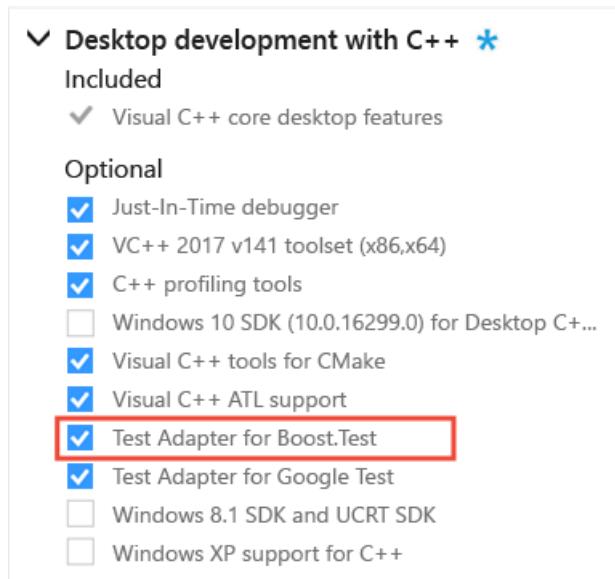
See also

[Write unit tests for C/C++](#)

How to use Boost.Test for C++ in Visual Studio

10/28/2019 • 3 minutes to read • [Edit Online](#)

In Visual Studio 2017 and later, the Boost.Test test adapter is integrated into the Visual Studio IDE as a component of the **Desktop development with C++** workload.



If you don't have the **Desktop development with C++** workload installed, open **Visual Studio Installer**. Select the **Desktop development with C++** workload, then choose the **Modify** button.

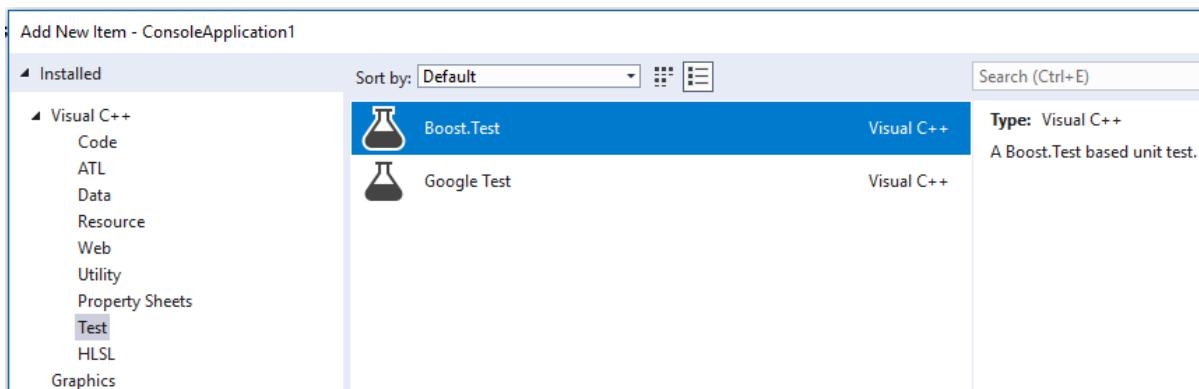
Install Boost

Boost.Test requires [Boost!](#) If you do not have Boost installed, we recommend that you use the Vcpkg package manager.

1. Follow the instructions at [Vcpkg: a C++ package manager for Windows](#) to install vcpkg (if you don't already have it).
2. Install the Boost.Test dynamic or static library:
 - Run **vcpkg install boost-test** to install the Boost.Test dynamic library.
-OR-
• Run **vcpkg install boost-test:x86-windows-static** to install the Boost.Test static library.
3. Run **vcpkg integrate install** to configure Visual Studio with the library and include paths to the Boost headers and binaries.

Add the item template (Visual Studio 2017 version 15.6 and later)

1. To create a .cpp file for your tests, right-click on the project node in **Solution Explorer** and choose **Add New Item**.



2. The new file contains a sample test method. Build your project to enable **Test Explorer** to discover the method.

The item template uses the single-header variant of Boost.Test, but you can modify the #include path to use the standalone library variant. For more information, see [Add include directives](#).

Create a test project

In Visual Studio 2017 version 15.5, no pre-configured test project or item templates are available for Boost.Test. Therefore, you have to create and configure a console application project to hold your tests.

1. In **Solution Explorer**, right click on the solution node and choose **Add > New Project**.
2. In the left pane, choose **Visual C++ > Windows Desktop**, and then choose the **Windows Console Application** template.
3. Give the project a name and choose **OK**.
4. Delete the `main` function in the `.cpp` file.
5. If you are using the single-header or dynamic library version of Boost.Test, go to [Add include directives](#). If you are using the static library version, then you have to perform some additional configuration:
 - a. To edit the project file, first unload it. In **Solution Explorer**, right-click the project node and choose **Unload Project**. Then, right-click the project node and choose **Edit <name>.vcxproj**.
 - b. Add two lines to the **Globals** property group as shown here:

```
<PropertyGroup Label="Globals">
  ...
  <VcpkgTriplet>x86-windows-static</VcpkgTriplet>
  <VcpkgEnabled>true</VcpkgEnabled>
</PropertyGroup>
```

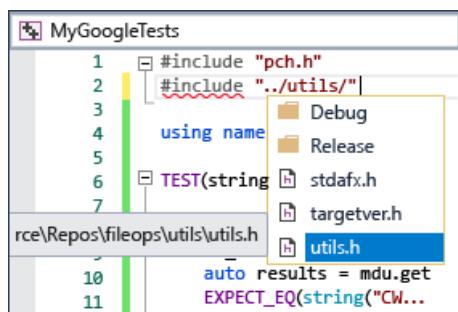
- c. Save and close the `*.vcxproj` file, and then reload the project.
- d. To open the **Property Pages**, right-click on the project node and choose **Properties**.
- e. Expand **C/C++ > Code Generation**, and then select **Runtime Library**. Select **/MTd** for debug static runtime library or **/MT** for release static runtime library.
- f. Expand **Linker > System**. Verify that **SubSystem** is set to **Console**.
- g. Choose **OK** to close the property pages.

Add include directives

1. In your test `.cpp` file, add any needed `#include` directives to make your program's types and functions

visible to the test code. Typically, the program is up one level in the folder hierarchy. If you type

```
#include "../"
```



You can use the standalone library with:

```
#include <boost/test/unit_test.hpp>
```

Or, use the single-header version with:

```
#include <boost/test/included/unit_test.hpp>
```

Then, define `BOOST_TEST_MODULE`.

The following example is sufficient for the test to be discoverable in **Test Explorer**:

```
#define BOOST_TEST_MODULE MyTest
#include <boost/test/included/unit_test.hpp> //single-header
#include "../MyProgram/MyClass.h" // project being tested
#include <string>

BOOST_AUTO_TEST_CASE(my_boost_test)
{
    std::string expected_value = "Bill";

    // assume MyClass is defined in MyClass.h
    // and get_value() has public accessibility
    MyClass mc;
    BOOST_CHECK(expected_value == mc.get_value());
}
```

Write and run tests

You're now ready to write and run Boost tests. See the [Boost test library documentation](#) for information about the test macros. See [Run unit tests with Test Explorer](#) for information about discovering, running, and grouping your tests by using **Test Explorer**.

See also

- [Write unit tests for C/C++](#)

How to use CTest for C++ in Visual Studio 2017 and later

10/1/2019 • 2 minutes to read • [Edit Online](#)

CMake (which includes CTest) is integrated into the Visual Studio IDE by default as a component of the **Desktop Development with C++** workload. If you need to install it on your machine, open the Visual Studio Installer program, click the **Desktop Development with C++** button, then click **Modify**. Select **C++ CMake tools for Windows** under the list of workload components.

To write tests

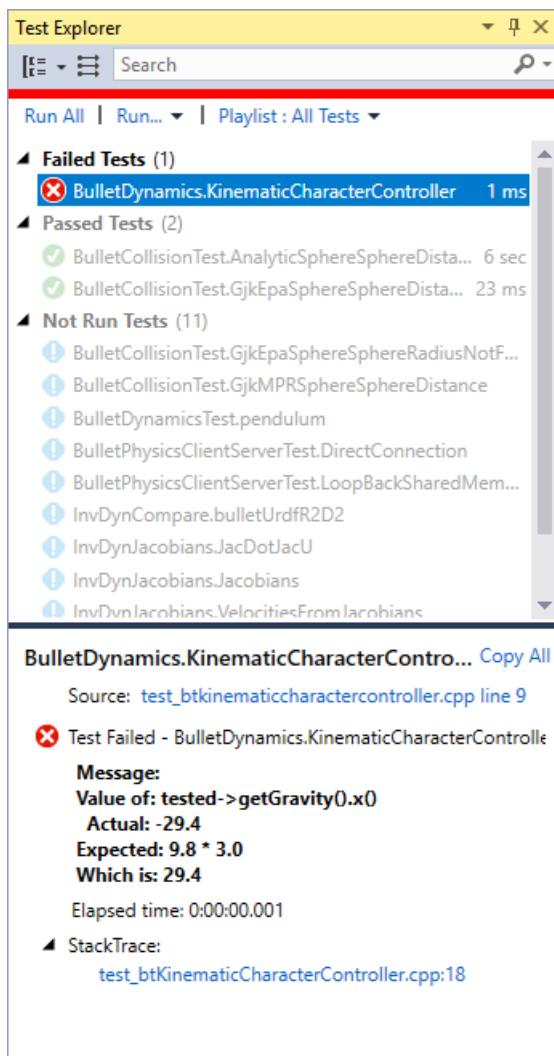
CMake support in Visual Studio doesn't involve the Visual Studio project system. Therefore, you write and configure CTest tests just as you would in any CMake environment. Use the `enable_testing()` command to enable testing, and the `add_test()` command to add a new test. To learn more about CTest, see the [CMake documentation](#).

For more information about using CMake in Visual Studio, see [CMake projects in Visual Studio](#).

To run tests

CTest is fully integrated with **Test Explorer** and also supports both the Google and Boost unit testing frameworks. Those frameworks are included by default as components in the **Desktop Development with C++** workload. However, if you are upgrading a project from an older version of Visual Studio, you may need to install those frameworks by using the Visual Studio Installer program.

The following illustration shows results of a CTest run using Google Test framework:



If you are using CTest but not the Google or Boost adapters, you see results at the CTest level instead of the individual test method level. You can debug and step-through CTest-only executables, but stack traces on individual tests aren't supported.

See also

- [Write unit tests for C/C++](#)

Write unit tests for C++ DLLs in Visual Studio

8/9/2019 • 4 minutes to read • [Edit Online](#)

There are several ways to test DLL code, depending on whether it exports the functions that you want to test. Choose one of the following ways:

The unit tests call only functions that are exported from the DLL: Add a separate test project as described in [Write unit tests for C/C++](#). In the test project, add a reference to the DLL project.

Go to the procedure [To reference exported functions from the DLL project](#).

The DLL is built as an .exe file: Add a separate test project. Link it to the output object file.

Go to the procedure [To link the tests to the object or library files](#).

The unit tests call non-member functions which are not exported from the DLL, and the DLL can be built as a static library: Change the DLL project so that it is compiled to a *.lib* file. Add a separate test project that references the project under test.

This approach has the benefit of allowing your tests to use non-exported members, but still keep the tests in a separate project.

Go to the procedure [To change the DLL to a static library](#).

The unit tests must call non-member functions that are not exported, and the code must be built as a dynamic link library (DLL): Add unit tests in the same project as the product code.

Go to the procedure [To add unit tests in the same project](#).

Create the tests

To change the DLL to a static library

- If your tests must use members that are not exported by the DLL project, and the project under test is built as a dynamic library, consider converting it to a static library.
 1. In **Solution Explorer**, on the shortcut menu of the project under test, choose **Properties**. The project **Properties** window opens.
 2. Choose **Configuration Properties** > **General**.
 3. Set **Configuration Type** to **Static Library (.lib)**.

Continue with the procedure [To link the tests to the object or library files](#).

To reference exported DLL functions from the test project

- If the DLL project exports the functions that you want to test, then you can add a reference to the code project from the test project.
 1. Create a Native Unit Test Project.
 1. On the **File** menu, choose **New > Project**. In the **Add a New Project** dialog, set **Language** to **C++** and type "test" into the search box. Then choose the **Native Unit Test Project**.
 1. On the **File** menu, choose **New > Project > Visual C++ > Test > C++ Unit Test Project**.
 2. In **Solution Explorer**, right-click on the test project, then choose **Add > Reference**.

3. Select **Projects**, and then the project to be tested.

Choose the **Add** button.

4. In the properties for the test project, add the location of the project under test to the Include Directories.

Choose **Configuration Properties > VC++ Directories > Include Directories**.

Choose **Edit**, and then add the header directory of the project under test.

Go to [Write the unit tests](#).

To link the tests to the object or library files

- If the DLL does not export the functions that you want to test, you can add the output **.obj** or **.lib** file to the dependencies of the test project.

1. Create a Native Unit Test Project.

1. On the **File** menu, choose **New > Project**. In the **Add a New Project** dialog, set **Language** to C++ and type "test" into the search box. Then choose the **Native Unit Test Project**.

1. On the **File** menu, choose **New > Project > Visual C++ > Test > C++ Unit Test Project**.

2. In **Solution Explorer**, on the shortcut menu of the test project, choose **Properties**.

3. Choose **Configuration Properties > Linker > Input > Additional Dependencies**.

Choose **Edit**, and add the names of the **.obj** or **.lib** files. Do not use the full path names.

4. Choose **Configuration Properties > Linker > General > Additional Library Directories**.

Choose **Edit**, and add the directory path of the **.obj** or **.lib** files. The path is typically within the build folder of the project under test.

5. Choose **Configuration Properties > VC++ Directories > Include Directories**.

Choose **Edit**, and then add the header directory of the project under test.

Go to [Write the unit tests](#).

To add unit tests in the same project

1. Modify the product code project properties to include the headers and library files that are required for unit testing.

a. In **Solution Explorer**, in the shortcut menu of the project under test, choose **Properties**. The project **Properties** window opens.

b. Choose **Configuration Properties > VC++ Directories**.

c. Edit the Include and Library directories:

DIRECTORY	PROPERTY
Include Directories	<code>\$(VCInstallDir)UnitTest\include;\$(IncludePath)</code>
Library Directories	<code>\$(VCInstallDir)UnitTest\lib;\$(LibraryPath)</code>

2. Add a C++ Unit Test file:

• In **Solution Explorer**, in the shortcut menu of the project, choose **Add > New Item > C++ Unit Test**.

Go to [Write the unit tests](#).

Write the unit tests

1. In each unit test code file, add an `#include` statement for the headers of the project under test.
2. Add test classes and methods to the unit test code files. For example:

```
#include "stdafx.h"
#include "CppUnitTest.h"
#include "MyProjectUnderTest.h"
using namespace Microsoft::VisualStudio::CppUnitTestFramework;
namespace MyTest
{
    TEST_CLASS(MyTests)
    {
        public:
            TEST_METHOD(MyTestMethod)
            {
                Assert::AreEqual(MyProject::Multiply(2,3), 6);
            }
    };
}
```

Run the tests

1. On the **Test** menu, choose **Windows > Test Explorer**.
2. If all your tests are not visible in the window, build the test project by right-clicking its node in **Solution Explorer** and choosing **Build** or **Rebuild**.
3. In **Test Explorer**, choose **Run All**, or select the specific tests you want to run. Right-click on a test for other options, including running it in debug mode with breakpoints enabled.

See also

- [Write unit tests for C/C++](#)
- [Microsoft.VisualStudio.TestTools.CppUnitTestFramework API Reference](#)
- [Debug native code](#)
- [Walkthrough: Creating and using a dynamic link library \(C++\)](#)
- [Import and export](#)
- [Quickstart: Test driven development with Test Explorer](#)

How to: Write unit tests for C++ DLLs

7/30/2019 • 8 minutes to read • [Edit Online](#)

This walkthrough describes how to develop a native C++ DLL using test-first methodology. The basic steps are as follows:

1. [Create a native test project](#). The test project is located in the same solution as the DLL project.
2. [Create a DLL project](#). This walkthrough creates a new DLL, but the procedure for testing an existing DLL is similar.
3. [Make the DLL functions visible to the tests](#).
4. [Iteratively augment the tests](#). We recommend a "red-green-refactor" cycle, in which development of the code is led by the tests.
5. [Debug failing tests](#). You can run tests in debug mode.
6. [Refactor while keeping the tests unchanged](#). Refactoring means improving the structure of the code without changing its external behavior. You can do it to improve the performance, extensibility, or readability of the code. Because the intention is not to change the behavior, you do not change the tests while making a refactoring change to the code. The tests help make sure that you do not introduce bugs while you are refactoring.
7. [Check coverage](#). Unit tests are more useful when they exercise more of your code. You can discover which parts of your code have been used by the tests.
8. [Isolate units from external resources](#). Typically, a DLL is dependent on other components of the system that you are developing, such as other DLLs, databases, or remote subsystems. It is useful to test each unit in isolation from its dependencies. External components can make tests run slowly. During development, the other components might not be complete.

Create a native unit test project

1. On the **File** menu, choose **New > Project**.

Visual Studio 2017 and earlier: Expand **Installed > Templates > Visual C++ > Test**.

2019: Set **Language** to C++ and type "test" into the search box.

Choose the **Native Unit Test Project** template, or whatever installed framework you prefer. If you choose another template such as Google Test or Boost.Test, the basic principles are the same although some details will differ.

In this walkthrough, the test project is named `NativeRooterTest`.

2. In the new project, inspect **unittest1.cpp**

```
#include "stdafx.h"
#include "CppUnitTest.h"

using namespace Microsoft::VisualStudio::CppUnitTest;

namespace UnitTest1
{
    TEST_CLASS (UnitTest1)
    {
        public:
            TEST_METHOD (TestMethod1)
            {
                // TODO: Your test code here
            }
    };
}
```

Notice that:

- Each test is defined by using `TEST_METHOD(YourTestMethod){...}`.

You do not have to write a conventional function signature. The signature is created by the macro `TEST_METHOD`. The macro generates an instance function that returns void. It also generates a static function that returns information about the test method. This information allows the test explorer to find the method.

- Test methods are grouped into classes by using `TEST_CLASS(YourClassName){...}`.

When the tests are run, an instance of each test class is created. The test methods are called in an unspecified order. You can define special methods that are invoked before and after each module, class, or method.

3. Verify that the tests run in Test Explorer:

- Insert some test code:

```
TEST_METHOD(TestMethod1)
{
    Assert::AreEqual(1,1);
}
```

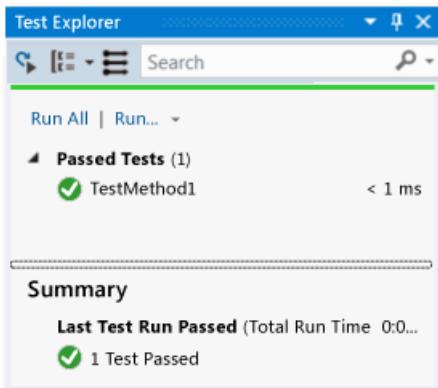
Notice that the `Assert` class provides several static methods that you can use to verify results in test methods.

- On the **Test** menu, choose **Run > All Tests**.

The test builds and runs.

Test Explorer appears.

The test appears under **Passed Tests**.



Create a DLL project

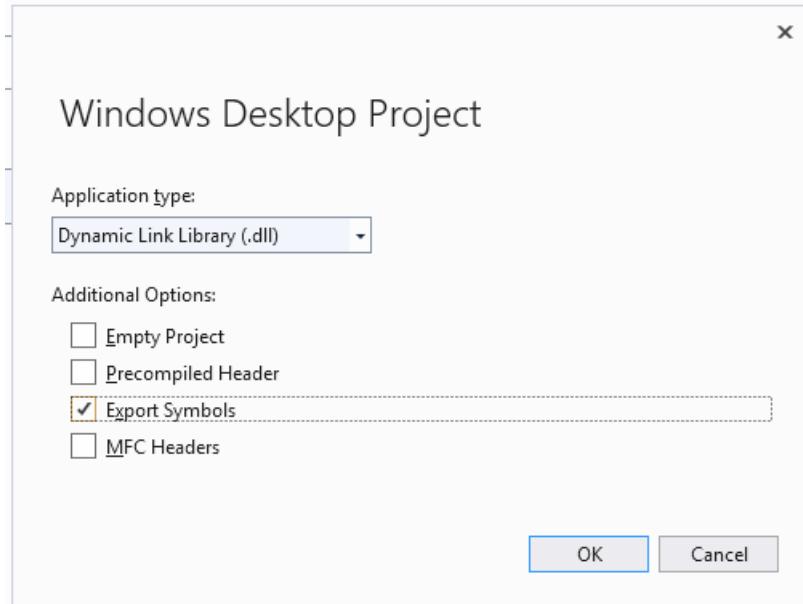
The following steps show how to create a DLL project in Visual Studio 2019.

1. Create a C++ project by using the **Windows Desktop Wizard**: Right-click on the solution name in **Solution Explorer** and choose **Add > New Project**. Set the **Language** to C++ and then type "windows" in the search box. Choose **Windows Desktop Wizard** from the results list.

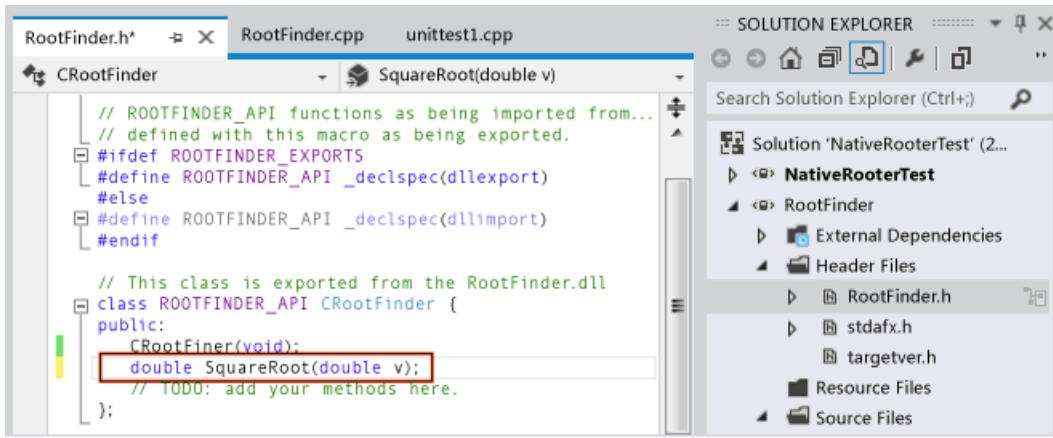
In this walkthrough, the project is named `RootFinder`.

2. Press **Create**. In the next dialog, under **Application type** choose **Dynamic Link Library (dll)** and also check **Export Symbols**.

The **Export Symbols** option generates a convenient macro that you can use to declare exported methods.



3. Declare an exported function in the principal .h file:



The declarator `__declspec(dllexport)` causes the public and protected members of the class to be visible outside the DLL. For more information, see [Using `dllimport` and `dllexport` in C++ Classes](#).

4. In the principal `.cpp` file, add a minimal body for the function:

```
// Find the square root of a number.
double CRootFinder::SquareRoot(double v)
{
    return 0.0;
}
```

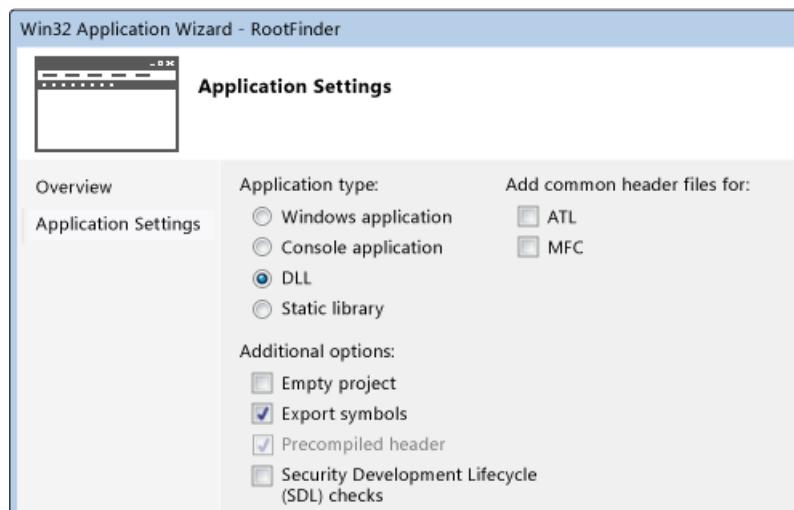
The following steps show how to create a DLL project in Visual Studio 2017.

1. Create a C++ project by using the **Win32 Project** template.

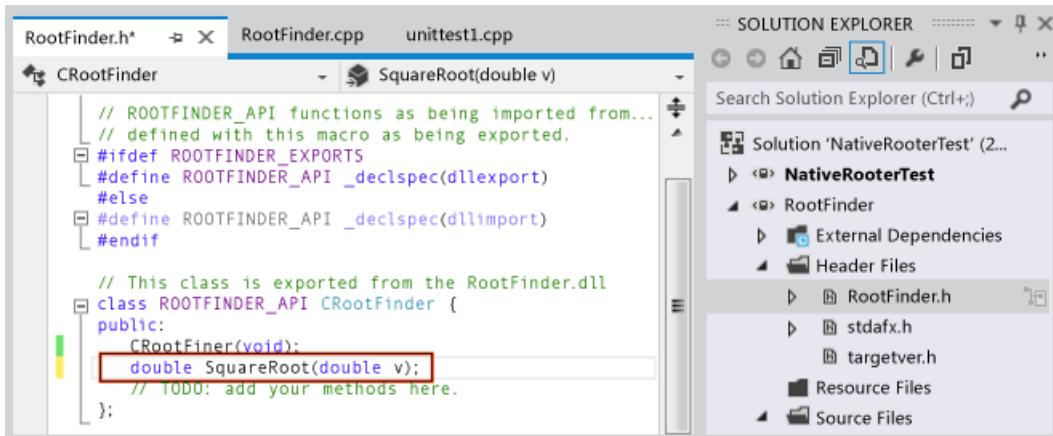
In this walkthrough, the project is named `RootFinder`.

2. Select **DLL** and **Export Symbols** in the Win32 Application Wizard.

The **Export Symbols** option generates a convenient macro that you can use to declare exported methods.



3. Declare an exported function in the principal `.h` file:



The declarator `__declspec(dllexport)` causes the public and protected members of the class to be visible outside the DLL. For more information, see [Using `dllimport` and `dllexport` in C++ Classes](#).

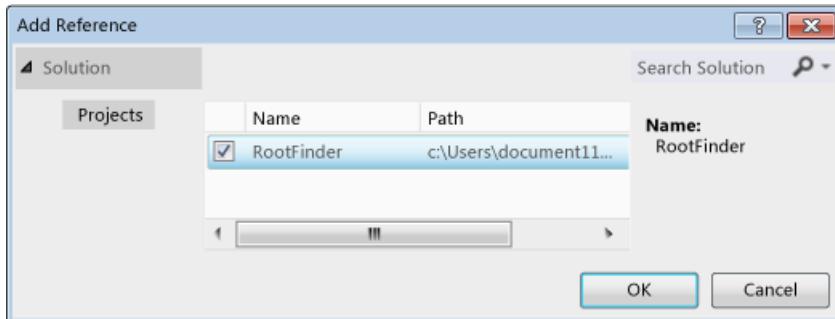
4. In the principal .cpp file, add a minimal body for the function:

```
// Find the square root of a number.
double CRootFinder::SquareRoot(double v)
{
    return 0.0;
}
```

Couple the test project to the DLL project

1. Add the DLL project to the project references of the test project:

- a. Right-click on the test project node in **Solution Explorer** and choose **Add > Reference**.
- b. In the **Add Reference** dialog box, select the DLL project and choose **Add**.



2. In the principal unit test .cpp file, include the .h file of the DLL code:

```
#include "..\RootFinder\RootFinder.h"
```

3. Add a basic test that uses the exported function:

```

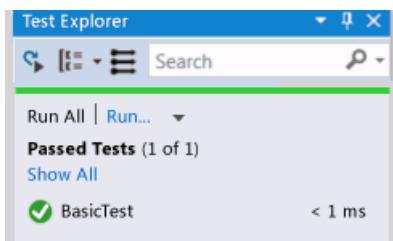
TEST_METHOD(BasicTest)
{
    CRootFinder rooter;
    Assert::AreEqual(
        // Expected value:
        0.0,
        // Actual value:
        rooter.SquareRoot(0.0),
        // Tolerance:
        0.01,
        // Message:
        L"Basic test failed",
        // Line number - used if there is no PDB file:
        LINE_INFO());
}

```

4. Build the solution.

The new test appears in **Test Explorer**.

5. In **Test Explorer**, choose **Run All**.



You have set up the test and the code projects, and verified that you can run tests that run functions in the code project. Now you can begin to write real tests and code.

Iteratively augment the tests and make them pass

1. Add a new test:

```

TEST_METHOD(RangeTest)
{
    CRootFinder rooter;
    for (double v = 1e-6; v < 1e6; v = v * 3.2)
    {
        double actual = rooter.SquareRoot(v*v);
        Assert::AreEqual(v, actual, v/1000);
    }
}

```

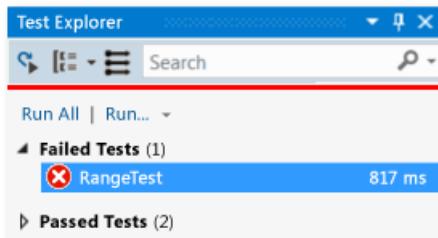
TIP

We recommend that you do not change tests that have passed. Instead, add a new test, update the code so that the test passes, and then add another test, and so on.

When your users change their requirements, disable the tests that are no longer correct. Write new tests and make them work one at a time, in the same incremental manner.

2. Build the solution, and then in **Test Explorer**, choose **Run All**.

The new test fails.



TIP

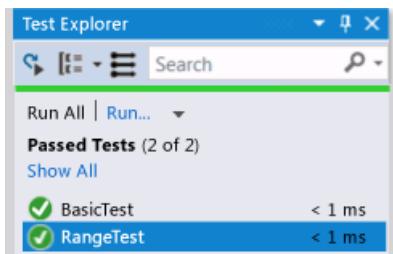
Verify that each test fails immediately after you have written it. This helps you avoid the easy mistake of writing a test that never fails.

3. Enhance your DLL code so that the new test passes:

```
#include <math.h>
...
double CRootFinder::SquareRoot(double v)
{
    double result = v;
    double diff = v;
    while (diff > result/1000)
    {
        double oldResult = result;
        result = result - (result*result - v)/(2*result);
        diff = abs (oldResult - result);
    }
    return result;
}
```

4. Build the solution and then in **Test Explorer**, choose **Run All**.

Both tests pass.



TIP

Develop code by adding tests one at a time. Make sure that all the tests pass after each iteration.

Debug a failing test

1. Add another test:

```

#include <stdexcept>
...
// Verify that negative inputs throw an exception.
TEST_METHOD(NegativeRangeTest)
{
    wchar_t message[200];
    CRootFinder rooter;
    for (double v = -0.1; v > -3.0; v = v - 0.5)
    {
        try
        {
            // Should raise an exception:
            double result = rooter.SquareRoot(v);

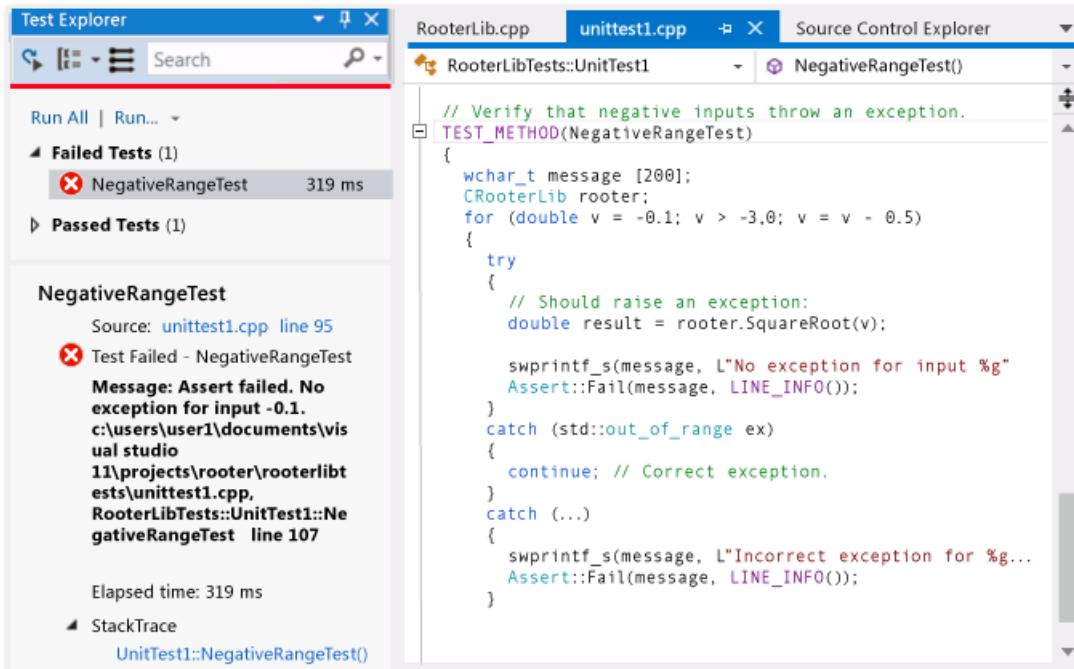
            _swprintf(message, L"No exception for input %g", v);
            Assert::Fail(message, LINE_INFO());
        }
        catch (std::out_of_range ex)
        {
            continue; // Correct exception.
        }
        catch (...)
        {
            _swprintf(message, L"Incorrect exception for %g", v);
            Assert::Fail(message, LINE_INFO());
        }
    }
}

```

2. Build the solution and choose **Run All**.

3. Open (or double-click) the failed test.

The failed assertion is highlighted. The failure message is visible in the detail pane of **Test Explorer**.



4. To see why the test fails, step through the function:

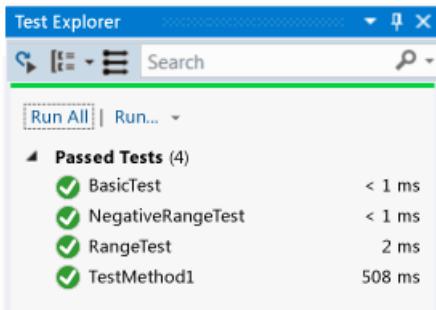
- Set a breakpoint at the start of the SquareRoot function.
- On the shortcut menu of the failed test, choose **Debug Selected Tests**.

When the run stops at the breakpoint, step through the code.

5. Insert code in the function that you are developing:

```
#include <stdexcept>
...
double CRootFinder::SquareRoot(double v)
{
    // Validate parameter:
    if (v < 0.0)
    {
        throw std::out_of_range("Can't do square roots of negatives");
    }
```

6. All tests now pass.



TIP

If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution with the  toggle button on the toolbar. This can noticeably reduce the time taken to run all the tests.

TIP

If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution in the settings menu of the toolbar. This can noticeably reduce the time taken to run all the tests.

Refactor the code without changing tests

1. Simplify the central calculation in the SquareRoot function:

```
// old code:
// result = result - (result*result - v)/(2*result);
// new code:
result = (result + v/result)/2.0;
```

2. Build the solution and choose **Run All**, to make sure that you have not introduced an error.

TIP

A good set of unit tests gives confidence that you have not introduced bugs when you change the code.

Keep refactoring separate from other changes.

Next steps

- **Isolation.** Most DLLs are dependent on other subsystems such as databases and other DLLs. These other components are often developed in parallel. To allow unit testing to be performed while the other components are not yet available, you have to substitute mock or
- **Build Verification Tests.** You can have tests performed on your team's build server at set intervals. This ensures that bugs are not introduced when the work of several team members is integrated.
- **Checkin tests.** You can mandate that some tests are performed before each team member checks code into source control. Typically this is a subset of the complete set of build verification tests.

You can also mandate a minimum level of code coverage.

See also

- [Add unit tests to existing C++ applications](#)
- [Using Microsoft.VisualStudio.TestTools.CppUnitTestFramework](#)
- [Debug native code](#)
- [Walkthrough: Creating and using a dynamic link library \(C++\)](#)
- [Import and export](#)

Microsoft.VisualStudio.TestTools.CppUnitTestFramework API reference

10/1/2019 • 7 minutes to read • [Edit Online](#)

This topic lists the public members of the `Microsoft::VisualStudio::CppUnitTestFramework` namespace. Use these APIs to write C++ unit tests based on the Microsoft Native Unit Test Framework. There is a [Usage Example](#) at the end of the topic.

The header and lib files are located under *<Visual Studio installation folder>\VC\Auxiliary\VS\UnitTest*.

Header and lib paths are automatically configured in a Native Test project.

In this topic

[CppUnitTest.h](#)

- [Create test classes and methods](#)
- [Initialize and cleanup](#)
 - [Test methods](#)
 - [Test classes](#)
 - [Test modules](#)
- [Create test attributes](#)
 - [Test method attributes](#)
 - [Test class attributes](#)
 - [Test module attributes](#)
 - [Pre-defined attributes](#)

[CppUnitTestAssert.h](#)

- [General Asserts](#)
 - [Are Equal](#)
 - [Are Not Equal](#)
 - [Are Same](#)
 - [Are Not Same](#)
 - [Is Null](#)
 - [Is Not Null](#)
 - [Is True](#)
 - [Is False](#)
 - [Fail](#)

- Windows Runtime Asserts

- Are Equal
- Are Same
- Are Not Equal
- Are Not Same
- Is Null
- Is Not Null

- Exception Asserts

- Expect Exception

CppUnitTestLogger.h

- Logger
- Write Message

- Usage Example

CppUnitTest.h

Create test classes and methods

```
TEST_CLASS(className)
```

Required for each class containing test methods. Identifies *className* as a test class. `TEST_CLASS` must be declared at namescope scope.

```
TEST_METHOD(methodName)
{
    // test method body
}
```

Defines *methodName* as a test method. `TEST_METHOD` must be declared in the scope of the method's class.

Initialize and cleanup

Test methods

```
TEST_METHOD_INITIALIZE(methodName)
{
    // method initialization code
}
```

Defines *methodName* as a method that runs before each test method is run. `TEST_METHOD_INITIALIZE` can only be defined once in a test class and must be defined in the scope of the test class.

```
TEST_METHOD_CLEANUP(methodName)
{
    // test method cleanup code
}
```

Defines *methodName* as a method that runs after each test method is run. `TEST_METHOD_CLEANUP` can only be

defined once in a test class and must be defined in the scope of the test class.

Test classes

```
TEST_CLASS_INITIALIZE(methodName)
{
    // test class initialization code
}
```

Defines *methodName* as a method that runs before each test class is created. `TEST_CLASS_INITIALIZE` can only be defined once in a test class and must be defined in the scope of the test class.

```
TEST_CLASS_CLEANUP(methodName)
{
    // test class cleanup code
}
```

Defines *methodName* as a method that runs after each test class is created. `TEST_CLASS_CLEANUP` can only be defined once in a test class and must be defined in the scope of the test class.

Test modules

```
TEST_MODULE_INITIALIZE(methodName)
{
    // module initialization code
}
```

Defines the method *methodName* that runs when a module is loaded. `TEST_MODULE_INITIALIZE` can only be defined once in a test module and must be declared at namespace scope.

```
TEST_MODULE_CLEANUP(methodName)
```

Defines the method *methodName* that runs when a module is unloaded. `TEST_MODULE_CLEANUP` can only be defined once in a test module and must be declared at namespace scope.

Create test attributes

Test method attributes

```
BEGIN_TEST_METHOD_ATTRIBUTE(testMethodName)
    TEST_METHOD_ATTRIBUTE(attributeName, attributeName)
    ...
END_TEST_METHOD_ATTRIBUTE()
```

Adds the attributes defined with one or more `TEST_METHOD_ATTRIBUTE` macros to the test method *testMethodName*.

A `TEST_METHOD_ATTRIBUTE` macro defines an attribute with the name *attributeName* and the value *attributeValue*.

Test class attributes

```
BEGIN_TEST_CLASS_ATTRIBUTE(testClassName)
    TEST_CLASS_ATTRIBUTE(attributeName, attributeName)
    ...
END_TEST_CLASS_ATTRIBUTE()
```

Adds the attributes defined with one or more `TEST_CLASS_ATTRIBUTE` macros to the test class *testClassName*.

A `TEST_CLASS_ATTRIBUTE` macro defines an attribute with the name *attributeName* and the value *attributeValue*.

Test module attributes

```
BEGIN_TEST_MODULE_ATTRIBUTE(testModuleName)
    TEST_MODULE_ATTRIBUTE(attributeName, attributeValue)
    ...
END_TEST_MODULE_ATTRIBUTE()
```

Adds the attributes defined with one or more `TEST_MODULE_ATTRIBUTE` macros to the test module `testModuleName`.

A `TEST_MODULE_ATTRIBUTE` macro defines an attribute with the name `attributeName` and the value `attributeValue`.

Pre-defined attributes

These pre-defined attribute macros are provided as a convenience for common cases. They can be substituted for the macro `TEST_METHOD_ATTRIBUTE` described above.

```
TEST_OWNER(ownerAlias)
```

Defines a `TEST_METHOD_ATTRIBUTE` with the name `Owner` and the attribute value of `ownerAlias`.

```
TEST_DESCRIPTION(description)
```

Defines a `TEST_METHOD_ATTRIBUTE` with the name `Description` and the attribute value of `description`.

```
TEST_PRIORITY(priority)
```

Defines a `TEST_METHOD_ATTRIBUTE` with the name `Priority` and the attribute value of `priority`.

```
TEST_WORKITEM(workitem)
```

Defines a `TEST_METHOD_ATTRIBUTE` with the name `WorkItem` and the attribute value of `workItem`.

```
TEST_IGNORE()
```

Defines a `TEST_METHOD_ATTRIBUTE` with the name `Ignore` and the attribute value of `true`.

CppUnitTestFixture.h

General Asserts

Are Equal

Verify that two objects are equal

```
template<typename T>
static void Assert::AreEqual(
    const T& expected,
    const T& actual,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Verify that two doubles are equal

```
static void Assert::AreEqual(
    double expected,
    double actual,
    double tolerance,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Verify that two floats are equal

```
static void Assert::AreEqual(
    float expected,
    float actual,
    float tolerance,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Verify that two char* strings are equal

```
static void Assert::AreEqual(
    const char* expected,
    const char* actual,
    bool ignoreCase = false,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Verify that two w_char* strings are equal

```
static void Assert::AreEqual(
    const wchar_t* expected,
    const wchar_t* actual,
    bool ignoreCase = false,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Are Not Equal

Verify that two doubles are not equal

```
static void Assert::AreNotEqual(
    double notExpected,
    double actual,
    double tolerance,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Verify that two floats are not equal

```
static void Assert::AreNotEqual(
    float notExpected,
    float actual,
    float tolerance,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Verify that two char* strings are not equal

```
static void Assert::AreNotEqual(
    const char* notExpected,
    const char* actual,
    bool ignoreCase = false,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Verify that two w_char* strings are not equal

```
static void Assert::AreNotEqual(
    const wchar_t* notExpected,
    const wchar_t* actual,
    bool ignoreCase = false,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Verify that two references are not equal based on operator==.

```
template<typename T>
static void Assert::AreNotEqual(
    const T& notExpected,
    const T& actual,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Are Same

Verify that two references refer to the same object instance (identity).

```
template<typename T>
static void Assert::AreSame(
    const T& expected,
    const T& actual,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Are Not Same

Verify that two references do not refer to the same object instance (identity).

```
template<typename T>
static void Assert::AreNotSame (
    const T& notExpected,
    const T& actual,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Is Null

Verify that a pointer is NULL.

```
template<typename T>
static void Assert::IsNull(
    const T* actual,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Is Not Null

Verify that a pointer is not NULL

```
template<typename T>
static void Assert::IsNotNull(
    const T* actual,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Is True

Verify that a condition is true

```
static void Assert::IsTrue(
    bool condition,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Is False

Verify that a condition is false

```
static void Assert::IsFalse(
    bool condition,
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Fail

Force the test case result to be failed

```
static void Assert::Fail(
    const wchar_t* message = NULL,
    const __LineInfo* pLineInfo = NULL)
```

Windows Runtime Asserts

Are Equal

Verifies that two Windows Runtime pointers are equal.

```
template<typename T>
static void Assert::AreEqual(
    T^ expected,
    T^ actual,
    Platform::String^ message = nullptr,
    const __LineInfo* pLineInfo= nullptr)
```

Verifies that two Platform::String^ strings are equal.

```
template<typename T>
static void Assert::AreEqual(
    T^ expected,
    T^ actual,
    Platform::String^ message= nullptr,
    const __LineInfo* pLineInfo= nullptr)
```

Are Same

Verifies that two Windows Runtime references reference the same object.

```
template<typename T>
static void Assert::AreSame(
    T% expected,
    T% actual,
    Platform::String^ message= nullptr,
    const __LineInfo* pLineInfo= nullptr)
```

Are Not Equal

Verifies that two Windows Runtime pointers are not equal.

```
template<typename T>
static void Assert::AreNotEqual(
    T^ notExpected,
    T^ actual,
    Platform::String^ message = nullptr,
    const __LineInfo* pLineInfo= nullptr)
```

Verifies that two Platform::String^ strings are not equal.

```
static void Assert::AreNotEqual(
    Platform::String^ notExpected,
    Platform::String^ actual,
    bool ignoreCase = false,
    Platform::String^ message= nullptr,
    const __LineInfo* pLineInfo= nullptr)
```

Are Not Same

Verifies that two Windows Runtime references do not reference the same object.

```
template<typename T>
static void Assert::AreNotSame(
    T% notExpected,
    T% actual,
    Platform::String^ message= nullptr,
    const __LineInfo* pLineInfo= nullptr)
```

Is Null

Verifies that a Windows Runtime pointer is a nullptr.

```
template<typename T>
static void Assert::IsNull(
    T^ actual,
    Platform::String^ message = nullptr,
    const __LineInfo* pLineInfo= nullptr)
```

Is Not Null

Verifies that a Windows Runtime pointer is not a nullptr.

```
template<typename T>
static void Assert::IsNotNull(
    T^ actual,
    Platform::String^ message= nullptr,
    const __LineInfo* pLineInfo= nullptr)
```

Exception Asserts

Expect Exception

Verify that a function raises an exception:

```
template<typename _EXPECTEDEXCEPTION, typename _FUNCTOR>
static void Assert::ExpectException(
    _FUNCTOR functor,
    const wchar_t* message= NULL,
    const __LineInfo* pLineInfo= NULL)
```

Verify that a function raises an exception:

```
template<typename _EXPECTEDEXCEPTION, typename _RETURNTYPE>
static void Assert::ExpectException(
    _RETURNTYPE (*func)(),
    const wchar_t* message= NULL,
    const __LineInfo* pLineInfo = NULL)
```

CppUnitTestLogger.h

Logger

The Logger class contains static methods to write to the **Output Window**.

Write Message

Write a string to the **Output Window**

```
static void Logger::WriteMessage(const wchar_t* message)
```

```
static void Logger::WriteMessage(const char* message)
```

Example

This code is an example of VSCppUnit usage. It includes examples of attribute metadata, fixtures, unit tests with assertions, and custom logging.

```

// USAGE EXAMPLE

#include <CppUnitTest.h>

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

BEGIN_TEST_MODULE_ATTRIBUTE()
    TEST_MODULE_ATTRIBUTE(L"Date", L"2010/6/12")
END_TEST_MODULE_ATTRIBUTE()

TEST_MODULE_INITIALIZE(ModuleInitialize)
{
    Logger::WriteMessage("In Module Initialize");
}

TEST_MODULE_CLEANUP(ModuleCleanup)
{
    Logger::WriteMessage("In Module Cleanup");
}

TEST_CLASS(Class1)
{
public:

    Class1()
    {
        Logger::WriteMessage("In Class1");
    }

    ~Class1()
    {
        Logger::WriteMessage("In ~Class1");
    }

    TEST_CLASS_INITIALIZE(ClassInitialize)
    {
        Logger::WriteMessage("In Class Initialize");
    }

    TEST_CLASS_CLEANUP(ClassCleanup)
    {
        Logger::WriteMessage("In Class Cleanup");
    }

    BEGIN_TEST_METHOD_ATTRIBUTE(Method1)
        TEST_OWNER(L"OwnerName")
        TEST_PRIORITY(1)
    END_TEST_METHOD_ATTRIBUTE()

    TEST_METHOD(Method1)
    {
        Logger::WriteMessage("In Method1");
        Assert::AreEqual(0, 0);
    }

    TEST_METHOD(Method2)
    {
        Assert::Fail(L"Fail");
    }
};

```

See also

- [Unit test your code](#)

- Write unit tests for C/C++

How to test a C++ DLL

10/21/2019 • 8 minutes to read • [Edit Online](#)

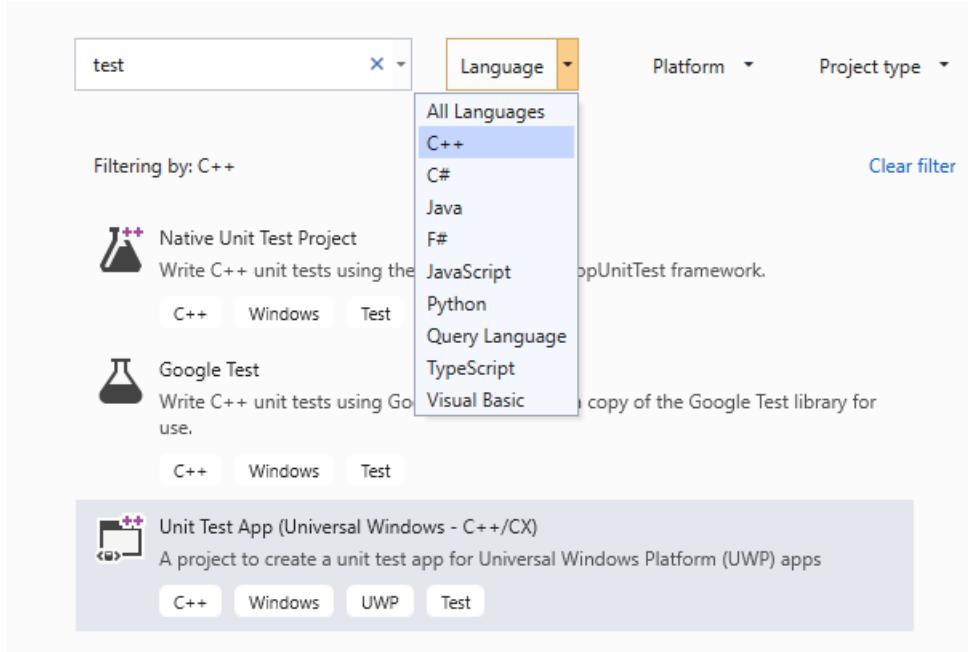
This topic describes one way to create unit tests for a C++ DLL for Universal Windows Platform (UWP) apps with the Microsoft Test Framework for C++. The RooterLib DLL demonstrates vague memories of limit theory from the calculus by implementing a function that calculates an estimate of the square root of a given number. The DLL might then be included in a UWP app that shows a user the fun things that can be done with math.

This topic shows you how to use unit testing as the first step in development. In this approach, you first write a test method that verifies a specific behavior in the system that you are testing and then you write the code that passes the test. By making changes in the order of the following procedures, you can reverse this strategy to first write the code that you want to test and then write the unit tests.

This topic also creates a single Visual Studio solution and separate projects for the unit tests and the DLL that you want to test. You can also include the unit tests directly in the DLL project, or you can create separate solutions for the unit tests and the .DLL. See [Adding unit tests to existing C++ applications](#) for tips on which structure to use.

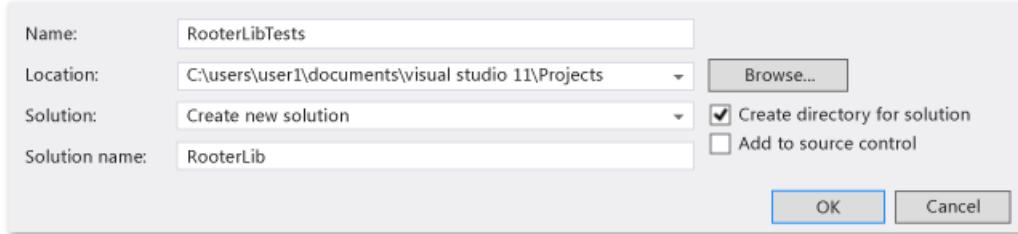
Create the solution and the unit test project

Start by creating a new test project. On the **File** menu, choose **New > Project**. In the **Create a New Project** dialog, type "test" into the search box and then set **Language** to C++. Then choose **Unit Test App (Universal Windows)** from the list of project templates.

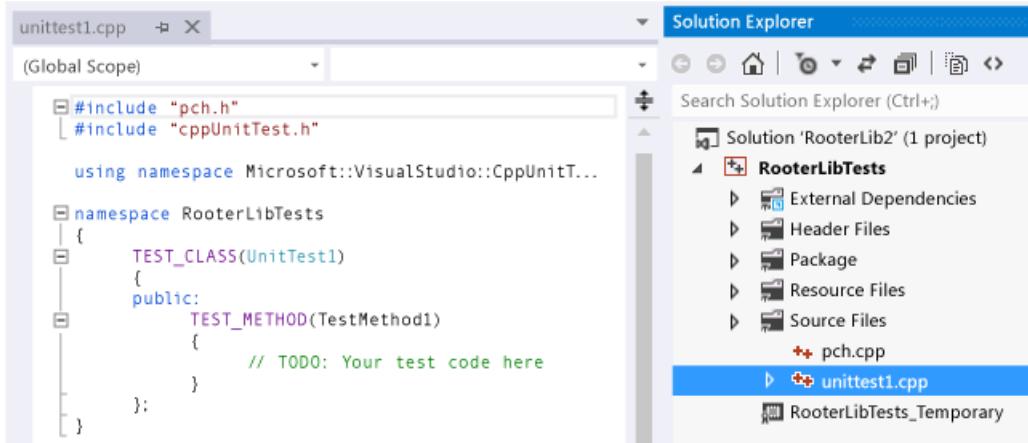


Start by creating a new test project. On the **File** menu, choose **New > Project**. In the **New Project** dialog, expand **Installed > Visual C++** and choose **Windows Universal**. Then choose **Unit Test App (Universal Windows)** from the list of project templates.

1. In the New Project dialog, expand **Installed > Visual C++** and choose **Windows Universal**. Then choose **Unit Test App (Universal Windows)** from the list of project templates.
2. Name the project `RooterLibTests`; specify the location; name the solution `RooterLib`; and make sure **Create directory for solution** is checked.



- In the new project, open **unittest1.cpp**.



Note that:

- Each test is defined by using `TEST_METHOD(YourTestMethod){...}`.

You do not have to write a conventional function signature. The signature is created by the macro `TEST_METHOD`. The macro generates an instance function that returns void. It also generates a static function that returns information about the test method. This information allows the test explorer to find the method.

- Test methods are grouped into classes by using `TEST_CLASS(YourClassName){...}`.

When the tests are run, an instance of each test class is created. The test methods are called in an unspecified order. You can define special methods that are invoked before and after each module, class, or method. For more information, see [Using Microsoft.VisualStudio.TestTools.CppUnitTestFramework](#).

Verify that the tests run in Test Explorer

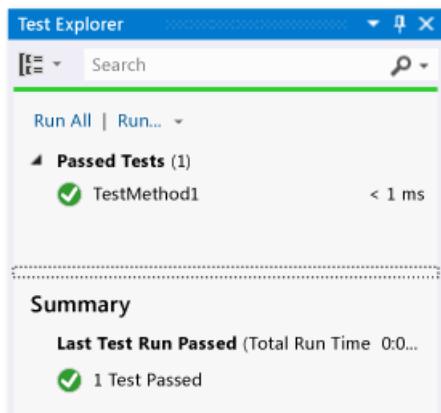
- Insert some test code:

```
TEST_METHOD(TestMethod1)
{
    Assert::AreEqual(1,1);
}
```

Notice that the `Assert` class provides several static methods that you can use to verify results in test methods.

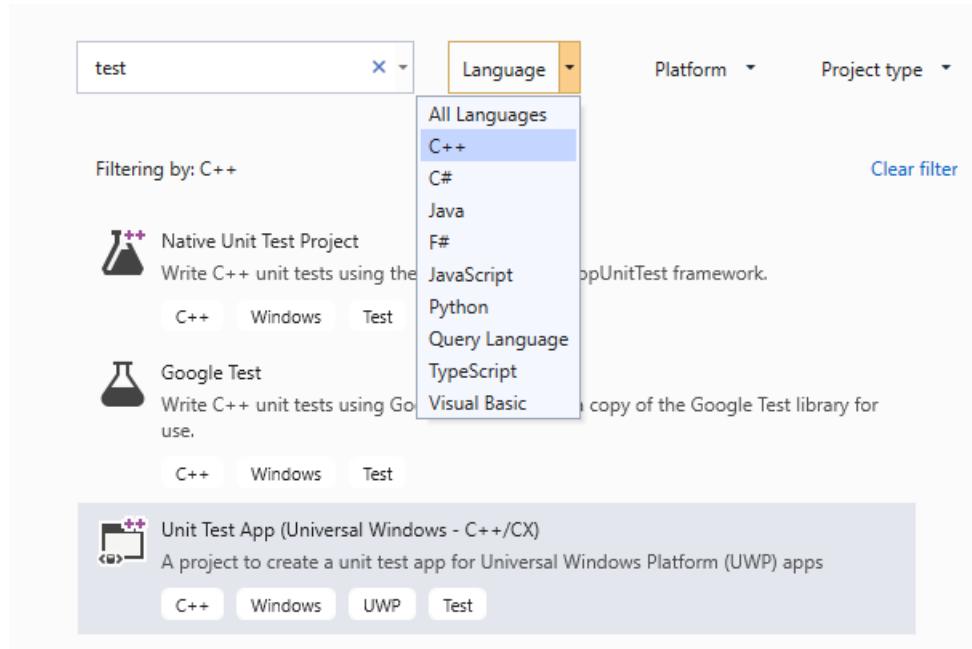
- On the **Test** menu, choose **Run** and then choose **Run All**.

The test project builds and runs. The **Test Explorer** window appears, and the test is listed under **Passed Tests**. The **Summary** pane at the bottom of the window provides additional details about the selected test.

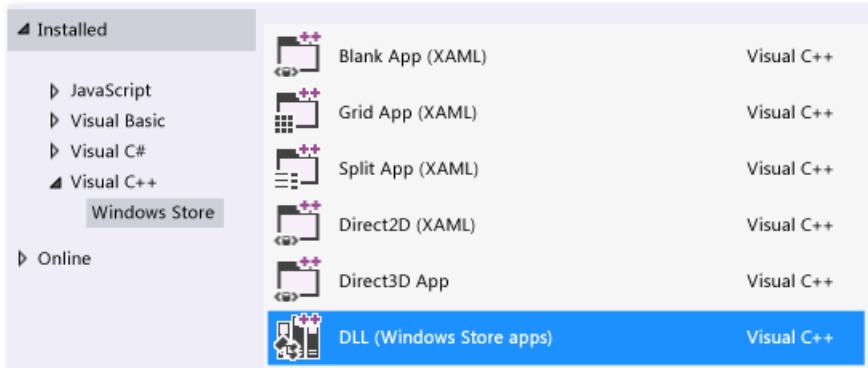


Add the DLL project to the solution

In **Solution Explorer**, choose the solution name. From the shortcut menu, choose **Add**, and then **New Project**. In the **Add a New Project** dialog, set **Language** to C++ and type "DLL" into the search box. From the results list, choose **Unit Test App (Universal Windows - C++/CX)**.



In **Solution Explorer**, choose the solution name. From the shortcut menu, choose **Add**, and then **New Project**.



1. In the **Add New Project** dialog box, choose **DLL (UWP apps)**.
2. Add the following code to the *RooterLib.h* file:

```

// The following ifdef block is the standard way of creating macros which make exporting
// from a DLL simpler. All files within this DLL are compiled with the ROOTERLIB_EXPORTS
// symbol defined on the command line. This symbol should not be defined on any project
// that uses this DLL. This way any other project whose source files include this file see
// ROOTERLIB_API functions as being imported from a DLL, whereas this DLL sees symbols
// defined with this macro as being exported.
#ifndef ROOTERLIB_EXPORTS
#define ROOTERLIB_API __declspec(dllexport)
#else
#define ROOTERLIB_API __declspec(dllimport)
#endif //ROOTERLIB_EXPORTS

class ROOTERLIB_API CRooterLib {
public:
    CRooterLib(void);
    double SquareRoot(double v);
};

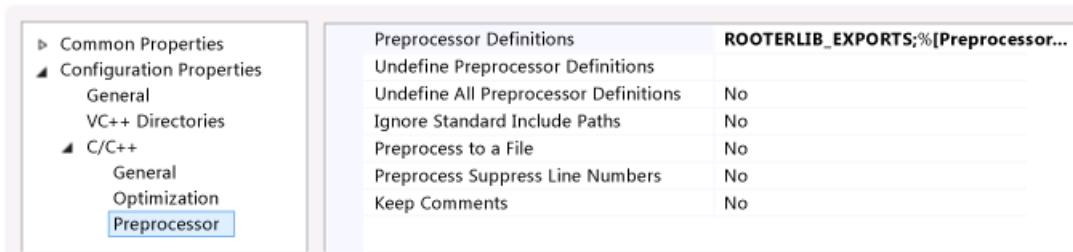
```

The comments explain the ifdef block not only to the developer of the dll, but to anyone who references the DLL in their project. You can add the ROOTERLIB_EXPORTS symbol to the command line by using the project properties of the DLL.

The `CRooterLib` class declares a constructor and the `SquareRoot` estimator method.

3. Add the ROOTERLIB_EXPORTS symbol to the command line.

- In **Solution Explorer**, choose the **RooterLib** project, and then choose **Properties** from the shortcut menu.



- In the **RooterLib Property Page** dialog box, expand **Configuration Properties**, expand **C++** and choose **Preprocessor**.
- Choose **<Edit...>** from the **Preprocessor Definitions** list, and then add `ROOTERLIB_EXPORTS` in the **Preprocessor Definitions** dialog box.

4. Add minimal implementations of the declared functions. Open *RooterLib.cpp* and add the following code:

```

// constructor
CRooterLib::CRooterLib()
{
}

// Find the square root of a number.
double CRooterLib::SquareRoot(double v)
{
    return 0.0;
}

```

Make the dll functions visible to the test code

- Add RooterLib to the RooterLibTests project.

- a. In **Solution Explorer**, choose the **RooterLibTests** project and then choose **Add > Reference** on the shortcut menu.
 - b. In the **Add Reference** dialog box, choose **Projects**. Then select the **RouterLib** item.
2. Include the RooterLib header file in *unittest1.cpp*.
- a. Open *unittest1.cpp*.
 - b. Add this code to below the `#include "CppUnitTest.h"` line:

```
#include "..\RooterLib\RooterLib.h"
```

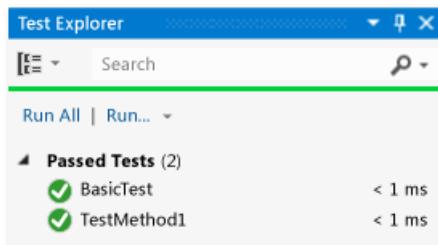
3. Add a test that uses the imported function. Add the following code to *unittest1.cpp*:

```
TEST_METHOD(BasicTest)
{
    CRooterLib rooter;
    Assert::AreEqual(
        // Expected value:
        0.0,
        // Actual value:
        rooter.SquareRoot(0.0),
        // Tolerance:
        0.01,
        // Message:
        L"Basic test failed",
        // Line number - used if there is no PDB file:
        LINE_INFO());
}
```

4. Build the solution.

The new test appears in **Test Explorer** in the **Not Run Tests** node.

5. In **Test Explorer**, choose **Run All**.



You have set up the test and the code projects, and verified that you can run tests that run functions in the code project. Now you can begin to write real tests and code.

Iteratively augment the tests and make them pass

1. Add a new test:

```

TEST_METHOD(RangeTest)
{
    CRooterLib rooter;
    for (double v = 1e-6; v < 1e6; v = v * 3.2)
    {
        double expected = v;
        double actual = rooter.SquareRoot(v*v);
        double tolerance = expected/1000;
        Assert::AreEqual(expected, actual, tolerance);
    }
};

```

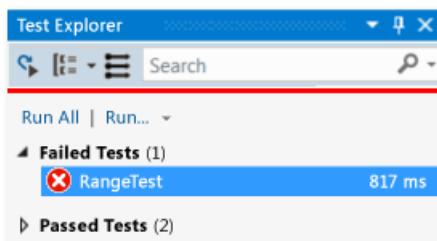
TIP

We recommend that you do not change tests that have passed. Instead, add a new test, update the code so that the test passes, and then add another test, and so on.

When your users change their requirements, disable the tests that are no longer correct. Write new tests and make them work one at a time, in the same incremental manner.

2. In **Test Explorer**, choose **Run All**.

3. The test fails.



TIP

Verify that each test fails immediately after you have written it. This helps you avoid the easy mistake of writing a test that never fails.

4. Enhance the code under test so that the new test passes. Add the following to *RooterLib.cpp*:

```

#include <math.h>
...
// Find the square root of a number.
double CRooterLib::SquareRoot(double v)
{
    double result = v;
    double diff = v;
    while (diff > result/1000)
    {
        double oldResult = result;
        result = result - (result*result - v)/(2*result);
        diff = abs (oldResult - result);
    }
    return result;
}

```

5. Build the solution and then in **Test Explorer**, choose **Run All**.

Both tests pass.

TIP

Develop code by adding tests one at a time. Make sure that all the tests pass after each iteration.

Debug a failing test

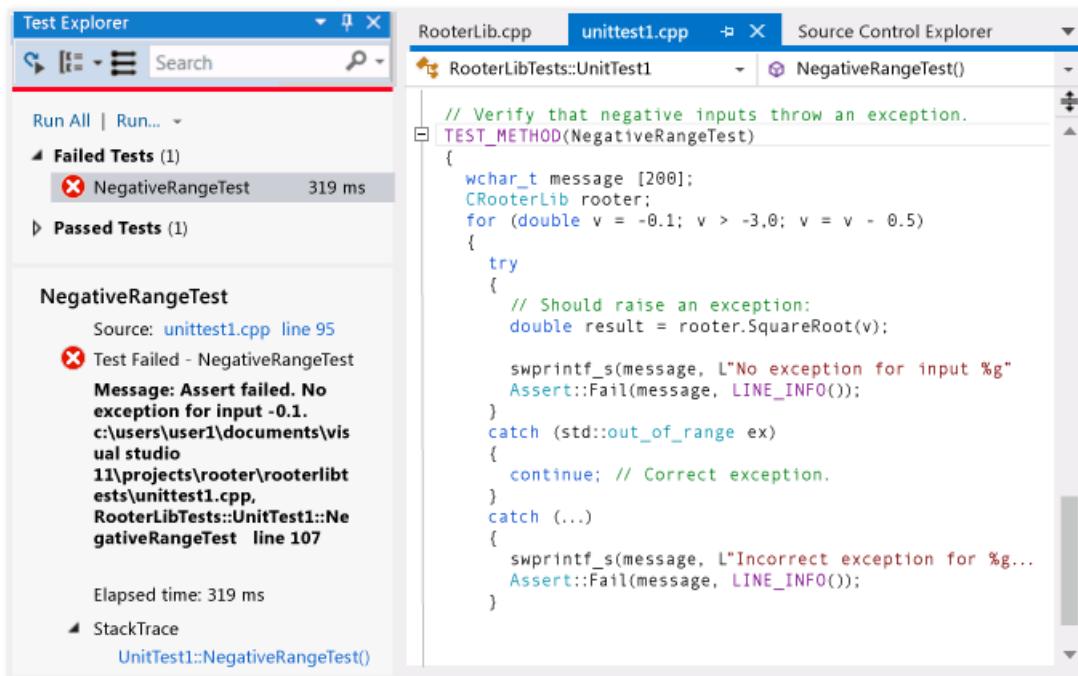
1. Add another test to *unittest1.cpp*:

```
// Verify that negative inputs throw an exception.
TEST_METHOD(NegativeRangeTest)
{
    wchar_t message[200];
    CRooterLib rooter;
    for (double v = -0.1; v > -3.0; v = v - 0.5)
    {
        try
        {
            // Should raise an exception:
            double result = rooter.SquareRoot(v);

            swprintf_s(message, L"No exception for input %g", v);
            Assert::Fail(message, LINE_INFO());
        }
        catch (std::out_of_range ex)
        {
            continue; // Correct exception.
        }
        catch (...)
        {
            swprintf_s(message, L"Incorrect exception for %g", v);
            Assert::Fail(message, LINE_INFO());
        }
    }
}
```

2. In **Test Explorer**, choose **Run All**.

The test fails. Choose the test name in **Test Explorer**. The failed assertion is highlighted. The failure message is visible in the detail pane of **Test Explorer**.



3. To see why the test fails, step through the function:

- Set a breakpoint at the start of the `SquareRoot` function.
- On the shortcut menu of the failed test, choose **Debug Selected Tests**.

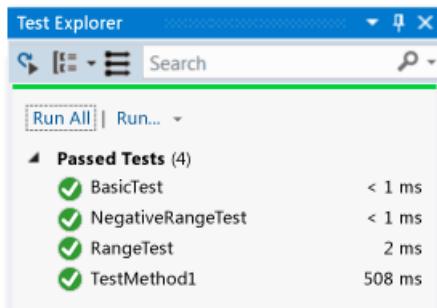
When the run stops at the breakpoint, step through the code.

- Add code to `RooterLib.cpp` to catch the exception:

```
#include <stdexcept>
...
double CRooterLib::SquareRoot(double v)
{
    //Validate the input parameter:
    if (v < 0.0)
    {
        throw std::out_of_range("Can't do square roots of negatives");
    }
    ...
}
```

- In **Test Explorer**, choose **Run All** to test the corrected method and make sure that you haven't introduced a regression.

All tests now pass.



Refactor the code without changing tests

1. Simplify the central calculation in the `SquareRoot` function:

```
// old code  
//result = result - (result*result - v)/(2*result);  
// new code  
result = (result + v/result) / 2.0;
```

2. Choose **Run All** to test the refactored method and make sure that you haven't introduced a regression.

TIP

A stable set of good unit tests gives confidence that you have not introduced bugs when you change the code.

Keep refactoring separate from other changes.

Unit test C# code

1/1/2020 • 5 minutes to read • [Edit Online](#)

This article describes one way to create unit tests for a C# class in a UWP app.

The **Rooter** class, which is the class under test, implements a function that calculates an estimate of the square root of a given number.

This article demonstrates *test-driven development*. In this approach, you first write a test that verifies a specific behavior in the system that you're testing, and then you write the code that passes the test.

Create the solution and the unit test project

1. On the **File** menu, choose **New > Project**.
2. Search for and select the **Blank App (Universal Windows)** project template.
3. Name the project **Maths**.
4. In **Solution Explorer**, right-click on the solution and choose **Add > New Project**.
5. Search for and select the **Unit Test App (Universal Windows)** project template.
6. Name the test project **RooterTests**.

Verify that the tests run in Test Explorer

1. Insert some test code into **TestMethod1** in the *UnitTest.cs* file:

```
[TestMethod]
public void TestMethod1()
{
    Assert.AreEqual(0, 0);
}
```

The [Assert](#) class provides several static methods that you can use to verify results in test methods.

2. On the **Test** menu, choose **Run > All Tests**.
2. On the **Test** menu, choose **Run All Tests**.

The test project builds and runs. Be patient because it may take a little while. The **Test Explorer** window appears, and the test is listed under **Passed Tests**. The **Summary** pane at the bottom of the window provides additional details about the selected test.

Add the Rooter class to the Maths project

1. In **Solution Explorer**, right-click on the **Maths** project, and then choose **Add > Class**.
2. Name the class file *Rooter.cs*.
3. Add the following code to the **Rooter** class *Rooter.cs* file:

```

public Rooter()
{
}

// estimate the square root of a number
public double SquareRoot(double x)
{
    return 0.0;
}

```

The **Rooter** class declares a constructor and the **SquareRoot** estimator method. The **SquareRoot** method is only a minimal implementation, just enough to test the basic structure of the testing setup.

4. Add the `public` keyword to the **Rooter** class declaration, so the test code can access it.

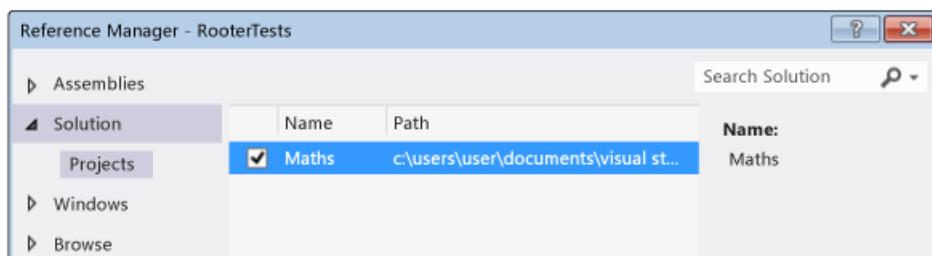
```

public class Rooter

```

Add a project reference

1. Add a reference from the **RooterTests** project to the **Maths** app.
 - a. In **Solution Explorer**, right-click on the **RooterTests** project, and then choose **Add > Reference**.
 - b. In the **Add Reference - RooterTests** dialog box, expand **Solution** and choose **Projects**. Select the **Maths** project.



2. Add a `using` statement to the *UnitTest.cs* file:
 - a. Open *UnitTest.cs*.
 - b. Add this code below the `using Microsoft.VisualStudio.TestTools.UnitTesting;` line:

```

using Maths;

```

3. Add a test that uses the **Rooter** function. Add the following code to *UnitTest.cs*:

```

[TestMethod]
public void BasicTest()
{
    Maths.Rooter rooter = new Rooter();
    double expected = 0.0;
    double actual = rooter.SquareRoot(expected * expected);
    double tolerance = .001;
    Assert.AreEqual(expected, actual, tolerance);
}

```

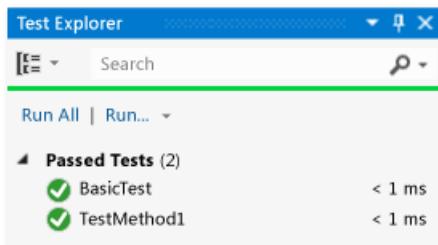
The new test appears in **Test Explorer** in the **Not Run Tests** node.

4. To avoid a "Payload contains two or more files with the same destination path" error, in **Solution Explorer**,

expand the **Properties** node under the **Maths** project, and then delete the *Default.rd.xml* file.

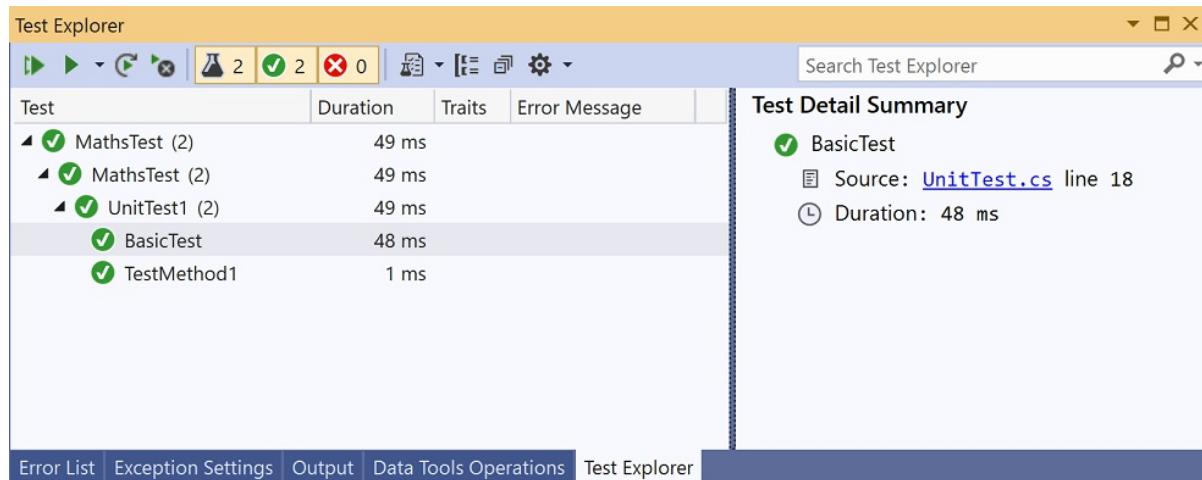
6. In **Test Explorer**, choose **Run All**.

The solution builds and the tests run and pass.



6. In **Test Explorer**, choose **Run All Tests**.

The solution builds and the tests run and pass.



You've set up the test and app projects and verified that you can run tests that call functions in the app project. Now you can begin to write real tests and code.

Iteratively augment the tests and make them pass

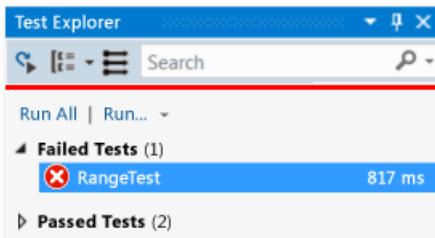
1. Add a new test called **RangeTest**:

```
[TestMethod]
public void RangeTest()
{
    Rooter rooter = new Rooter();
    for (double v = 1e-6; v < 1e6; v = v * 3.2)
    {
        double expected = v;
        double actual = rooter.SquareRoot(v*v);
        double tolerance = expected/1000;
        Assert.AreEqual(expected, actual, tolerance);
    }
}
```

TIP

We recommend that you do not change tests that have passed. Add a new test instead.

2. Run the **RangeTest** test and verify that it fails.



TIP

Immediately after you write a test, run it to verify that it fails. This helps you avoid the easy mistake of writing a test that never fails.

3. Enhance the code under test so that the new test passes. Change the **SquareRoot** function in *Rooter.cs* to this:

```
public double SquareRoot(double x)
{
    double estimate = x;
    double diff = x;
    while (diff > estimate / 1000)
    {
        double previousEstimate = estimate;
        estimate = estimate - (estimate * estimate - x) / (2 * estimate);
        diff = Math.Abs(previousEstimate - estimate);
    }
    return estimate;
}
```

4. In **Test Explorer**, choose **Run All**.

4. In **Test Explorer**, choose **Run All Tests**.

All three tests now pass.

TIP

Develop code by adding tests one at a time. Make sure that all the tests pass after each iteration.

Refactor the code

In this section, you refactor both app and test code, then rerun tests to make sure they still pass.

Simplify the square root estimation

1. Simplify the central calculation in the **SquareRoot** function by changing one line of code, as follows:

```
// Old code
//estimate = estimate - (estimate * estimate - x) / (2 * estimate);

// New code
estimate = (estimate + x/estimate) / 2.0;
```

2. Run all tests to make sure that you haven't introduced a regression. They should all pass.

TIP

A stable set of good unit tests gives confidence that you have not introduced bugs when you change the code.

Eliminate duplicated code

The **RangeTest** method hard codes the denominator of the *tolerance* variable that's passed to the **Assert** method. If you plan to add additional tests that use the same tolerance calculation, the use of a hard-coded value in multiple locations makes the code harder to maintain.

1. Add a private helper method to the **UnitTest1** class to calculate the tolerance value, and then call that method from **RangeTest**.

```
private double ToleranceHelper(double expected)
{
    return expected / 1000;
}

...

[TestMethod]
public void RangeTest()
{
    ...
    // Old code
    // double tolerance = expected/1000;

    // New code
    double tolerance = ToleranceHelper(expected);
}
...
```

2. Run **RangeTest** to make sure that it still passes.

TIP

If you add a helper method to a test class, and you don't want it to appear in **Test Explorer**, don't add the **TestMethodAttribute** attribute to the method.

See also

- [Walkthrough: Test-driven development using Test Explorer](#)

Walkthrough: Create and run unit tests for UWP apps

1/1/2020 • 2 minutes to read • [Edit Online](#)

Visual Studio includes support for unit testing Universal Windows Platform (UWP) apps. Visual Studio provides unit test project templates for C#, Visual Basic, and C++.

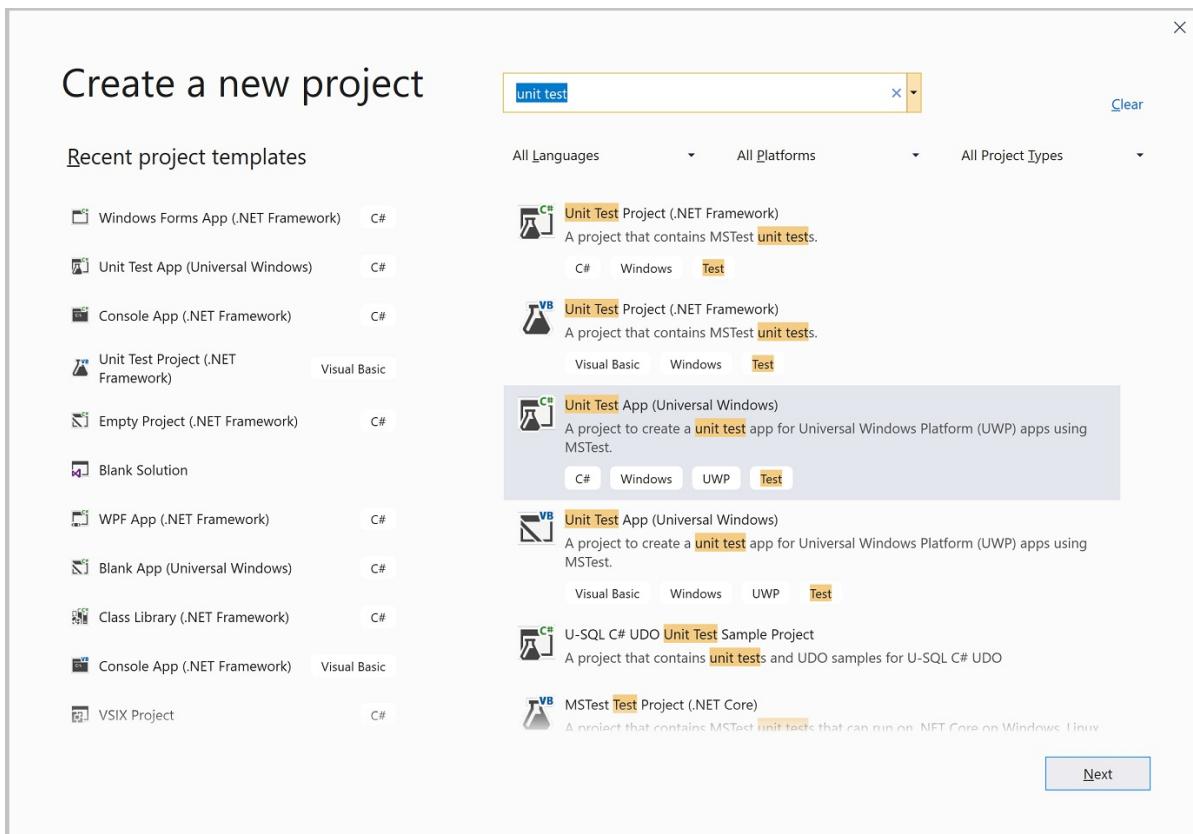
TIP

For more information about developing UWP apps, see [Getting started with UWP apps](#).

The following procedures describe the steps to create, run, and debug unit tests for a UWP app.

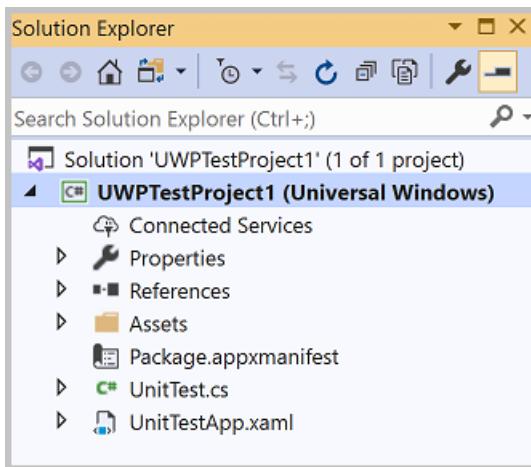
Create a unit test project for a UWP app

1. Open Visual Studio. On the start window, choose **Create a new project**.
2. In the search box of the **Create a new project** page, enter **unit test**.
The list of templates filters to those for unit testing.
3. Select the **Unit Test App (Universal Windows)** template for either C# or Visual Basic, and then select **Next**.



4. Optionally change the project or solution name and location, and then select **Create**.
5. Optionally change the target and minimum platform versions, and then select **OK**.

After completing these steps, the unit test project is created and displays in Solution Explorer.

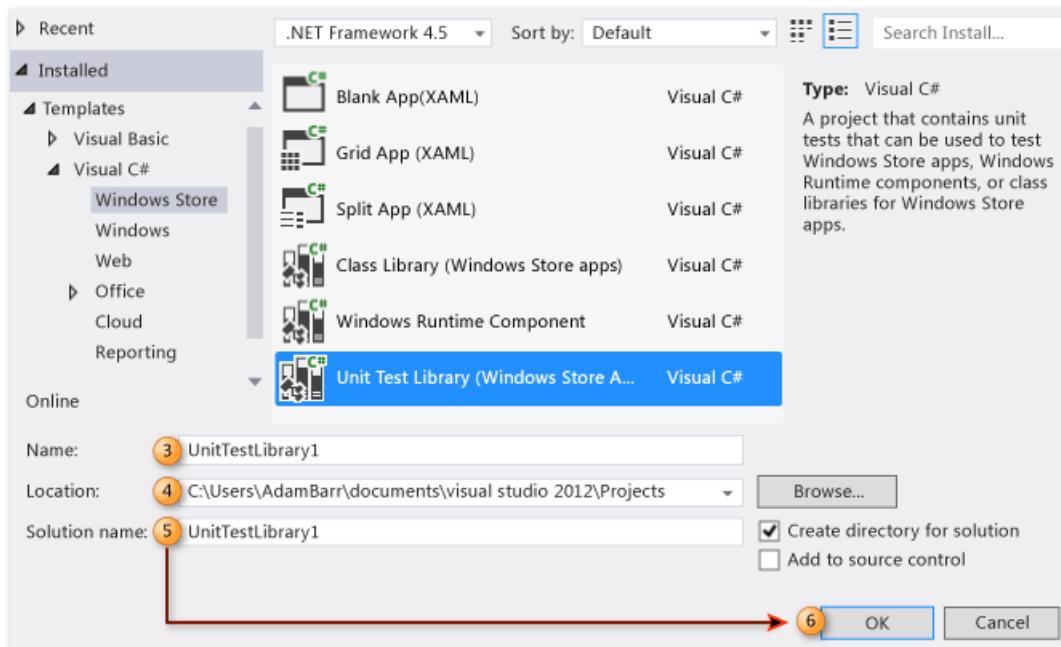


1. From the **File** menu, choose **New Project**.

The **New Project** dialog displays.

2. Under Templates, choose the programming language you want to create unit tests in, and then choose the associated Windows Universal unit test library. For example, choose **Visual C#**, then choose **Windows Universal**, and then choose **Unit Test Library (Universal Windows)**.
3. (Optional) In the **Name** textbox, enter the name you want to use for the project.
4. (Optional) Modify the path where you want to create the project by entering it in the **Location** textbox, or by choosing the **Browse** button.
5. (Optional) In the **Solution** name textbox, enter that name you want to use for your solution.

6. Leave the **Create directory for solution** option selected and choose the **OK** button.



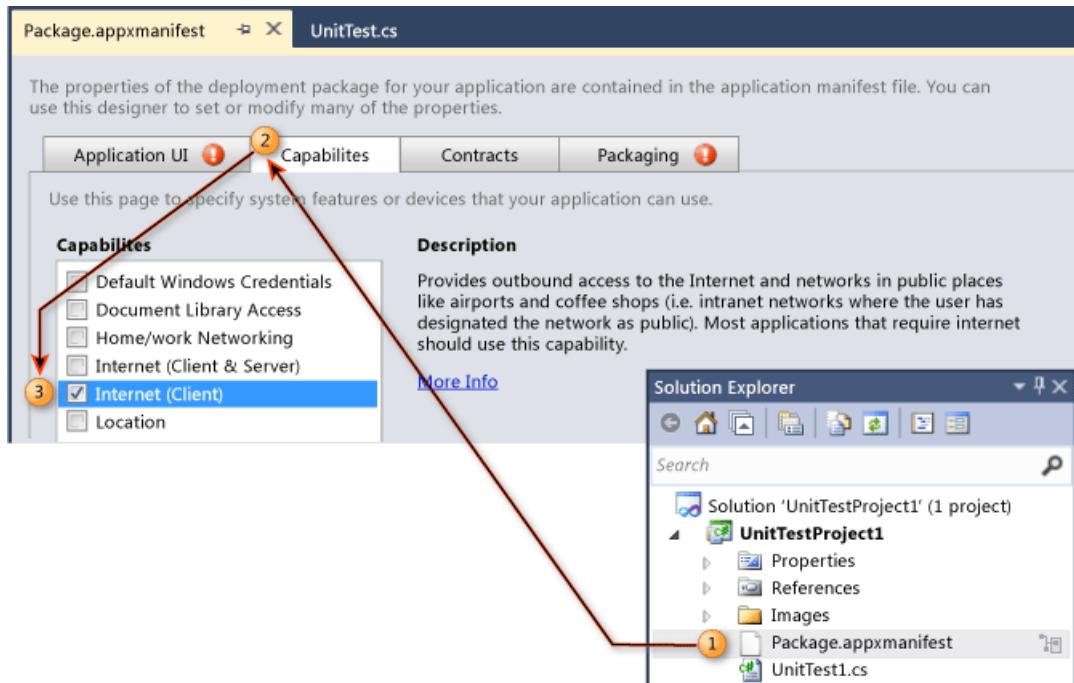
Solution Explorer is populated with the UWP unit test project, and the code editor displays the default unit test titled `UnitTest1`.

Edit the unit test project's UWP application manifest file

1. In **Solution Explorer**, right-click the *Package.appxmanifest* file and choose **Open**.
2. In the **Manifest Designer**, choose the **Capabilities** tab.
3. In the list under **Capabilities**, select the capabilities that you need your unit test and the code that it testing to have. For example, select the **Internet** checkbox if the unit test needs and the code it is testing need to have the capability to access the internet.

NOTE

The capabilities you select should only include capabilities that are necessary for the unit test to function correctly.



Code the unit test for a UWP app

In the code editor, edit the unit test, and add the asserts and logic required for your test.

Run unit tests

To build the solution and run the unit test using Test Explorer:

1. On the **Test** menu, choose **Windows**, and then choose **Test Explorer**.

2. From the **Build** menu, choose **Build Solution**.

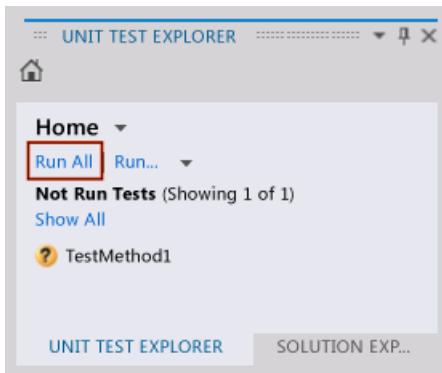
Your unit test is now shown in Test Explorer.

NOTE

You must build the solution to update the list of unit tests in Test Explorer.

3. In **Test Explorer**, choose the unit test you created.

4. Choose **Run All**.



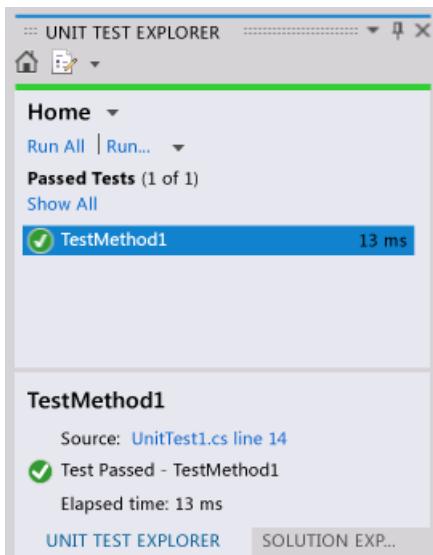
TIP

You can select one or more unit tests listed in Test Explorer, and then right-click and choose **Run Selected Tests**.

Additionally, you can choose to **Debug Selected Tests**, **Open Test**, and use the **Properties** option.



The unit test runs. Upon completion, Test Explorer displays the test status and elapsed time and provides a link to the source.



See also

- [Test UWP apps with Visual Studio](#)
- [Build and test a UWP app](#)

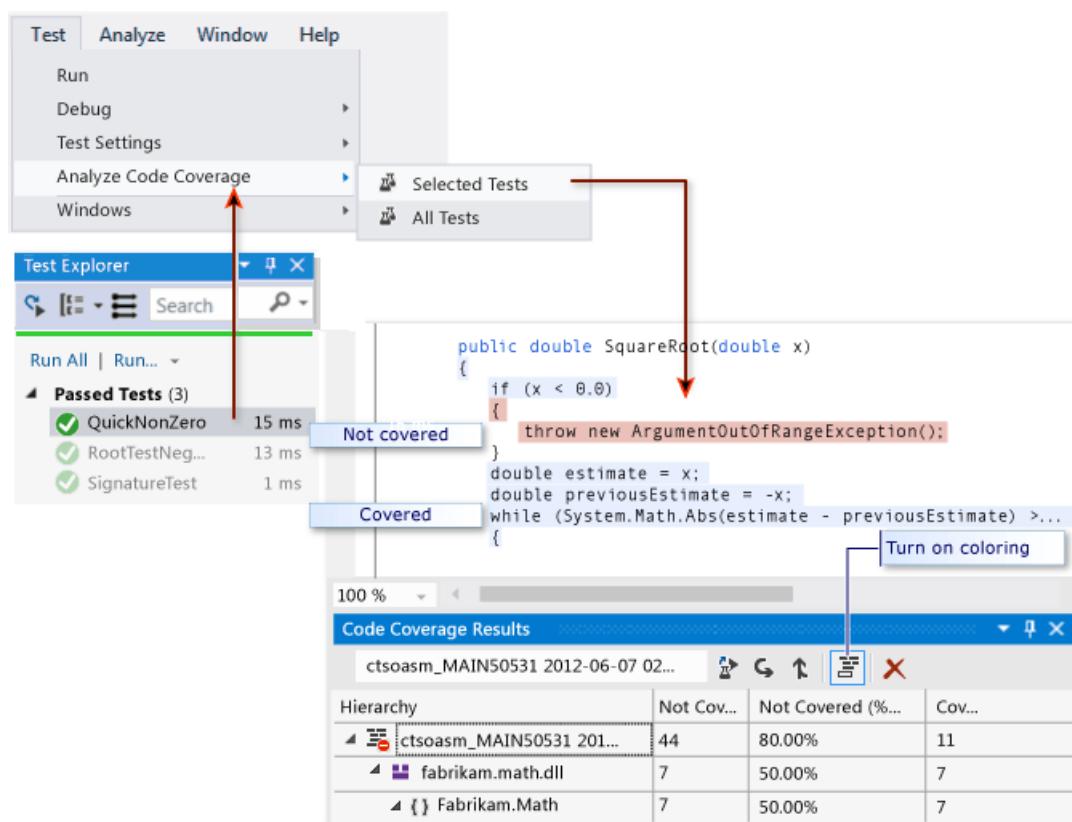
Use code coverage to determine how much code is being tested

1/1/2020 • 8 minutes to read • [Edit Online](#)

To determine what proportion of your project's code is actually being tested by coded tests such as unit tests, you can use the code coverage feature of Visual Studio. To guard effectively against bugs, your tests should exercise or 'cover' a large proportion of your code.

Code coverage analysis can be applied to both managed (CLI) and unmanaged (native) code.

Code coverage is an option when you run test methods using Test Explorer. The results table shows the percentage of the code that was run in each assembly, class, and method. In addition, the source editor shows you which code has been tested.

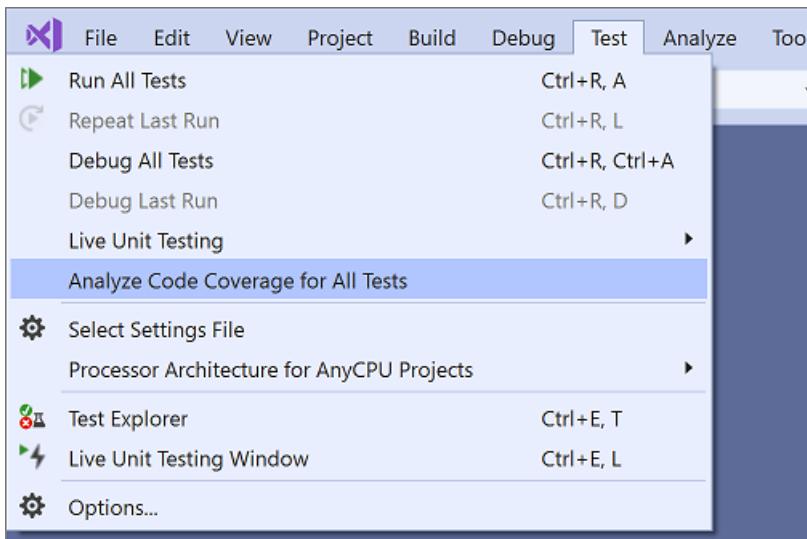


Requirements

The code coverage feature is available only in Visual Studio Enterprise edition.

Analyze code coverage

1. On the **Test** menu, choose **Analyze Code Coverage**.
1. On the **Test** menu, select **Analyze Code Coverage for All Tests**.

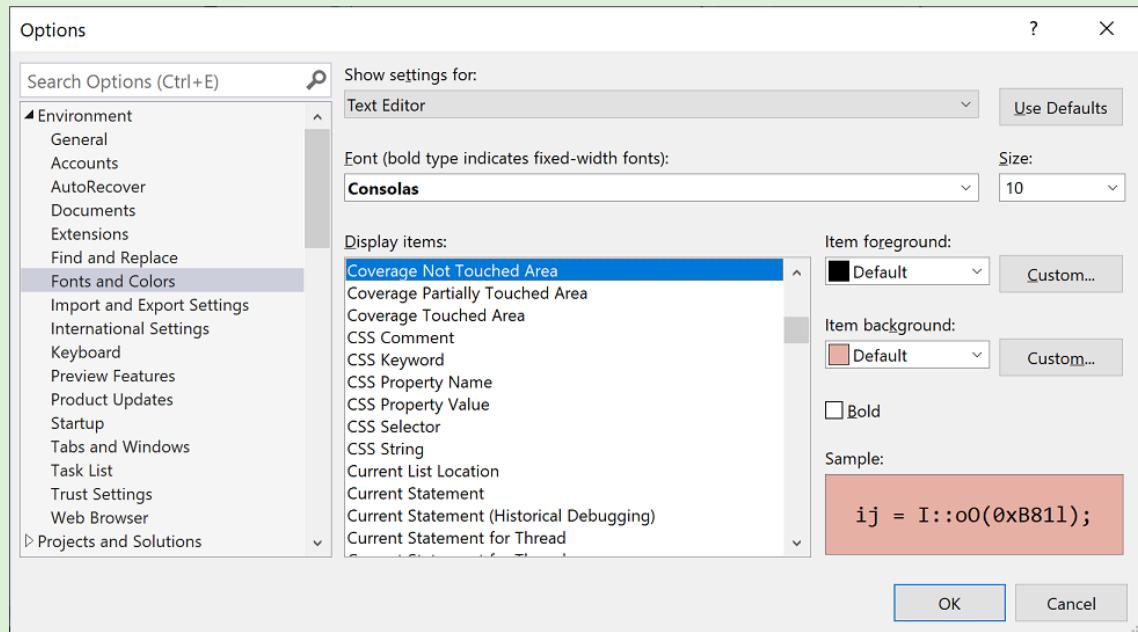


You can also run code coverage from the Test Explorer tool window.

2. After the tests have run, to see which lines have been run, choose **Show Code Coverage Coloring** in the **Code Coverage Results** window. By default, code that is covered by tests is highlighted in light blue.

TIP

To change the colors or to use bold face, choose **Tools > Options > Environment > Fonts and Colors > Show settings for: Text Editor**. Under **Display items**, adjust settings for the "Coverage" items, for example, **Coverage Not Touched Area**.



3. If the results show low coverage, investigate which parts of the code are not being exercised, and write more tests to cover them. Development teams typically aim for about 80% code coverage. In some situations, lower coverage is acceptable. For example, lower coverage is acceptable where some code is generated from a standard template.

TIP

- Turn compiler optimization off
- If you're working with unmanaged (native) code, use a debug build
- Generate .pdb (symbol) files for each assembly

If you don't get the results you expect, see [Troubleshoot code coverage](#).

Don't forget to run code coverage again after updating your code. Coverage results and code coloring are not automatically updated after you modify your code or when you run tests.

Report in blocks or lines

Code coverage is counted in *blocks*. A block is a piece of code with exactly one entry and exit point. If the program's control flow passes through a block during a test run, that block is counted as covered. The number of times the block is used has no effect on the result.

You can also have the results displayed in terms of lines by choosing **Add/Remove Columns** in the table header. Some users prefer a count of lines because the percentages correspond more closely to the size of the fragments that you see in the source code. A long block of calculation would count as a single block even if it occupies many lines.

TIP

A line of code can contain more than one code block. If this is the case, and the test run exercises all the code blocks in the line, it is counted as one line. If some but not all code blocks in the line are exercised, it is counted as a partial line.

Manage code coverage results

The **Code Coverage Results** window usually shows the result of the most recent run. The results will vary if you change your test data, or if you run only some of your tests each time.

The code coverage window can also be used to view previous results, or results obtained on other computers.

You can merge the results of several runs, for example from runs that use different test data.

- **To view a previous set of results**, select it from the drop-down menu. The menu shows a temporary list that is cleared when you open a new solution.
- **To view results from a previous session**, choose **Import Code Coverage Results**, navigate to the **TestResults** folder in your solution, and import a `.coverage` file.

The coverage coloring might be incorrect if the source code has changed since the `.coverage` file was generated.

- **To make results readable as text**, choose **Export Code Coverage Results**. This generates a readable `.coveragexml` file, which you could process with other tools or send easily in mail.
- **To send results to someone else**, send either a `.coverage` file or an exported `.coveragexml` file. They can then import the file. If they have the same version of the source code, they can see coverage coloring.

Merge results from different runs

In some situations, different blocks in your code will be used depending on the test data. Therefore, you might want to combine the results from different test runs.

For example, suppose that when you run a test with input "2", you find that 50% of a particular function is covered. When you run the test a second time with the input "-2", you see in the coverage coloring view that the other 50% of the function is covered. Now you merge the results from the two test runs, and the report and coverage coloring view show that 100% of the function was covered.

Use  **Merge Code Coverage Results** to do this. You can choose any combination of recent runs or imported results. If you want to combine exported results, you must import them first.

Use **Export Code Coverage Results** to save the results of a merge operation.

Limitations in merging

- If you merge coverage data from different versions of the code, the results are shown separately, but they are not combined. To get fully combined results, use the same build of the code, changing only the test data.
- If you merge a results file that has been exported and then imported, you can only view the results by lines, not by blocks. Use the **Add/Remove Columns** command to show the line data.
- If you merge results from tests of an ASP.NET project, the results for the separate tests are displayed, but not combined. This applies only to the ASP.NET artifacts themselves: results for any other assemblies will be combined.

Exclude elements from the code coverage results

You might want to exclude specific elements in your code from the coverage scores, for example if the code is generated from a text template. Add the [System.Diagnostics.CodeAnalysis.ExcludeFromCodeCoverageAttribute](#) attribute to any of the following code elements: class, struct, method, property, property setter or getter, event.

TIP

Excluding a class does not exclude its derived classes.

For example:

```
using System.Diagnostics.CodeAnalysis;
...
public class ExampleClass1
{
    [ExcludeFromCodeCoverage]
    void ExampleMethod() {...}

    [ExcludeFromCodeCoverage] // exclude property
    int ExampleProperty1
    { get {...} set{...} }

    int ExampleProperty2
    {
        get
        {
            ...
        }
        [ExcludeFromCodeCoverage] // exclude setter
        set
        {
            ...
        }
    }
}

[ExcludeFromCodeCoverage]
class ExampleClass2 { ... }
```

```

Imports System.Diagnostics.CodeAnalysis

Class ExampleClass1
    <ExcludeFromCodeCoverage()
    Public Sub ExampleSub1()
        ...
    End Sub

    ' Exclude property
    <ExcludeFromCodeCoverage()
    Property ExampleProperty1 As Integer
        ...
    End Property

    ' Exclude setter
    Property ExampleProperty2 As Integer
        Get
            ...
        End Get
        <ExcludeFromCodeCoverage()
        Set(ByVal value As Integer)
            ...
        End Set
    End Property
End Class

<ExcludeFromCodeCoverage()
Class ExampleClass2
    ...
End Class

```

```

// A .cpp file compiled as managed (CLI) code.
using namespace System::Diagnostics::CodeAnalysis;
...
public ref class ExampleClass1
{
public:
    [ExcludeFromCodeCoverage]
    void ExampleFunction1() { ... }

    [ExcludeFromCodeCoverage]
    property int ExampleProperty2 {....}

    property int ExampleProperty2 {
        int get() { ... }
        [ExcludeFromCodeCoverage]
        void set(int value) { ... }
    }
}

[ExcludeFromCodeCoverage]
public ref class ExampleClass2
{ ... }

```

Exclude elements in Native C++ code

To exclude unmanaged (native) elements in C++ code:

```

#include <CodeCoverage\CodeCoverage.h>
...
// Exclusions must be compiled as unmanaged (native):
#pragma managed(push, off)

// Exclude a particular function:
ExcludeFromCodeCoverage(Exclusion1, L"MyNamespace::MyClass::MyFunction");

// Exclude all the functions in a particular class:
ExcludeFromCodeCoverage(Exclusion2, L"MyNamespace::MyClass2::*");

// Exclude all the functions generated from a particular template:
ExcludeFromCodeCoverage(Exclusion3, L"*::MyFunction<*>");

// Exclude all the code from a particular .cpp file:
ExcludeSourceFromCodeCoverage(Exclusion4, L"*\\unittest1.cpp");

// After setting exclusions, restore the previous managed/unmanaged state:
#pragma managed(pop)

```

Use the following macros:

`ExcludeFromCodeCoverage(ExclusionName , L" FunctionName ");`

`ExcludeSourceFromCodeCoverage(ExclusionName , L" SourceFilePath ");`

- *ExclusionName* is any unique name.
- *FunctionName* is a fully qualified function name. It may contain wildcards. For example, to exclude all the functions of a class, write `MyNamespace::MyClass::*`
- *SourceFilePath* is the local or UNC path of a .cpp file. It may contain wildcards. The following example excludes all files in a particular directory: `\\\MyComputer\Source\UnitTests*.cpp`
- `#include <CodeCoverage\CodeCoverage.h>`
- Place calls to the exclusion macros in the global namespace, not within any namespace or class.
- You can place the exclusions either in the unit test code file or the application code file.
- The exclusions must be compiled as unmanaged (native) code, either by setting the compiler option or by using `#pragma managed(off)`.

NOTE

To exclude functions in C++/CLI code, apply the attribute

`[System::Diagnostics::CodeAnalysis::ExcludeFromCodeCoverage]` to the function. This is the same as for C#.

Include or exclude additional elements

Code coverage analysis is performed only on assemblies that are loaded and for which a .pdb file is available in the same directory as the .dll or .exe file. Therefore in some circumstances, you can extend the set of assemblies that is included by getting copies of the appropriate .pdb files.

You can exercise more control over which assemblies and elements are selected for code coverage analysis by writing a .runsettings file. For example, you can exclude assemblies of particular kinds without having to add attributes to their classes. For more information, see [Customize code coverage analysis](#).

Analyze code coverage in Azure Pipelines

When you check in your code, your tests run on the build server along with tests from other team members. It's useful to analyze code coverage in Azure Pipelines to get the most up-to-date and comprehensive picture of coverage in the whole project. It also includes automated system tests and other coded tests that you don't usually run on the development machines. For more information, see [Run unit tests with your builds](#).

Analyze code coverage from the command line

To run tests from the command line, use `vstest.console.exe`. Code coverage is an option of the `vstest.console.exe` utility.

1. Launch the Developer Command Prompt for Visual Studio:

In the Windows **Start** menu, choose **Visual Studio 2017 > Developer Command Prompt for VS 2017**.

In the Windows **Start** menu, choose **Visual Studio 2019 > Developer Command Prompt for VS 2019**.

2. At the command prompt, run the following command:

```
vstest.console.exe MyTestAssembly.dll /EnableCodeCoverage
```

For more information, see [VSTest.Console.exe command-line options](#).

Troubleshoot

If you do not see code coverage results, the [Troubleshoot code coverage](#) article might help you.

See also

- [Customize code coverage analysis](#)
- [Troubleshoot code coverage](#)
- [Unit test your code](#)

Customize code coverage analysis

1/13/2020 • 6 minutes to read • [Edit Online](#)

By default, code coverage analyzes all solution assemblies that are loaded during unit tests. We recommend that you use this default behavior, because it works well most of the time. For more information, see [Use code coverage to determine how much code is tested](#).

To exclude test code from the code coverage results and only include application code, add the [ExcludeFromCodeCoverageAttribute](#) attribute to your test class.

To include assemblies that aren't part of your solution, obtain the *.pdb* files for these assemblies and copy them into the same folder as the assembly *.dll* files.

Run settings file

The [run settings file](#) is the configuration file used by unit testing tools. Advanced code coverage settings are specified in a *.runsettings* file.

To customize code coverage, follow these steps:

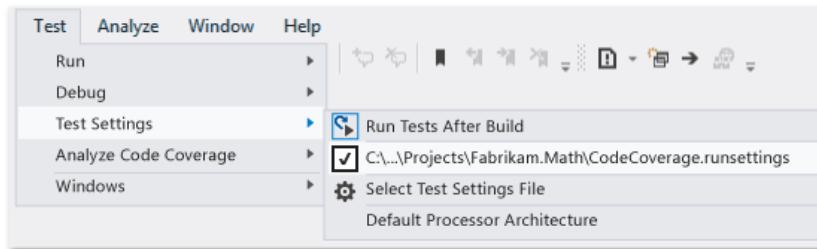
1. Add a run settings file to your solution. In **Solution Explorer**, on the shortcut menu of your solution, choose **Add > New Item**, and select **XML File**. Save the file with a name such as *CodeCoverage.runsettings*.
2. Add the content from the example file at the end of this article, and then customize it to your needs as described in the sections that follow.
3. To select the run settings file, on the **Test** menu, choose **Test Settings > Select Test Settings File**. To specify a run settings file for running tests from the command line, see [Configure unit tests](#).
3. To select the run settings file, on the **Test** menu, choose **Select Settings File**. To specify a run settings file for running tests from the command line, see [Configure unit tests](#).

When you select **Analyze Code Coverage**, the configuration information is read from the run settings file.

TIP

Any previous code coverage results and code coloring aren't automatically hidden when you run tests or update your code.

To turn the custom settings off and on, deselect or select the file in the **Test > Test Settings** menu.



To turn the custom settings off and on, deselect or select the file on the **Test** menu.

Symbol search paths

Code coverage requires symbol files (*.pdb* files) for assemblies. For assemblies built by your solution, symbol files are usually present alongside the binary files, and code coverage works automatically. In some cases, you might want to include referenced assemblies in your code coverage analysis. In such cases, the *.pdb* files might not be adjacent to the binaries, but you can specify the symbol search path in the *.runsettings* file.

```
<SymbolSearchPaths>
  <Path>\\mybuildshare\\builds\\ProjectX</Path>
  <!--More paths if required-->
</SymbolSearchPaths>
```

NOTE

Symbol resolution can take time, especially when using a remote file location with many assemblies. Therefore, consider copying *.pdb* files to the same local location as the binary (*.dll* and *.exe*) files.

Include or exclude assemblies and members

You can include or exclude assemblies or specific types and members from code coverage analysis. If the **Include** section is empty or omitted, then all assemblies that are loaded and have associated PDB files are included. If an assembly or member matches a clause in the **Exclude** section, then it is excluded from code coverage. The **Exclude** section takes precedence over the **Include** section: if an assembly is listed in both **Include** and **Exclude**, it will not be included in code coverage.

For example, the following XML excludes a single assembly by specifying its name:

```
<ModulePaths>
  <Exclude>
    <ModulePath>Fabrikam.Math.UnitTest.dll</ModulePath>
    <!-- Add more ModulePath nodes here. -->
  </Exclude>
</ModulePaths>
```

The following example specifies that only a single assembly should be included in code coverage:

```
<ModulePaths>
  <Include>
    <ModulePath>Fabrikam.Math.dll</ModulePath>
    <!-- Add more ModulePath nodes here. -->
  </Include>
</ModulePaths>
```

The following table shows the various ways that assemblies and members can be matched for inclusion in or exclusion from code coverage.

XML ELEMENT	WHAT IT MATCHES
ModulePath	Matches assemblies specified by assembly name or file path.
CompanyName	Matches assemblies by the Company attribute.
PublicKeyToken	Matches signed assemblies by the public key token.
Source	Matches elements by the path name of the source file in which they're defined.

XML ELEMENT	WHAT IT MATCHES
Attribute	<p>Matches elements that have the specified attribute. Specify the full name of the attribute, for example</p> <pre data-bbox="833 235 1507 258"><Attribute>^System\.Diagnostics\.DebuggerHiddenAttribute\$</Attrib</pre> <p>If you exclude the CompilerGeneratedAttribute attribute, code that uses language features such as <code>async</code>, <code>await</code>, <code>yield return</code>, and auto-implemented properties is excluded from code coverage analysis. To exclude truly generated code, only exclude the GeneratedCodeAttribute attribute.</p>
Function	<p>Matches procedures, functions, or methods by fully qualified name, including the parameter list. You can also match part of the name by using a regular expression.</p> <p>Examples:</p> <pre data-bbox="833 696 1333 718">Fabrikam.Math.LocalMath.SquareRoot(double); (C#)</pre> <pre data-bbox="833 752 1372 774">Fabrikam::Math::LocalMath::SquareRoot(double) (C++)</pre>

Regular expressions

Include and exclude nodes use regular expressions, which aren't the same as wildcards. All matches are case-insensitive. Some examples are:

- `.*` matches a string of any characters
- `\.` matches a dot `"."`
- `\(\)` matches parentheses `"()"`
- `\\\` matches a file path delimiter `"\"`
- `^` matches the start of the string
- `$` matches the end of the string

The following XML shows how to include and exclude specific assemblies by using regular expressions:

```

<ModulePaths>
  <Include>
    <!-- Include all loaded .dll assemblies (but not .exe assemblies): -->
    <ModulePath>.*\.dll$</ModulePath>
  </Include>
  <Exclude>
    <!-- But exclude some assemblies: -->
    <ModulePath>.*\\Fabrikam\\.MyTests1\\.dll$</ModulePath>
    <!-- Exclude all file paths that contain "Temp": -->
    <ModulePath>.*Temp.*</ModulePath>
  </Exclude>
</ModulePaths>
```

The following XML shows how to include and exclude specific functions by using regular expressions:

```

<Functions>
<Include>
    <!-- Include methods in the Fabrikam namespace: -->
    <Function>^Fabrikam\..*</Function>
    <!-- Include all methods named EqualTo: -->
    <Function>.*\EqualTo\(.*\</Function>
</Include>
<Exclude>
    <!-- Exclude methods in a class or namespace named UnitTest: -->
    <Function>.*\UnitTest\..*</Function>
</Exclude>
</Functions>

```

WARNING

If there is an error in a regular expression, such as an unescaped or unmatched parenthesis, code coverage analysis won't run.

For more information about regular expressions, see [Use regular expressions in Visual Studio](#).

Sample .runsettings file

Copy this code and edit it to suit your needs.

```

<?xml version="1.0" encoding="utf-8"?>
<!-- File name extension must be .runsettings -->
<RunSettings>
    <DataCollectionRunSettings>
        <DataCollectors>
            <DataCollector friendlyName="Code Coverage" uri="datacollector://Microsoft/CodeCoverage/2.0"
assemblyQualifiedName="Microsoft.VisualStudio.Coverage.DynamicCoverageDataCollector,
Microsoft.VisualStudio.TraceCollector, Version=11.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a">
                <Configuration>
                    <CodeCoverage>
                <!--
                    Additional paths to search for .pdb (symbol) files. Symbols must be found for modules to be instrumented.
                    If .pdb files are in the same folder as the .dll or .exe files, they are automatically found. Otherwise, specify them here.
                    Note that searching for symbols increases code coverage runtime. So keep this small and local.
                -->
            <!--
                <SymbolSearchPaths>
                    <Path>C:\Users\User\Documents\Visual Studio 2012\Projects\ProjectX\bin\Debug</Path>
                    <Path>\mybuildshare\builds\ProjectX</Path>
                </SymbolSearchPaths>
            -->
            <!--
                About include/exclude lists:
                Empty "Include" clauses imply all; empty "Exclude" clauses imply none.
                Each element in the list is a regular expression (ECMAScript syntax). See /visualstudio/ide/using-regular-expressions-in-visual-studio.
                An item must first match at least one entry in the include list to be included.
                Included items must then not match any entries in the exclude list to remain included.
            -->
            <!-- Match assembly file paths: -->
            <ModulePaths>
                <Include>
                    <ModulePath>.*\.dll$</ModulePath>
                    <ModulePath>.*\.exe$</ModulePath>
                </Include>
                <Exclude>
                    <ModulePath>.*CPPUnitTestFramework.*</ModulePath>
                </Exclude>
            </ModulePaths>

            <!-- Match fully qualified names of functions: -->
            <!-- (Use "\." to delimit namespaces in C# or Visual Basic, "::" in C++) -->

```

```

<Functions>
  <Exclude>
    <Function>^Fabrikam\\UnitTest\..*</Function>
    <Function>^std::.*</Function>
    <Function>^ATL::.*</Function>
    <Function>.*::__GetTestMethodInfo.*</Function>
    <Function>^Microsoft::VisualStudio::CppCodeCoverageFramework::.*</Function>
    <Function>^Microsoft::VisualStudio::CppUnitTestFramework::.*</Function>
  </Exclude>
</Functions>

<!-- Match attributes on any code element: -->
<Attributes>
  <Exclude>
    <!-- Don't forget "Attribute" at the end of the name -->
    <Attribute>^System\.\Diagnostics\.\DebuggerHiddenAttribute$</Attribute>
    <Attribute>^System\.\Diagnostics\.\DebuggerNonUserCodeAttribute$</Attribute>
    <Attribute>^System\.\CodeDom\.\Compiler\.\GeneratedCodeAttribute$</Attribute>
    <Attribute>^System\.\Diagnostics\.\CodeAnalysis\.\ExcludeFromCodeCoverageAttribute$</Attribute>
  </Exclude>
</Attributes>

<!-- Match the path of the source files in which each method is defined: -->
<Sources>
  <Exclude>
    <Source>.*\\atlmfc\\.*</Source>
    <Source>.*\\vctools\\.*</Source>
    <Source>.*\\public\\sdk\\.*</Source>
    <Source>.*\\microsoft sdks\\.*</Source>
    <Source>.*\\vc\\include\\.*</Source>
  </Exclude>
</Sources>

<!-- Match the company name property in the assembly: -->
<CompanyNames>
  <Exclude>
    <CompanyName>.*microsoft.*</CompanyName>
  </Exclude>
</CompanyNames>

<!-- Match the public key token of a signed assembly: -->
<PublicKeyTokens>
  <!-- Exclude Visual Studio extensions: -->
  <Exclude>
    <PublicKeyToken>^B77A5C561934E089$</PublicKeyToken>
    <PublicKeyToken>^B03F5F7F11D50A3A$</PublicKeyToken>
    <PublicKeyToken>^31BF3856AD364E35$</PublicKeyToken>
    <PublicKeyToken>^89845DCD8080CC91$</PublicKeyToken>
    <PublicKeyToken>^71E9BCE111E9429C$</PublicKeyToken>
    <PublicKeyToken>^8F50407C4E9E73B6$</PublicKeyToken>
    <PublicKeyToken>^E361AF139669C375$</PublicKeyToken>
  </Exclude>
</PublicKeyTokens>

<!-- We recommend you do not change the following values: -->

<!-- Set this to True to collect coverage information for functions marked with the "SecuritySafeCritical" attribute. Instead of writing directly into a memory location from such functions, code coverage inserts a probe that redirects to another function, which in turns writes into memory. -->
<UseVerifiableInstrumentation>True</UseVerifiableInstrumentation>
<!-- When set to True, collects coverage information from child processes that are launched with low-level ACLs, for example, UWP apps. -->
<AllowLowIntegrityProcesses>True</AllowLowIntegrityProcesses>
<!-- When set to True, collects coverage information from child processes that are launched by test or production code. -->
<CollectFromChildProcesses>True</CollectFromChildProcesses>
<!-- When set to True, restarts the IIS process and collects coverage information from it. -->
<CollectAspDotNet>False</CollectAspDotNet>

</CodeCoverage>
</Configuration>
</DataCollector>
</DataCollectors>
</DataCollectionRunSettings>

```

```
</RunSettings>
```

See also

- [Configure unit tests by using a run settings file](#)
- [Use code coverage to determine how much code is tested](#)
- [Unit test your code](#)

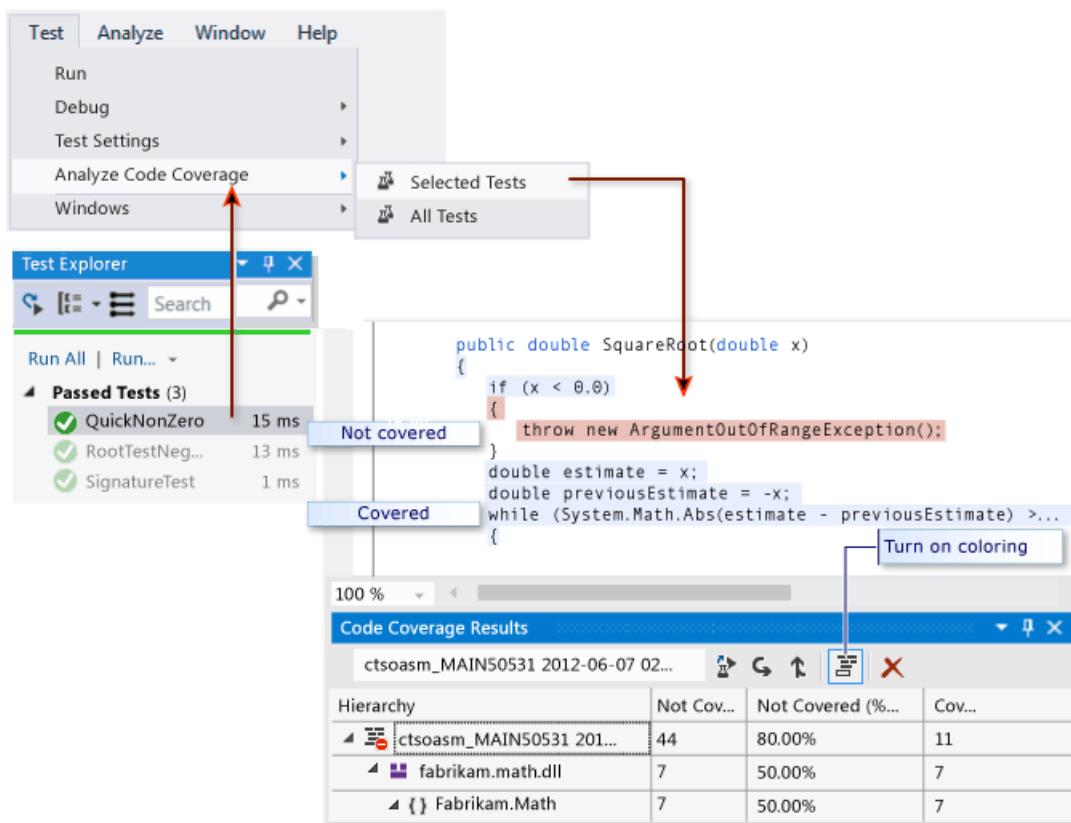
Troubleshoot code coverage

1/1/2020 • 5 minutes to read • [Edit Online](#)

The code coverage analysis tool in Visual Studio collects data for native and managed assemblies (.dll or .exe files). However, in some cases, the **Code Coverage Results** window displays an error similar to "Empty results generated:" There are several reasons why you can get empty results. This article helps you resolve those issues.

What you should see

If you choose an **Analyze Code Coverage** command on the **Test** menu, and if the build and tests run successfully, then you should see a list of results in the **Code Coverage** window. You might have to expand the items to see the detail.



For more information, see [Use code coverage to determine how much code is being tested](#).

Possible reasons for seeing no results or old results

Do you have the right edition of Visual Studio?

You need Visual Studio Enterprise.

No tests were executed

Analysis—Check your output window. In the **Show Output from** drop-down list, choose **Tests**. See if there are any warnings or errors logged.

Explanation—Code coverage analysis is done while tests are running. It only includes assemblies that are loaded into memory when the tests run. If none of the tests are executed, there's nothing for code coverage to report.

Resolution—In Test Explorer, choose **Run All** to verify that the tests run successfully. Fix any failures before using **Analyze Code Coverage**.

You're looking at a previous result

When you modify and rerun your tests, a previous code coverage result can still be visible, including the code coloring from that old run.

1. Run **Analyze Code Coverage**.

2. Make sure that you have selected the most recent result set in the **Code Coverage Results** window.

.pdb (symbol) files are unavailable

Analysis—Open the compile target folder (typically *bin\debug*), and verify that for each assembly, there's a *.pdb* file in the same directory as the *.dll* or *.exe* file.

Explanation—The code coverage engine requires that every assembly has its associated *.pdb* file accessible during the test run. If there's no *.pdb* file for a particular assembly, the assembly is not analyzed.

The *.pdb* file must be generated from the same build as the *.dll* or *.exe* files.

Resolution—Make sure that your build settings generate the *.pdb* file. If the *.pdb* files are not updated when the project is built, then open the project properties, select the **Build** page, choose **Advanced**, and inspect **Debug Info**.

For C++ projects, ensure that the generated *.pdb* files have full debug information. Open the project properties and verify that **Linker > Debugging > Generate Debug Info** is set to **Generate Debug Information optimized for sharing and publishing (/DEBUG:FULL)**.

If the *.pdb* and *.dll* or *.exe* files are in different places, copy the *.pdb* file to the same directory. It is also possible to configure code coverage engine to search for *.pdb* files in another location. For more information, see [Customize code coverage analysis](#).

Use an instrumented or optimized binary

Analysis—Determine if the binary has undergone any form of advanced optimization such as Profile Guided Optimization, or has been instrumented by a profiling tool such as *vsinstr.exe* or *vsperfmon.exe*.

Explanation—If an assembly has already been instrumented or optimized by another profiling tool, the assembly is omitted from the code coverage analysis. Code coverage analysis can't be performed on such assemblies.

Resolution—Switch off optimization and use a new build.

Code is not managed (.NET) or native (C++) code

Analysis—Verify that you're running some tests on managed or C++ code.

Explanation—Code coverage analysis in Visual Studio is available only on managed and native (C++) code. If you're working with third-party tools, some or all of the code might execute on a different platform.

Resolution—None available.

Assembly has been installed by NGen

Analysis—Verify that the assembly is not loaded from the native image cache.

Explanation—For performance reasons, native image assemblies are not analyzed. For more information, see [Ngen.exe \(Native Image Generator\)](#).

Resolution—Use an MSIL version of the assembly. Don't process it with NGen.

Custom .runsettings file with bad syntax

Analysis—If you're using a custom *.runsettings* file, it might contain a syntax error. Code coverage is not run, and either the code coverage window doesn't open at the end of the test run, or it shows old results.

Explanation—You can run your unit tests with a custom *.runsettings* file to configure code coverage options. The

options allow you to include or exclude files. For more information, see [Customize code coverage analysis](#).

Resolution—There are two possible types of faults:

- **XML error**

Open the `.runsettings` file in the Visual Studio XML editor. Look for error indications.

- **Regular expression error**

Each string in the file is a regular expression. Review each one for errors, and in particular look for:

- Mismatched parentheses (...) or unescaped parentheses \(...\). If you want to match a parenthesis in the search string, you must escape it. For example, to match a function use: `.*MyFunction\(double\)`
- Asterisk or plus at the start of an expression. To match any string of characters, use a dot followed by an asterisk: `.*`

Custom `.runsettings` file with incorrect exclusions

Analysis—If you're using a custom `.runsettings` file, make sure that it includes your assembly.

Explanation—You can run your unit tests with a custom `.runsettings` file to configure code coverage options. The options allow you to include or exclude files. For more information, see [Customize code coverage analysis](#).

Resolution—Remove all the `Include` nodes from the `.runsettings` file, and then remove all the `Exclude` nodes. If that fixes the problem, put them back in stages.

Make sure the `DataCollectors` node specifies Code Coverage. Compare it with the sample in [Customize code coverage analysis](#).

Some code is always shown as not covered

Initialization code in native DLLs is executed before instrumentation

Analysis—In statically linked native code, part of the initialization function **DllMain** and code that it calls is sometimes shown as not covered, even though the code has been executed.

Explanation—The code coverage tool works by inserting instrumentation into an assembly just before the application starts running. In any assembly loaded beforehand, the initialization code in **DllMain** executes as soon as the assembly loads, and before the application runs. That code appears to be not covered, which typically applies to statically loaded assemblies.

Resolution—None.

See also

- [Use code coverage to determine how much code is being tested](#)

Live Unit Testing overview

1/1/2020 • 2 minutes to read • [Edit Online](#)

Live Unit Testing executes your unit tests automatically and in real time as you make code changes. This lets you refactor and change code with greater confidence. Live Unit Testing automatically executes all impacted tests as you edit your code to ensure that your changes do not introduce regressions.

Live Unit Testing indicates whether your unit tests adequately cover your code. It graphically depicts code coverage in real time. You can see at a glance how many tests cover each line of code and which lines are not covered by any unit tests.

If you have a solution that includes one or more unit test projects, you can enable Live Unit Testing by selecting **Test > Live Unit Testing > Start** from the top-level menu bar in Visual Studio.

NOTE

Live Unit Testing is only available in Visual Studio Enterprise edition.

To learn more about Live Unit Testing:

- Try an introductory tutorial: [Get started with Live Unit Testing](#).
- Read detailed documentation: [Use Live Unit Testing with Visual Studio Enterprise Edition](#).
- Read the [Live Unit Testing FAQ](#) to learn what's new in Live Unit Testing as well as tips and techniques.
- Watch the Channel 9 video for an overview of Live Unit Testing and its features.

Related resources

- [Code testing tools](#)
- [Unit test your code](#)

What's new in Live Unit Testing for Visual Studio 2017

1/16/2020 • 3 minutes to read • [Edit Online](#)

This topic lists the new features added to Live Unit Testing in each version of Visual Studio starting with Visual Studio 2017 version 15.3. For an overview of how to use Live Unit Testing, see [Live Unit Testing with Visual Studio](#).

Version 15.4

Starting with Visual Studio 2017 version 15.4, Live Unit Testing includes improvements and enhancements in a number of areas:

- **Improved discoverability.** For users who do not know that the Live Unit Testing feature exists, the Visual Studio IDE shows a gold bar that mentions Live Unit Testing whenever the user opens a solution that includes unit tests but Live Unit Testing is not enabled. The information presented in the gold bar allows the user to learn more about Live Unit Testing and to enable it. The gold bar also displays information when Live Unit Testing prerequisites are not met. These include:
 - Test adapters are missing.
 - Older versions of test adapters are present.
 - A restore of NuGet packages referenced by the solution is needed.
- **Integration with Task Center notifications.** The Visual Studio IDE now shows a Live Unit Testing background processing notification in Task Center so that users can easily tell what is happening when Live Unit Testing is enabled. This addresses the key problem of starting Live Unit Testing on a large solution. Previously, for a few minutes until the coverage icons appeared, users couldn't determine whether Live Unit Testing was really enabled and whether it was working. Not anymore!
- **Support for the MSTest framework version 1:** Live Unit Testing already works with three popular unit testing frameworks: xUnit, NUnit, and MSTest. Previously, Live Unit Testing only worked when MSTest unit test projects used MS Test version 2. Starting with Visual Studio 2017 version 15.4, it now also supports MSTest version 1 as well.
- **Reliability & Performance:** Live Unit Testing now ensures that the system can better detect when projects haven't completed loading fully and avoids crashing Live Unit Testing. Build performance improvements also avoid reevaluating MSBuild projects when the system knows that nothing in the project file has changed.
- **Miscellaneous user interface refinements:** The confusing **Live Test Set – Include/Exclude** option from the right click gesture has been renamed to **Live Unit Testing Include/Exclude**. The **Reset clean** option on the **Test > Live Unit Testing** menu has been removed. It is now accessible by selecting **Tools > Options > Live Unit Testing** and selecting **Delete Persisted Data**.

Version 15.3

Starting with Visual Studio 2017 version 15.3, Live Unit Testing features improvements and enhancements in two major areas:

- Support for .NET Core and .NET Standard. You can use Live Unit Testing on .NET Core and .NET Standard solutions written in either C# or Visual Basic.
- Performance improvements. You'll notice that performance is significantly faster after the first full build and run of tests under Live Unit Testing. You'll also notice significant performance improvement in subsequent

starts of Live Unit Testing on the same solution. We now persist data generated by Live Unit Testing and reuse it as much as possible with up-to-date checks.

In addition to these major additions, Live Unit Testing includes the following enhancements:

- A new beaker icon is now used to distinguish a test method from regular methods. An empty beaker icon indicates that the specific test is not included in Live Unit Testing.
- When clicking on a test method from the pop-up UI window of a Live Unit Testing coverage icon, you now have the option to debug the test right from that context within the UI window and without having to leave the code editor. This is especially useful when you are looking at a failed test.
- Several additional configurable options have been added to Tools/Options/Live Unit Testing/General. You can cap the memory used for Live Unit Testing. You can also specify the file path for persisted Live Unit Testing data for your open solution.
- Several additional menu items have been added under the menu bar of Test/Live Unit Testing. **Reset Clean** deletes the persisted data and generates it again. **Option** jumps to Tools/Options/Live Unit Testing/General.
- You can now use the following attributes to specify in source code that you want to exclude targeted test methods from Live Unit Testing:
 - For xUnit: `[Trait("Category", "SkipWhenLiveUnitTesting")]`
 - For NUnit: `[Category("SkipWhenLiveUnitTesting")]`
 - For MSTest: `[TestCategory("SkipWhenLiveUnitTesting")]`

See also

- [Introducing Live Unit Testing](#)
- [Live Unit Testing with Visual Studio](#)

Get started with Live Unit Testing

1/1/2020 • 11 minutes to read • [Edit Online](#)

When you enable Live Unit Testing in a Visual Studio solution, it visually depicts your test coverage and the status of your tests. Live Unit Testing also dynamically executes tests whenever you modify your code and immediately notifies you when your changes cause tests to fail.

Live Unit Testing can be used to test solutions that target either .NET Framework or .NET Core. In this tutorial, you'll learn to use Live Unit Testing by creating a simple class library that targets .NET Standard, and you'll create an MSTest project that targets .NET Core to test it.

The complete C# solution can be downloaded from the [MicrosoftDocs/visualstudio-docs](#) repo on GitHub.

Prerequisites

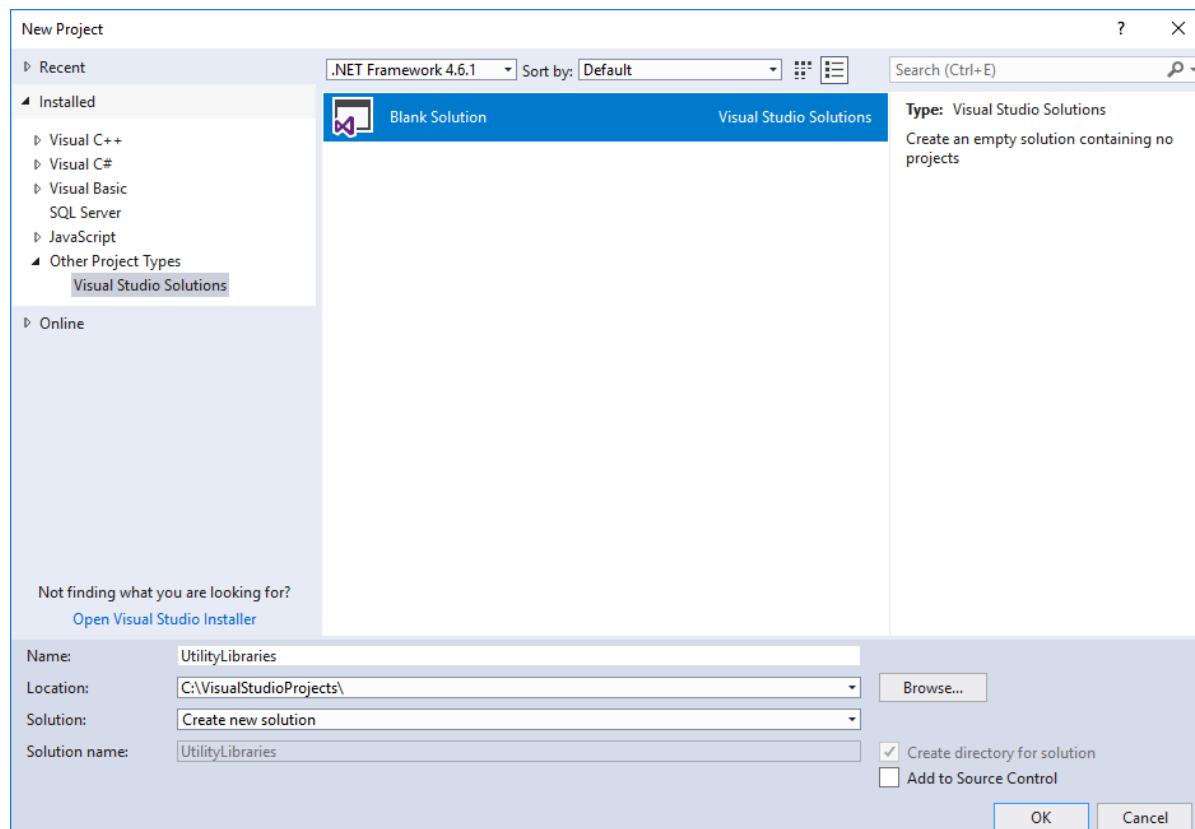
This tutorial requires that you've installed Visual Studio Enterprise edition with the **.NET Core cross-platform development** workload.

Create the solution and the class library project

Begin by creating a Visual Studio solution named UtilityLibraries that consists of a single .NET Standard class library project, StringLibrary.

The solution is just a container for one or more projects. To create a blank solution, open Visual Studio and do the following:

1. Select **File > New > Project** from the top-level Visual Studio menu.
2. Type **solution** into the template search box, and then select the **Blank Solution** template.



3. Finish creating the solution.

Now that you've created the solution, you'll create a class library named StringLibrary that contains a number of extension methods for working with strings.

1. In **Solution Explorer**, right-click on the UtilityLibraries solution and select **Add > New Project**.

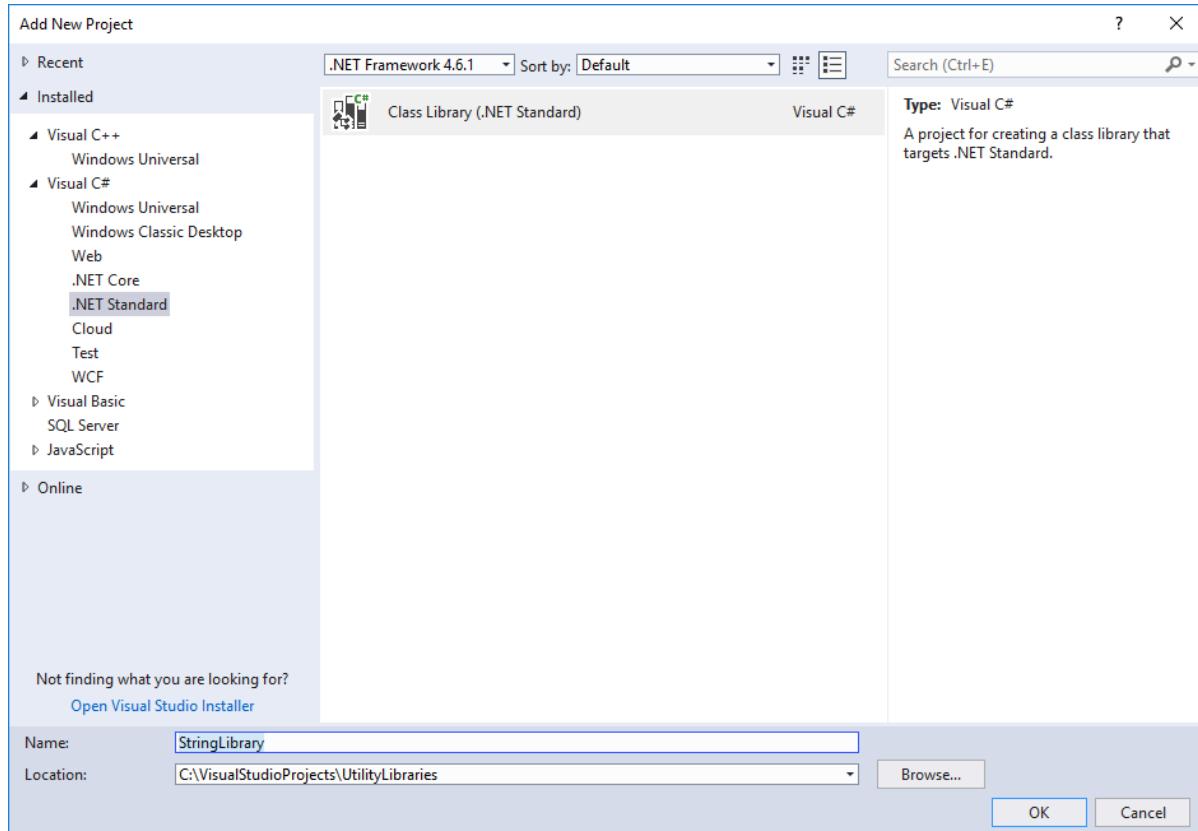
2. In the **Add New Project** dialog, select the C# node, then select **.NET Standard**.

NOTE

Because our library targets .NET Standard rather than a particular .NET implementation, it can be called from any .NET implementation that supports that version of .NET Standard. For more information, see [.NET Standard](#).

3. Select the **Class Library (.NET Standard)** template in the right pane, and enter **StringLibrary** in the

Name text box, as the following image shows:



4. Select **OK** to create the project.

2. Type **class library** into the template search box, and the select the **Class Library (.NET Standard)** template. Click **Next**.

NOTE

Because our library targets .NET Standard rather than a particular .NET implementation, it can be called from any .NET implementation that supports that version of .NET Standard. For more information, see [.NET Standard](#).

3. Name the project **StringLibrary**.

4. Click **Create** to create the project.

5. Replace all of the existing code in the code window with the following code:

```

using System;

namespace UtilityLibraries
{
    public static class StringLibrary
    {
        public static bool StartsWithUpper(this string s)
        {
            if (String.IsNullOrWhiteSpace(s))
                return false;

            return Char.IsUpper(s[0]);
        }

        public static bool StartsWithLower(this string s)
        {
            if (String.IsNullOrWhiteSpace(s))
                return false;

            return Char.IsLower(s[0]);
        }

        public static bool HasEmbeddedSpaces(this string s)
        {
            foreach (var ch in s.Trim())
            {
                if (ch == ' ')
                    return true;
            }
            return false;
        }
    }
}

```

StringLibrary has three static methods:

- `StartsWithUpper` returns `true` if a string starts with an uppercase character; otherwise, it returns `false`.
- `StartsWithLower` returns `true` if a string starts with a lowercase character; otherwise, it returns `false`.
- `HasEmbeddedSpaces` returns `true` if a string contains an embedded whitespace character; otherwise, it returns `false`.

6. Select **Build > Build Solution** from the top-level Visual Studio menu. The build should succeed.

Create the test project

The next step is to create the unit test project to test the StringLibrary library. Create the unit tests by performing the following steps:

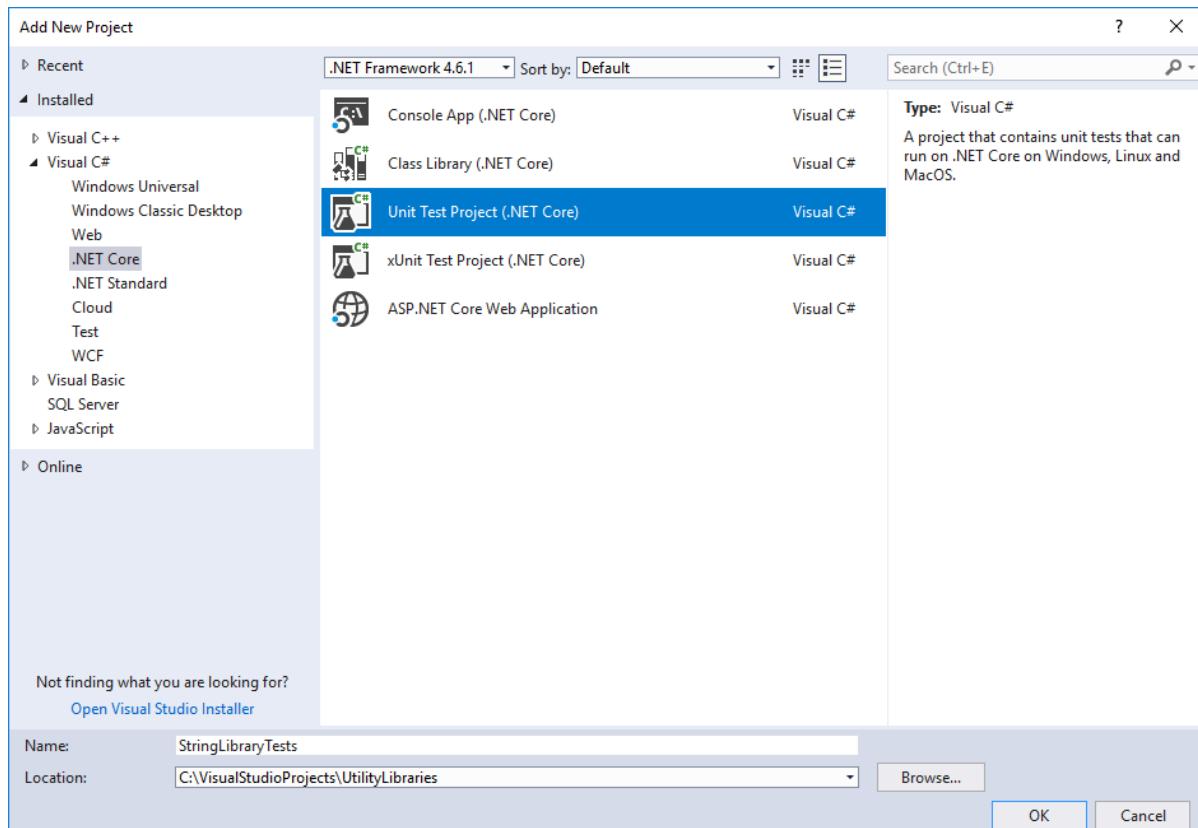
1. In **Solution Explorer**, right-click on the UtilityLibraries solution and select **Add > New Project**.
2. In the **Add New Project** dialog, select the C# node, then select **.NET Core**.

NOTE

You do not have to write your unit tests in the same language as your class library.

3. Select the **Unit Test Project (.NET Core)** template in the right pane, and enter **StringLibraryTests** in the

Name text box, as the following image shows:

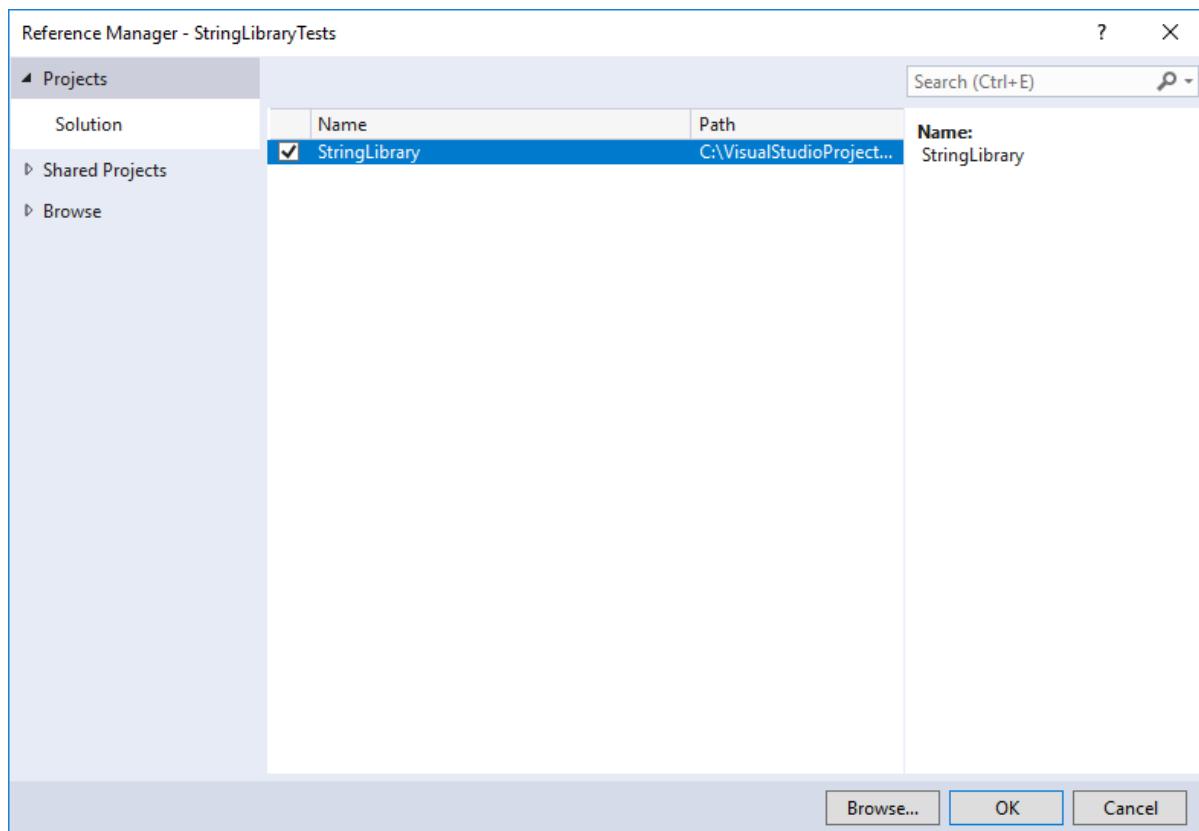


4. Select **OK** to create the project.
2. Type **unit test** into the template search box, and the select the **Unit Test Project (.NET Core)** template. Click **Next**.
3. Name the project **StringLibraryTests**.
4. Click **Create** to create the project.

NOTE

This getting started tutorial uses Live Unit Testing with the MSTest test framework. You can also use the xUnit and NUnit test frameworks.

5. The unit test project can't automatically access the class library that it is testing. You give the test library access by adding a reference to the class library project. To do this, right-click on the **StringLibraryTests** project and select **Add > Reference**. In the **Reference Manager** dialog, make sure the **Solution** tab is selected, and select the StringLibrary project, as shown in the following image.



6. Replace the boilerplate unit test code provided by the template with the following code:

```

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

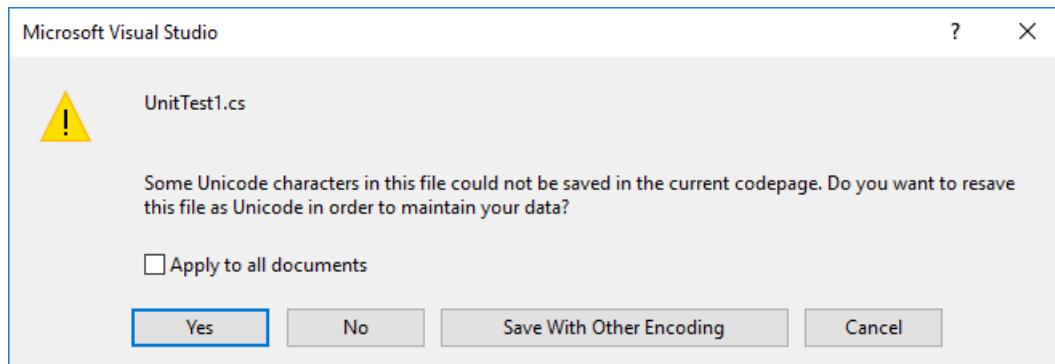
namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestStartsWithUpper()
        {
            // Tests that we expect to return true.
            string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα", "Москва" };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsTrue(result,
                    $"Expected for '{word}': true; Actual: {result}");
            }
        }

        [TestMethod]
        public void TestDoesNotStartWithUpper()
        {
            // Tests that we expect to return false.
            string[] words = { "alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                "1234", ".", ";", " " };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsFalse(result,
                    $"Expected for '{word}': false; Actual: {result}");
            }
        }

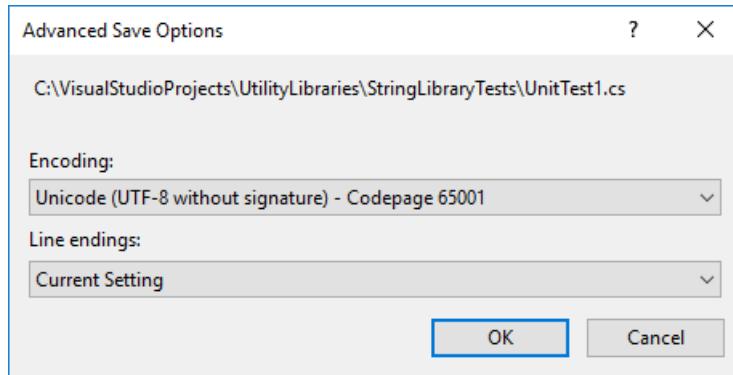
        [TestMethod]
        public void DirectCallWithNullOrEmpty()
        {
            // Tests that we expect to return false.
            string[] words = { String.Empty, null };
            foreach (var word in words)
            {
                bool result = StringLibrary.StartsWithUpper(word);
                Assert.IsFalse(result,
                    $"Expected for '{(word == null ? "<null>" : word)}': " +
                    $"false; Actual: {result}");
            }
        }
    }
}

```

7. Save your project by selecting the **Save** icon on the toolbar.
8. Because the unit test code includes some non-ASCII characters, Visual Studio displays the following dialog to warn that some characters will be lost if you save the file in its default ASCII format. Choose the **Save with Other Encoding** button.



9. In the **Encoding** drop-down list of the **Advance Save Options** dialog, choose **Unicode (UTF-8 without signature) - Codepage 65001**, as the following image shows:



10. Compile the unit test project by selecting **Build > Rebuild Solution** from the top-level Visual Studio menu.

You've created a class library as well as some unit tests for it. You've now finished the preliminaries needed to use Live Unit Testing.

Enable Live Unit Testing

So far, although you've written the tests for the StringLibrary class library, you haven't executed them. Live Unit Testing executes them automatically once you enable it. To do that, do the following:

1. Optionally, select the code window that contains the code for StringLibrary. This is either *Class1.cs* for a C# project or *Class1.vb* for a Visual Basic project. (This step lets you visually inspect the result of your tests and the extent of your code coverage once you enable Live Unit Testing.)
2. Select **Test > Live Unit Testing > Start** from the top-level Visual Studio menu.
3. Visual Studio starts Live Unit Test, which automatically runs all of your tests.

When it finishes running your tests, **Test Explorer** displays both the overall results and the result of individual tests. In addition, the code window graphically displays both your test code coverage and the result for your tests. As the following image shows, all three tests have executed successfully. It also shows that our tests have covered all code paths in the `StartsWithUpper` method, and those tests all executed successfully (which is indicated by the green check mark, "✓"). Finally, it shows that none of the other methods in StringLibrary have code coverage (which is indicated by a blue line, "□").

You can also get more detailed information about test coverage and test results by selecting a particular code coverage icon in the code window. To examine this detail, do the following:

1. Click on the green check mark on the line that reads `if (String.IsNullOrEmpty(s))` in the `StartsWithUpper` method. As the following image shows, Live Unit Testing indicates that three tests cover that line of code, and that all have executed successfully.

2. Click on the green check mark on the line that reads `return Char.ToUpper(s[0]);` in the `StartsWithUpper` method. As the following image shows, Live Unit Testing indicates that only two tests cover that line of code, and that all have executed successfully.

The major issue that Live Unit Testing identifies is incomplete code coverage. You'll address it in the next section.

Expand test coverage

In this section, you'll extend your unit tests to the `StartsWithLower` method. While you do that, Live Unit Testing will dynamically continue to test your code.

To extend code coverage to the `StartsWithLower` method, do the following:

1. Add the following `TestStartsWithLower` and `TestDoesNotStartWithLower` methods to your project's test source code file:

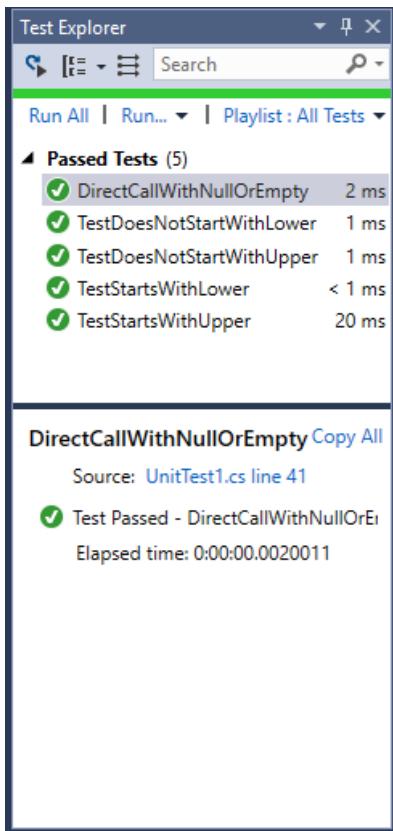
```
// Code to add to UnitTest1.cs
[TestMethod]
public void TestStartsWithLower()
{
    // Tests that we expect to return true.
    string[] words = { "alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство" };
    foreach (var word in words)
    {
        bool result = word.StartsWithLower();
        Assert.IsTrue(result,
            $"Expected for '{word}': true; Actual: {result}");
    }
}

[TestMethod]
public void TestDoesNotStartWithLower()
{
    // Tests that we expect to return false.
    string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα", "Москва",
        "1234", ".", ";", " "};
    foreach (var word in words)
    {
        bool result = word.StartsWithLower();
        Assert.IsFalse(result,
            $"Expected for '{word}': false; Actual: {result}");
    }
}
```

2. Modify the `DirectCallWithNullOrEmpty` method by adding the following code immediately after the call to the `Microsoft.VisualStudio.TestTools.UnitTesting.Assert.IsFalse` method.

```
// Code to add to UnitTest1.cs
result = StringLibrary.StartsWithLower(word);
Assert.IsFalse(result,
    $"Expected for '{{(word == null ? "<null>" : word)}}': " +
    $"false; Actual: {result}");
```

3. Live Unit Testing automatically executes new and modified tests when you modify your source code. As the following image of **Test Explorer** shows, all of the tests, including the two you've added and the one you've modified, have succeeded.



4. Switch to the window that contains the source code for the StringLibrary class. Live Unit Testing now shows that our code coverage is extended to the `StartsWithLower` method.

The screenshot shows a Visual Studio code editor window. The top bar displays tabs for 'Class1.cs' and 'UnitTest1.cs'. Below the tabs, the status bar shows 'C# StringLibrary' and 'UtilityLibraries.StringLibrary'. A tooltip for the method 'StartsWithLower(string s)' is visible. The code itself is as follows:

```
1     using System;
2
3     namespace UtilityLibraries
4     {
5         public static class StringLibrary
6         {
7             [✓] public static bool StartsWithUpper(this string s)
8             {
9                 if (String.IsNullOrWhiteSpace(s))
10                    return false;
11
12                 return Char.IsUpper(s[0]);
13             }
14
15             [✓] public static bool StartsWithLower(this string s)
16             {
17                 if (String.IsNullOrWhiteSpace(s))
18                    return false;
19
20                 return Char.IsLower(s[0]);
21             }
22
23             [!] public static bool HasEmbeddedSpaces(this string s)
24             {
25                 foreach (var ch in s.Trim())
26                 {
27                     if (ch == ' ')
28                         return true;
29
30                 }
31             }
32
33         }
34     }
```

A green checkmark icon is placed next to the first two methods, indicating they have been successfully tested. A red exclamation mark icon is placed next to the 'HasEmbeddedSpaces' method, indicating it has not been tested or is currently being tested.

In some cases, successful tests in **Test Explorer** may be grayed-out. That indicates that a test is currently executing, or that the test has not run again because there have been no code changes that would impact the test since it was last executed.

So far, all of our tests have succeeded. In the next section, we'll examine how you can handle test failure.

Handle a test failure

In this section, you'll explore how you can use Live Unit Testing to identify, troubleshoot, and address test failures. You'll do this by expanding test coverage to the `HasEmbeddedSpaces` method.

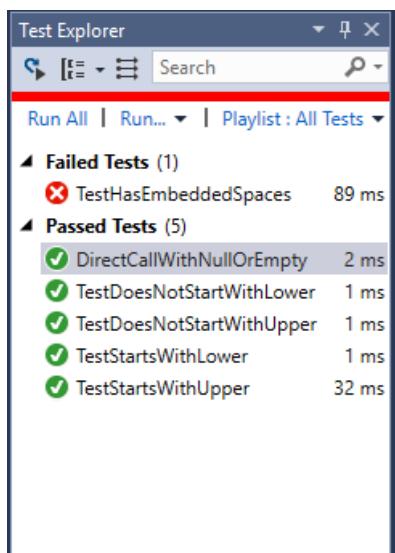
1. Add the following method to your test file:

```

[TestMethod]
public void TestHasEmbeddedSpaces()
{
    // Tests that we expect to return true.
    string[] phrases = { "one car", "Name\u0009Description",
        "Line1\nLine2", "Line3\u000ALine4",
        "Line5\u000BLine6", "Line7\u000CLine8",
        "Line0009\u000DLine10", "word1\u000Aword2" };
    foreach (var phrase in phrases)
    {
        bool result = phrase.HasEmbeddedSpaces();
        Assert.IsTrue(result,
            $"Expected for '{phrase}': true; Actual: {result}");
    }
}

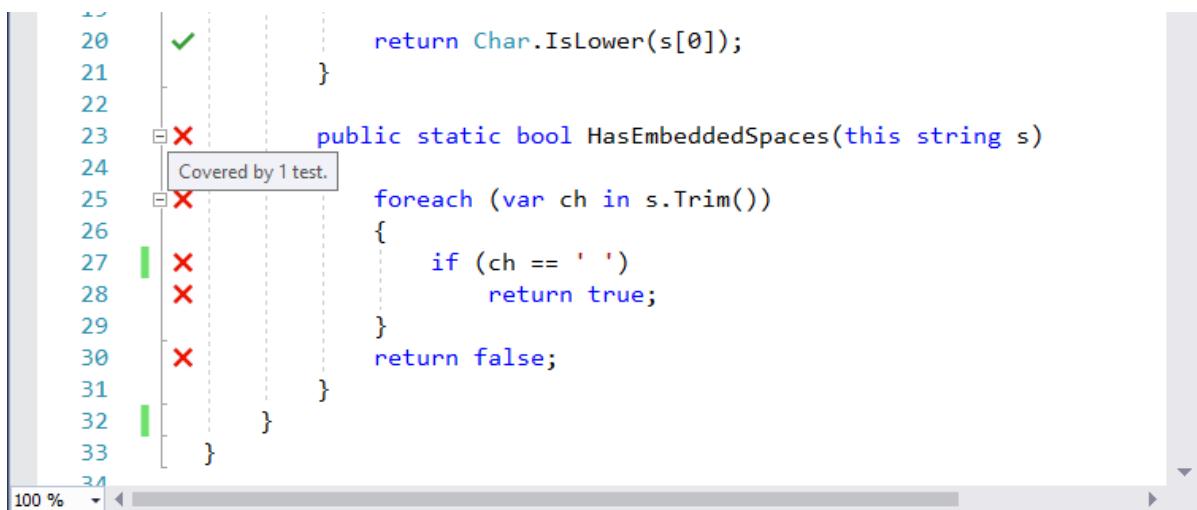
```

2. When the test executes, Live Unit Testing indicates that the `TestHasEmbeddedSpaces` method has failed, as the following image shows:



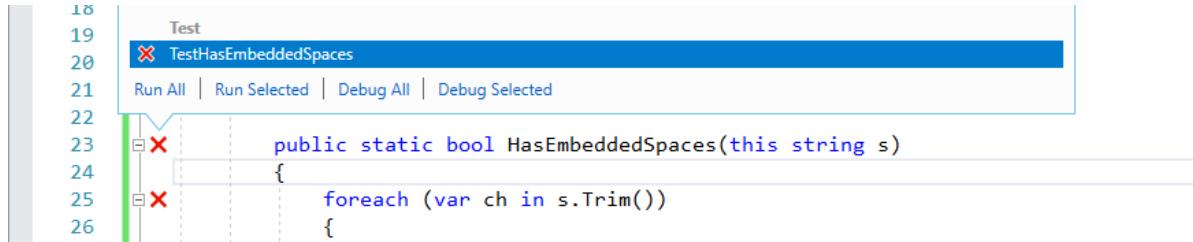
3. Select the window that displays the library code. Live Unit Testing has expanded code coverage to the `HasEmbeddedSpaces` method. It also reports the test failure by adding a red "✗" to lines covered by failing tests.

4. Hover over the line with the `HasEmbeddedSpaces` method signature. Live Unit Testing displays a tooltip that reports that the method is covered by one test, as the following image shows:



5. Select the failed `TestHasEmbeddedSpaces` test. Live Unit Testing gives you a number of options, such as running all tests, running the selected tests, debugging all tests, and debugging selected tests, as the

following image shows:



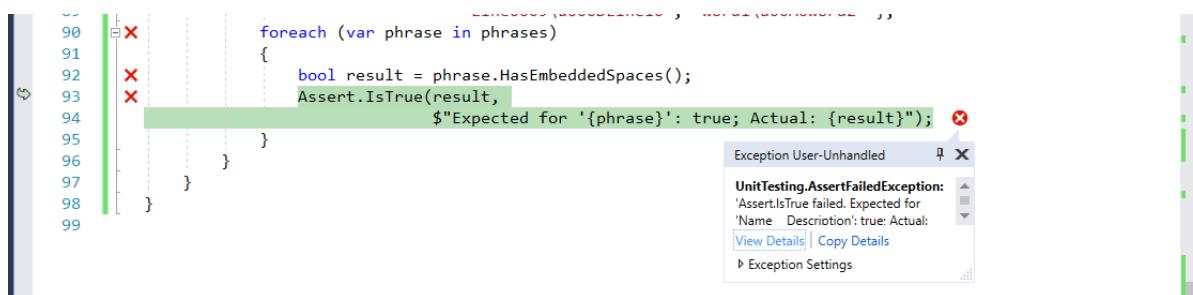
6. Select **Debug Selected** to debug the failed test.

7. Visual Studio executes the test in debug mode.

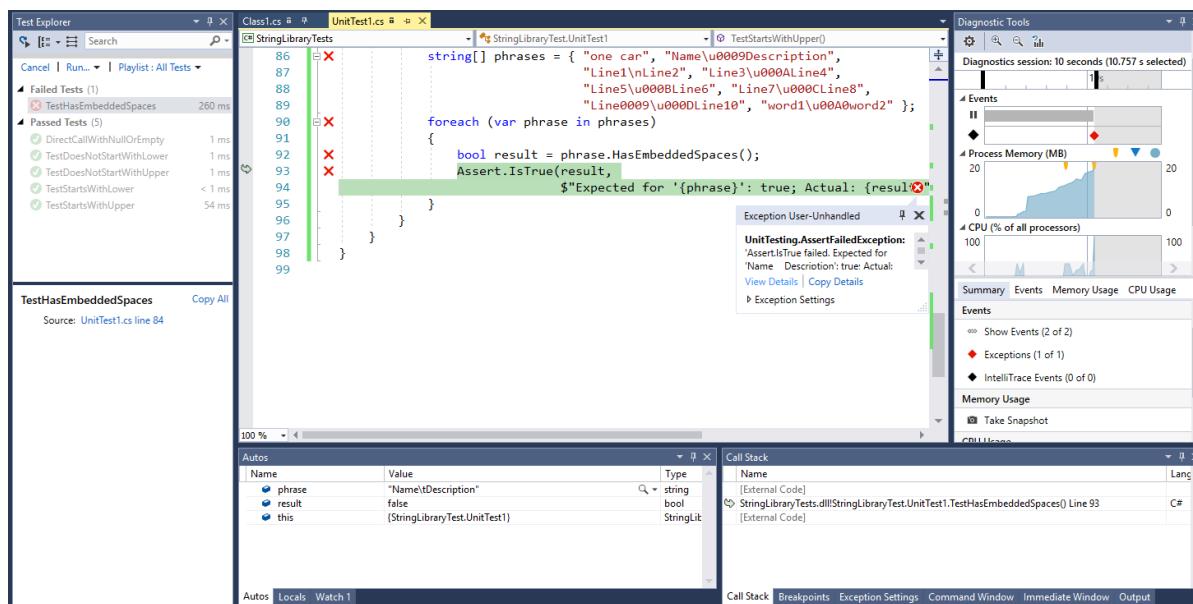
The test assigns each string in an array to a variable named `phrase` and passes it to the `HasEmbeddedSpaces` method. Program execution pauses and invokes the debugger the first time the assert expression is `false`.

The exception dialog that results from the unexpected value in the

`Microsoft.VisualStudio.TestTools.UnitTesting.Assert.IsTrue` method call is shown in the following image.



In addition, all of the debugging tools that Visual Studio provides are available to help us troubleshoot our failed test, as the following image shows:



Note in the **Autos** window that the value of the `phrase` variable is "Name\tDescription", which is the second element of the array. The test method expects `HasEmbeddedSpaces` to return `true` when it is passed this string; instead, it returns `false`. Evidently, it does not recognize "\t", the tab character, as an embedded space.

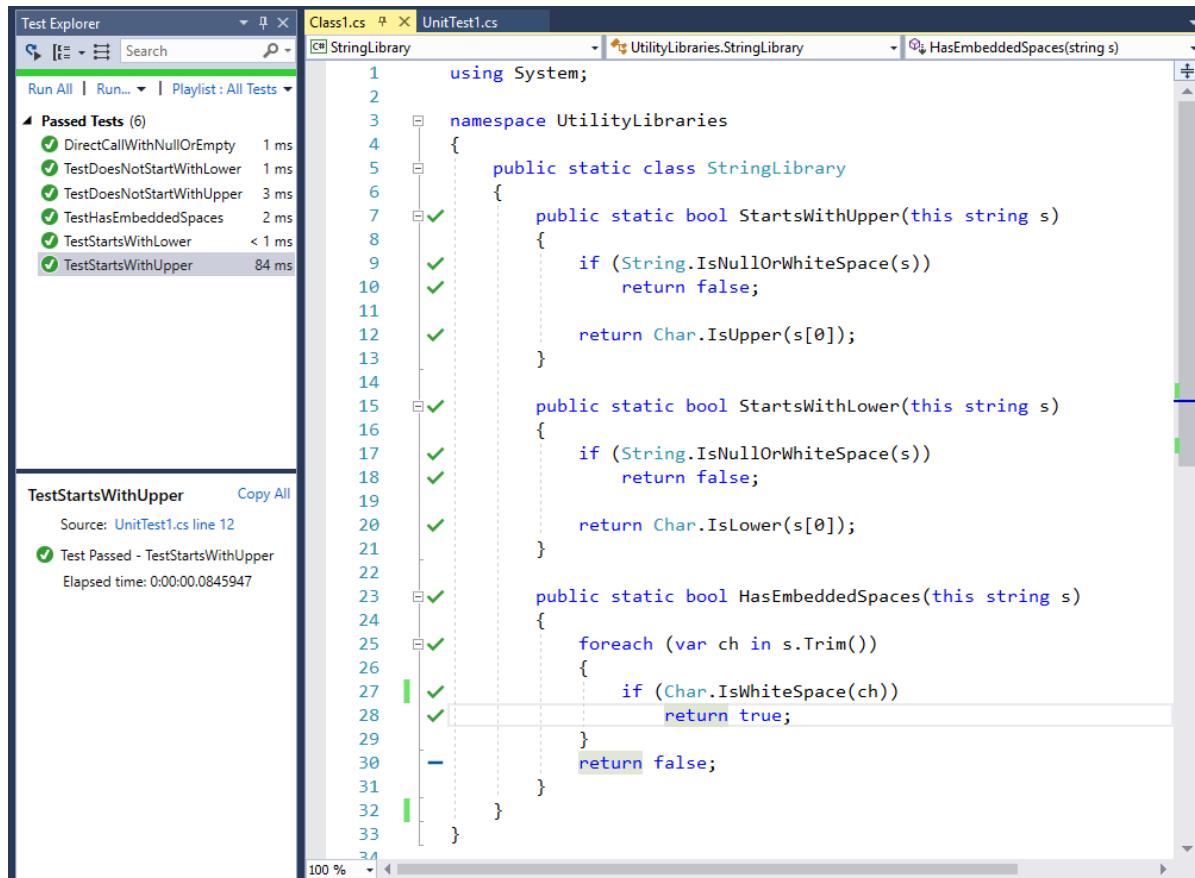
8. Select **Debug > Continue**, press **F5**, or click the **Continue** button on the toolbar to continue executing the test program. Because an unhandled exception occurred, the test terminates. This provides enough information for a preliminary investigation of the bug. Either `TestHasEmbeddedSpaces` (the test routine) made an incorrect assumption, or `HasEmbeddedSpaces` does not correctly recognize all embedded spaces.

9. To diagnose and correct the problem, start with the `StringLibrary.HasEmbeddedSpaces` method. Look at the comparison in the `HasEmbeddedSpaces` method. It considers an embedded space to be U+0020. However, the Unicode Standard includes a number of other space characters. This suggests that the library code has incorrectly tested for a whitespace character.

10. Replace the equality comparison with a call to the `System.Char.IsWhiteSpace` method:

```
if (Char.IsWhiteSpace(ch))
```

11. Live Unit Testing automatically reruns the failed test method and updates the results in the code window and in **Test Explorer**, as the following image shows:



See also

- [Live Unit Testing in Visual Studio](#)
- [Live Unit Testing Frequently Asked Questions](#)

How to configure and use Live Unit Testing

1/10/2020 • 10 minutes to read • [Edit Online](#)

As you're developing an application, Live Unit Testing automatically runs any impacted unit tests in the background and presents the results and code coverage in real time. As you modify your code, Live Unit Testing provides feedback on how your changes impacted existing tests and whether the new code you've added is covered by one or more existing tests. This gently reminds you to write unit tests as you're making bug fixes or adding new features.

NOTE

Live Unit Testing is available for C# and Visual Basic projects that target .NET Core or .NET Framework in the Enterprise edition of Visual Studio.

When you use Live Unit Testing for your tests, it persists data about the status of your tests. Using persisted data allows Live Unit Testing to offer superior performance while running your tests dynamically in response to code changes.

Supported test frameworks

Live Unit Testing works with the three popular unit testing frameworks listed in the following table. The minimum supported version of their adapters and frameworks is also shown. The unit testing frameworks are all available from NuGet.org.

TEST FRAMEWORK	VISUAL STUDIO ADAPTER MINIMUM VERSION	FRAMEWORK MINIMUM VERSION
xUnit.net	xunit.runner.visualstudio version 2.2.0-beta3-build1187	xunit 1.9.2
NUnit	NUnit3TestAdapter version 3.5.1	NUnit version 3.5.0
MSTest	MSTest.TestAdapter 1.1.4-preview	MSTest.TestFramework 1.0.5-preview

If you have older MSTest-based test projects that reference Microsoft.VisualStudio.QualityTools.UnitTestingFramework, and you don't wish to move to the newer MSTest NuGet packages, upgrade to Visual Studio 2019 or Visual Studio 2017.

In some cases, you may need to explicitly restore the NuGet packages referenced by a project in order for Live Unit Testing to work. You can do this either by doing an explicit build of the solution (select **Build > Rebuild Solution** from the top-level Visual Studio menu) or by restoring packages in the solution (right-click on the solution and select **Restore NuGet Packages**).

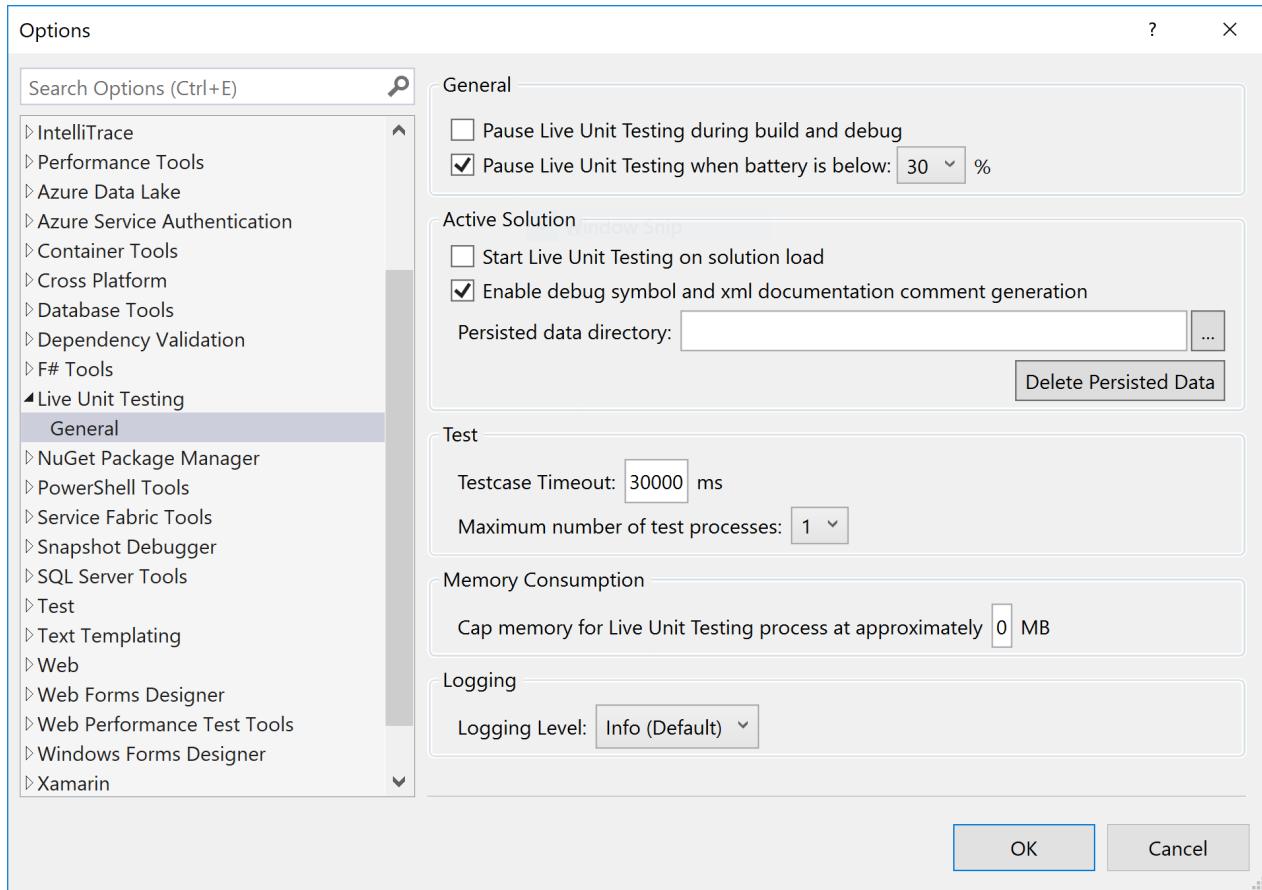
Configure

Configure Live Unit Testing by selecting **Tools > Options** from the top-level Visual Studio menu bar, and then selecting **Live Unit Testing** in the left pane of the **Options** dialog.

TIP

After Live Unit Testing is enabled (see the next section, [Start, pause, and stop Live Unit Testing](#)), you can also open the **Options** dialog by selecting **Test > Live Unit Testing > Options**.

The following image shows the Live Unit Testing configuration options available in the dialog:



The configurable options include:

- Whether Live Unit Testing pauses when a solution is built and debugged.
- Whether Live Unit Testing pauses when a system's battery power falls below a specified threshold.
- Whether Live Unit Testing runs automatically when a solution is opened.
- Whether to enable debug symbol and XML documentation comment generation.
- The directory in which to store persisted data.
- The ability to delete all persisted data. This is useful when Live Unit Testing is behaving in an unpredictable or unexpected way, which suggests that the persisted data has become corrupted.
- The interval after which a test case times out. The default is 30 seconds.
- The maximum number of test processes that Live Unit Testing creates.
- The maximum amount of memory that Live Unit Testing processes can consume.
- The level of information written to the Live Unit Testing **Output** window.

Options include no logging (**None**), error messages only (**Error**), error and informational messages (**Info**, the default), or all detail (**Verbose**).

You can also display verbose output in the Live Unit Testing **Output** window by assigning a value of "1" to

a user-level environment variable named `VS_UTE_DIAGNOSTICS`, and then restarting Visual Studio.

To capture detailed MSBuild log messages from Live Unit Testing in a file, set the `LiveUnitTesting_BuildLog` user-level environment variable to the name of the file to contain the log.

Start, pause, and stop

To enable Live Unit Testing, select **Test > Live Unit Testing > Start** from the top-level Visual Studio menu. When Live Unit Testing is enabled, the options available on the **Live Unit Testing** menu change from a single item, **Start, to Pause and Stop**:

- **Pause** temporarily suspends Live Unit Testing.

When Live Unit Testing is paused, coverage visualization does not appear in the editor, but all the data that was collected is preserved. To resume Live Unit Testing, select **Continue** from the Live Unit Testing menu. Live Unit Testing does the necessary work to catch up with all the edits that have been made while it was paused and updates the glyphs appropriately.

- **Stop** completely stops Live Unit Testing. Live Unit Testing discards all data that it has collected.

NOTE

If you start Live Unit Testing in a solution that does not include a unit test project, the **Pause** and **Stop** options appear on the **Live Unit Testing** menu, but Live Unit Testing does not start. The **Output** window displays a message that begins, "No supported test adapters are referenced by this solution...".

At any time, you can temporarily pause or completely stop Live Unit Testing. You may want to do this, for example, if you're in the middle of a refactoring and know that your tests will be broken for a while.

View coverage visualization

After it's enabled, Live Unit Testing updates each line of code in the Visual Studio editor to show you whether the code you're writing is covered by unit tests and whether the tests that cover it are passing. The following image shows lines of code with both passing and failing tests, as well as lines of code that are not covered by tests. Lines decorated with a green "✓" are covered only by passing tests, lines decorated with a red "x" are covered by one or more failing tests, and lines decorated by a blue "□" are not covered by any test.

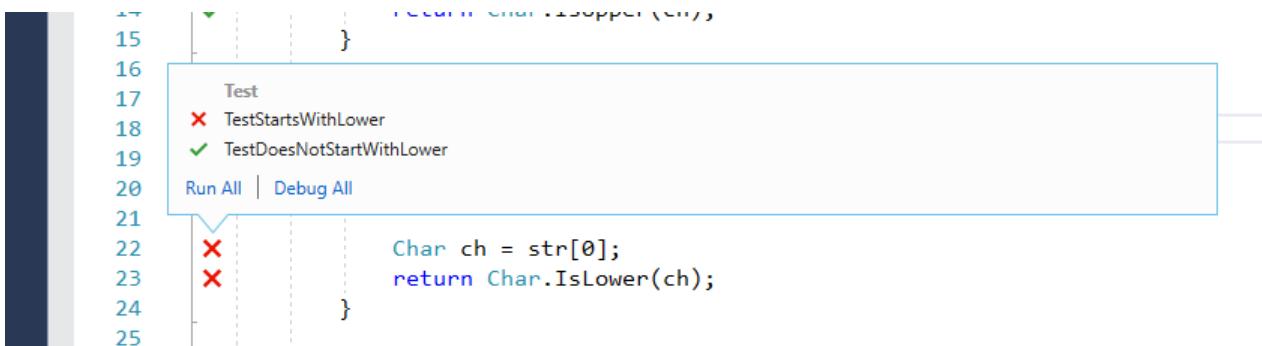
```
1  using System;
2  using System.Text.RegularExpressions;
3
4  namespace UtilityLibraries
5  {
6      public static class StringLibrary
7      {
8          public static bool StartsWithUpper(this String str)
9          {
10             if (String.IsNullOrWhiteSpace(str))
11                 return false;
12
13             Char ch = str[0];
14             return Char.IsUpper(ch);
15         }
16
17         public static bool StartsWithLower(this String str)
18         {
19             if (String.IsNullOrWhiteSpace(str))
20                 return false;
21
22             Char ch = str[0];
23             return Char.IsLower(ch);
24         }
25
26         public static int GetWordCount(this String str)
27         {
28             string pattern = @"\w+";
29             return Regex.Matches(str, pattern).Count;
30         }
31     }
32 }
33
```

Live Unit Testing coverage visualization is updated immediately when you modify code in the code editor. While processing the edits, visualization changes to indicate that the data is not up-to-date by adding a round timer image below the passing, failing, and not covered symbols, as the following image shows.

```
1  using System;
2  using System.Text.RegularExpressions;
3
4  namespace UtilityLibraries
5  {
6      public static class StringLibrary
7      {
8          public static bool StartsWithUpper(this String str)
9          {
10             if (String.IsNullOrWhiteSpace(str))
11                 return false;
12
13             Char ch = str[0];
14             return Char.IsUpper(ch);
15         }
16
17         public static bool StartsWithLower(this String str)
18         {
19             if (String.IsNullOrWhiteSpace(str))
20                 return false;
21
22             Char ch = str[0];
23             return Char.IsLower(ch);
24         }
25
26         public static int GetWordCount(this String str)
27         {
28             string pattern = @"\w+";
29             return Regex.Matches(str, pattern).Count;
30         }
31     }
32 }
33
```

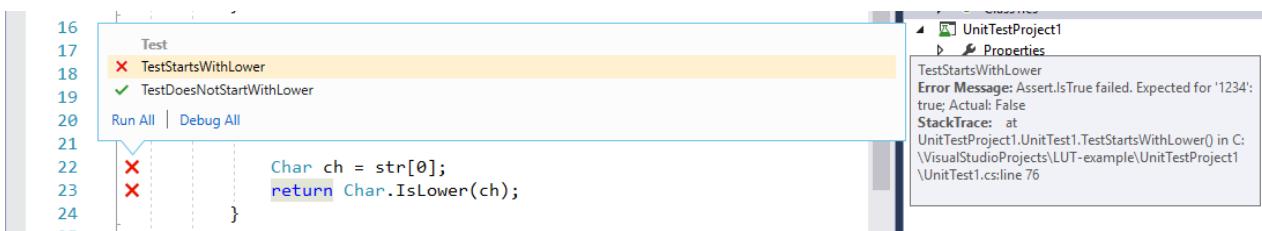
Get information about test status

By hovering over the succeeded or failed symbol in the code window, you can see how many tests are hitting that line. To see the status of the individual tests, select the symbol:



In addition to providing the names and result of tests, the tooltip lets you rerun or debug the set of tests. If you select one or more of the tests in the tooltip, you can also run or debug just those tests. This allows you to debug your tests without having to leave the code window. When debugging, in addition to observing any breakpoints you may have already set, program execution pauses when the debugger executes an `Assert` method that returns an unexpected result.

When you hover over a failed test in the tooltip, it expands to provide additional info about the failure, as shown in the following image. To navigate directly to a failed test, double-click it in the tooltip.



When you navigate to the failed test, Live Unit Testing visually indicates in the method signature the tests that have:

- passed (indicated by a half-full beaker along with a green "✓")
- failed (a half-full beaker along with a red "✗")
- are not involved in Live Unit Testing (a half-full beaker along with a blue "□")

Non-test methods are not decorated with a symbol. The following image illustrates all four types of methods.

```
67
68
69     [TestMethod]
70     public void TestStartsWithLower()
71     {
72         // Tests that we expect to return true.
73         string[] words = { "alphabet", "1234", "zebra", "abc",
74             "αυτοκινητοβιομηχανία", "государство" };
75         foreach (var word in words)
76         {
77             bool result = word.StartsWithLower();
78             Assert.IsTrue(result,
79                 String.Format($"Expected for '{word}': true; Actual: {result}"));
80         }
81     }
82
83     [TestMethod]
84     public void LowerCallWithNullOrEmpty()
85     {
86         // Tests that we expect to return false.
87         string[] words = { string.Empty, null };
88         foreach (var word in words)
89         {
90             bool result = StringLibrary.StartsWithLower(word);
91             Assert.IsFalse(result,
92                 String.Format("Expected for '{0}': false; Actual: {1}",
93                     word == null ? "<null>" : word, result));
94         }
95     }
96
97     [TestMethod]
98     public void TestIfNull(object obj)
99     {
100        DetermineIfNull(obj);
101    }
102
103    private void DetermineIfNull(object obj)
104    {
105        Assert.IsNotNull(obj);
106    }

```

Diagnose and correct test failures

From the failed test, you can easily debug the product code, make edits, and continue developing your application. Because Live Unit Testing runs in the background, you don't have to stop and restart Live Unit Testing during the debug, edit, and continue cycle.

For example, the test failure shown in the previous image was caused by an incorrect assumption in the test method that non-alphabetic characters return `true` when passed to the `System.Char.IsLower` method. After you correct the test method, all the tests should pass. You don't have to pause or stop Live Unit Testing.

Test Explorer

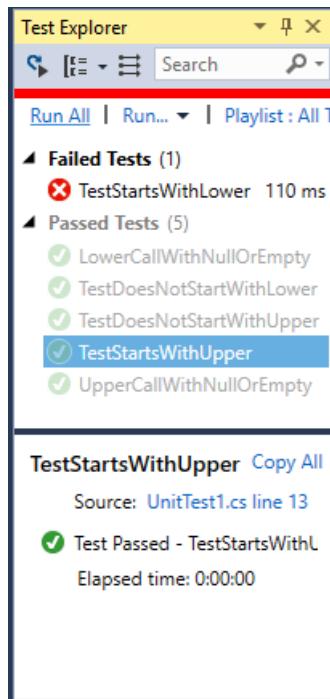
Test Explorer provides an interface that lets you run and debug tests and analyze test results. Live Unit Testing integrates with **Test Explorer**. When Live Unit Testing is not enabled or is stopped, **Test Explorer** displays the status of unit tests the last time a test was run. Source code changes require that you rerun the tests. In contrast, when Live Unit Testing is enabled, the status of unit tests in **Test Explorer** is updated immediately. You don't need to explicitly run the unit tests.

TIP

Open **Test Explorer** by selecting **Test > Windows > Test Explorer** from the top-level Visual Studio menu.

You may notice in the **Test Explorer** window that some tests are faded out. For example, when you enable Live Unit Testing after opening a previously saved project, the **Test Explorer** window had faded out all but the failed

test, as the following image shows. In this case, Live Unit Testing has rerun the failed test, but it has not rerun the successful tests. This is because Live Unit Testing's persisted data indicates that there were no changes since the tests were last run successfully.



You can rerun any tests that appear faded by selecting the **Run All** or **Run** options from the **Test Explorer** menu. Or, select one or more tests in the **Test Explorer** menu, right-click, and then select **Run Selected Tests** or **Debug Selected Tests** from the popup menu. As tests are run, they bubble up the top.

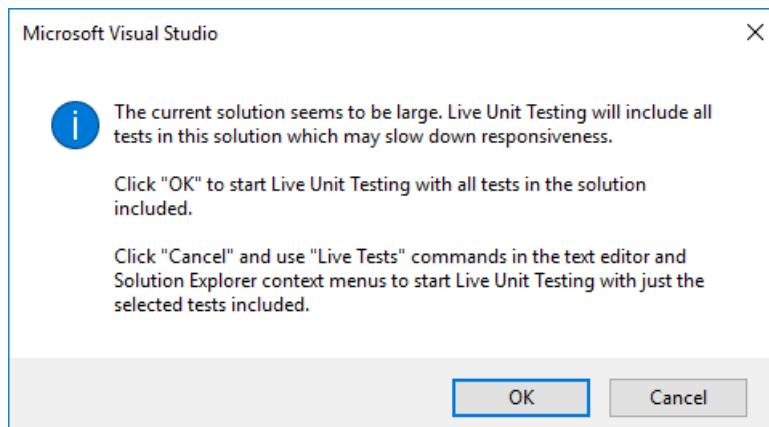
There are some differences between Live Unit Testing automatically running and updating test results and explicitly running tests from **Test Explorer**. These differences include:

- Running or debugging tests from the Test Explorer window runs regular binaries, whereas Live Unit Testing runs instrumented binaries.
- Live Unit Testing does not create a new application domain to run tests, but rather runs tests from the default domain. Tests run from the **Test Explorer** window do create a new application domain.
- Live Unit Testing runs tests in each test assembly sequentially. In the **Test Explorer** window, you can choose to run multiple tests in parallel.

Large solutions

If your solution has 10 or more projects, Visual Studio displays the following dialog when you:

- start Live Unit Testing and there is no persisted data
- select **Tools > Options > Live Unit Testing > Delete Persisted Data**



The dialog warns you that dynamic execution of large numbers of tests in large projects can severely impact performance. If you select **OK**, Live Unit Testing executes all tests in the solution. If you select **Cancel**, you can select the tests to execute. The following section explains how to do this.

Include and exclude test projects and test methods

For solutions with many test projects, you can control which projects and individual methods in a project participate in Live Unit Testing. For example, if you have a solution with hundreds of test projects, you can select a targeted set of test projects to participate in Live Unit Testing. There are a number of ways to do this, depending on whether you want to exclude all the tests in the project or solution, include or exclude most tests, or exclude individual tests. Live Unit Testing saves include/exclude state as a user setting and remembers it when a solution is closed and reopened.

Exclude all tests in a project or solution

To select the individual projects in unit tests, do the following after Live Unit Testing is started:

1. Right-click the solution in **Solution Explorer** and choose **Live Tests > Exclude** to exclude the entire solution.
2. Right-click each test project that you'd like to include in the tests and choose **Live Tests > Include**.

Exclude individual tests from the code editor window

You can use the code editor window to include or exclude individual test methods. Right-click on the signature of the test method in the code editor window, and then select one of the following options:

- **Live Tests > Include <selected method>**
- **Live Tests > Exclude <selected method>**
- **Live Tests > Exclude All But <selected method>**

Exclude tests programmatically

You can apply the [ExcludeFromCodeCoverageAttribute](#) attribute to programmatically exclude methods, classes, or structures from reporting their coverage in Live Unit Testing.

Use the following attributes to exclude individual methods from Live Unit Testing:

- For xUnit: `[Trait("Category", "SkipWhenLiveUnitTesting")]`
- For NUnit: `[Category("SkipWhenLiveUnitTesting")]`
- For MSTest: `[TestCategory("SkipWhenLiveUnitTesting")]`

Use the following attributes to exclude an entire assembly of tests from Live Unit Testing:

- For xUnit: `[assembly: AssemblyTrait("Category", "SkipWhenLiveUnitTesting")]`
- For NUnit: `[assembly: Category("SkipWhenLiveUnitTesting")]`
- For MSTest: `[assembly: TestCategory("SkipWhenLiveUnitTesting")]`

See also

- [Code testing tools](#)
- [Live Unit Testing blog](#)
- [Live Unit Testing FAQ](#)
- [Channel 9 video: Live Unit Testing in Visual Studio](#)

Live Unit Testing frequently asked questions

1/1/2020 • 10 minutes to read • [Edit Online](#)

Supported frameworks

What test frameworks does Live Unit Testing support and what are the minimum supported versions?

Live Unit Testing works with the three popular unit testing frameworks listed in the table that follows. The minimum supported version of their adapters and frameworks is also listed in the table. The unit testing frameworks are all available from NuGet.org.

TEST FRAMEWORK	VISUAL STUDIO ADAPTER MINIMUM VERSION	FRAMEWORK MINIMUM VERSION
xUnit.net	xunit.runner.visualstudio version 2.2.0-beta3-build1187	xunit 1.9.2
NUnit	NUnit3TestAdapter version 3.7.0	NUnit version 3.5.0
MSTest	MSTest.TestAdapter 1.1.4-preview	MSTest.TestFramework 1.0.5-preview

If you have older MSTest based test projects that reference

`Microsoft.VisualStudio.QualityTools.UnitTestFramework` and you don't wish to move to the newer MSTest NuGet packages, upgrade to Visual Studio 2019 or Visual Studio 2017.

In some cases, you may need to explicitly restore the NuGet packages referenced by the projects in the solution in order for Live Unit Testing to work. You can restore the packages either by doing an explicit build of the solution (select **Build > Rebuild Solution** from the top-level Visual Studio menu), or by right-clicking on the solution and selecting **Restore NuGet Packages** before enabling Living Unit Testing.

.NET Core support

Does Live Unit Testing work with .NET Core?

Yes. Live Unit Testing works with .NET Core and the .NET Framework.

Configuration

Why doesn't Live Unit Testing work when I turn it on?

The Output window (when the Live Unit Testing drop-down is selected) should tell you why Live Unit Testing is not working. Live Unit Testing may not work for one of the following reasons:

- If NuGet packages referenced by the projects in the solution have not been restored, Live Unit Testing will not work. Doing an explicit build of the solution or restoring NuGet packages in the solution before turning on Live Unit Testing should resolve this issue.
- If you are using MSTest-based tests in your projects, make sure that you remove the reference to `Microsoft.VisualStudio.QualityTools.UnitTestFramework`, and add references to the latest MSTest NuGet packages, `MSTest.TestAdapter` (a minimum version of 1.1.11 is required) and `MSTest.TestFramework` (a minimum version of 1.1.11 is required). For more information, see the "Supported test frameworks" section of the [Use Live Unit Testing in Visual Studio](#) article.

- At least one project in your solution should have either a NuGet reference or direct reference to the xUnit, NUnit, or MSTest test framework. This project should also reference a corresponding Visual Studio test adapters NuGet package. The Visual Studio test adapter can also be referenced through a *.runsettings* file. The *.runsettings* file must have an entry like the following example:

```
<RunSettings>
  <RunConfiguration>
    <TestAdaptersPaths>path-to-your-test-adapter</TestAdaptersPaths>
  </RunConfiguration>
</RunSettings>
```

Incorrect coverage after upgrade

Why does Live Unit Testing show incorrect coverage after you upgrade the test adapter referenced in your Visual Studio Projects to the supported version?

- If multiple projects in the solution reference the NuGet test adapter package, each of them must be upgraded to the supported version.
- Make sure the MSBuild *.props* file imported from the test adapter package is correctly updated as well. Check the NuGet package version/path of the import, which can usually be found near the top of the project file, like the following:

```
<Import
Project=".\\packages\\xunit.runner.visualstudio.2.2.0\\build\\net20\\xunit.runner.visualstudio.props"
Condition="Exists('.\\packages\\xunit.runner.visualstudio.2.2.0\\build\\net20\\xunit.runner.visualstudio.pr
ops')"/>
```

Customize builds

Can I customize my Live Unit Testing builds?

If your solution requires custom steps to build for instrumentation (Live Unit Testing) that are not required for the "regular" non-instrumented build, then you can add code to your project or *.targets* files that checks for the `BuildingForLiveUnitTesting` property and performs custom pre/post build steps. You can also choose to remove certain build steps (like publishing or generating packages) or to add build steps (like copying prerequisites) to a Live Unit Testing build based on this project property. Customizing your build based on this property does not alter your regular build in any way, and only impacts Live Unit Testing builds.

For example, there may be a target that produces NuGet packages during a regular build. You probably do not want NuGet packages to be generated after every edit you make. So you can disable that target in the Live Unit Testing build by doing something like the following:

```
<Target Name="GenerateNuGetPackages" BeforeTargets="AfterBuild" Condition="$(BuildingForLiveUnitTesting) != 'true'>
  <Exec Command="$(MSBuildThisFileDirectory)..\tools\GenPac" />
</Target>
```

Error messages with `<OutputPath>`, `<OutDir>` or `<IntermediateOutputPath>`

Why do I get the following error when Live Unit Testing tries to build my solution: "...appears to unconditionally set `<OutputPath>` or `<OutDir>`. Live Unit Testing will not execute tests from the output

assembly"?

You can get this error if the build process for your solution has custom logic that specifies where binaries should be generated. By default the location of your binaries depends on `<OutputPath>`, `<OutDir>` or `<IntermediateOutputPath>` as well as `<BaseOutputPath>` or `<BaseIntermediateOutputPath>`.

Live Unit Testing overrides those variables to ensure that build artifacts are dropped to a Live Unit Testing artifacts folder and will fail if your build process also overrides these variables.

There are two main approaches to make Live Unit Testing build successfully. For easier build configurations, you can base your output paths on `<BaseIntermediateOutputPath>`. For more complex configurations you can base your output paths on `<LiveUnitTestingBuildRootPath>`.

Overriding `<OutputPath>` / `<IntermediateOutputPath>` **conditionally based on** `<BaseOutputPath>` / `<BaseIntermediateOutputPath>`.

NOTE

To use this approach, each project needs to be able to build independently from one another. Do not have one project reference artifacts from another project during build. Do not have one project dynamically load assemblies from another project during runtime (for example call `Assembly.Loadfile("../..\Project2\Release\Project2.dll")`).

During build, Live Unit Testing automatically overrides the `<BaseOutputPath>` / `<BaseIntermediateOutputPath>` variables to target the Live Unit Testing artifacts folder.

For example, if your build overrides the as shown below:

```
<Project>
  <PropertyGroup>
    <OutputPath>$({SolutionDir})Artifacts\${Configuration}\bin\${MSBuildProjectName}</OutputPath>
  </PropertyGroup>
</Project>
```

then you can replace it with the following XML:

```
<Project>
  <PropertyGroup>
    <BaseOutputPath Condition="'$(BaseOutputPath)' == 
'${SolutionDir})Artifacts\${Configuration}\bin\${MSBuildProjectName}'></BaseOutputPath>
    <OutputPath Condition="'$(OutputPath)' == '$(BaseOutputPath)'"></OutputPath>
  </PropertyGroup>
</Project>
```

This ensures that `<OutputPath>` lies within the `<BaseOutputPath>` folder.

Do not override `<OutDir>` directly in your build process; override `<OutputPath>` instead to drop build artifacts to a specific location.

Overriding your properties based on the `<LiveUnitTestingBuildRootPath>` property.

NOTE

In this approach, you need to be careful about files added under the artifacts folder that are not generated during build. The example below shows what to do when placing the packages folder under artifacts. Because the contents of this folder are not generated during the build, the MSBuild property **should not be changed**.

During a Live Unit Testing build, the `<LiveUnitTestingBuildRootPath>` property is set to the location of Live Unit

Testing artifacts folder.

For example, assume that your project has the structure shown here.

```
.vs\...\lut\0\b  
artifacts\{binlog,obj,bin,nupkg,testresults,packages}  
src\{proj1,proj2,proj3}  
tests\{testproj1,testproj2}  
Solution.sln
```

During the Live Unit Testing build, the `<LiveUnitTestingBuildRootPath>` property is set to the full path of `.vs\...\lut\0\b`. If the project defines `<ArtifactsRoot>` property that maps to the solution dir, you can update the MSBuild project as follows:

```
<Project>  
  <PropertyGroup Condition="\"$(LiveUnitTestingBuildRootPath)" == '\"'>  
    <SolutionDir>$([MSBuild]::GetDirectoryNameOfFileAbove(`$(MSBuildProjectDirectory)`,  
`YOUR_SOLUTION_NAME.sln`))</SolutionDir>  
  
    <ArtifactsRoot>Artifacts\</ArtifactsRoot>  
    <ArtifactsRoot Condition="\"$(LiveUnitTestingBuildRootPath)" != '\"'>$(LiveUnitTestingBuildRootPath)  
  </ArtifactsRoot>  
  </PropertyGroup>  
  
<PropertyGroup>  
  <BinLogPath>$(ArtifactsRoot)\BinLog</BinLogPath>  
  <ObjPath>$(ArtifactsRoot)\Obj</ObjPath>  
  <BinPath>$(ArtifactsRoot)\Bin</BinPath>  
  <NupkgPath>$(ArtifactsRoot)\Nupkg</NupkgPath>  
  <TestResultsPath>$(ArtifactsRoot)\TestResults</TestResultsPath>  
  
  <!-- Note: Given that a build doesn't generate packages, the path should be relative to the solution  
dir, rather than artifacts root, which will change during a Live Unit Testing build. -->  
  <PackagesPath>$(SolutionDir)\artifacts\packages</PackagesPath>  
  </PropertyGroup>  
</Project>
```

Build artifact location

I want the artifacts of a Live Unit Testing build to go to a specific location instead of the default location under the .vs folder. How can I change that?

Set the `LiveUnitTesting_BuildRoot` user-level environment variable to the path where you want the Live Unit Testing build artifacts to be dropped.

Test Explorer versus Live Unit Testing

How is running tests from Test Explorer window different from running tests in Live Unit Testing?

There are several differences:

- Running or debugging tests from the **Test Explorer** window runs regular binaries, whereas Live Unit Testing runs instrumented binaries. If you want to debug instrumented binaries, adding a `Debugger.Launch` method call in your test method causes the debugger to launch whenever that method is executed (including when it is executed by Live Unit Testing), and you can then attach and debug the instrumented binary. However, our hope is that instrumentation is transparent to you for most user scenarios, and that you do not need to debug instrumented binaries.
- Live Unit Testing does not create a new application domain to run tests, but tests run from the **Test**

Explorer window do create a new application domain.

- Live Unit Testing runs tests in each test assembly sequentially. In **Test Explorer**, you can choose to run multiple tests in parallel.
- **Test Explorer** runs tests in a single-threaded apartment (STA) by default, whereas Live Unit Testing runs tests in a multi-threaded apartment (MTA). To run MSTest tests in STA in Live Unit Testing, decorate the test method or the containing class with the `<STATestMethod>` or `<STATestClass>` attribute that can be found in the `MSTest.STAExtensions 1.0.3-beta` NuGet package. For NUnit, decorate the test method with the `<RequiresThread(ApartmentState.STA)>` attribute, and for xUnit, with the `<STAFact>` attribute.

Exclude tests

How do I exclude tests from participating in Live Unit Testing?

See the "Include and exclude test projects and test methods" section of the [Use Live Unit Testing in Visual Studio](#) article for the user-specific setting. Including or excluding tests is useful when you want to run a specific set of tests for a particular edit session or to persist your own personal preferences.

For solution-specific settings, you can apply the

`System.Diagnostics.CodeAnalysis.ExcludeFromCodeCoverageAttribute` attribute programmatically to exclude methods, properties, classes, or structures from being instrumented by Live Unit Testing. Additionally, you can also set the `<ExcludeFromCodeCoverage>` property to `true` in your project file to exclude the whole project from being instrumented. Live Unit Testing will still run the tests that have not been instrumented, but their coverage will not be visualized.

You can also check whether `Microsoft.CodeAnalysis.LiveUnitTesting.Runtime` is loaded in the current application domain and disable tests based on why. For example, you can do something like the following with xUnit:

```
[ExcludeFromCodeCoverage]
public class SkipLiveFactAttribute : FactAttribute
{
    private static bool s_lutRuntimeLoaded = AppDomain.CurrentDomain.GetAssemblies().Any(a => a.GetName().Name
== "Microsoft.CodeAnalysis.LiveUnitTesting.Runtime");
    public override string Skip => s_lutRuntimeLoaded ? "Test excluded from Live Unit Testing" : "";
}

public class Class1
{
    [SkipLiveFact]
    public void F()
    {
        Assert.True(true);
    }
}
```

Win32 PE headers

Why are Win32 PE headers different in instrumented assemblies built by Live Unit testing?

This issue is fixed and does not exist in Visual Studio 2017 version 15.3 and later.

For older versions of Visual Studio 2017, there is a known bug that may result in Live Unit Testing builds failing to embed the following Win32 PE Header data:

- File Version (specified by `AssemblyFileVersionAttribute` in code).
- Win32 Icon (specified by `/win32icon:` on the command line).

- Win32 Manifest (specified by `/win32manifest:` on the command line).

Tests that rely on these values may fail when executed by Live Unit testing.

Continuous builds

Why does Live Unit testing keep building my solution all the time even if I am not making any edits?

Your solution can build even if you're not making edits if the build process generates source code that's part of the solution itself, and your build target files don't have appropriate inputs and outputs specified. Targets should be given a list of inputs and outputs so that MSBuild can perform the appropriate up-to-date checks and determine whether a new build is required.

Live Unit Testing starts a build whenever it detects that source files have changed. Because the build of your solution generates source files, Live Unit Testing gets into an infinite build loop. If, however, the inputs and outputs of the target are checked when Live Unit Testing starts the second build (after detecting the newly generated source files from the previous build), it breaks out of the build loop because the inputs and outputs checks indicate that everything is up-to-date.

Editor icons

Why do I not see any icons in the editor even though Live Unit Testing seems to be running the tests based on the messages in the Output window?

You might not see icons in the editor if the assemblies that Live Unit Testing is operating on aren't instrumented for any reason. For example, Live Unit Testing is not compatible with projects that set

`<UseHostCompilerIfAvailable>false</UseHostCompilerIfAvailable>`. In this case, your build process needs to be updated to either remove this setting or to change it to `true` for Live Unit Testing to work.

Capture logs

How do I collect more detailed logs to file bug reports?

You can do several things to collect more detailed logs:

- Go to **Tools > Options > Live Unit Testing** and change the logging option to **Verbose**. Verbose logging causes more detailed logs to be shown in the **Output** window.
- Set the `LiveUnitTesting_BuildLog` user environment variable to the name of the file you want to use to capture the MSBuild log. Detailed MSBuild log messages from Live Unit Testing builds can then be retrieved from that file.
- Set the `LiveUnitTesting_TestPlatformLog` user environment variable to `1` to capture the Test Platform log. Detailed Test Platform log messages from Live Unit Testing runs can then be retrieved from `[Solution Root]\.vs\[Solution Name]\log\[VisualStudio Process ID]`.
- Create a user-level environment variable named `vs_UTE_DIAGNOSTICS` and set it to 1 (or any value) and restart Visual Studio. Now you should see lots of logging in the **Output - Tests** tab in Visual Studio.

See also

- [Live Unit Testing](#)

Quickstart: Create a load test project

1/1/2020 • 3 minutes to read • [Edit Online](#)

In this 10-minute quickstart, you'll learn how to create and run a web performance and load test project in Visual Studio. Load tests execute web performance or unit tests to simulate many users accessing a server at the same time.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

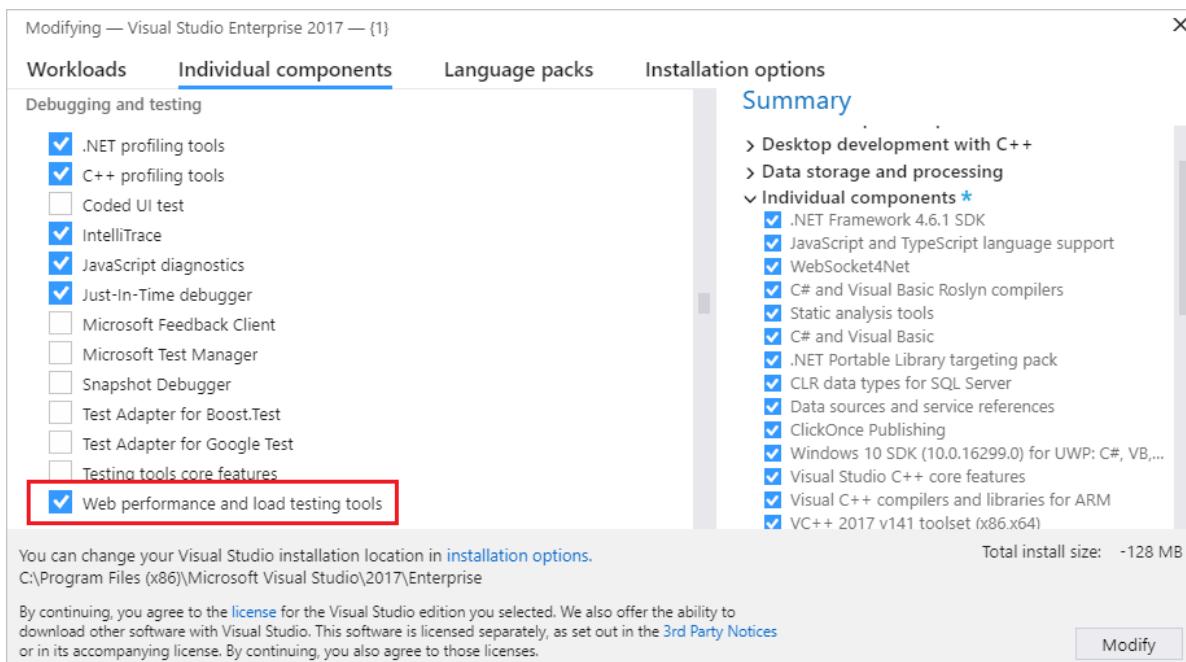
Software requirements

Web performance and load test projects are only available in the **Enterprise edition** of Visual Studio.

Install the load testing component

If you don't already have the web performance and load testing tools component installed, you'll need to install it through the Visual Studio Installer.

1. Open **Visual Studio Installer** from the **Start** menu of Windows. You can also access it in Visual Studio from the new project dialog box or by choosing **Tools > Get Tools and Features** from the menu bar.
2. In **Visual Studio Installer**, choose the **Individual components** tab, and scroll down to the **Debugging and testing** section. Select **Web performance and load testing tools**.



3. Choose the **Modify** button.

The web performance and load testing tools component is installed.

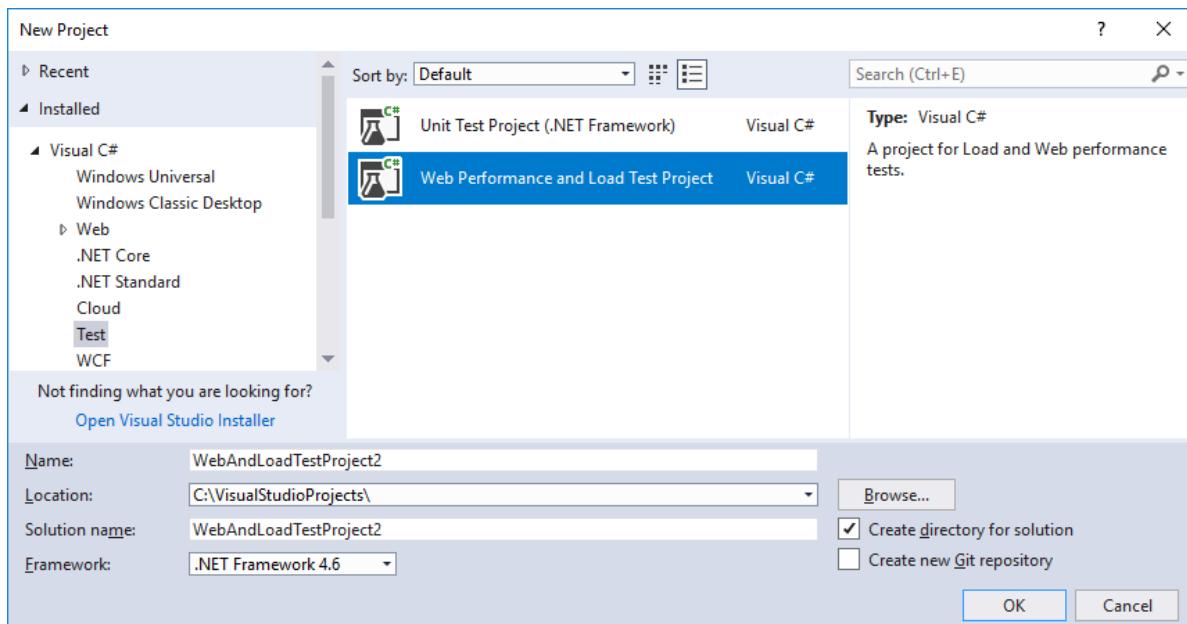
Create a load test project

In this section, we'll create a C# load test project. You can also create a Visual Basic load test project, if you prefer.

1. Open Visual Studio.
2. Choose **File > New > Project** from the menu bar.

The **New Project** dialog box opens.

3. In the **New Project** dialog box, expand **Installed** and **Visual C#**, and then select the **Test** category. Choose the **Web Performance and Load Test Project** template.



4. Enter a name for the project if you don't want to use the default name, and then choose **OK**.

1. Open Visual Studio.
2. On the start window, choose **Create a new project**.
3. On the **Create a new project** page, type **web test** into the search box, and then select the **Web Performance and Load Test Project [Deprecated]** template for C#. Choose **Next**.
4. Enter a name for the project if you don't want to use the default name, and then choose **Create**.

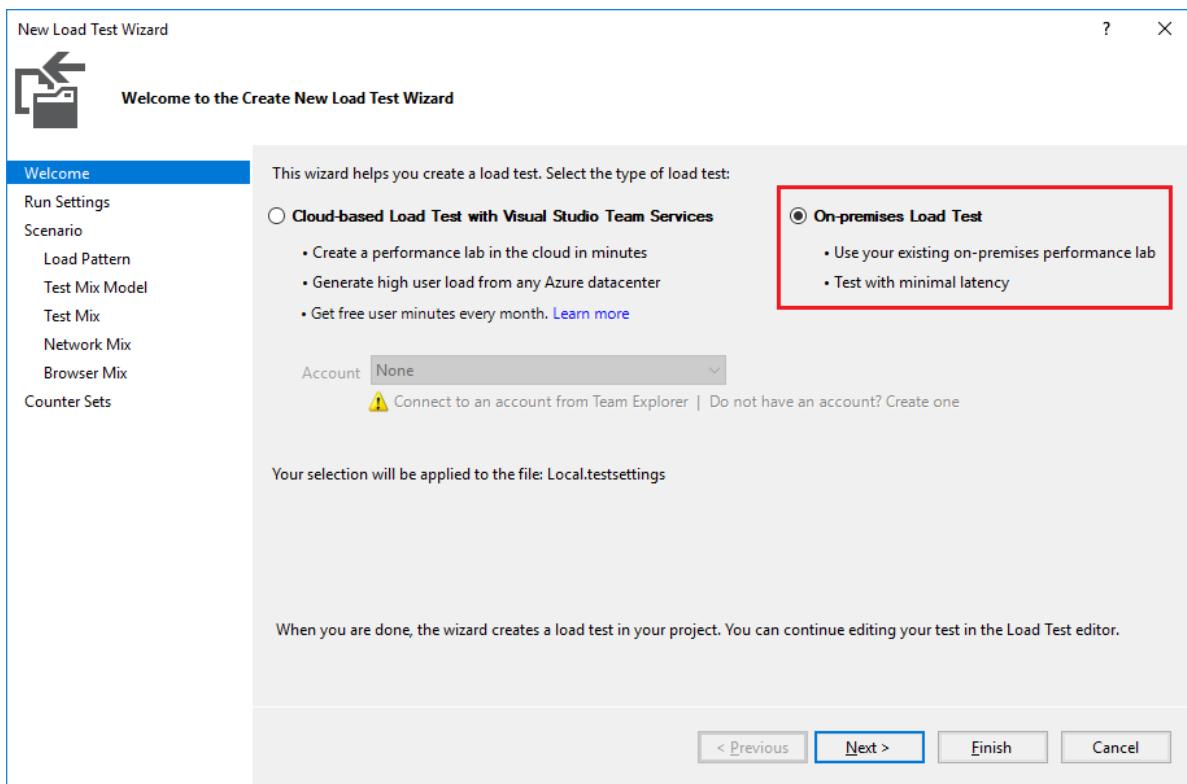
Visual Studio creates the project and displays the files in **Solution Explorer**. The project initially contains one web test file named *WebTest1.webtest*.

Add a load test to the project

1. From the right-click menu, or context menu, of the project node in **Solution Explorer**, choose **Add > Load Test**.

The **New Load Test Wizard** opens.

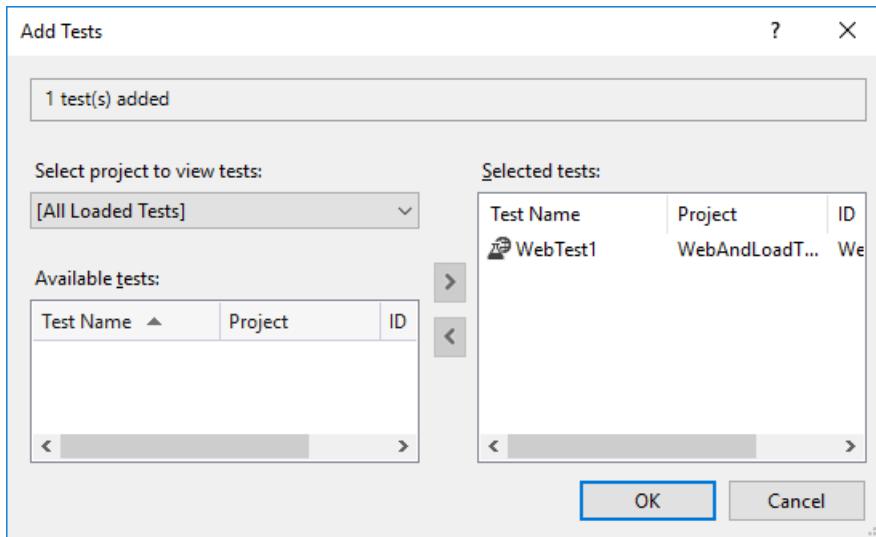
2. Select the **On-premises Load Test** option, and then choose **Next**. You can learn more about cloud-based load testing [here](#).



3. Choose **Next** to step through the wizard until you reach the **Add tests to a load test scenario and edit the test mix** page. Choose the **Add** button.

The **Add Tests** dialog box opens.

4. Under **Available tests**, select **WebTest1**, and then choose the right arrow to move it over to the **Selected tests** box. Choose the **OK** button.



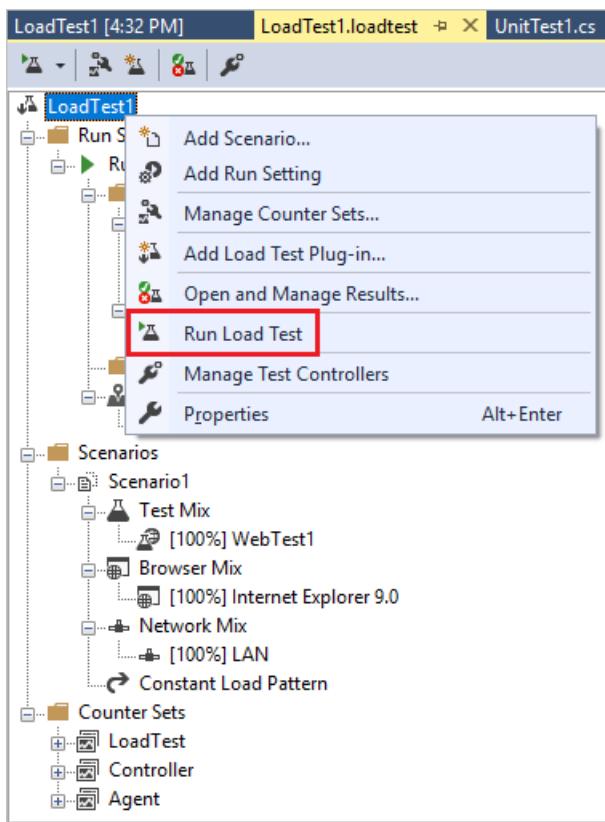
5. Back in the **New Load Test Wizard**, choose the **Finish** button.

The load test is added to the project, and the load test file opens in the editor window.

Run the load test

We've created a load test that doesn't do very much, but let's run it anyway.

From the right-click menu, or context menu, of the load test that's open in the editor, choose **Run Load Test**.



The load test starts running. The **Test Results** window shows that the test is in progress, and the load test analyzer is displayed in the editor window. After the test completes, which should be five minutes if you accepted the defaults, a summary is shown in the editor. You can choose **Graphs**, **Tables**, or **Detail** to get different information about the results of the load test.

The screenshot shows the 'LoadTest1 [4:32 PM]' window with the 'Summary' tab selected (highlighted with a red box). The status bar indicates 'Test Completed' with '40 threshold violations'. The main area displays the 'Load Test Summary' with sections for 'Test Run Information' and 'Key Statistic: Top 5 Slowest Tests'. Below this is the 'Overall Results' table, and at the bottom is the 'Test Results' table.

Test Run Information		Key Statistic: Top 5 Slowest Tests	
Load test name	LoadTest1	Name	95% Test Time (sec)
Description		WebTest1	0.000080
Start time	3/14/2018 4:32:25 PM		
End time	3/14/2018 4:37:25 PM		
Warm-up duration	00:00:00		
Duration	00:05:00		
Controller	Local run		
Number of agents	1		
Run settings used	Run Settings1		

Overall Results	
Max User Load	25
Tests/Sec	1,501
Tests Failed	0
Avg. Test Time (sec)	0.0000024
Transactions/Sec	0
Avg. Transaction Time (sec)	0
Pages/Sec	0
Avg. Page Time (sec)	0
Requests/Sec	0
Requests Failed	0
Requests Cached Percentage	-
Avg. Response Time (sec)	0
Avg. Content Length (bytes)	0

Test Results				
Name	Scenario	Total Tests	Failed Tests (% of total)	Avg. Test Time (sec)
WebTest1	Scenario1	450,309	0 (0)	0.0000024

Next steps

Now that you've created a simple load test project, the next step is to configure scenarios, counter sets, and run settings.

[Edit test settings](#)

Edit load tests

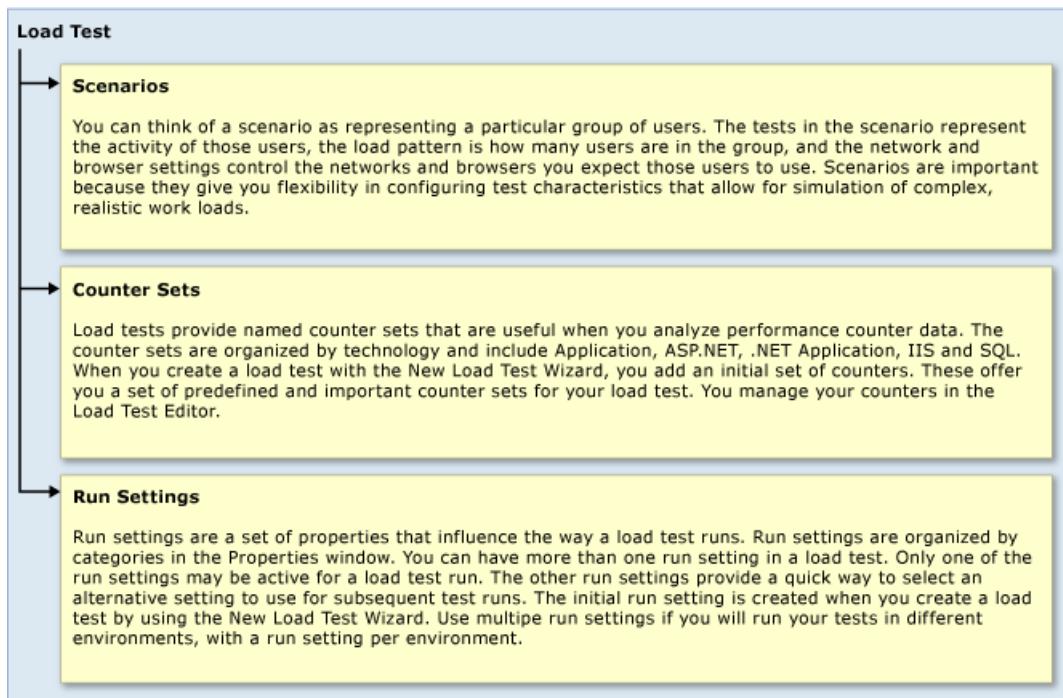
1/1/2020 • 2 minutes to read • [Edit Online](#)

Load tests run web performance tests or unit tests to simulate many users accessing a server at the same time. A load test gives you access to application stress and performance data. A load test can be configured to emulate various load conditions such as user loads and network types.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

A load test is defined by *scenarios*, *counter sets*, and *run settings*. The following illustration explains the differences between [scenarios](#), [counter sets](#), and [run settings](#):



Software requirements

Web performance and load test projects are only available in the Enterprise edition of Visual Studio.

Edit load test scenario settings

A scenario is used to model how a group of users interacts with a server application. A scenario consists of a load pattern, a test mix model, a test mix, a browser mix, and network mix. A load test can have more than one scenario and a single scenario can contain web performance tests and unit tests. By grouping similar settings together, a scenario lets you to group and run tests of a similar nature together.

For more information, see [Edit load test scenarios](#) and [Load test scenario properties](#).

Configure and manage performance counter sets

Load tests provide named counter sets, organized by technology, that are useful when you analyze performance

counter data. The counter sets include Load Test, IIS, ASP.NET, and SQL. When you create a load test with the **New Load Test Wizard**, an initial set of predefined and important counters is configured for the computers that you specify to include in the load test. You manage your counters in the **Load Test Editor**.

For more information, see [Specify the counter sets and threshold rules for computers in a load test](#).

Configure and manage load test run settings

Run settings are properties that influence the way a load test runs. Run settings are organized by categories in the **Properties** window.

For more information, see [Configure load test run settings](#) and [Load test run settings properties](#).

See also

- [Analyze load test results](#)
- [Analyzing threshold rule violations](#)

Edit load test scenarios

1/1/2020 • 5 minutes to read • [Edit Online](#)

A load test *scenario* specifies the load pattern, test mix, browser mix, and network mix. Scenarios are important because they enable you to configure tests to simulate complex, realistic workloads.

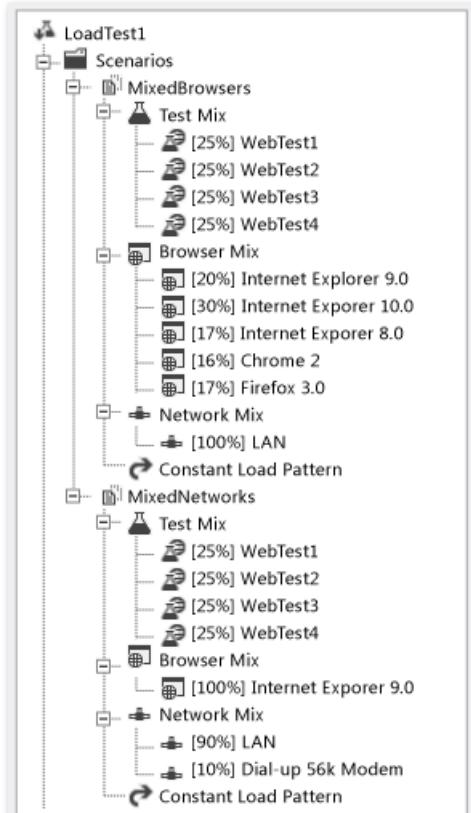
For example, you might be testing an e-commerce site that has an Internet front-end used by hundreds of concurrent customers coming in over many connection speeds and using different browsers. The same site might also have an administration function that is used by internal employees to update products and to view statistics. These internal users would typically access the site by using the same browser and a high-speed LAN connection. You would want to encapsulate the properties of these two different groups of users in different scenarios. Each scenario can contain a virtual user type. In this case, a load test scenario can be made to represent virtual customers and another scenario can be made to represent virtual internal users of a website.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Scenario components

Any initial configuration options and settings that you specify when you create a load test, can be modified later in the **Load Test Editor**. You can also add new scenarios, run settings, and counter sets to a load test.



Scenarios contain the following components:

TERM	DEFINITION
Browser Mix	Simulates that virtual users access a website through a variety of web browsers.
Load Pattern	Specifies the number of virtual users that are active during a load test, and the rate at which new users are started. For example: step, constant, and goal-based.
Test Mix Model	Specifies the probability of a virtual user running a given test in a load test scenario. For example: 20% chance to run TestA and 80% chance to run TestB. The test mix model should reflect the objectives of your test for a particular scenario.
Test Mix	The test mix is the selection of web performance and unit tests that constitute the scenario, and the distribution of those tests.
Network Mix	Simulates that virtual users access a website through a variety of network connections. The Network Mix offers options that include LAN, cable modem, and other options.

A scenario has several other properties that you can edit by using the **Load Test Editor**. For more information, see [Load test scenario properties](#).

Tasks

TASKS	ASSOCIATED TOPICS
Add artificial human interaction pauses in your scenario: Think times are used to simulate human behavior that causes people to wait between interactions with a website. Think times occur between requests in a web performance test and between test iterations in a load test scenario. Using think times in a load test can be useful in creating more accurate load simulations.	- Edit think times to simulate website human interaction delays
Specify the number of virtual users for your scenario: You can configure the load pattern properties to specify how the simulated user load is adjusted during a load test. You get three built-in load patterns: constant, step, and goal-based. You choose the load pattern and adjust the properties to appropriate levels for your load test goals.	- Edit load patterns to model virtual user activities
Configure the probability of a virtual user running a test in the scenario: You can use the test mix, which specifies the probability of a virtual user running a given test in a load test scenario. This lets you simulate load more realistically. Instead of having just one workflow through your applications, you can have several workflows, which is a closer approximation of how end-users interact with your applications.	- Edit test mix models
Add or remove a Web performance or unit test from a load test scenario: You can add or remove a web performance or unit test from a load test in a scenario. A load test contains one or more scenarios, each of which contains one or more web performance or unit tests.	- Edit the test mix

TASKS	ASSOCIATED TOPICS
<p>Configure the desired network mix for your scenario: Using the network mix, you can simulate network load more realistically in a load test scenario. Load is generated by using a heterogeneous mix of network types instead of one single network type. You create a closer approximation of how end-users interact with your applications. The network mix model should reflect the objectives of that scenario.</p>	<ul style="list-style-type: none"> - Specify virtual network types
<p>Select the appropriate Web browser mix for your scenario: Using the browser mix, you can simulate web load more realistically in a load test scenario. Load is generated by using a heterogeneous mix of browsers instead of one single browser. You create a closer approximation of the browsers that will be used with your applications.</p>	<ul style="list-style-type: none"> - Specify web browsers types
<p>Configure test iteration settings for your scenario: You can edit a load test scenario to configure test iteration settings using the Load Test Editor and the Properties window. By default, a scenario is set up with no maximum test iterations. You can optionally configure the maximum number of iterations in the scenario and how long to pause between them.</p>	<ul style="list-style-type: none"> - Configure test iterations for scenarios
<p>Configure delay settings for your scenario: Using the Load Test Editor and the Properties window, you can specify a delay before starting a scenario in a load test. An example of when you might want to use the Delay Start Time property is if you need one scenario to start producing items that another scenario consumes. You can delay the consuming scenario to enable the producing scenario to populate some data.</p>	<ul style="list-style-type: none"> - Configuring scenario start delays
<p>Specify remote machines to use in a load test scenario: After you create a load test, you can edit the properties of your load test scenario to indicate which test agents you want to include. For more information, see Test controllers and test agents.</p>	<ul style="list-style-type: none"> - How to: Specify test agents to use

See also

- [Edit load tests](#)
- [Load test scenario properties](#)

Load test scenario properties

1/1/2020 • 4 minutes to read • [Edit Online](#)

Change your load test scenario property settings in Visual Studio to meet your load testing requirements. This article lists the various load test scenario properties by category.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

General

PROPERTY	DEFINITION
Name	The name for the scenario.

Mix

PROPERTY	DEFINITION
Browser Mix	<p>Specifies the web browser mix for the load test. You can specify different web browser types and their load distribution.</p> <p>Choose the ellipsis (...) button to open the Edit Browser Mix dialog and use Add and Remove to select the web browser types in the load test.</p> <p>For more information, see Specify web browsers types.</p>
Network Mix	<p>Specifies the network mix for the load test. You can specify which network types to include and their load distribution.</p> <p>Choose the ellipsis (...) button to open the Edit Network Mix dialog box and use Add and Remove to select the network types in the load test.</p> <p>For more information, see Specify virtual network types.</p>
Test Mix	<p>Specifies the web performance and unit test mix for the load test. You can specify which tests to include and their load distribution.</p> <p>Choose the ellipsis (...) button to open the Edit Test Mix dialog box and use Add and Remove to select the tests in the load test.</p> <p>For more information, Edit the test mix for a load test scenario.</p>

PROPERTY	DEFINITION
Test Mix Type	<p>Specifies the test mix model for the load test.</p> <p>Choose the ellipsis (...) button to open the Edit Test Mix dialog box and use the drop-down under Test mix model to select the test mix model to use in the load test.</p> <p>For more information, see Edit text mix models.</p>

Options

PROPERTY	DEFINITION
Agents to Use	<p>Specifies the agents that you want your scenario to use if you are running the load test remotely. For example, you might want to specify a specific set of agents so that you maintain consistency when you analyze performance trends. Also, agents may be geographically distributed so that there is an affinity between which scripts they run and where the agent is located.</p> <p>Agents must be separated by commas, for example "Agent1, Agent2, Agent3". Leaving the property blank specifies that the scenario should use all available agents.</p> <p>For more information, see How to: Specify test agents to use.</p>
Apply Distribution to Pacing Delay	<p>Boolean value that's used to specify if you want to apply typical distribution delays in the user pacing test mix model. This property only applies if the Test Mix Type property is set to Based on user pace.</p> <p>For more information, see How to: Apply distribution to pacing delay</p>
IP Switching	<p>Boolean value that's used to specify if IP switching is used.</p> <p>IP switching enables a test agent to send requests to a server by using a range of different IP addresses. This simulates calls that come from different client computers. IP switching is important when you test against a load balanced web farm. Most load balancers establish affinity between a client and a particular web server by using the IP address of the client. If all requests seem like they are coming from a single client, the load balancer will not balance the load. To obtain good load balance in the web farm, it is important that requests come from a range of IP addresses.</p> <p>IP switching is available only with the test agent.</p>
Maximum Test Iterations	<p>Numeric value that is used to specify the maximum number of tests to run in the scenario. A value of 0 specifies no maximum.</p> <p>For more information, see Configure test iterations for scenarios.</p>

PROPERTY	DEFINITION
Percentage of New Users	<p>Numeric value that specifies the percentage of new users or first time visitors in the scenario.</p> <p>For more information, see How to: Specify the percentage of virtual users that use web cache data.</p>
Think Profile	<p>Specifies if the scenario will use Normal Distribution, or if the think profile is On or Off.</p> <p>For more information, see Edit think times to simulate website human interaction delays.</p>

Timing

PROPERTY	DEFINITION
Delay Start Time	<p>A time value that indicates how many hours, minutes and seconds to delay starting the scenario after the load test starts. If Disable During Warmup property is set to True, the amount of time to wait applies after the completion of the warm-up period.</p> <p>For more information, see Configure scenario start delays.</p>
Disable During Warmup	<p>Boolean value that is used to specify if the scenario should run or not during the Warm Up Duration property time value specified in the load test's run setting.</p> <p>For more information about the load test run setting properties, see Load test run settings properties.</p> <p>For more information, see Configure scenario start delays.</p>
Think Times Between Test Iterations	<p>Numeric value that is used to specify the wait time in seconds between test iterations.</p> <p>For more information, see Edit think times to simulate website human interaction delays.</p>

See also

- [Edit load test scenarios](#)

Edit the test mix to specify which web performance, unit and coded UI tests to include in a load test scenario

1/1/2020 • 4 minutes to read • [Edit Online](#)

The *test mix* of a scenario is a combination of the selection of web performance and unit tests that are contained in the scenario and the distribution of those tests in the scenario. The distribution is a setting that you can specify for the probability that a particular test will be selected by a virtual user during a load test run.

After you add a set of tests to a load test, the *test mix* works like other mix options. A virtual user randomly selects a test, based on the probability that you specified in the mix. For example, if you have two tests, each 50 percent in the mix, a new virtual user chooses to run the first test approximately half the time. In a 50/50 mix, if one test is long and another is short, more load comes from the long test.

After you have added tests to the mix, you can remove them. You can also change the distribution of the test mix by using the mix control. The mix control lets you easily adjust the distribution of tests in a scenario.

NOTE

Distribution is a measure of the probability that a particular test will be selected by a virtual user during a load test run.

Distribution is expressed as a percentage. Therefore, the sum of the distribution numbers for all tests that are contained in a scenario is 100. For example, if a scenario contains only one test, the distribution for that test is 100 percent.

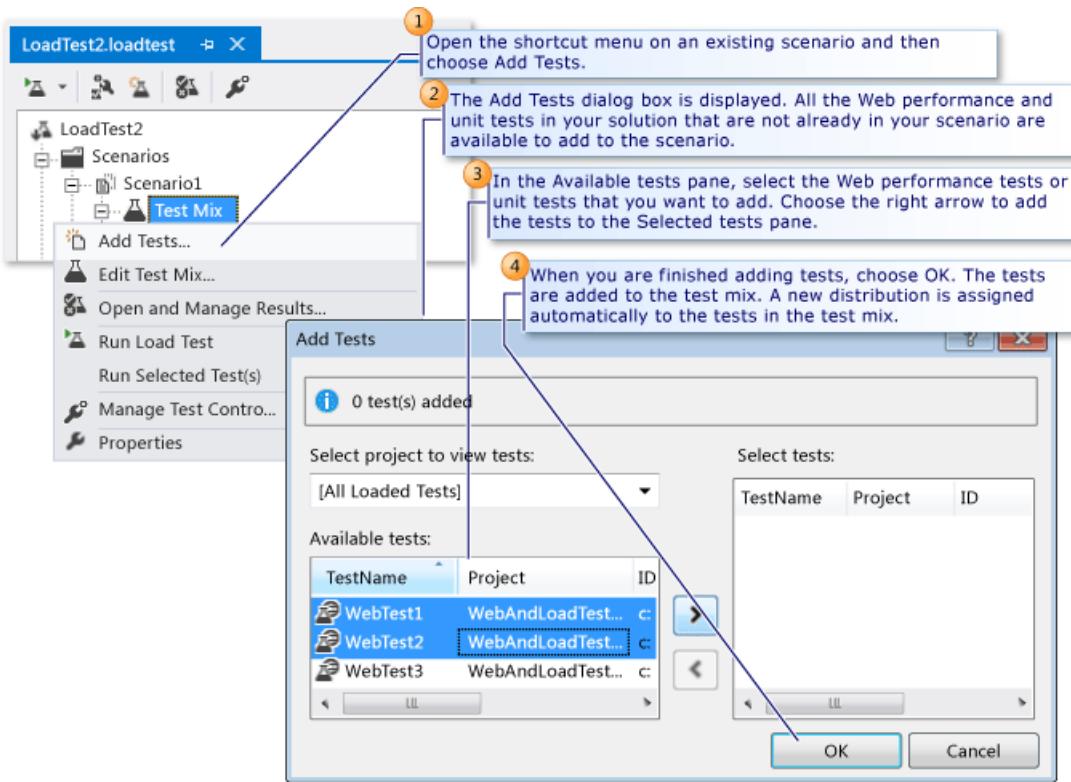
NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Add new tests to a test mix in an existing scenario

When you create a new scenario by using the **New Load Test Wizard**, you can specify the web performance and unit tests to add to the test mix of the new scenario.

You can add more web performance and unit tests to the test mix of the scenario by using the **Load Test Editor**.



To add more tests to an existing scenario

1. Open a load test.
2. In the **Load Test Editor**, right-click an existing scenario and then choose **Add Tests**.

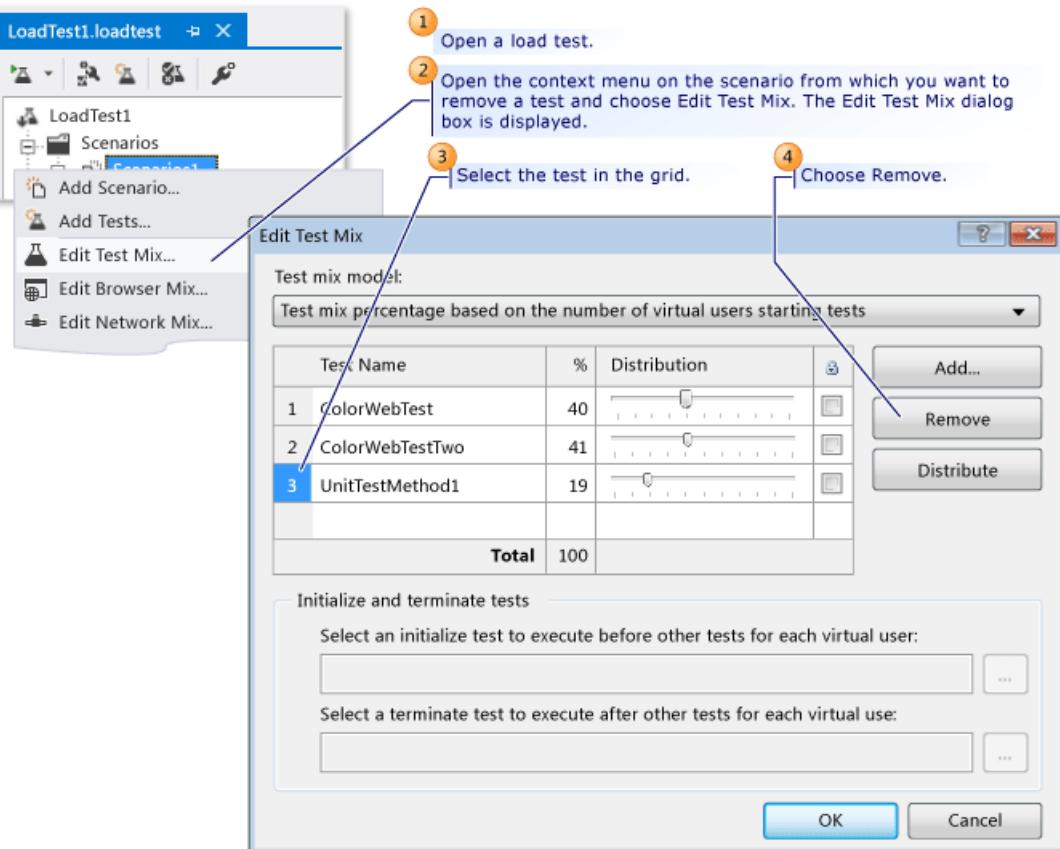
The **Add Tests** dialog box is displayed. All web performance, unit, and coded UI tests in your solution that are not already in your scenario are available to add to the scenario.

3. In the **Available tests** pane, select the web performance, unit, and coded UI tests that you want to add. Choose the right arrow to add the tests to the **Selected tests** pane.
4. When you finish adding tests, choose **OK**.

The tests are added to the test mix. A new distribution is assigned automatically to the tests in the test mix.

5. (Optional) Adjust the mix control to specify the test distribution. For more information, see [About the mix control](#).

Remove tests from a scenario



To remove tests from a scenario

1. Open a load test.
2. In the **Load Test Editor**, in the load test tree, right-click the scenario from which you want to remove a test and select **Edit Test Mix**. The **Edit Test Mix** dialog box is displayed.
3. Select the web performance, unit, or coded UI test in the grid and then choose **Remove**.

NOTE

After you remove the test, adjust the test mix to your preferred distribution.

4. When you finish removing tests, choose **OK**.

About the Mix Control

The mix control allows you to adjust the percentage of load that is distributed among tests, browser types, or network types in a load test scenario. You adjust the percentage values by moving sliders. Adjusting the mix for tests specifies the probability of a virtual user running a specific test in a load test scenario.

When you move a slider, the percentage values of all available items change. If you have more than two items, the amount you add or remove is distributed evenly among the other items. It is possible to override this behavior. If you select the check box in the lock column for a particular item, you lock the specified percentage value for that item. Then, when you move a slider, the amount you add or remove is only applied to any remaining unlocked items.

The **Distribute** button is used to allocate the percentages equally among all items. For example, if you have three items, choosing **Distribute** sets the percentage values to 34, 33, and 33.

WARNING

The **Distribute** button overrides any items that are locked.

It is also possible to type the percentage values directly into the **%** column instead of using the sliders. If you enter a percentage value directly, the other items will not adjust automatically.

NOTE

The sliders are disabled when the total does not add up to 100%, or when the percentage values entered into the **%** column are decimals.

When you enter percentage values manually, you should make sure that the sum of all items is 100%. When you save a mix, if the sum is not 100%, you will be prompted to accept the percentage values as they are, or to go back and adjust them. If you choose to accept them as they are, they will be prorated to 100%. For example, if you have two items and you manually set them to 80% and 40%, the first item will be set to 66.67% (80 divided by 120) and the second item will be set to 33.33% (40 divided by 120).

See also

- [Editing load test scenarios](#)

Test mix models overview

1/1/2020 • 7 minutes to read • [Edit Online](#)

You use load modeling options to more accurately predict the expected real-world usage of a website or application that you are load testing. It's important to do this because a load test that's not based on an accurate load model can generate misleading results.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Test mix model enhancements

Using the Load Test Editor or the test mix model wizard, you can specify the following types of test mix for a load test scenario. For more information, see [Change the test mix model in a scenario](#).

You can specify one of the following test mix model options for your load test scenario:

- **Based on the total number of tests:** Determines which web performance or unit test is run when a virtual user starts a test iteration. At the end of the load test, the number of times that a particular test run matched the assigned test distribution. Use this test mix model when you are basing the test mix on transaction percentages in an IIS log or in production data. For more information, see [Percentage based on tests started](#).
- **Based on the number of virtual users:** Determines the percentage of virtual users who will run a particular web performance or unit test. At any point in the load test, the number of users who are running a particular test matches the assigned distribution. Use this test mix model when you are basing the test mix on the percentage of users who are running a particular test. For more information, see [Percentage based on virtual users](#).
- **Based on user pace:** Over the course of the load test, each web performance test or unit test is run a specified number of times per users, per hour. Use this test mix model when you want virtual users to run test at a certain pace throughout the load test. For more information, see [Pacing test mix](#).

TIP

When do you choose **Percentage test mix** and when do you choose **Percentage based on virtual users**? The difference between these two choices is important when some tests in the test mix have a much longer duration than other tests. In this situation, you should probably choose **Percentage based on virtual users**. This choice helps avoid a test run in which the probability increases that too many users will be running long-duration tests. However, if the tests all have similar durations, you can more safely choose **Percentage test mix**.

- **Based on sequential order:** Each virtual user runs the web performance or unit tests in the order that the tests are defined in the scenario. The virtual user continues cycling through the tests in this order until the load test is complete. For more information, see [Sequential order](#).

Percentage based on tests started

For each test in the mix, you can specify a percentage that determines how frequently the test is selected as the next test to run. For example, you might assign the following percentage values to three tests:

- TestA (50%)
- TestB (35%)
- TestC (15%)

If you use this setting, the next test to start is based on the assigned percentages. You do this without taking into account the number of virtual users who are currently running each test.

Percentage based on virtual users

This model of test mix determines the percentage of virtual users who will run a particular test. If you use this model of test mix, the next test to start is based not only on the assigned percentages but also on the percentage of virtual users who are currently running a particular test. At any point in the load test, the number of users who are running a particular test matches the assigned distribution as closely as possible.

Pacing test mix

If you specify a pacing test mix, you set a rate of test execution for each virtual user for each test in the test mix. For each test, this rate is expressed as tests run per virtual user per hour. For example, you might assign the following pacing test mix to the following tests:

- TestA: 4 tests per user per hour
- TestB: 2 tests per user per hour
- TestC: 0.125 tests per user per hour

If you use the pacing test mix model, the load test runtime engine guarantees that the actual rate at which tests are started is less than or equal to the specified rate. If the tests run too long for the assigned number to be completed, an error is returned.

The **Think Time Between Test Iterations** setting does not apply when you use a pacing test mix.

Apply distribution to pacing delay

The value for the **Apply Distribution to Pacing Delay** property in a load test scenario can be set to either true or false:

- **True:** The scenario will apply typical statistical distribution delays specified by the value in the **Tests Per User Per Hour** column in the **Edit Test Mix** dialog. For more information, see [Edit text mix models to specify the probability of a virtual user running a test](#).

For example, assume that you have **Tests Per User Per Hour** value in the **Edit Test Mix** dialog for the test set to 2 users per hour. If **Apply Distribution to Pacing Delay** property is set to **True**, a typical statistical distribution is applied to the wait time between the tests. The tests will still run 2 tests per hour, but it will not necessarily be 30 minutes between them. The first test could run after 4 minutes and the second test after 45 minutes.

- **False:** The tests will run at the specific pace you specified for the value in the **Tests Per User Per Hour** column in the **Edit Test Mix** dialog. For more information, see [Edit text mix models to specify the probability of a virtual user running a test](#).

For example, assume that you have **Tests Per User Per Hour** value in the **Edit Test Mix** dialog for the test set to 2 users per hour. If the **Apply Distribution to Pacing Delay** property is set to **False**, you are basically giving no leeway when your tests run. The test will run every 30 minutes. This makes sure that you execute 2 tests per hour.

For more information, see [How to: Apply distribution to pacing delay when using a user pace test mix model](#).

Sequential order

Selecting the Based on sequential test order option makes each virtual user run all the tests in the scenario in the order that the tests were defined.

Test iterations property

In the Run Settings properties, you can specify a value for the Test Iterations property. This value is the number of test iterations to run in a load test. After the specified number of test iterations has been started, no additional test iterations will be started despite the settings of any of the load profiles. After the number of test iterations specified has been completed, the load test ends. For more information, see [How to: Specify the number of test iterations in a run setting](#).

Initialize and terminate tests

You can select tests to run at the beginning and end of each virtual user's load testing session. For more information, see [Edit text mix models to specify the probability of a virtual user running a test](#).

- **Initialize test.** This test is run by each virtual user before any of the tests in the test mix are run.
- **Terminate test.** This test is run after all tests for a particular virtual user are run.

Please note the following about the initialize test and the terminate test:

- You can specify the load test duration by time instead of by iteration count. In this case, when the load test run duration has completed, the terminate test will not be run.
- If the initialize test is a unit test or a web performance test, the state of the `TestContext`, or `WebTestContext`, object after the completion of the initialize test is saved. It will then be used as the starting context for iterations of tests in the test mix.
- New Users, as defined in the scenario property Percentage of New Users, always execute the initialize test, one iteration of a test from the test mix, and the terminate test.

See also

- [Edit text mix models to specify the probability of a virtual user running a test](#)
- [Edit load patterns to model virtual user activities](#)
- [Edit the test mix to specify which tests to include in a load test scenario](#)
- [Configure load test run settings](#)
- [Load test scenario properties](#)
- [Change the test mix model in a scenario](#)

Edit test mix models to specify the probability of a virtual user running a test

1/1/2020 • 4 minutes to read • [Edit Online](#)

The *test mix model* specifies the probability of a virtual user running a given test in a load test scenario. This lets you simulate load more realistically. Instead of having just one workflow through your applications, you can have several workflows, which is a closer approximation of how end-users interact with your applications.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Test mix model options

You can specify one of the following test mix model options for your load test scenario:

- **Based on the total number of tests:** Determines which web performance or unit test is run when a virtual user starts a test iteration. At the end of the load test, the number of times that a particular test was run matches the assigned test distribution. Use this test mix model when you are basing the test mix on transaction percentages in an IIS log or in production data.
- **Based on the number of virtual users:** Determines the percentage of virtual users who will run a particular web performance or unit test. At any point in the load test, the number of users who are running a particular test matches the assigned distribution. Use this test mix model when you are basing the test mix on the percentage of users running a particular test.
- **Based on user pace:** Over the course of the load test, each web performance test or unit test is run a specified number of times per users, per hour. Use this test mix model when you want virtual users to run test at a certain pace throughout the load test.
- **Based on sequential order:** Each virtual user runs the web performance or unit tests in the order that the tests are defined in the scenario. The virtual user continues cycling through the tests in this order until the load test is complete.

Tasks

TASKS	ASSOCIATED TOPICS
<p>Specifying the test mix for your load test: When you create a load test, you specify settings for the load test in the New Load Test Wizard. In the New Load Test Wizard, you choose existing web and unit tests to add to the initial scenario. After you have added tests to the scenario, you specify the test mix for the scenario.</p> <p>You use load modeling options to more accurately predict the expected real-world usage of a website or application that you are load-testing. It is important to do this because a load test that is not based on an accurate load model can generate misleading results.</p>	<p>- Emulate expected real-world usage of a website or application</p>

TASKS	ASSOCIATED TOPICS
<p>Edit the test mix model: You can change a load test scenario to use one of the test mix models by using the Load Test Editor.</p>	
<p>Configure pacing delay for a user paced test mix model: If your load test scenario is configured to use the Based on user pace test mix model, you can specify how you want the distribution Pacing Delay configured.</p>	<ul style="list-style-type: none"> - How to: Apply distribution to pacing delay when using a user pace test mix model

Change the test mix model in a scenario

After you create your load test by using the **New Load Test Wizard**, you can use the **Load Test Editor** to change the scenarios properties to meet your testing needs and goals.

NOTE

For a complete list of the load settings properties and their descriptions, see [Load test scenario properties](#).

Using the **Load Test Editor**, you can change the test mix model in a load test scenario by editing the **Test Mix Type** property in the **Properties** window.

To change the test mix model

1. Open a load test.

The **Load Test Editor** appears. The load test tree is displayed.

2. In *Scenarios* folder of the load test tree, choose the scenario node for which you want to specify the maximum number of test iterations.

3. On the **View** menu, select **Properties Window**.

The categories and properties of the scenario are displayed.

4. In the **Test Mix Type** property, choose the ellipsis button (...).

The **Edit Test Mix** dialog box is displayed.

5. Choose the drop-down list under **Test mix model** and select the test mix model that you want to use for the scenario.

6. (Optional) Modify the test mix by using the **Add**, **Remove** and **Distribute** buttons and distribution sliders. For more information, see [Edit the test mix to specify which tests to include in a load test scenario](#).

7. (Optional) Specify a web performance and unit test to initialize or end by using the check boxes and selecting the desired tests. For more information, see [Emulate expected real-world usage of a website or application](#).

8. Choose **OK**.

The **Properties** window displays the new test mix model for the **Test Mix Type** property.

9. After you change the property, choose **Save** on the **File** menu. You can then run your load test by using the new **Test Mix Type** value.

See also

- [Edit load test scenarios](#)
- [Load test scenario properties](#)

Edit the test mix to specify which web browsers types in a load test scenario

1/16/2020 • 3 minutes to read • [Edit Online](#)

The *browser mix* gives you a way to simulate load more realistically in a load test scenario. Load is generated by using a heterogeneous mix of web browsers instead of one single web browser. You create a closer approximation of the web browsers that will be used with your applications.

A browser mix specifies the probability of a virtual user running a particular web browser type in a load test scenario. When you create a load test, you might want to simulate that the load is generated through more than one web browser. When you add a web browser type to the mix from the set of web browsers that are provided, a set of associated headers for the selected web browser is added to each HTTP request that is submitted by a web performance test.

The browser mix works like other mix options. A web browser type is randomly associated with a virtual user, based on the browser mix. The tests of that user are run on a particular web browser, based on the probability that you specified in the mix.

After you have specified a browser mix, you can later add and remove web browser types to the mix. You can also change the distribution of the browser mix by using the mix control. The mix control lets you easily adjust the distribution of browsers in a scenario.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Add new browsers to a scenario

To add new browsers to a scenario

1. While in the process of specifying the browser mix for a scenario choose **Add**.

A new browser entry is added to the grid.

NOTE

To display the **Edit Browser Mix** dialog box, right-click an existing scenario and then choose **Edit Browser Mix**.

2. In the **Browser Type** column, choose the arrow for the new entry and choose the desired browser type.
3. (Optional) Adjust the mix control to specify the test distribution.
4. When you are finished adding browsers, choose **OK**.

Remove browsers from a scenario

To remove browsers from a scenario

1. Open a load test.
2. Right-click the scenario from which you want to remove a browser and then choose **Edit Browser Mix**.

The **Edit Browser Mix** dialog box is displayed.

3. Select the browser in the grid and then choose **Remove**.
4. (Optional) Adjust the mix control to specify the test distribution.
5. When you are finished removing browsers, choose **OK**.

About the mix control

The mix control allows you to adjust the percentage of load that is distributed among tests, browser types, or network types in a load test scenario. You adjust the percentage values by moving sliders. Adjusting the mix for the browser types specifies the probability of a virtual user running a specific browser type in a load test scenario.

When you move a slider, the percentage values of all available items change. If you have more than two items, the amount you add or remove is distributed evenly among the other items. It is possible to override this behavior. If you select the check box in the lock column for a particular item, you lock the specified percentage value for that item. Then, when you move a slider, the amount you add or remove is only applied to any remaining unlocked items.

The **Distribute** button is used to allocate the percentage values equally among all items. For example, if you have three items, choosing **Distribute** sets the percentage values to 34, 33, and 33.

WARNING

The **Distribute** button overrides any items that are locked.

It is also possible to type the percentage values directly into the % column instead of using the sliders. If you enter a percentage value directly, the other items will not adjust automatically.

NOTE

The sliders are disabled when the total does not add up to 100%, or when the percentage values entered into the % column are decimals.

When you enter percentage values manually, you should make sure that the sum of all items is 100%. When you save a mix, if the sum is not 100%, you will be prompted to accept the percentage values as they are, or to go back and adjust them. If you choose to accept them as they are, they will be prorated to 100%. For example, if you have two items and you manually set them to 80% and 40%, the first item will be set to 66.67% (80 divided by 120) and the second item will be set to 33.33% (40 divided by 120).

See also

- [Edit load test scenarios](#)

Specify virtual network types in a load test scenario

1/16/2020 • 4 minutes to read • [Edit Online](#)

The *network mix* gives you a way to simulate load more realistically in a load test scenario. Load is generated by using a heterogeneous mix of network types instead of one single network type. You create a closer approximation of how end-users interact with your applications.

A network mix specifies the probability of a virtual user running a given *network profile*. A network profile is a simulation of network bandwidth at the application layer. It does not simulate latency.

When you create a load test, you might want to simulate that load is being generated through more than one type of network connection. The network mix offers several network types. The different networks are simulated. When you choose an option such as `Cable-DSL 1.5Mbps`, wait times are injected into the test to simulate the selected bandwidth.

The network mix works like other mix options. A network type is selected randomly associated with a virtual user, based on network mix. That user's tests are run using a particular network type, based on the probability you specified in the mix.

After you have specified a network mix, you can add and remove network types. You can also change the distribution of the network mix using the mix control.

The mix control lets you easily adjust the distribution of networks in a scenario.

For more information, see [About the mix control](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

True network emulation

Visual Studio uses software-based true network emulation for all test types including load tests. True network emulation simulates network conditions by direct manipulation of the network packets. The true network emulator can emulate the behavior of both wired and wireless networks by using a reliable physical link, such as an Ethernet. The following network attributes are incorporated into true network emulation:

- Round-trip time over the network (latency)
- The amount of available bandwidth
- Queuing behavior
- Packet loss
- Reordering of packets
- Error propagations.

True network emulation also provides flexibility in filtering network packets based on IP addresses or protocols such as TCP, UDP, and ICMP.

True network emulation can be used by network-based application developers and testers to emulate a desired

test environment, assess performance, predict the impact of change, or make decisions about technology optimization. When compared to hardware test beds, true network emulation is a much cheaper and more flexible solution.

To add new networks to a scenario

1. During the process of specifying the network mix for a scenario, choose **Add**.

A new network entry is added to the grid.

NOTE

To display the **Edit Network Mix** dialog box, right-click an existing scenario and then choose **Edit Network Mix**.

2. In the **Network Type** column, choose the arrow for the new entry. Choose the desired network type.
3. (Optional) Adjust the mix control to specify the test distribution. For more information, see [About the mix control](#).
4. When you are finished adding networks, choose **OK**.

To remove networks from a scenario

1. Open a load test.
2. Right-click the scenario from which you want to remove a network, and choose **Edit Network Mix**. The **Edit Network Mix** dialog box is displayed.
3. Select the network in the grid and then choose **Remove**.
4. (Optional) Adjust the mix control to specify the test distribution. For more information, see [About the mix control](#).
5. When you are finished removing networks, choose **OK**.

About the mix control

The mix control lets you adjust the percentage of load that is distributed among tests, browser types, or network types in a load test scenario. To adjust the percentage values, move the sliders. Adjusting the mix for network types specifies the probability of a virtual user running a specific network profile in a load test scenario.

When you move a slider, the percentage values of all available items change. If you have more than two items, the amount you add or remove is distributed evenly among the other items. It is possible to override this behavior. If you select the check box in the lock column for a particular item, you lock the specified percentage value for that item. Then, when you move a slider, the amount you add or remove is only applied to any remaining unlocked items.

The **Distribute** button is used to allocate the percentage values equally among all items. For example, if you have three items, choosing **Distribute** sets the percentage values to 34, 33, and 33.

WARNING

The **Distribute** button overrides any items that are locked.

It is also possible to type the percentage values directly into the % column instead of using the sliders. If you enter a percentage value directly, the other items will not adjust automatically.

NOTE

The sliders are disabled when the total does not add up to 100%, or when the percentage values entered into the % column are decimals.

When you enter percentage values manually, you should make sure that the sum of all items is 100%. When you save a mix, if the sum is not 100%, you will be prompted to accept the percent values as they are, or to go back and adjust them. If you choose to accept them as they are, they will be prorated to 100%. For example, if you have two items and you manually set them to 80% and 40%, the first item will be set to 66.67% (80 divided by 120) and the second item will be set to 33.33% (40 divided by 120).

Edit think times to simulate website human interaction delays in load tests scenarios

1/1/2020 • 2 minutes to read • [Edit Online](#)

Think times are used to simulate human behavior that causes people to wait between interactions with a website. Think times occur between requests in a web performance test and between test iterations in a load test scenario. Using think times in a load test can be useful in creating more accurate load simulations. You can change whether think times are used or ignored in load tests. You change whether think times are used in your load tests in the **Load Test Editor**.

The *think profile* is a setting that applies to a scenario in a load test. The setting determines whether the think times that are saved in the individual web performance tests are used during the load test. If you want to use think times in some web performance tests but not in others, you must place them in different scenarios. For more information about scenarios, see [Edit load test scenarios](#).

Initially, you set whether you use think times in your load tests when you create the load test using the **New Load Test Wizard**. For more information, see [Edit load test scenarios](#).

The **Think Profile** options are described in the following list:

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Off

Think times are ignored. Use this setting when you want to generate maximum load to heavily stress your web server. Do not use it when you are trying to create more realistic user interactions with a web server.

On

Think times are used exactly as they were recorded in the web performance test. Simulates multiple users running the web performance tests exactly as recorded. Because a load test simulates multiple users, using the same think time could create an unnatural load pattern of synchronized virtual users.

Normal Distribution

Think times are used, but varied on a normal curve. Provides a more realistic simulation of virtual users by slightly varying the think time between requests.

NOTE

For a complete list of the load test scenario properties and their descriptions, see [Load test scenario properties](#).

Change the think profile

To change a think profile in a load test scenario

1. From the web performance and load test project, open a load test.
2. In the **Load Test Editor**, choose the scenario node where you want to change the **Think Profile**. The

Think Profile is displayed in the **Properties** window. Press **F4** to display the **Properties** window.

3. Change the **Think Profile** property in the **Properties** window.
4. After you have finished changing the properties, choose **Save** on the **File** menu. You can then run your load test with the new think profile.

See also

- [Edit load test scenarios](#)

Edit load patterns to model virtual user activities

1/1/2020 • 9 minutes to read • [Edit Online](#)

The load pattern properties specify how the simulated user load is adjusted during a load test. Visual Studio provides three built-in load patterns: constant, step, and goal-based. You choose the load pattern and adjust the properties to appropriate levels for your load test goals.

The load pattern is a component of a scenario. The scenarios, together with their defined load patterns, comprise a load test.

NOTE

In all Load Patterns, the load that Visual Studio generates is a simulated load of virtual users.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Load patterns

Constant

The constant load pattern is used to specify a user load that does not change during the load test. For example, when you run a smoke test on a web application, you might want to set a light, constant load of 10 users.

Constant load pattern considerations

A constant load pattern is used to run the same user load during the run of a load test. Be careful about using a constant load pattern that has a high user count; doing so may place an unreasonable and unrealistic demand on your server or servers at the beginning of the load test. For example, if your load test contains a web test that starts with a request to a home page, and you set up the load test with a constant load of 1,000 users, the load test will submit the first 1,000 requests to the home page as fast as possible. This may not be a realistic simulation of real-world access to your website. To mitigate this, consider using a step load pattern that increases gradually to 1,000 users, or specify a warm-up period in the Load Test Run Settings. If a warm-up period is specified, the load test will automatically increase the load gradually during the warm-up period. For more information, see [Configure scenario start delays](#).

Step

The step load pattern is used to specify a user load that increases with time up to a defined maximum user load. For stepping loads, you specify the **Initial User Count**, **Maximum User Count**, **Step Duration (seconds)**, and **Step User Count**.

For example a Step load with an **Initial User Count** of one, **Maximum User Count** of 100, **Step Duration (seconds)** of 10, and a **Step User Count** of 1 creates a user load pattern that starts at 1, increases by 1 every 10 seconds until it reaches 100 Users.

NOTE

If the total test duration is shorter than the time that is required to step up to the maximum user load, then the test stops after the elapsed duration and does not reach the **Maximum User Count** target.

You can use the Step goal to increase the load until the server reaches a point where performance diminishes significantly. As load increases, the server will eventually run out of resources. The step load is a good way to determine the number of users at which this occurs. With the stepping load, you also have to monitor agent resources closely to make sure that the agents can generate the desired load.

Ordinarily, you should conduct several runs that have different step durations and step user counts so that you can obtain good measurements for a given load. Frequently, loads show an initial spike for each step as users are added. Holding the load at that rate allows you to measure system performance after the system recovers from the initial spike.

Step load pattern considerations

A step load pattern can be used to increase the load on the server or servers as the load test runs so that you can see how performance varies as the user load increases. For example, to see how your server or servers perform as the user load increases to 2,000 users, you might run a 10-hour load test by using a step load pattern that has the following properties:

- **Initial User Count:** 100
- **Maximum User Count:** 2,000
- **Step Duration (seconds):** 1,800
- **Step Ramp Time (seconds):** 20
- **Step User Count:** 100

These settings run the load test for 30 minutes (1,800 seconds) at user loads of 100, 200, 300, and up to 2,000 users. The **Step Ramp Time** property is worth special mention, because it is the only one of these properties that is not available for selection in the **New Load Test Wizard**. This property allows the increase from one step to the next (for example from 100 to 200 users) to occur gradually rather than immediately. In the example, the user load would be increased from 100 to 200 users over a 20 second period (an increase of five users every second). For more information, see [How to: Specify the step ramp time property for a step load pattern](#).

Goal-based

A goal-based load pattern resembles the step pattern but adjusts the user load based on performance counter thresholds versus periodic user load adjustments. Goal based loads are useful for a variety of different purposes:

- Maximizing output from the agents: measure the key limiting metric on the agent to maximize the output of the agents. Typically, it is CPU; However, it could also be memory.
- Reaching some target resource level, typically CPU, on the target server, then measuring throughput at that level. This enables you to do run-to-run comparisons of throughput given a consistent level of resource usage on the server.
- Reaching a target throughput level on the server.

In the following table, an example shows a goal-based pattern with the following property settings:

PROPERTY GROUP	PROPERTY	VALUE
Performance Counter	Category	Processor
Performance Counter	Computer	ContosoServer1
Performance Counter	Counter	% Processor Time

PROPERTY GROUP	PROPERTY	VALUE
Performance Counter	Instance	_Total
Target Range for Performance Counter	High End	90
Target Range for Performance Counter	Low End	70
User Count Limits	Initial User Count	1
User Count Limits	Maximum User Count	100
User Count Limits	Maximum User Count Decrement	5
User Count Limits	Maximum User Count Increment	5
User Count Limits	Minimum User Count	1

Those settings cause the **Load Test Analyzer** to adjust the user load between 1 and 100 during a test run in such a way that the **Counter** for `% Processor Time` of the WebServer01 hovers between 70% and 90%.

The size of the each user load adjustment is determined by **Maximum User Count Increment** and **Maximum User Count Decrement** settings. The user count limits are set by the **Maximum User Count** and **Minimum User Count** properties.

Goal-based load pattern considerations

A goal-based load pattern is useful when you want to determine the number of users that your system can support before it reaches some level of resource utilization. This option works best when you have already identified the limiting resource (that is, the bottleneck) in your system.

For example, suppose you know that the limiting resource in your system is the CPU on your database server, and you want to see how many users can be supported when the CPU on the database server is approximately 75 percent busy. You could use a goal-based load pattern that has the goal of keeping the value of the performance counter "%Processor Time" between 70 percent and 80 percent.

One thing to watch out for is if some other resource is limiting the throughput of the system. Such resources can cause the goal that is specified by the goal-based load pattern to never be reached. Also, the user load will continue to rise until the value that is specified for the **Maximum User Count** is reached. This is usually not the desired load, so be careful about the choice of the performance counter in the goal-based load pattern.

Tasks

TASKS	ASSOCIATED TOPICS
Specifying the initial load pattern for your load test: When you create a load test by using the New Load Test Wizard , you select a load pattern.	<ul style="list-style-type: none"> - Change the load pattern
Editing the load pattern for your load test: After you create your load test, you can edit the load pattern in the Load Test Editor .	<ul style="list-style-type: none"> - How to: Specify the step ramp time property for a step load pattern

TASKS	ASSOCIATED TOPICS
<p>Specifying whether the virtual users in your load test scenario should include web cache data: You can change the Percentage of new Users property to affect the way in which the load test simulates the web caching that would be performed by a web browser for the virtual users.</p>	<ul style="list-style-type: none"> - How to: Specify the percentage of virtual users that use web cache data
<p>Specifying the step ramp time for a step load pattern: The Step Ramp Time property allows the increase from one step to the next (for example from 100 to 200 users) to occur gradually rather than immediately.</p>	<ul style="list-style-type: none"> - How to: Specify the step ramp time property for a step load pattern

Change the load pattern

After you create your load test with the **New Load Test Wizard**, you can use the **Load Test Editor** to change the load pattern properties associated with a scenario to levels that meet your test goals.

NOTE

For a full list of the load test scenario properties and their descriptions, see [Load test scenario properties](#).

A load pattern specifies the number of virtual users active during a load test, and the rate at which new users are added. You can choose from the three available patterns: step pattern, constant and goal based. For more information, see [Specify the number of virtual users with load patterns in a load test scenario](#).

NOTE

You can also change your load properties programmatically by using a load test plug-in. For more information, see [How to: Create a load test plug-in](#).

To change the load pattern

1. Open a load test.
2. In the **Load Test Editor**, in the *Scenarios* folder, expand the scenario you want to edit the load pattern for and choose the load pattern for the scenario.

NOTE

The wording of the load pattern node, as it is displayed in the scenario tree of your load test, reflects the load profile you chose when you created the load test. It can be either **Constant Load Profile** or **Step Load Profile**.

3. Press **F4** to display the **Properties** window.

The **Load Pattern** and the **Parameters** categories are displayed in the **Properties** window.

4. (Optional) Change the **Pattern** property in the **Load Pattern** category.

Your choices for the **Pattern** property are **Step**, **Constant**, and **Goal Based**. For more information about the load pattern types, see [Specify the number of virtual users with load patterns in a load test scenario](#).

5. (Optional) In the **Parameters** category, change the values.

NOTE

The values you can set for **Parameters** differ according to the value that was selected for **Pattern** property.

6. After you have finished changing the properties, choose **Save** on the **File** menu. You can then run your load test with the new load pattern.

See also

- [Edit load test scenarios](#)
- [How to: Specify the percentage of virtual users that use web cache data](#)
- [How to: Specify the step ramp time property for a step load pattern](#)

How to: Specify the step ramp time property for a step load pattern

1/1/2020 • 2 minutes to read • [Edit Online](#)

After you create your load test with the **New Load Test Wizard**, you can use the **Load Test Editor** to change the scenarios properties to meet your testing needs and goals. For more information, see [Walkthrough: Create and run a load test](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

NOTE

For a full list of the load test scenario properties and their descriptions, see [Load test scenario properties](#).

The **Step Ramp Time** property is set in the **Properties** window. You edit load test scenario properties in the **Load Test Editor**.

The **Step Ramp Time** property is only used with a step load pattern. For more information, see [Edit load patterns to model virtual user activities](#).

A step load pattern is used to increase the load on the server or servers as the load test runs so that you can see how performance varies as the user load increases. For example, to see how your server or servers perform as the user load increasing to 2,000 users, you might run a 10-hour load test using a step load pattern with the following properties:

- Initial User Count: 100
- Maximum User Count: 2000
- Step Duration (seconds): 1800
- Step Ramp Time (seconds): 20
- Step User Count: 100

These settings have the load test running for 30 minutes (1800 seconds) at user loads of 100, 200, 300, up to 2,000 users.

NOTE

The **Step Ramp Time** property is the only one of these properties that is not available to choose in the **New Load Test Wizard**.

The **Step Ramp Time** property allows the increase from one step to the next (for example from 100 to 200 users) to be gradual rather than immediate. In the example, the user load would be increased from 100 to 200 users over a 20 second period (an increase of 5 users every second).

To edit the step ramp time property for a step load pattern

1. Open a load test.

The **Load Test Editor** appears. The load test tree is displayed.

2. In the load test trees **Scenarios** folder, open the scenario node you want to specify the step ramp time for.
3. Select the **Step Load Pattern** node.

NOTE

The load pattern for the scenario must be a step load pattern. If it is not, the load pattern will display the load pattern type that is currently associated with the scenario. For more information, see [Edit load patterns to model virtual user activities](#).

4. On the **View** menu, select **Properties Window**.

The scenario's categories and properties are displayed in the **Properties** window.

5. Set the value for the **Step Ramp Time** property by entering a number for the seconds taken in each step to gradually add the users specified by the **Step User Count** property.
6. After you have finished changing the property, choose **Save** on the **File** menu. You can then run your load test using the new **Step Ramp Time** value.

See also

- [Edit load test scenarios](#)
- [Test controllers and test agents](#)
- [Load test scenario properties](#)
- [Edit load patterns to model virtual user activities](#)

Configure scenario start delays in load tests

1/16/2020 • 3 minutes to read • [Edit Online](#)

Specify a delay before a scenario starts in a load test by using the Load Test Editor and the **Properties** window.

For example, you might want to use the **Delay Start Time** property if you need one scenario to start producing items that another scenario consumes. You can delay the consuming scenario to enable the producing scenario to populate some data.

Another example is that you might have one scenario that is only run at a certain time of the day. So, you want to delay the start of the scenario to simulate this.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Specify the delay start time of a scenario

You can specify a delay before the start of a scenario in a load test by using the Load Test Editor to change the **Delay Start Time** property in the **Properties** window.

NOTE

For a full list of the load test scenario properties and their descriptions, see [Load test scenario properties](#).

An example of an instance when you might want to use the **Delay Start Time** property is when you need one scenario to start producing items that another scenario consumes. You can delay the consuming scenario to enable the producing scenario to populate some data.

Another example is that you might have one scenario that is run only at a certain time of day. Therefore, you want to delay the start of the scenario to simulate this.

NOTE

For a full list of the run settings properties and their descriptions, see [Load test scenario properties](#).

To specify the delay start time for a scenario

1. Open a load test.

The Load Test Editor appears. The load test tree is displayed.

2. In the load test trees **Scenarios** folder, choose the scenario node for which you want to specify the delay start time.
3. On the **View** menu, select **Properties Window**.

The categories and properties of the scenario are displayed in the **Properties** window.

4. In the text box for the **Delay Start Time** property, type a time value that indicates the time to wait after the load test starts before starting the scenario when the load test is run.

NOTE

If the value for the **Disable During Warmup** property for the scenario is set to **True**, then the **Delay Start Time** properties time value will be applied after the warm-up period. You can control which scenarios are included in warm-up by using the **Disable During Warmup** scenario property.

5. After you change the property, choose **Save** on the **File** menu. You can then run your load test by using the new **Delay Start Time** value.

Enable and disable whether a scenario runs during the warm-up period

The **Disable During Warmup** property is set by using the **Properties** window. Editing load test scenario properties is set by the Load Test Editor.

The **Disable During Warmup** property is used to indicate whether the scenario should run or not run during the warm-up period that is specified in the **Delay Start Time** property. For more information, review the previous procedure [Specify the delay start time of a scenario](#).

NOTE

For a complete list of the run settings properties and their descriptions, see [Load test scenario properties](#).

To enable or disable the warm-up period for a scenario

1. Open a load test.

The **Load Test Editor** appears. The load test tree is displayed.

2. In the load test trees **Scenarios** folder, choose the scenario node that you want to change the warmup behavior for.

3. On the **View** menu, select **Properties Window**.

The scenario's categories and properties are displayed in the **Properties** window.

In the **Disable During Warmup** property, select either **True** or **False**.

4. After you have finished changing the property, choose **Save** on the **File** menu. You can then run your load test using the new **Disable During Warmup** value.

See also

- [Editing load test scenarios](#)
- [Configure test agents and test controllers for load tests](#)
- [Load test scenario properties](#)

Configure test iterations in a load test scenario

1/1/2020 • 3 minutes to read • [Edit Online](#)

To configure test iteration settings, edit a load test scenario using the Load Test Editor and the **Properties** window. By default, a load test scenario is set up without specifying maximum test iterations. You have the option to configure the maximum number of iterations in the scenario and how long to pause between them.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Specify the maximum test iterations for a scenario

You can specify the maximum number of times that you want your tests to run for a scenario by using the Load Test Editor to change the **Maximum Test Iterations** property in the **Properties** window.

The **Maximum Test Iterations** property controls the maximum number of test iterations to run for the scenario. Just as for the **Test Iterations** property in the load test run settings, this is the maximum across all users on all agents, not a per user setting.

NOTE

For a full list of the load test scenario properties and their descriptions, see [Load test scenario properties](#).

For sequential test mix, one iteration is one pass through all the tests in the mix. For all other test mixes, each test execution counts as an iteration. For more information, see [About the mix control](#).

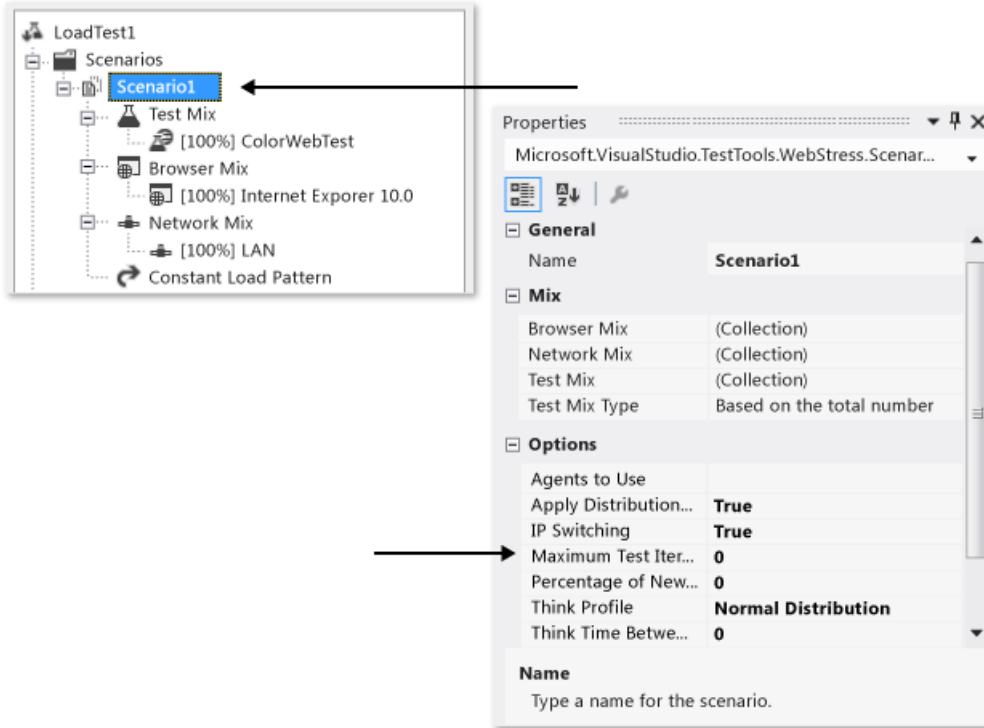
If the load test is a duration-based load test, and the duration expires before the iteration count is complete, the test will still stop. If the test is iteration-based, and the test iterations are met before scenario iterations, the test will stop. The duration is configured by using the **Run Duration** property in the **Properties** window associated with a run setting in a load test.

When the scenario iteration count is met, the scenario will stop running, but any other active scenarios will continue to run.

NOTE

A related property is the **Unique** property on a web test data source, which moves sequentially through the data, row-by-row, but only one time for each record. For more information, see [Add a data source to a web performance test](#).

The **Maximum Test Iterations** property is useful for a variety of situations. Some load testers prefer to conduct iteration-based testing, whereas other load testers prefer to conduct duration-based testing.



To specify the maximum test iterations

1. Open a load test.
2. The Load Test Editor appears. The load test tree is displayed.
3. In the load test trees **Scenarios** folder, choose the scenario node for which you want to specify the maximum number of test iterations.
4. On the **View** menu, select **Properties Window**.

The categories and properties of the scenario are displayed in the **Properties** window.

5. In the text box for the **Maximum Test Iterations** property, type a value that indicates the maximum number of tests to run for the scenario when the load test is run.

NOTE

Using a value of 0 for the **Maximum Test Iterations** property specifies no maximum iterations.

6. After you have finished changing the property, choose **Save** on the **File** menu. You can then run your load test by using the new **Maximum Test Iterations** value.

Specify think times between test iterations for a scenario

The **Think Time Between Test Iterations** property is set using the **Properties** window while editing load test scenario properties in the Load Test Editor.

The **Think Time Between Test Iterations** property is used to specify the amount of seconds to wait before starting a test iteration.

NOTE

For a complete list of the load test scenario properties and their descriptions, see [Load test scenario properties](#).

To specify the think time between test iterations

1. Open a load test.

The **Load Test Editor** appears. The load test tree is displayed.

2. In the load test trees **Scenarios** folder, choose the scenario node you want to specify think time for.

3. On the **View** menu, select **Properties Window**.

The scenario's categories and properties are displayed in the **Properties** window.

4. In the value for **Think Time Between Test Iterations** property, enter a number representing the number of seconds to wait before starting the next test iteration.
5. After you have finished changing the property, choose **Save** on the **File** menu. You can then run your load test using the new **Think Time Between Test Iterations** value.

See also

- [Editing load test scenarios](#)
- [Configure test agents and test controllers for load tests](#)
- [Load test scenario properties](#)
- [Edit think times to simulate website human interaction delays](#)

How to: Specify the percentage of virtual users that use web cache data

1/16/2020 • 2 minutes to read • [Edit Online](#)

After you create your load test with the **New Load Test Wizard**, you can change the scenarios properties to meet your testing needs and goals by using the **Load Test Editor**. For a full list of the load test scenario properties and their descriptions, see [Load test scenario properties](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

The **Percentage of new Users** property is set in the **Properties** window. You edit load test scenario properties in the **Load Test Editor**.

The **Percentage of new Users** property affects the way in which the load test simulates the caching that would be performed by a web browser. By default, the **Percentage of new Users** property is set to 0%. If the value for the **Percentage of new Users** property is set to 100%, each web performance test run in a load test is treated like a first time user to the website who does not have any content from the website in their browser cache from previous visits. Thus, all requests in the web test, including all dependent requests such as images, are downloaded.

NOTE

When the same cacheable resource is requested more than once in a web test, the requests are not downloaded.

If you are load testing a website that has a significant number of return users who are likely to have images and other cacheable content cached locally, then a setting of 100% for **Percentage of new Users** property will generate more download requests than would occur in real-world usage. In this case, you should estimate the percentage of visits to your website that are from first time users of the website, and set **Percentage of new Users** property accordingly.

To specify the percentage of new users for a scenario

1. Open a load test.

The **Load Test Editor** appears. The load test tree is displayed.

2. In the load test trees **Scenarios** folder, choose the scenario node you want to change the new user percentage value for.

3. On the **View** menu, select **Properties Window**.

The scenario's categories and properties are displayed in the **Properties** window.

4. Set the value for the **Percentage of New Users** property by entering a number for the percentage of new users.
5. After you have finished changing the property, choose **Save** on the **File** menu. You can then run your load test using the new **Percentage of New Users** value.

See also

- [Edit load test scenarios](#)
- [Walkthrough: Create and run a load test](#)
- [Test controllers and test agents](#)
- [Load test scenario properties](#)
- [Edit load patterns to model virtual user activities](#)

How to: Apply distribution to pacing delay for a user pace test mix model

1/16/2020 • 2 minutes to read • [Edit Online](#)

After you create your load test by using the **New Load Test Wizard**, you can use the Load Test Editor to change the scenario's properties to meet your testing needs and goals.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

The **Apply Distribution to Pacing Delay** property is set by using the **Properties** window. Load test scenario properties are modified by using the Load Test Editor.

NOTE

The **Apply Distribution to Pacing Delay** property applies only if the *load test mix* is configured based on the user pace. For more information, see [Edit text mix models to specify the probability of a virtual user running a test](#).

The value for the **Apply Distribution to Pacing Delay** can be set to either true or false:

- **True:** The scenario applies normal statistical distribution delays that are specified by the value in the **Tests Per User Per Hour** column in the **Edit Test Mix** dialog box. For more information, see [Edit text mix models to specify the probability of a virtual user running a test](#).

For example, assume that you have **Tests Per User Per Hour** value in the **Edit Test Mix** dialog box for the test set to two users per hour. If **Apply Distribution to Pacing Delay** property is set to **True**, a normal statistical distribution is applied to the wait time between the tests. The tests will still run two tests per hour, but it will not necessarily be 30 minute delay between them. The first test could run after four minutes and the second test after 45 minutes.

- **False:** The tests run at the pace that you specified for the value in the **Tests Per User Per Hour** column in the **Edit Test Mix** dialog box. For more information, see [Edit text mix models to specify the probability of a virtual user running a test](#).

For example, assume that you have **Tests Per User Per Hour** value in the **Edit Test Mix** dialog box for the test set to two users per hour. If the **Apply Distribution to Pacing Delay** property is set to **False**, you are giving no leeway when your tests run. The test will run every 30 minutes. This makes sure that you execute two tests per hour.

To specify the Apply Distribution to Pacing Delay property setting for a scenario

1. Open a load test.

The **Load Test Editor** appears. The load test tree is displayed.

2. In the **Scenarios** folder of the load test tree, select the scenario node you want to apply pacing distribution to.

3. On the **View** menu, select **Properties Window**.

The categories and properties of the scenario are displayed in the **Properties** window.

4. In the property value for the **Apply Distribution to Pacing Delay**, select either **True** or **False**.

5. Select **File > Save**. You can now run your load test with the new **Apply Distribution to Pacing Delay** value.

See also

- [Edit load test scenarios](#)
- [Walkthrough: Create and run a load test](#)
- [Test controllers and test agents](#)
- [Load test scenario properties](#)

Specify counter sets and threshold rules for computers in a load test

1/1/2020 • 6 minutes to read • [Edit Online](#)

Load tests provide named counter sets that are useful when you analyze performance counter data. The counter sets are organized by technology and include Application, ASP.NET, .NET Application, IIS, and SQL. When you create a load test by using the **New Load Test Wizard**, you add an initial set of counters. These offer you a set of predefined and important counter sets for your load test. You manage your counters in the **Load Test Editor**.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

NOTE

If your load tests are distributed across remote machines, controller and agent counters are mapped to the controller and agent counter sets. For more information about how to use remote machines in your load test, see [Test controllers and test agents](#).

Counter sets are gathered on computers that you specify. The association between a counter set and a computer that is used during a load test is a *counter set map*. For example, the web server that you are testing might have ASP.NET, IIS, and .NET application counter set mappings.

By default, performance counters are collected on the controller and agents. For more information, see [Test controllers and test agents](#).

It is important that you add the servers under test to the list of computers on which to collect counters. Then, any important system data is collected and monitored during the load test.

Tasks

TASKS	ASSOCIATED TOPICS
Manage counter sets for your load test: After you create your load test, you can edit the Counter Set in the Load Test Editor. Managing counter sets involves choosing the set of computers from which you want to collect performance data and assigning a set of counter sets to collect from each individual computer. You manage your counters in the Load Test Editor.	- How to: Manage counter sets
Add counter sets to your load test: When you create a load test with the New Load Test Wizard , you add an initial set of counters. These offer you a set of predefined counter sets for your load test. After you create a load test, you can add new counters to existing counter sets using the Load Test Editor.	- How to: Add counters to counter sets - How to: Add custom counter sets

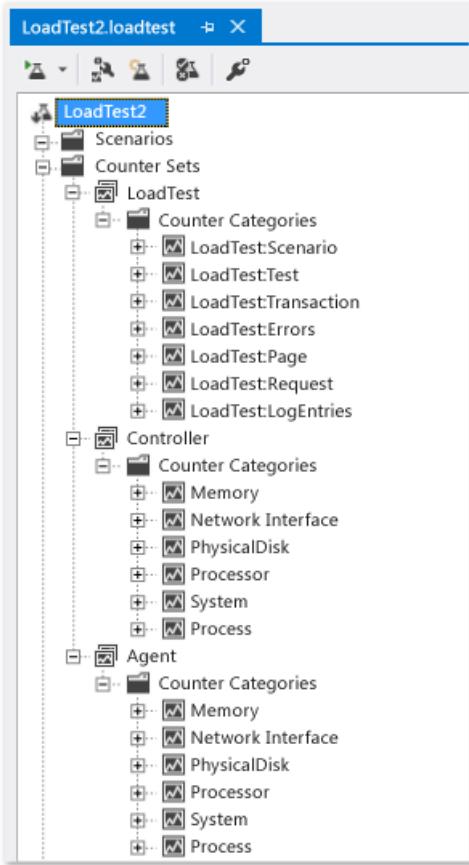
TASKS	ASSOCIATED TOPICS
<p>Specify a threshold rule using counters for your load test: A threshold rule is a rule that is set on an individual performance counter to monitor system resource usage during a load test. Counter set definitions contain predefined threshold rules for many key performance counters. Threshold rules in load tests compare a performance counter value with either a constant value or another performance counter value.</p>	<ul style="list-style-type: none"> - How to: Add a threshold rule
<p>Assign friendly names to the computers to which counter sets are mapped: You can add computer tags that enable you to apply an easily recognized name to a computer. The tags are displayed in the Counter Set Mappings node for the tree in the Load Test Editor. More important, the tags are displayed in Excel reports that help stakeholders identify what role the computer has in the load test, for example, "Web Server1 in lab2" or "SQL Server2 in Phoenix office".</p> <p>For more information, see Report load tests results for test comparisons or trend analysis.</p>	

Use counter sets

The load test tools collect and graph performance data by using counters over time. Counter data is collected at user-specified intervals during a load test run. For more information, see [How to: Specify the sample rate](#). You can view the counters at run time or you can view them after a load test run by using the *Load Test Analyzer*.

Counter data is gathered on the server and on any computer where a test is run. If you have set up a set of agent computers on which to run your tests, counters are gathered on those computers also.

There are three counter categories: percentages, counts, and averages. Some examples are % CPU usage, SQL Server lock counts, and IIS requests per second.



Performance data for individual HTTP requests is reported by the computer that runs a test, such as an agent computer. For requests, you might monitor data such as Average Time to First Byte, Response Time, and Requests per Second.

To ease collection of performance data on a web server, Visual Studio Enterprise also provides predefined, named counter sets, based on technology for use in load tests. These sets are useful when you are analyzing a server that is running IIS, ASP.NET, or SQL Server. Counters not provided in the default set of counter can be added by using the Load Test Editor. It is important that you add the computers or servers under test to your load test to make sure that you can monitor resource use on these computers. For more information, see [How to: Manage counter sets](#).

Results analysis of load runs frequently requires domain-specific knowledge of a particular area in order to know what data to gather, where to set threshold rules, and how to tell when a measurement reflects a specific problem in the application. For more information, see [About threshold rules](#).

Performance counter sampling interval considerations

Select an appropriate value for the **Sample Rate** property in the load test run settings based on the length of your load test. A smaller sample rate, such as the default value of five seconds, requires more space in the load test results database. For longer load tests, increasing the sample rate reduces the amount of data collected. For more information, see [How to: Specify the sample rate](#).

The following are some guidelines for sample rates.

LOAD TEST DURATION	RECOMMENDED SAMPLE RATE
< 1 Hour	5 seconds
1–8 Hours	15 seconds
8–24 Hours	30 seconds

LOAD TEST DURATION	RECOMMENDED SAMPLE RATE
> 24 Hours	60 seconds

Store performance data

During a load test run, the performance counter data is collected and stored in the *Load Test Results Repository*. For more information, see [Manage load test results in the load test results repository](#).

About threshold rules

A *threshold rule* is a rule that is set on an individual performance counter to monitor system resource usage during a load test. Counter set definitions contain predefined threshold rules for many key performance counters. For more information, see [Use counter sets to help analyze performance counter data in load tests](#).

Threshold rules and levels

When you create threshold rules in your load tests, you choose between two types of rules:

Compare Constant—Compare a performance counter value with a constant value.

Compare Counters—Compare a performance counter value with another performance counter value.

When you create threshold rules, you also set the levels for the rule. The levels are the warning threshold and the critical threshold. When you view a load test run, warning level threshold violations are indicated by a yellow symbol, and critical level threshold violations are indicated by a red symbol.

The Alert If Over property

Set the **Alert If Over** property to **True** to indicate that exceeding a threshold is a problem. For example, if the threshold rule is set on **% Processor Time**, and you want to be alerted if the value is greater than 90, use the **Compare Constant** rule type, set the **Critical Threshold Value** to 90, and set **Alert If Over** to **True**.

Set the **Alert If Over** property to **False** to indicate that falling below a threshold is a problem. For example, if the threshold rule is set on **Requests/Sec**, and you want to be alerted if the value is below 50, use the **Compare Constant** rule type, set the **Critical Threshold Value** to 50, and set **Alert If Over** to **False**.

See also

- [How to: Add a threshold rule](#)
- [Analyze threshold rule violations](#)

How to: Add counters to counter sets using the Load Test Editor

1/1/2020 • 2 minutes to read • [Edit Online](#)

When you create a load test with the **Load Test Wizard**, you add an initial set of counters. These offer you a set of predefined counter sets for your load test. For more information, see [Specify the counter sets and threshold rules for computers in a load test](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

NOTE

If your load tests are distributed across remote machines, controller and agent counters are mapped to the controller and agent counter sets. For more information about how to use remote machines in your load test, see [Test controllers and test agents](#).

You manage your counters in the **Load Test Editor**. The counter sets that are already added to the test are visible in the **Counter Sets** node of the load test. After you create a load test, you can add new counters to existing counter sets.

To add counters to a counter set

1. Open a load test.
2. Expand the **Counter Sets** node. All the counter sets that have been added to the load test are visible.

NOTE

The load test hierarchy tree also contains the **Run Settings** node. This node contains the **Counter Set Mappings** node, which shows all the computers and the counter sets that are mapped to those computers.

3. Right-click an existing counter set and then choose **Add Counters**.

The **Pick Performance Counters** dialog box is displayed.

4. In the **Computer** drop-down combo box, type the name of the computer you want to map to. Alternatively, select one of the computers in the drop-down list.

NOTE

Because counter sets must be mapped to a computer before performance data is collected, you must specify a computer on which to collect performance data.

5. Select a **Performance category** to filter the categories of performance data counters. You will see two columns of data from which to select performance counters.

NOTE

Some counter categories will require that you select an instance also. For example, if you select a SQL counter, you must select a SQL instance because there may be more than one instance of SQL installed on the target computer.

6. Select a counter and an instance to add to your custom counter set.

- or -

Select the **All counters** radio button to select all available counters.

7. Choose **OK**.

NOTE

It is also possible to add counters to a counter set by choosing an existing counter or counter category, choosing copy, and then pasting it to a different counter set node. Extra counters that are copied, but not needed, can be deleted.

See also

- [Specify the counter sets and threshold rules for computers in a load test](#)
- [Configure load test run settings](#)

How to: Add custom counter sets using the Load Test Editor

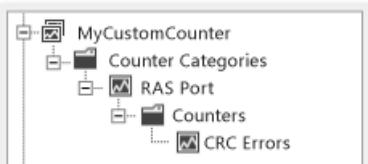
1/16/2020 • 2 minutes to read • [Edit Online](#)

When you create a load test with the **New Load Test Wizard**, you add an initial set of counters. These offer you a set of predefined counter sets for your load test.

NOTE

If your load tests are distributed across remote machines, controller and agent counters are mapped to the controller and agent counter sets. For more information on using remote machines in your load test, see [Test controllers and test agents](#).

You manage your counters in the **Load Test Editor**. The counter sets that are already added to the test are visible in the **Counter Sets** node of the load test. After you create a Load test, you can add new custom counter sets to it.



NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To add a custom counter set to a Load Test

1. Open a load test.
2. Expand the **Counter Sets** node. All the counter sets that have been added to the load test are visible.
3. Right-click the **Counter Sets** node and select **Add Custom Counter Set**.

NOTE

The counter set is given a default name, such as **Custom1**. You can change the name by using the **Properties** window. Press **F4** to display the **Properties** window.

4. To add counters to your custom counter set, right-click the new counter set and then choose **Add Counters**. For more information about how to add counters, see [How to: Add counters to counter sets](#).

NOTE

It is also possible to add a custom counter set by right-clicking an existing counter set, choosing copy, and then pasting it to the counter sets node. Additional counters that are copied, but not needed, can be deleted. You can change the name of the new counter set by using the **Properties** window.

See also

- [Specify the counter sets and threshold rules for computers in a load test](#)
- [Configure load test run settings](#)

How to: Manage counter sets using the Load Test Editor

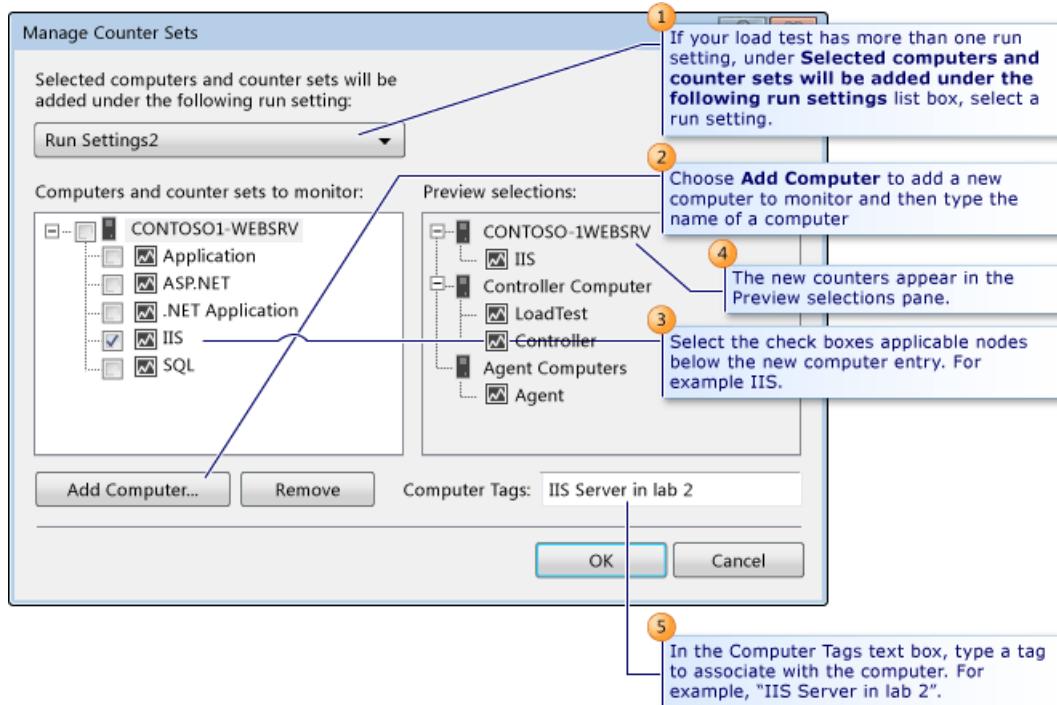
1/16/2020 • 2 minutes to read • [Edit Online](#)

When you create a load test with the **New Load Test Wizard**, you add an initial set of counters. These offer you a set of predefined counter sets for your load test.

NOTE

If your load tests are distributed across remote machines, controller and agent counters are mapped to the controller and agent counter sets. For more information about how to use remote machines in your load test, see [Test controllers and test agents](#).

Managing counter sets involves choosing the set of computers you want to collect performance data from, and assigning a set of counter sets to collect from each individual computer. You manage your counters in the **Load Test Editor**.



NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To manage counter sets

1. Open a load test.
2. Choose the **Manage Counter Sets** button.

– or –

Right-click **Counter Sets** folder in the load test tree and choose **Manage Counter Sets**.

The **Manage Counter Sets** dialog box is displayed.

3. (Optional) In the **Selected computers and counter sets will be added under the following run settings** list box, select a different run setting.

NOTE

This only applies if you have more than one run setting in your load test.

4. (Optional) Choose **Add Computer** to add a new computer to monitor. You will be prompted for a name. Type the name of a computer, and you will see nodes below the new entry. For example **ASP.NET, IIS, SQL**, and others. Select the check boxes in front of the nodes you want to select. The new counters appear in the **Preview selections** pane.
5. (Optional) In the **Computer Tags** text box, type a tag to associate with the computer. For example, "TestMachine12 in lab3".

Computer tags let you identify a computer with an easy-to-recognize name.
The tags are displayed in the **Counter Set Mappings** node in the tree in the Load Test Editor. More important, the tags are displayed in Excel reports, which help stakeholders identify what role the computer has in the load test. For example, "Web Server1 in lab2" or "SQL Server2 in Phoenix office". For more information, see [Report load tests results for test comparisons or trend analysis](#).
6. Choose **OK**.

See also

- [Test controllers and test agents](#)
- [Specify the counter sets and threshold rules for computers in a load test](#)
- [Configure load test run settings](#)

How to: Add a threshold rule using the load test editor

1/1/2020 • 2 minutes to read • [Edit Online](#)

Threshold rules in load tests compare a performance counter value with either a constant value or another performance counter value.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To add a threshold rule

1. Open a load test.
2. In the Load Test Editor, expand the **Counter Sets** node.
3. Expand one of the **Counter Categories** in one of the Counter Sets. For example, you can select **LoadTest:Scenario**. Expand the node.
4. Right-click one of the counters, for example, **User Load**, under **LoadTest:Scenario**. Select **Add Threshold Rule**.

The **Add Threshold Rule** dialog box is displayed.

5. You can choose from two types of rules: **Compare Constant** and **Compare Counter**. Select the appropriate type and set the values.

NOTE

Set the **Alert If Over** property to **True** to indicate that exceeding a threshold is a problem, or to **False** to indicate that falling below a threshold is a problem.

See also

- [Analyze threshold rule violations](#)
- [Specify the counter sets and threshold rules for computers in a load test](#)
- [Analyze load test results](#)

Configure load test run settings

1/1/2020 • 3 minutes to read • [Edit Online](#)

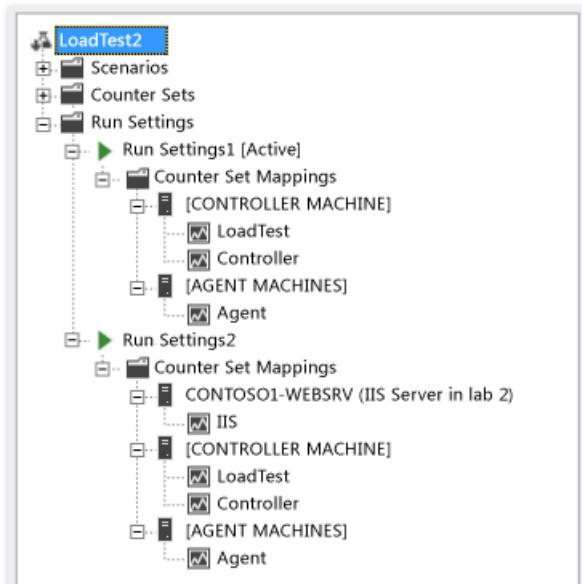
Run settings are a set of properties that influence the way a load test runs. Run settings are organized by categories in the **Properties** window.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

You can have more than one run setting in a load test, however only one of the run settings may be active per run. The other run settings provide a quick way to select an alternative setting to use for subsequent test runs.

The initial run setting is created when you create a load test by using the **New Load Test Wizard**.



Tasks

TASKS	ASSOCIATED TOPICS
Add more run settings to your load test: In addition to the run setting that is created when you run the New Load Test Wizard , you can add more run settings to your load test so that you can run the test under different conditions.	- How to: Add additional run settings to a load test
Specify the active run setting to use with the load test: You can select the run setting that you want to use with your load test using the Load Test Editor. The active run setting is identified by the "[Active]" suffix.	- How to: Select the active run setting for a load test

TASKS	ASSOCIATED TOPICS
<p>Edit run setting properties: You can edit your run setting properties for such things as logging options (see more below), determining the length of the test, warm-up duration, maximum number of error details reported, sampling rate, connection model (web performance tests only), results storage type, validation level and SQL tracing. The run settings should reflect the goals of your load test.</p>	<ul style="list-style-type: none"> - Load test run settings properties - Changing run setting properties
<p>Specify test iteration count in load test run settings: You can specify the number of times to run all of the web performance and unit tests in all of the scenarios of your load tests by configuring the Test Iterations property.</p>	<ul style="list-style-type: none"> - How to: Specify the number of test iterations in a run setting
<p>Specify the sampling rate for a load test run setting: You can specify how frequently to have the load test collect performance counter data by configuring the Sample Rate property.</p>	<ul style="list-style-type: none"> - How to: Specify the sample rate
<p>Specify the timing details storage option: You can specify how you want the details of the load test saved by configuring the Timing Details Storage property.</p>	<ul style="list-style-type: none"> - How to: Specify the timing details storage property
<p>Specify the test resource retention period: Speed up the test > fix > retest cycle by retaining the test resources for a specified period by setting the Resources Retention Time property.</p>	<ul style="list-style-type: none"> - Retain the resources to speed up load testing
<p>Use context parameters: You can use context parameters to parameterize a string. For example, if your load test contains a web performance test that uses a parameterized web server, you can add a context parameter to the run settings that maps to a different server.</p>	<ul style="list-style-type: none"> - How to: Add context parameters to a run setting
<p>Configuring test logging properties: You can configure how frequently data is written to the log that is associated with your load test run settings. This can be important when you are running a large or complex load test because the log could become several gigabytes.</p> <p>You can also configure the log file to be automatically saved when your load test fails to help in debugging and analyzing your application.</p>	<ul style="list-style-type: none"> - Modifying load test logging settings

Load test run settings properties

1/1/2020 • 9 minutes to read • [Edit Online](#)

The run settings of a load test determine a variety of other settings, including the duration of the test, results collection detail level, and the counter sets that are collected when the test runs. You can create and store multiple run settings for each load test, and then select one particular setting to use when you run the test. An initial run setting is added to your load test when you create your load test using the **New Load Test Wizard**.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

The following tables describe the various properties for load test run settings. You can modify these properties to meet your specific load testing requirements.

For more information, see [Configure load test run settings](#).

General properties

PROPERTY	DEFINITION
Description	A description of the Run Settings.
Maximum Error per Type	The maximum number of errors per type to save for the load test. You can increase this number if you have to, but doing this will also increase the size and processing time of the load test result.
Maximum Request URLs Reported	The maximum number of unique web performance test request URLs on which to report results in this load test. You can increase this number if you have to, but doing this will also increase the size and processing time of the load test result.
Maximum Threshold violations	The maximum number of threshold violations to save for this load test. You can increase this number if you have to, but doing this will also increase the size and processing time of the load test result.
Run unit tests in application domain	A Boolean value that determines whether each unit test assembly will run in a separate application domain when the load test contains unit tests. The default setting is True. If your unit tests do not require a separate application domain or app.config file to function correctly, your unit tests might run faster by setting the value of this property to <code>False</code> .

PROPERTY	DEFINITION
Name	The name of the Run Setting as it appears in the Run Settings node of the Load Test Editor .
Validation Level	This defines the highest level of validation rule that will run in a load test. Validation rules are associated with web performance test requests. Each validation rule has an associated validation level: High , Medium , or Low . This load test run setting will specify which validation rules will run while the web performance test is run in the load test. For example, if this run setting is set to Medium , all validation rules marked as Medium , or Low will be run.

Logging properties

PROPERTY	DEFINITION
Maximum Test Logs	Specifies the maximum number of test logs to save for the load test. When the value entered for the maximum number of test logs is reached, the load test will stop collecting logs. Therefore, the logs will be collected at the beginning of the test, not the end. The load test will continue to run until it is completed.
Save Log Frequency for Completed Tests	Specifies the frequency at which the test log will be written. The number indicates that one out of every entered number of tests will be saved to the test log. For example, entering the value of ten specifies that the tenth, twentieth, thirtieth and so on will be written to the test log. Setting the value to 0 specifies that no test logs will be saved.
Save Log on Test Failure	A Boolean value that determines whether if test logs are saved if a test fails in a load test. The default is <input checked="" type="checkbox"/> True . For more information, see How to: Specify if test failures are saved to test logs

For more information, see [Modify load test logging settings](#).

Results properties

PROPERTY	DEFINITION
Storage Type	The way to store the performance counters that are obtained in a load test. The options are as follows: <ul style="list-style-type: none"> - Database - Requires an SQL database that has a Load Test Results Store. - None.

PROPERTY	DEFINITION
Timing Details Storage	<p>This is used to determine which details will be stored in the Load Test Results Store. Three values are available:</p> <ul style="list-style-type: none"> - AllIndividualDetails - Collect and store individual timing values for each test, transaction, and page that was run or issued during the load test in the Load Test Results Store. It is required if you intend to use the Virtual User Activity Chart in the Load Test Analyzer. For more information, see Analyze virtual user activity in the Details view. - None - Do not collect any individual timing values. This is the default value for Visual Studio 2013 Update 4 and later releases. - StatisticsOnly - Collect and store only the statistics instead of storing the individual timing values for each test, transaction, and page that was executed or issued during the load test in the Load Test Results Store. <p>For more information, see How to: Specify the timing details storage property.</p>

SQL tracing properties

PROPERTY	DEFINITION
Minimum Duration of Traced SQL Operations	<p>The minimum duration of a SQL operation to be captured by the SQL Trace, in milliseconds. For example, this lets you ignore operations that complete quickly if you are trying to find SQL operations that are slow under load.</p>
SQL Tracing Connect String	<p>The connection string that is used to access the database to be traced.</p>
SQL Tracing Directory	<p>The location where the SQL Trace file is put after the trace ends. This directory must have write permissions for SQL Server and read permissions for the controller.</p>
SQL Tracing Enabled	<p>This enables the tracing of SQL operations. The default value is <code>False</code>.</p>

Test iterations properties

PROPERTY	DEFINITION
Test Iterations	<p>Specifies the total number of individual tests to run before the load test is complete. This property only applies when the property "Use Test Iterations" is <code>True</code>.</p>
Use Test Iterations	<p>If Use Test Iterations is <code>True</code>, then the load test runs until the number of individual tests completed within the load test reaches the number that is specified by the "Test Iterations" property. In this case, the time-based settings, which are Warm up Duration, Run Duration, and Cool-down Duration, are ignored. If "Use Test Iterations" is <code>False</code>, all the timing settings apply, and "Test Iterations" is ignored.</p>

For more information, see [How to: Specify the number of test iterations in a run setting](#).

Timing properties

PROPERTY	DEFINITION
Cool-down Duration	The duration of the test cool-down period, expressed in hh:mm:ss format. Individual tests within a load test might still be running when the load test finishes. During the cool-down period, those tests can continue until they complete or the end of the cool-down period is reached. By default, there is no cool-down period, and individual tests are terminated when the load test finishes based on the Run Duration setting.
Run Duration	The length of the test, in hh:mm:ss format.
Sample Rate	The interval at which to capture performance counter values, in hh:mm:ss format. For more information, see How to: Specify the sample rate .
Warm up Duration	The period between the beginning of the test and when the data samples start being recorded, in hh:mm:ss format. This is frequently used to step load virtual users to reach a certain load level before recording sample values. The sample values that are captured before the warm-up period ends are shown in the Load Test Analyzer .

WebTest connections properties

PROPERTY	DEFINITION
----------	------------

PROPERTY	DEFINITION
WebTest Connection Model	<p>This controls the usage of connections from the load test agent to the web server for web performance tests that run inside a load test. Three web performance test connection model options are available:</p> <ul style="list-style-type: none"> - The Connection Per User model simulates the behavior of a user who is using a real browser. When Internet Explorer 6 or Internet Explorer 7 is simulated, each virtual user who is running a web performance test uses one or two dedicated connections to the web server. The first connection is established when the first request in the web performance test is issued. A second connection may be used when a page contains more than one dependent request. These requests are issued in parallel by using the two connections. These connections are reused for subsequent requests in the web performance test. The connections are closed when the web performance test finishes. A drawback to this model is that the number of connections that is held open on the agent computer might be high (up to two times the user load). Consequently, the resources that are required to support this high connection count might limit the user load that can be driven from a single load test agent. When Internet Explorer 8 is simulated, six concurrent connections are supported. - The Connection Pool model conserves the resources on the load test agent by sharing connections to the web server among multiple virtual web performance test users. If the user load is larger than the connection pool size, the web performance tests that are run by different virtual users will share a connection. This could mean that one web performance test might have to wait before it issues a request when another web performance test is using the connection. The average time that a web performance test waits before it submits a request is tracked by the load test performance counter Average Connection Wait Time. This number should be less than the average response time for a page. If it is not, the connection pool size is probably too small. - The Connection Per Test Iteration model specifies the use of dedicated connections for each test iteration.
WebTest Connection Pool Size	<p>This specifies the maximum number of connections to make between the load test agent and the Web server. This applies only to the Connection Pool model.</p>

Change run setting properties

You can add more run settings to your load test with different property settings so that you can run the load test under different conditions. For example, you can add a new test setting and use a different sample rate, or specify a longer run duration. You can only use one run setting at a time and you must specify which run setting to use by marking it as active. For an example, see [How to: Select the active run setting for a load test](#).

To change run settings:

1. Open a load test.
2. Expand the **Run Settings** folder.
3. Choose a **Run Settings** node.
4. On the **View** menu, choose **Properties Window**.

The **Properties Window** is displayed and the properties for the selected run setting are displayed.

5. Use the **Properties Window** to change the run settings. For example, change the run duration to **00:05:00** to run your test for five minutes.

NOTE

For a complete list of the run settings properties and their descriptions, see [Load test run setting properties](#).

6. When you are finished changing properties, save your load test. On the **File** menu, choose **Save**.

NOTE

Counter set mappings are also a part of run settings. For more information, see [Specify the counter sets and threshold rules for computers in a load test](#).

See also

- [Configure load test run settings](#)

How to: Add additional run settings to a load test

1/1/2020 • 2 minutes to read • [Edit Online](#)

The run settings of a load test determine a variety of other settings. These include the duration of the test, results collection detail level, and the counter sets that are collected when the test runs. You can create and store multiple run settings for each load test, and then select one particular setting to use when you run the test. An initial run setting is added to your load test when you create your load test by using the **New Load Test Wizard**.

You can add more run settings to your load test with different property settings so that you can run the load test under different conditions. For example, you can add a new test setting and use a different sample rate, or specify a longer run duration. You can use only one run setting at a time and must specify which run setting to use by marking it as active.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To add another run setting

1. Open a load test.
2. (Optional) Expand the **Run Settings** folder.
3. Right-click the **Run Settings** folder and select **Add Run Settings**.

A new run setting is added to the **Run Settings** folder.

4. On the **View** menu, choose **Properties Window**.

The **Properties** window is displayed with the properties for the selected run setting.

5. In the **Properties** window, use the text box for the **Name** property to give the new run setting a name that describes the intent of the run setting (for example, **Run Setting: Five minute run**).
6. Use the **Properties** window to change the run settings. For example, change the run duration to **00:05:00** to run your test for five minutes.

NOTE

For a full list of the run settings properties and their descriptions, see [Load test run settings properties](#).

You can now specify that you want to use the added run setting by setting it to active. For more information, see [How to: Select the active run setting for a load test](#).

See also

- [Configure load test run settings](#)
- [Specify the counter sets and threshold rules for computers in a load test](#)

How to: Add context parameters to a load test run setting

1/1/2020 • 3 minutes to read • [Edit Online](#)

After you create your load test by using the **New Load Test Wizard**, you can use the **Load Test Editor** to change the scenarios properties to meet your testing needs and goals.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

NOTE

For a full list of the run settings properties and their descriptions, see [Load test run settings properties](#).

You can create context parameters to use in a load test run setting by using the Load Test Editor. Context parameters let you parameterize a string.

Suppose your load test contains a web performance test that already uses a parameterized web server URL by using a context parameter. You can add a context parameter to a load test run setting that uses the same name value as the one that is used in the web performance test. This will map the web performance test to a different server when you run the load test. For example, if your load test includes a web performance test that uses a context parameter that is named WebServer1 for the name of the web server in the URL. If you then specify a context parameter in your load test run setting that is also named WebServer1, the load test will use the context parameter that you assigned in the load test run setting. To clarify, if the web performance test in the load test uses the same context parameter name as a context parameter in the load test, the context parameter in the load test will override the context parameter that is used in the web performance test.

WARNING

Be careful not to unintentionally override the context parameter of a web performance test when you use context parameters in a run setting. Avoid using the same context parameter names unless you do this intentionally.

If you assign the value of the Webserver1 context parameter to `http://CorporateStagingWebServer`, you can then use `WebServer1` throughout the load test and thereby easily change the value to a different web server at any time.

Additionally, by assigning different values to a context parameter by using the same name in different load test run settings, you can run the load test by using different environments:

- Corporate Staging Web Server run setting: The context parameter that is named
`WebServer1=http://CorporateStagingWebServer`
- Corporate Production Web Server run setting: The Context parameter that is named
`WebServer1=http://CorporateProductionWebServer`

Changing the Run Setting from the Command Line

If you want to use different run settings from the command line to take advantage of the context parameter

strategy, use the following commands:

Set Test.UseRunSetting= CorporateStagingWebServer

-and-

mstest /testcontainer:loadtest1.loadtest

To add a context parameter to a run setting

1. Open a load test.
2. Expand the **Run Settings** folder in the load test tree in the Load Test Editor.
3. Right-click the specific run setting to which you want to add a context parameter and then choose **Add Context Parameter**.

A new context parameter is added to the **Context Parameters** folder in the **Run Settings** folder in the load test tree.

-or-

If the run setting already contains a **Context Parameters** folder, you can right-click it and then choose **Add Context Parameter**.

4. In the **Properties** window, change the value for **Name** as appropriate (for example, WebServer1). In the **Properties** window, change **Value** to the parameter that you want to use (for example, `http://CorporateStagingWebServer`).
5. (Optional) Repeat steps 3 through 5 and use a different string for the **Value** property (for example, `http://CorporateProductionWebServer`).
6. Choose which run settings that you want to be active. Open the shortcut menu on the run settings and choose **Set As Active**.

See also

- [Configure load test run settings](#)

Modify load test logging settings

1/1/2020 • 2 minutes to read • [Edit Online](#)

The load test result for the completed load test contains performance counter samples and error information that was collected in a log periodically from the computers-under-test. A large number of performance counter samples can be collected over the course of a load test run. The amount of performance data that is collected depends on the length of the run, the sampling interval, the number of computers under test, and the number of counters to collect. For a large load test, the amount of performance data that is collected can easily be several gigabytes; therefore, you might consider modifying how often data is saved to the log. See [Test controllers and test agents](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

The *test controller* spools all collected load test sample data to a database log while the test is running. Additional data, such as timing details and error details, is loaded into the database when the test completes.

TASK	ASSOCIATED TOPICS
Save logs if a load test fails: You can specify if you want to save the test log whenever a load test fails.	- How to: Specify if test failures are saved to test logs
Set the maximum file size for the log file: You can edit the XML configuration file that is associated with the test controller service to specify the maximum file size you want to use for the log file.	Modify <code><add key="LogSizeLimitInMegs" value="20"/></code> in the <code>QTCController.exe.config</code> XML configuration file.

See also

- [Configure load test run settings](#)

How to: Specify if test failures are saved to test logs using the Load Test Editor

1/1/2020 • 2 minutes to read • [Edit Online](#)

After you create your load test with the **New Load Test Wizard**, you can use the **Load Test Editor** to change the load test properties to meet your testing needs and goals. See [Walkthrough: Create and run a load test](#). You can specify if you want to have the test log saved if a test fails in a load test by changing the **Save Log on Test Failure** property.

NOTE

For a complete list of the run settings properties and their descriptions, see [Load test run settings properties](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To specify if the test log is saved when a test fails in a scenario

1. Open a load test.

The **Load Test Editor** appears. The load test tree is displayed.

2. In the load test trees **Run Settings** folder, choose the run settings node that you want to specify the maximum number of test iterations for.
3. On the **View** menu, select **Properties Window**.

The run settings categories and properties are displayed in the **Properties** window.

4. In the **Save Log on Test Failure** property, select either **True** or **False** to specify if you want to save the test log in the event of a test failure in the scenario.

After you have finished changing the property, choose **Save** on the **File** menu.

The data that is saved in the log can be viewed using the Load Test Analyzer's Tables view. For more information, see [Analyze load test results and errors in the Tables view](#).

See also

- [Edit load test scenarios](#)
- [Walkthrough: Create and run a load test](#)

How to: Specify the timing details storage property for a load test run setting

1/1/2020 • 3 minutes to read • [Edit Online](#)

After you create your load test with the **New Load Test Wizard**, you can use the **Load Test Editor** to change the settings to meet your testing needs and goals.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

You can edit a run setting's **Timing Details Storage** property's value in the **Properties** window. The **Timing Details Storage** property can be set to any of the following options:

- **All Individual Details:** Collects and stores individual timing data for each test, transaction, and page issued during the test.

NOTE

The **All Individual Details** option must be selected to enable virtual user data information in your load test results. For more information, see [Analyze virtual user activity in the Details view](#).

- **None:** Does not collect any individual timing details. However, the average values are still available.
- **Statistics Only:** Stores individual timing data, but only as percentile data. This saves space resources.

Considerations for the Timing Details Storage Property

If the **Timing Details Storage** property is enabled, then the time to execute each individual test, transaction, and page during the load test will be stored in the load test results repository. This allows for 90th and 95th percentile data to be shown in the **Load Test Analyzer** in the **Tests**, **Transactions**, and **Pages** tables.

If the **Timing Details Storage** property is enabled, by setting its value to either **StatisticsOnly** or **AllIndividualDetails**, all the individual tests, pages, and transactions are timed, and percentile data is calculated from the individual timing data. The difference is that with the **StatisticsOnly** option, after the percentile data has been calculated, the individual timing data is deleted from the repository. This reduces the amount of space that is required in the repository when timing details are used. However, you might want to process the timing detail data in other ways by using SQL tools, in which case the **AllIndividualDetails** option should be used so that the timing detail data is available for that processing. Additionally, if you set the property to **AllIndividualDetails**, then you can analyze the virtual user activity using the **Virtual User Activity Chart** in the **Load Test Analyzer** after the load test completes running. For more information, see [Analyze virtual user activity in the Details view](#).

The amount of space required in the load test results repository to store the timing details data can be very large, especially for longer running load tests. Also, the time to store this data in the load test results repository at the end of the load test is longer because this data is stored on the load test agents until the load test has finished executing, at which time the data is stored into the repository. The **Timing Details Storage** property is enabled by default. If this is an issue for your testing environment, you may wish to set

the **Timing Details Storage** to **None**.

The timing details data is stored in the *LoadTestItemResults.dat* file during the run and is sent back to the controller after the load test is complete. For a load test running for a long duration, the size of the file is large. If there is not enough disk space on the agent machine, this will be an issue.

If you are upgrading a project from a previous version of Visual Studio load test, use the following procedure to enable full detail collection.

To configure the timing details storage property in a load test

1. Open a load test in the load test editor.
2. Expand the **Run Settings** node in the load test.
3. Choose on the run settings that you want to configure, for example **Run Settings1[Active]**.
4. Open the **Properties** Window. On the **View** menu, select **Properties Window**.
5. Under the **Results** category, choose the **Timing Details Storage** property and select **All Individual Details**.

After you have configured the **All Individual Details** setting for the **Timing Details Storage** property, you can run your load test and view the **Virtual User Activity Chart**. For more information, see [How to: Analyze what virtual users are doing during a load test](#).

See also

- [Analyze virtual user activity in the Details view](#)
- [Walkthrough: Using the Virtual User Activity Chart to isolate issues](#)

How to: Specify the number of test iterations in a load test run setting

1/1/2020 • 2 minutes to read • [Edit Online](#)

After you create your load test with the **New Load Test Wizard**, you can use the **Load Test Editor** to change the scenarios properties to meet your testing needs and goals. For more information, see [Walkthrough: Create and run a load test](#).

Using the **Load Test Editor**, you can edit the **Test Iterations** property of a run settings value in the **Properties** window. The **Test Iterations** property specifies the number of iterations to run on all the web performance and unit tests in all the scenarios in a load test using the **Load Test Editor**.

NOTE

For a complete list of the run settings properties and their descriptions, see [Load test run settings properties](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To specify the number of test iterations in a run setting

1. Open a load test.

The **Load Test Editor** appears and displays the load test tree.

2. In the load test tree, in the **Run Settings** folder, choose a run setting.
3. On the **View** menu, select **Properties Window** to view the load run setting's categories and properties.
4. Set the **Use Test Iterations** property to **True**.
5. In the **Test Iterations** property, enter a number that indicates the number of test iterations to run during the load test.
6. After you have finished changing the property, choose **Save** on the **File** menu. You can then run your load test using the new **Test Iterations** value.

See also

- [Configure load test run settings](#)
- [Load test scenario properties](#)

How to: Specify the sample rate for a load test run setting

1/1/2020 • 2 minutes to read • [Edit Online](#)

After you create your load test with the **New Load Test Wizard**, you can use the **Load Test Editor** to change the properties to meet your testing needs and goals.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Using the **Load Test Editor**, you can edit a run setting's **Sample Rate** property's value in the **Properties** window. For a complete list of the run settings properties and their descriptions, see [Load test run settings properties](#).

Choose an appropriate value for the **Sample Rate** property for the load test run setting based on the length of your load test. A smaller sample rate, such as the default value of five seconds, requires more space in the load test results database. For longer load tests, increasing the sample rate reduces the amount of data that you collect. For more information, see [How to: Specify the sample rate for a load test run setting](#).

Here are some guidelines for sample rates:

LOAD TEST DURATION	RECOMMENDED SAMPLE RATE
< 1 Hour	5 seconds
1 - 8 Hours	15 seconds
8 - 24 Hours	30 seconds
> 24 Hours	60 seconds

To specify performance counter sampling rate in a run setting

1. Open a load test.

The **Load Test Editor** appears. The load test tree is displayed.

2. In the load test tree, in the **Run Settings** folder, choose the run setting that you want to specify the sample rate for.
3. On the **View** menu, select **Properties Window**.

The load run setting's categories and properties are displayed in the **Properties** window.

4. In the **Sample Rate** property, enter a time value that indicates the frequency at which the load test will collect performance counter data.
5. After you have finished changing the property, choose **Save** on the **File** menu. You can then run your load test using the new **Sample Rate** value.

See also

- [Configure load test run settings](#)
- [Load test scenario properties](#)

How to: Select the active run setting for a load test

1/1/2020 • 2 minutes to read • [Edit Online](#)

After you create your load test with the **New Load Test Wizard**, you can use the **Load Test Editor** to change the scenarios properties to meet your testing needs and goals.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

A load test can contain one or more *run settings* which are a set of properties that influence the way a load test runs. Run settings are organized by categories in the **Properties** window. When a load test is run, it uses the run setting that is currently set as active.

NOTE

For a complete list of the run settings properties and their descriptions, see [Load test run settings properties](#).

If your load test contains only one run setting node under the **Run Settings** folder, that node is always the active node. If your load test contains multiple run settings nodes, you can select the one to use when you run a load test. See [How to: Add Additional Run Settings to a Load Test](#).

In the **Load Test Editor**, the active run setting is identified by the "[Active]" suffix.

Select the active run setting

1. Open a load test.
2. Expand the **Run Settings** folder.
3. Right-click the run settings node that you want to be the active node, and then choose **Set As Active**.

In the **Load Test Editor**, the affected run setting node is updated with the "[Active]" suffix.

The run setting selected becomes active, and remains active until you select a different run setting to be active.

NOTE

You can override the active run setting by setting an environment variable named

`Test.UseRunSetting=<run setting name>`. This is useful when you run a load test from the command line or from a batch file. This lets you choose different run settings without opening your load test.

Specify the run setting to use from the command line

You can override the default run settings in your load test by setting an environment variable from the command line:

Set `Test.UseRunSetting=PreProdEnvironment`

And to run the test:

```
mstest /testcontainer:loadtest1.loadtest
```

See also

- [Configure load test run settings](#)
- [Specify the counter sets and threshold rules for computers in a load test](#)
- [How to: Add additional run settings to a load test](#)

How to: Select a load test run setting to use from the command line

1/1/2020 • 2 minutes to read • [Edit Online](#)

A load test can include *run settings*, which are properties that influence the way a load test runs. Run settings are organized by categories in the **Properties** window. When a load test is run, it uses the run setting that is currently set as active.

If your load test contains only one run setting, it is always the active node. If your load test contains multiple run settings nodes, you can select the one to use when you run a load test from the command line. See [How to: Add additional run settings to a load test](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To change the run setting from the command line

1. If you want to use different run settings from the command line to take advantage of the context parameter strategy, use the following command:

```
Set Test.UseRunSetting= CorporateStagingWebServer
```

2. Run the load test using mstest:

```
mstest /testcontainer:loadtest1.loadtest
```

See also

- [Configure load test run settings](#)
- [Specify the counter sets and threshold rules for computers in a load test](#)
- [How to: Add additional run settings to a load test](#)
- [How to: Select the active run setting for a load test](#)

Walkthrough: Create and run a load test that contains unit tests

1/1/2020 • 4 minutes to read • [Edit Online](#)

In this walkthrough you create a load test that contains unit tests.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

This walkthrough steps you through creating and then running a load test using Visual Studio Enterprise. A load test is a container of web performance tests and unit tests. You create load tests with the **New Load Test Wizard**.

A load test also exposes many run-time properties that can be modified to generate the desired load simulation. In this walkthrough, you use the **New Load Test Wizard** to add unit tests to a load test.

In this walkthrough, you will complete the following tasks:

- Create a load test that uses unit tests.
- Change some of the load test settings.
- Run a load test.
- Perform the steps in [Walkthrough: Creating and running unit tests for managed code](#) to create a simple C# class library that contains a web performance and load test project with some unit tests in it.

Create a load test containing unit tests using the New Load Test Wizard

To start the New Load Test Wizard

1. Open the Bank solution that you created in [Walkthrough: Creating and running unit tests for managed code](#).
2. In **Solution Explorer**, open the shortcut menu for the Bank solution node, choose **Add**, and then choose **New Project**.

The **Add New Project** dialog box displays.

3. In the **Add New Project** dialog box, expand **Visual C#** and choose **Test**. From the list of templates, choose **Web Performance and Load Test Project** and in the **Name** field, type `BankLoadTest`. Choose **OK**.

The BankLoadTest web performance and load test project is added to the solution.

4. Open the shortcut menu for the new BankLoadTest web performance and load test project, choose **Add**, and then choose **Load Test**.
5. The **New Load Test Wizard** starts.
6. The **Welcome** page of the **New Load Test Wizard** is the first page.

7. Choose **Next**.

To edit settings for load test scenario

1. In the **Enter a name for the load test scenario** text box, type **ScenarioSample**.

A **scenario** is a grouping mechanism. It consists of a set of tests and the properties for running those tests under load.

2. Set the **Time Profile Think** to `Use normal distribution centered on recorded think times`. Think times represent the time that a user would ponder a web page before going on to the next page.

3. Choose **Next** when you are finished.

To edit load pattern setting for test scenario

1. Choose **Step load**.

NOTE

You can choose from two types of load patterns: constant and step. Each type has its function in load testing, but for the purposes of this walkthrough choose **Step load**.

2. Set **Start user count** to 10 users.

3. Set **Step duration** to 10 seconds.

4. Set **Step user count** to 10 users/step.

5. Set **Maximum user count** to 100 users.

6. Choose **Next**.

To select test mix model for the scenario

1. Under **How should the test mix be modeled**, select **Based on the total number of test**.

2. Choose **Next**.

To add unit tests to the scenario

1. The next step is to **Add tests to a load test scenario and edit test mix**.

2. Choose **Add** to select tests.

3. Choose the **CreditTest** unit tests listed in the **Available Tests** pane, which lists all the web performance tests and unit tests in the web performance and load test project.

4. Choose the arrow to add the **CreditTest** unit test to the **Selected Tests** pane.

5. Repeat steps 3 and 4 for the **DebitTest** and **FreezeAccountTest** unit tests.

6. When you have finished adding the three unit tests, choose **OK**.

You are presented with the test mix.

7. Move the slider under **Distribution** for the **CreditTest** slightly to the right to adjust the test distribution. Notice that the other sliders move to the left automatically so that the distribution remains at 100%.

8. Choose **Next**.

To select network mix for test scenario

1. Select the LAN connection type to add to the network bandwidth mix.

You can add more network types. Use the sliders to adjust the test distribution and weighting.

2. Choose **Next**.

To specify computers to monitor with counter sets during load test run

1. Choose **Next**.

For more information about the counter sets, see [Specify the counter sets and threshold rules for computers in a load test](#).

To edit run setting for load test

1. Select **Load test duration** and then set **Run Duration** to 2 minutes in order to *smoke test* your load test.

When you build your load tests, it is a good practice to validate that everything is configured correctly and running as expected by running a short, light load test. This process is known as *smoke testing*.

2. Choose **Finish**. Your Load test is opened in the **Load Test Editor**.

Run the load test

After you have created the Load test, run it to view how your bank application responds to the load simulation. While a load test is running, you see the **Load Test Analyzer** window.

To run the load test

1. With a Load test open in the **Load Test Editor**, choose the green **Run Test** button in the toolbar. Your load test starts to run.
2. If your test simulation exceeds any thresholds, icons appear in the tree control nodes to indicate a threshold violation. Errors have a red circle overlay, warnings have a yellow triangle overlay. You can find a counter that exceeded the threshold and graph it by dragging the icon onto the graph. You can do this while the test is running.

See also

- [Edit the test mix to specify which tests to include in a load test scenario](#)
- [Specify virtual network types](#)
- [Edit load test scenarios](#)
- [Edit load patterns to model virtual user activities](#)
- [Edit text mix models to specify the probability of a virtual user running a test](#)

How to: Create a web service test

1/1/2020 • 2 minutes to read • [Edit Online](#)

You can use a web performance test to test web services. By using the **Insert Request** and **Insert Web Service Request** options, you can customize the individual requests in the **Web Performance Test Editor** to locate web service pages. Typically, you do not display these pages in the web application. Therefore, you must customize the request to gain access to these pages.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

The following procedures use a web service that is contained within the Commerce Starter Kit. You can download it from [ASP.NET commerce starter kit](#).

Requirements

Visual Studio Enterprise

To test a web service

1. Create a new web performance test. As soon as the browser opens, choose **Stop**.
2. In the **Web Performance Test Editor**, right-click the web performance test and select **Add Web Service Request**.
3. In the **Url** property of the new request, type the name of the web service, such as **http://localhost/storecsvs/InstantOrder.asmx**.
4. Open a separate session of the browser and type the URL of the .asmx page in the **Address** toolbar. Select the method that you want to test and examine the SOAP message. It contains a **SOAPAction**.
5. In the **Web Performance Test Editor**, right-click the request and select **Add Header** to add a new header. In the **Name** property, type **SOAPAction**. In the **Value** property, type the value that you see in **SOAPAction**, such as **"http://tempuri.org/CheckStatus"**.
6. Expand the URL node in the editor, choose the **String Body** node and in the **Content Type** property enter a value of **text/xml**.
7. Return to the browser in step 4, select the XML portion of the SOAP request from the web service description page and copy it to the clipboard.
8. The XML content resembles the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <CheckStatus xmlns="http://tempuri.org/">
            <userName>string</userName>
            <password>string</password>
            <orderID>int</orderID>
        </CheckStatus>
    </soap:Body>
</soap:Envelope>
```

9. Return to the **Web Performance Test Editor** and then choose the ellipsis (...) in the **String Body** property. Paste the contents of the clipboard into the property.
10. You must replace any placeholder values in the XML with valid values for the test to pass. In the previous sample you would replace the two instances of `string` and one `int`. This web service operation will only complete if there is a registered user who has placed an order.
11. Right-click the web service request and select **Add URL QueryString Parameter**.
12. Assign the query string parameter a name and value. In the previous example, the name is `op` and the value is `CheckStatus`. This identifies the web service operation to perform.

NOTE

You can use data binding in the SOAP body to replacing any placeholder value with data bound values by using the `{{DataSourceName.TableName.ColumnName}}` syntax.

13. Run the test. In the top pane of the **Web Performance Test Results Viewer**, select the web service request. In the bottom pane, select the web Browser tab. The XML that is returned by the web service, and the results of any operations, will be displayed.

See also

- [Create custom code and plug-ins for load tests](#)

Generate and run a coded web performance test

1/1/2020 • 2 minutes to read • [Edit Online](#)

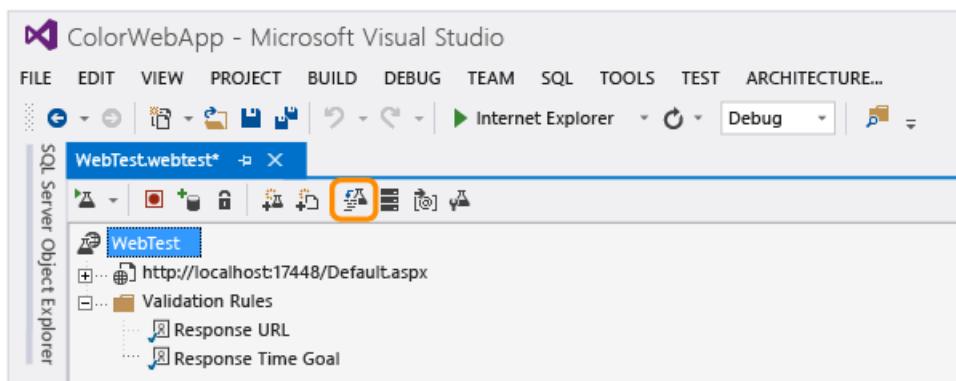
Web performance tests are recorded by browsing through your web app. The tests are included in load tests to measure the performance of your web application under the stress of multiple users. A web performance test can be converted to a code-based script that you can edit and customize like any other source code. For example, you can add looping and branching constructs.

NOTE

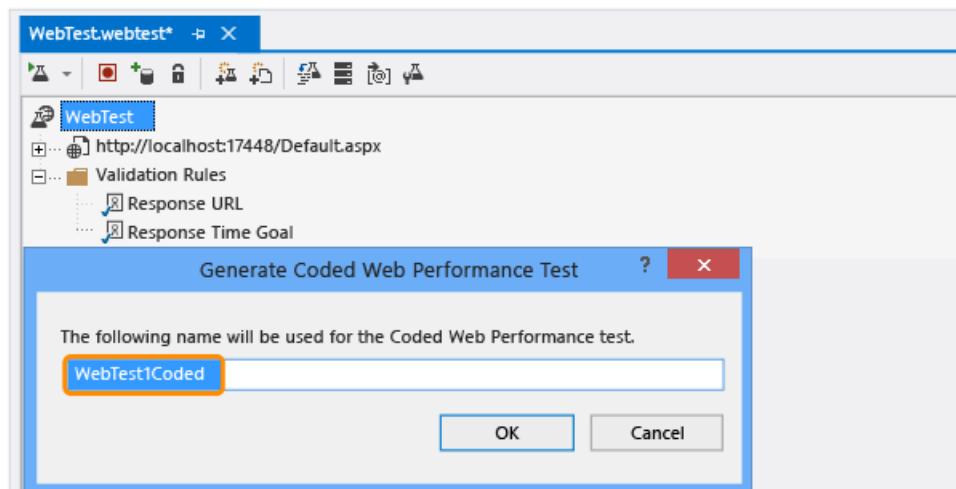
Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Generate a coded web performance test

1. If you have not created a web performance test, see [Record a web performance test](#).
2. Generate the coded test.

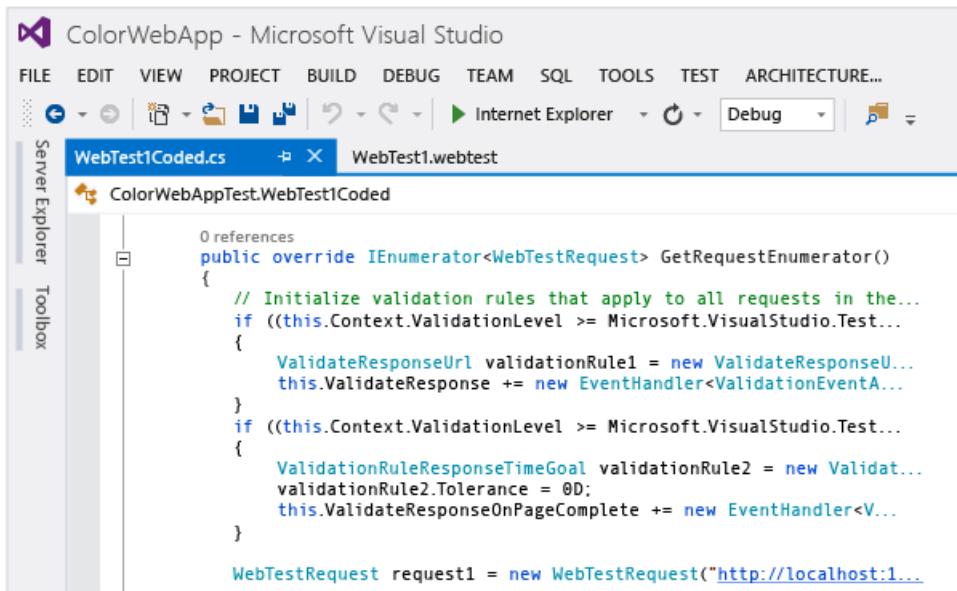


3. Name the test.



The new coded test opens in the code editor.

Depending on which web performance and load test project template you added to your solution, the code will be generated in either Visual Basic, or Visual C#.



```
0 references
public override IEnumerator<WebTestRequest> GetRequestEnumerator()
{
    // Initialize validation rules that apply to all requests in the...
    if ((this.Context.ValidationLevel >= Microsoft.VisualStudio.TestTools.UnitTestingValidationLevel.Warning) && (this.Context.TestMethod != null))
    {
        ValidateResponseUrl validationRule1 = new ValidateResponseUrl();
        validationRule1.Url = "http://localhost:12345";
        this.ValidateResponse += new EventHandler<ValidationEventArgs>(validationRule1.ValidateResponse);
    }
    if ((this.Context.ValidationLevel >= Microsoft.VisualStudio.TestTools.UnitTestingValidationLevel.Error) && (this.Context.TestMethod != null))
    {
        ValidationRuleResponseTimeGoal validationRule2 = new ValidationRuleResponseTimeGoal();
        validationRule2.Tolerance = 0D;
        validationRule2.Timeout = 10000;
        this.ValidateResponseOnPageComplete += new EventHandler<ValidationEventArgs>(validationRule2.ValidateResponseOnPageComplete);
    }
}

WebTestRequest request1 = new WebTestRequest("http://localhost:12345");
```

You can see in the code that the GetRequestEnumerator() method in C#, or the Run() method in Visual Basic, contains each validation rule and web request that was in the recoded test.

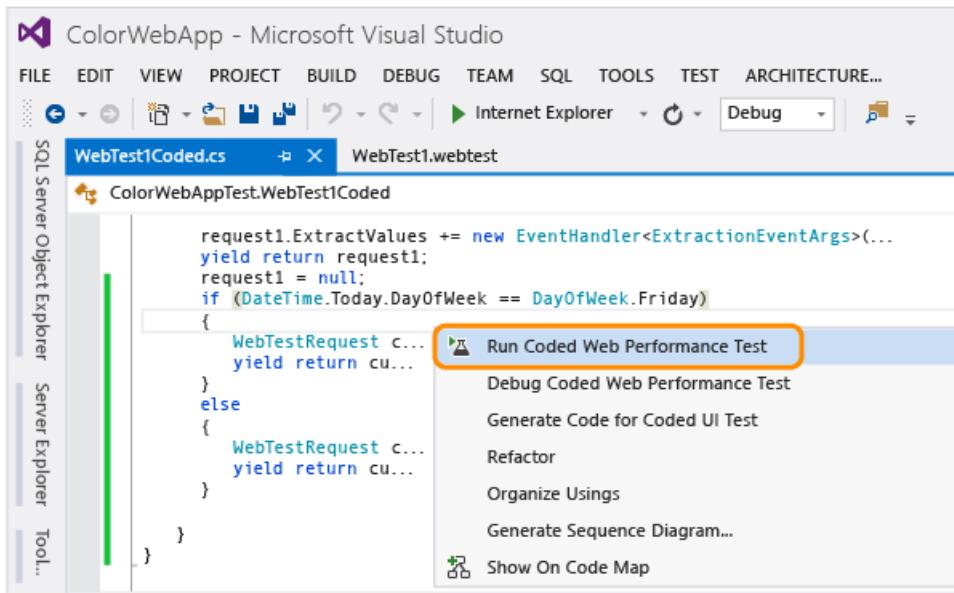
4. To demonstrate adding some simple code, scroll down to the end of the method and after the code for the last web request, and add the following code:

```
if (DateTime.Today.DayOfWeek == DayOfWeek.Wednesday)
{
    WebTestRequest customRequest = new WebTestRequest("http://weather.msn.com/");
    yield return customRequest;
}
else
{
    WebTestRequest customRequest = new WebTestRequest("https://msdn.microsoft.com/");
    yield return customRequest;
}
```

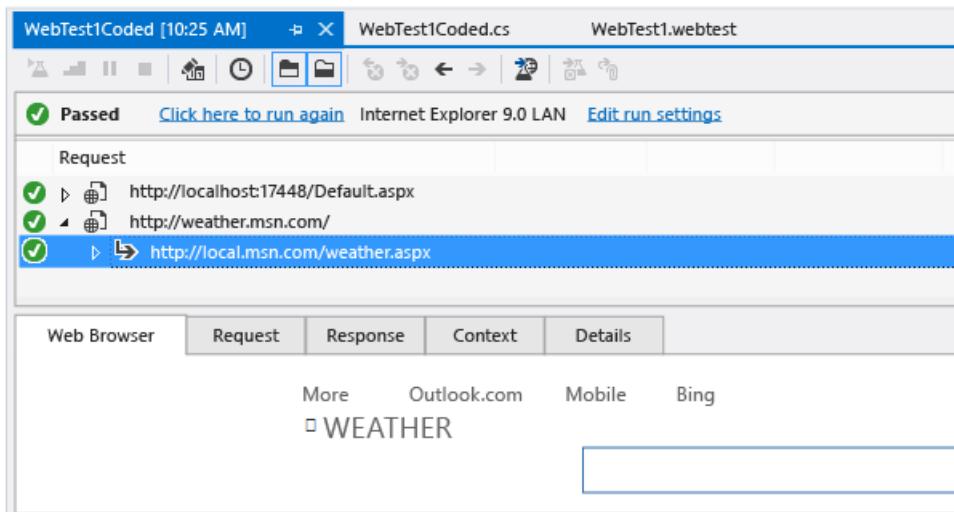
```
If DateTime.Today.DayOfWeek = DayOfWeek.Wednesday Then
    Dim customRequest As WebTestRequest = New WebTestRequest("http://weather.msn.com/")
    MyBase.Send(customRequest)
Else
    Dim customRequest As WebTestRequest = New WebTestRequest("https://msdn.microsoft.com/")
    MyBase.Send(customRequest)
End If
```

5. Build the solution to verify that your custom code compiles.

6. Run the test.



And because the day this was run happened to be a Wednesday...



Q&A

Q: Can I run more than one test at a time?

A: Yes, use the right-click (context) menu in **Solution Explorer**.

Q: Should I add a data source before or after I generate a coded test?

A: It is easier to add a [data source](#) before you generate the coded test because the code will be automatically generated for you.

When you run a coded test with a data source, you might see the following error message:

Could not run test <Test Name> on agent <Computer Name>: Object reference not set to an instance of an object.

This can occur because you have a `DataSourceAttribute` defined for the test class, without a corresponding `DataBindingAttribute`. To resolve this error, add an appropriate `DataBindingAttribute`, delete it, or comment it out of the code.

Q: Should I add validation and extraction rules before or after I generate a coded test?

A: It is easier to add validation rules and extraction rules before you generate the coded test; however, we recommend that you use [coded UI tests](#) for validation purposes.

Add a data source to a web performance test

1/1/2020 • 4 minutes to read • [Edit Online](#)

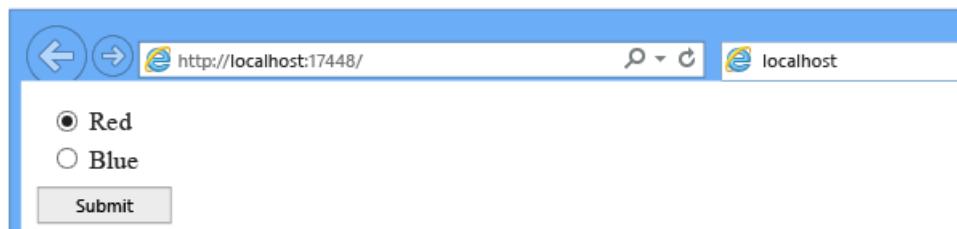
Bind data to provide different values to the same test, for example, to provide different values to your form post parameters.

NOTE

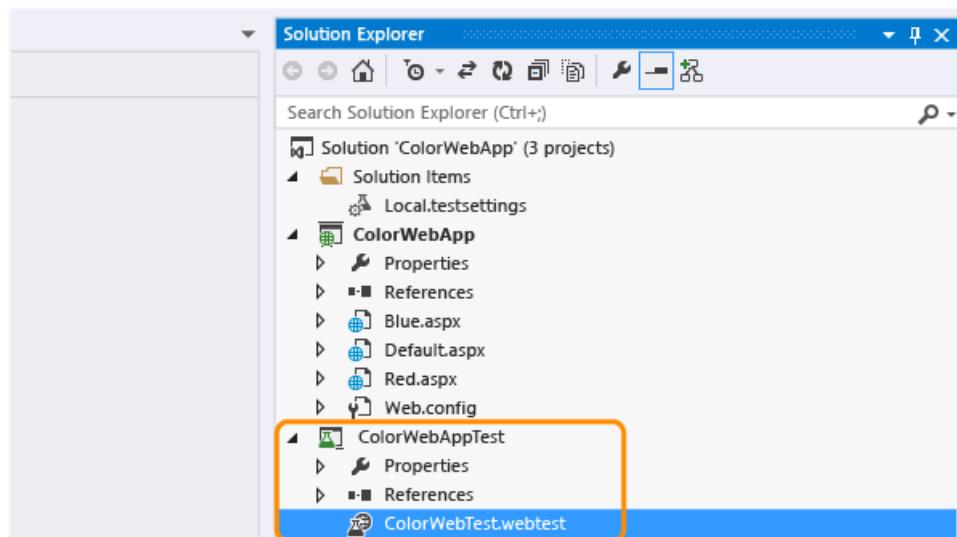
Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.



We're going to use a sample ASP.NET app. It has three `.aspx` pages – the default page, a Red page, and a Blue page. The default page has a radio control to choose either red or blue and a submit button. The other two `.aspx` pages are very simple. One has a label named Red and the other has a label named Blue. When you choose submit on the default page, we display one of the other pages. You can download the [ColorWebApp](#) sample, or just follow along with your own web app.

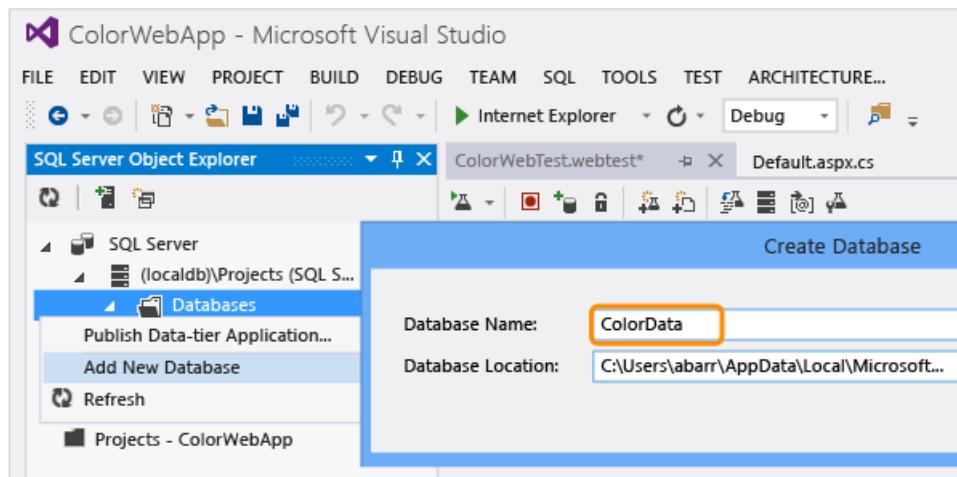


Your solution should also include a web performance test that browses through the pages of the web application.

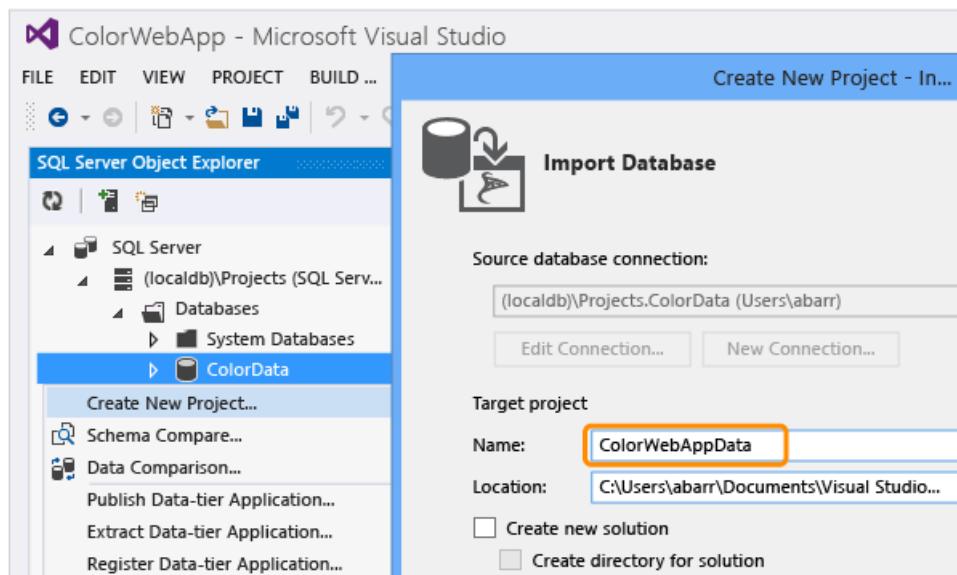


Create a SQL database

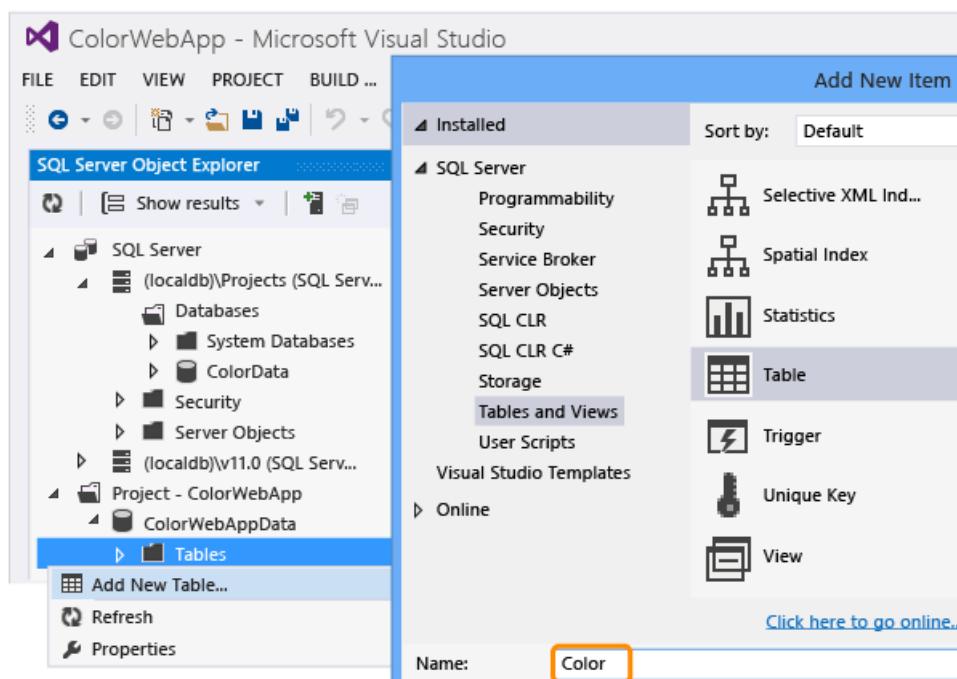
1. If you don't have Visual Studio Enterprise, you can download it from the [Visual Studio Downloads](#) page.
2. Create a SQL database.



3. Create a database project.



4. Add a table to the database project.



5. Add fields to the table.

...Object Explorer

...L Server
(localdb)\Projects (SQL Server 11.0.3000 - R
...Objects - ColorWebApp
ColorWebAppData
Tables
Views

Color_1.sql [Design]*

Script File: Color_1.sql*

Name	Data Type	Allow Nulls	Def...
ColorID	int	<input type="checkbox"/>	
ColorName	text	<input type="checkbox"/>	

6. Publish the database project.

Import
Snapshot Project
Schema Compare...
Build
Rebuild
Clean
Publish
Run Code Analysis
Scope to This
New Solution Explorer View
Project Dependencies...
Project Build Order...

Solution Explorer

Search Solution Explorer (Ctrl+;)

- Solution 'ColorWebApp' (3 projects)
 - Solution Items
 - Local.testsettings
 - ColorWebApp**
 - Properties
 - References
 - Blue.aspx
 - Default.aspx
 - Red.aspx
 - Web.config
 - ColorWebAppData**
 - Properties
 - References
 - Colors.sql
 - ColorWebAppData.refactorlog

7. Add data to the fields.

SQL Server Object Explorer

SQL Server
(localdb)\Projects (SQL Server 11.0.3000...
Databases
System Databases
ColorData
ColorWebAppData
Tables
System Tables
dbo.Color

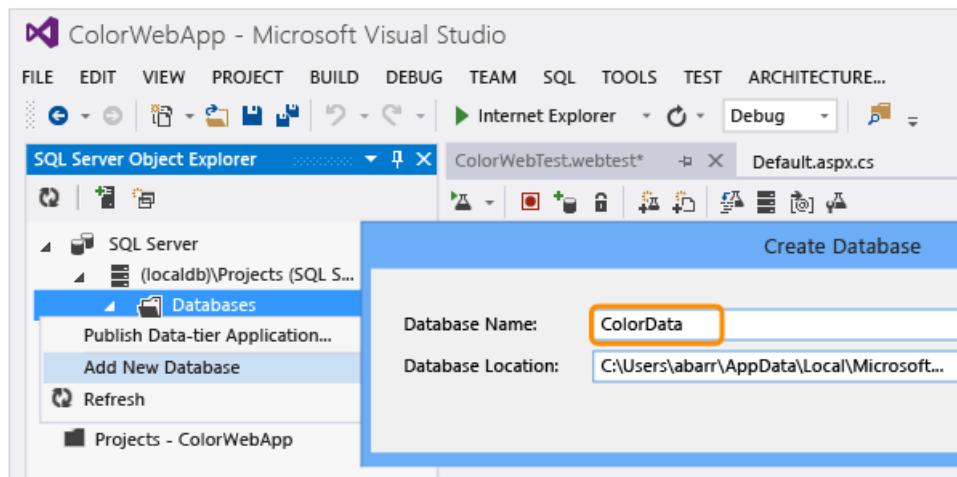
dbo.Color [Data]

Colors.sql [Design]

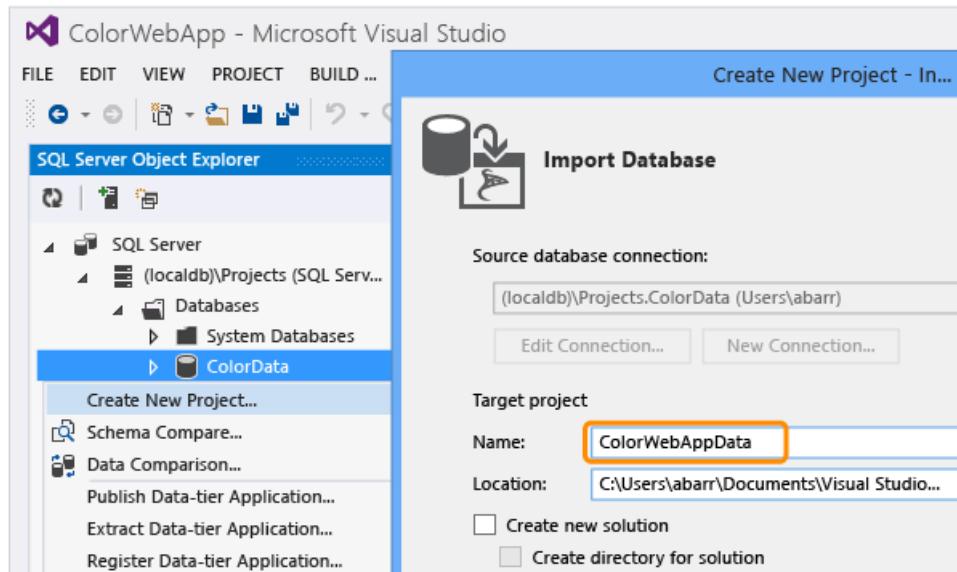
ColorID	ColorName
0	Red
1	Blue
NULL	NULL

Data Comparison...
Script As
View Code
View Designer
View Permissions
View Data

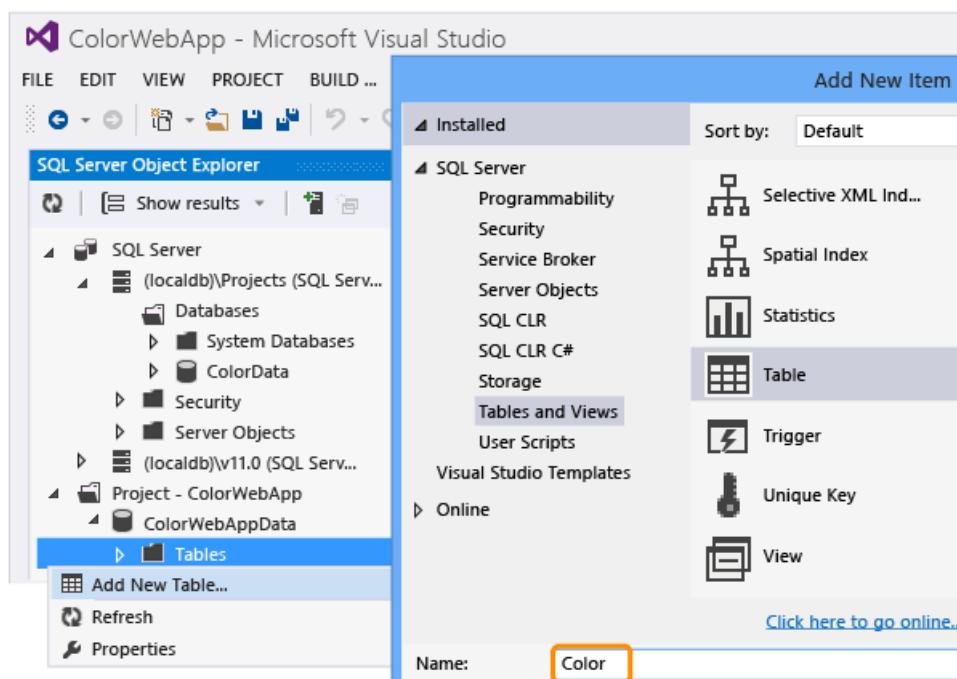
- If you don't have Visual Studio Enterprise, you can download it from the [Visual Studio Downloads](#) page.
- Create a SQL database.



3. Create a database project.



4. Add a table to the database project.



5. Add fields to the table.

...Object Explorer

...L Server
(localdb)\Projects (SQL Server 11.0.3000 - R
...Objects - ColorWebApp
ColorWebAppData
Tables
Views

Color_1.sql [Design]

Script File: Color_1.sql*

Name	Data Type	Allow Nulls	Def...
ColorID	int	<input type="checkbox"/>	
ColorName	text	<input type="checkbox"/>	

6. Publish the database project.

Import
Snapshot Project
Schema Compare...
Build
Rebuild
Clean
Publish
Run Code Analysis
Scope to This
New Solution Explorer View
Project Dependencies...
Project Build Order...

Solution Explorer

Search Solution Explorer (Ctrl+;)

Solution 'ColorWebApp' (3 projects)
Solution Items
Local.testsettings
ColorWebApp
Properties
References
Blue.aspx
Default.aspx
Red.aspx
Web.config
ColorWebAppData
Properties
References
Colors.sql
ColorWebAppData.refactorlog

7. Add data to the fields.

SQL Server Object Explorer

SQL Server
(localdb)\Projects (SQL Server 11.0.3000...
Databases
System Databases
ColorData
ColorWebAppData
Tables
System Tables
dbo.Color

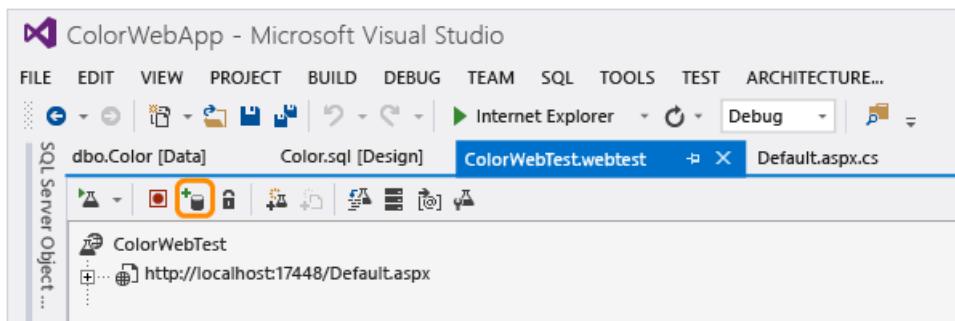
dbo.Color [Data]

Colors.sql [Design]

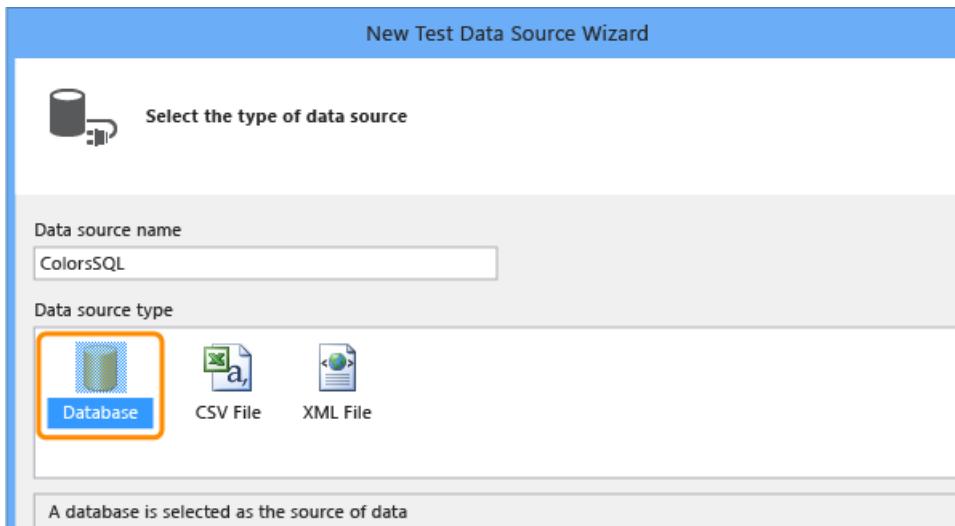
ColorID	ColorName
0	Red
1	Blue
*	NULL

Add the data source

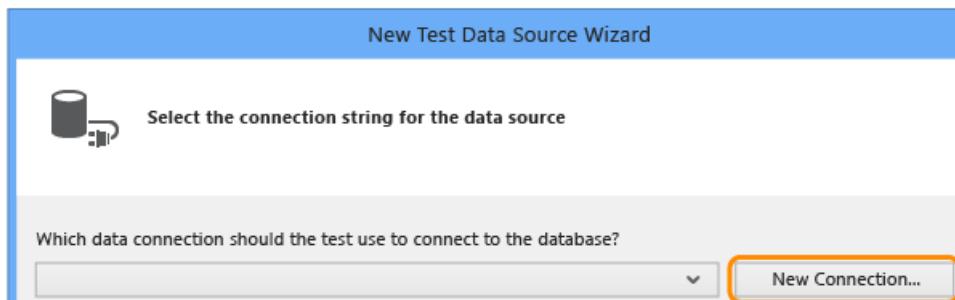
1. Add a data source.



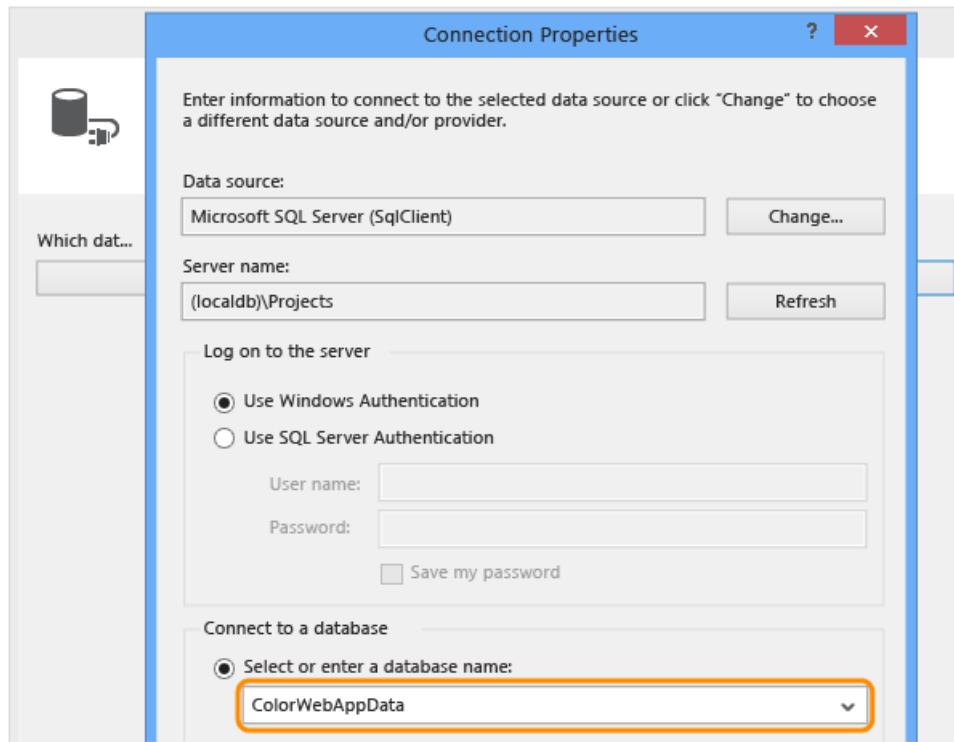
2. Choose the type of data source and name it.



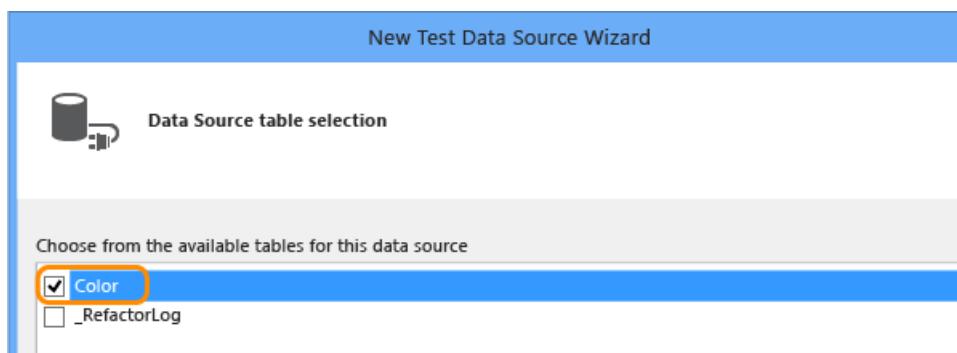
3. Create a connection.



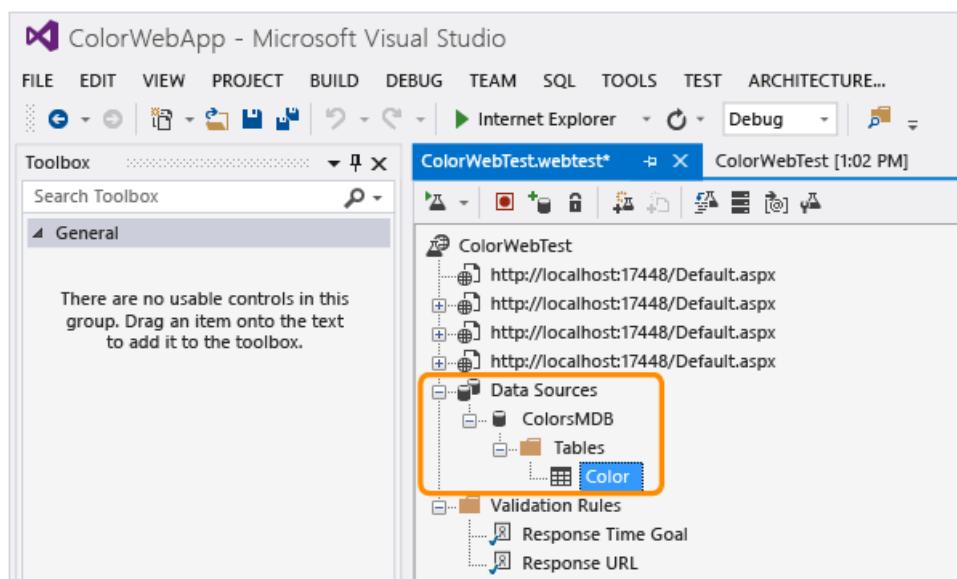
Enter the connection details.



4. Select the table that you want to use for your test.



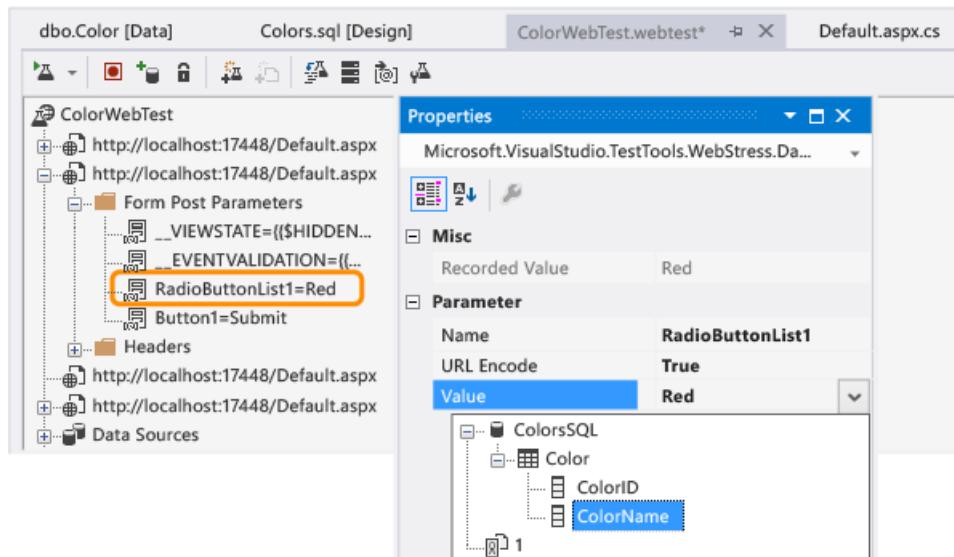
The table is bound to the test.



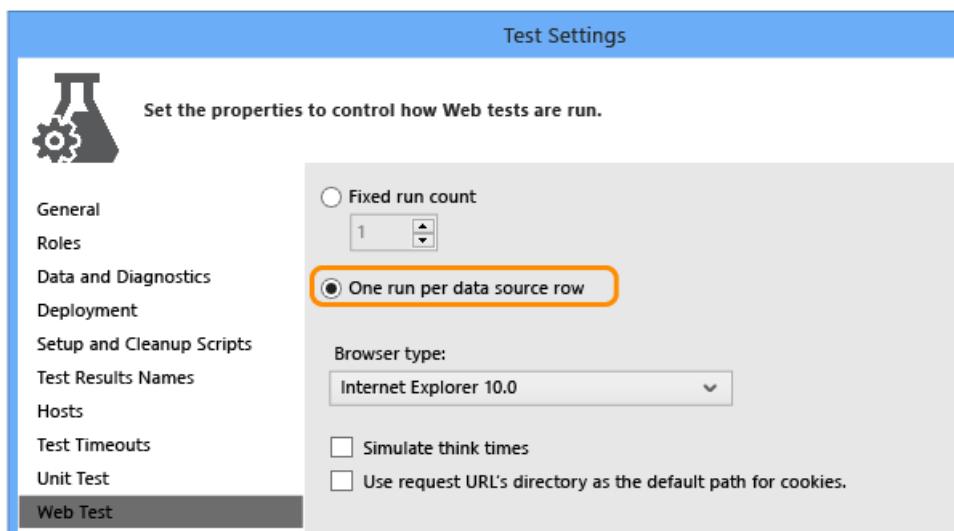
5. Save the test.

Bind the data

1. Bind the **ColorName** field.



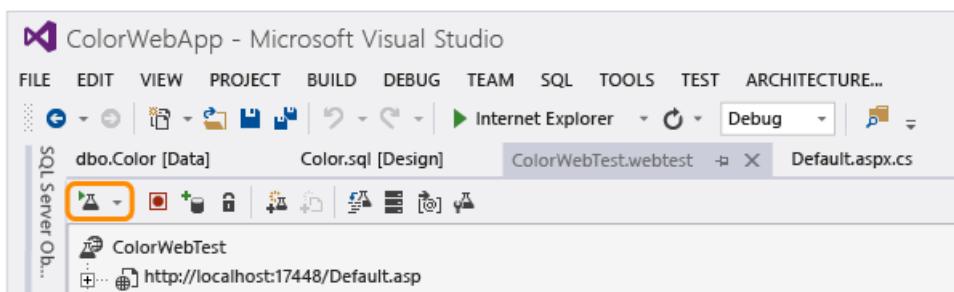
2. Open the *Local.testsettings* file in **Solution Explorer** and select the **One run per data source row** option.



3. Save the web performance test.

Run the test with the data

1. Run the test.

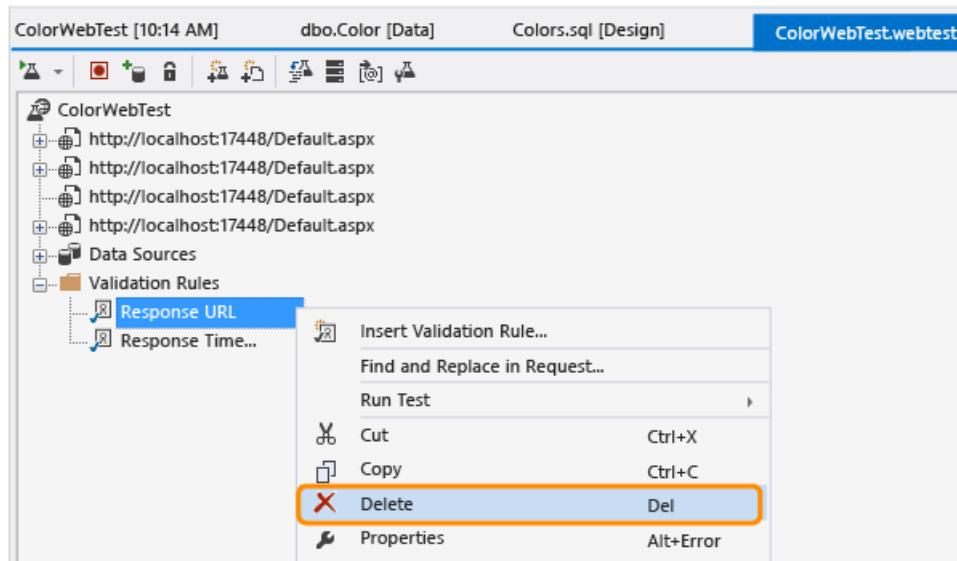


The two runs are displayed for each data row. Run 1 sends a request for the page *Red.aspx*, and Run 2 sends a request for the page *Blue.aspx*.

ColorWebTest [9:51 AM]				
		dbo.Color [Data]	Colors.sql [Design]	ColorWebTest.webtest
Failed Click here to run again Internet Explorer 10.0 LAN Edit run settings				
Request	Status	Total Time	Request Time	
Run 1				
✓ ▷ http://localhost:17448/Default.aspx	200 OK	0.131 sec	0.117 sec	
✓ ▲ http://localhost:17448/Default.aspx	302 Found	0.412 sec	0.355 sec	
✓ ▷ http://localhost:17448/Red.aspx	200 OK	-	0.046 sec	
✓ ▷ http://localhost:17448/Default.aspx	200 OK	0.183 sec	0.171 sec	
✓ ▲ http://localhost:17448/Default.aspx	302 Found	0.341 sec	0.286 sec	
✓ ▷ http://localhost:17448/Blue.aspx	200 OK	-	0.046 sec	
Run 2				
✓ ▷ http://localhost:17448/Default.aspx	200 OK	0.179 sec	0.166 sec	
✓ ▲ http://localhost:17448/Default.aspx	302 Found	0.306 sec	0.255 sec	
✗ ▷ http://localhost:17448/Blue.aspx	200 OK	-	0.051 sec	
✓ ▷ http://localhost:17448/Default.aspx	200 OK	0.115 sec	0.104 sec	
✓ ▲ http://localhost:17448/Default.aspx	302 Found	0.265 sec	0.212 sec	
✓ ▷ http://localhost:17448/Blue.aspx	200 OK	-	0.046 sec	

When you bind to a data source, you could violate the default response URL rule. In this case, the error in Run 2 is caused by the rule which expects the *Red.aspx* page from the original test recording, but the data binding now directs it to the *Blue.aspx* page.

2. Correct the validation error by deleting the **Response URL** validation rule and running the test again.



The web performance test now passes using data binding.

ColorWebTest [10:30 AM]		X	ColorWebTest [10:14 AM]	dbo.Color [Data]	Col...
Passed Click here to run again Internet Explorer 10.0 LAN Edit run settings					
Request		Status	Tot...		
✓ ▲ Run 1					
✓ ▷ http://localhost:17448/Default.aspx		200 OK	0.0		
✓ ▲ http://localhost:17448/Default.aspx		302 Found	0.3		
✓ ▷ http://localhost:17448/Red.aspx		200 OK			
✓ ▷ http://localhost:17448/Default.aspx		200 OK	0.1		
✓ ▲ http://localhost:17448/Default.aspx		302 Found	0.3		
✓ ▷ http://localhost:17448/Blue.aspx		200 OK			
✓ ▲ Run 2					
✓ ▷ http://localhost:17448/Default.aspx		200 OK	0.0		
✓ ▲ http://localhost:17448/Default.aspx		302 Found	0.3		
✓ ▷ http://localhost:17448/Blue.aspx		200 OK			
✓ ▷ http://localhost:17448/Default.aspx		200 OK	0.1		
✓ ▲ http://localhost:17448/Default.aspx		302 Found	0.3		
✓ ▷ http://localhost:17448/Blue.aspx		200 OK			

Q & A

Q: What databases can I use as a data source?

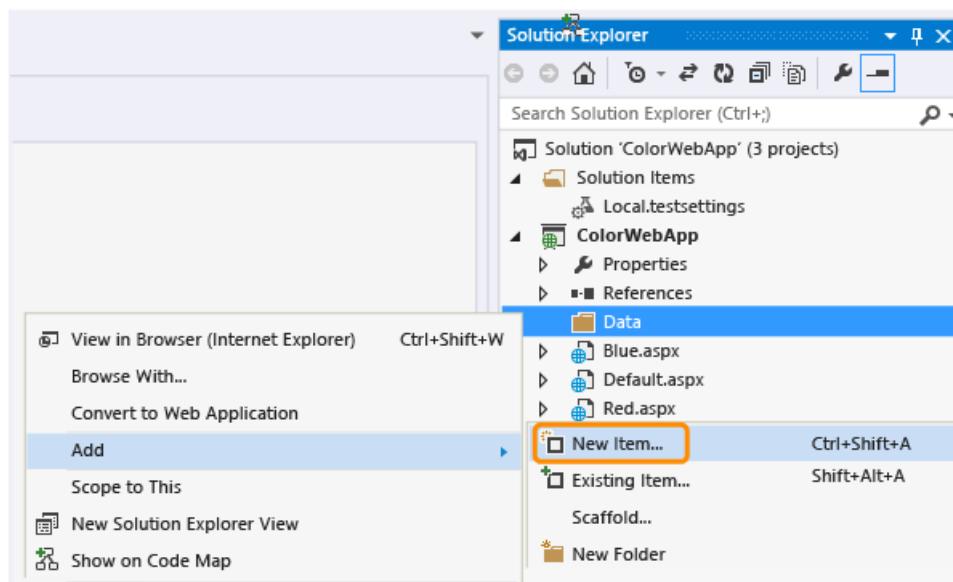
A: You can use the following:

- Microsoft SQL Azure.
- Any version of Microsoft SQL Server 2005 or later.
- Microsoft SQL Server database file (including SQL Express).
- Microsoft ODBC.
- Microsoft Access file using the .NET Framework provider for OLE DB.
- Oracle 7.3, 8i, 9i, or 10g.

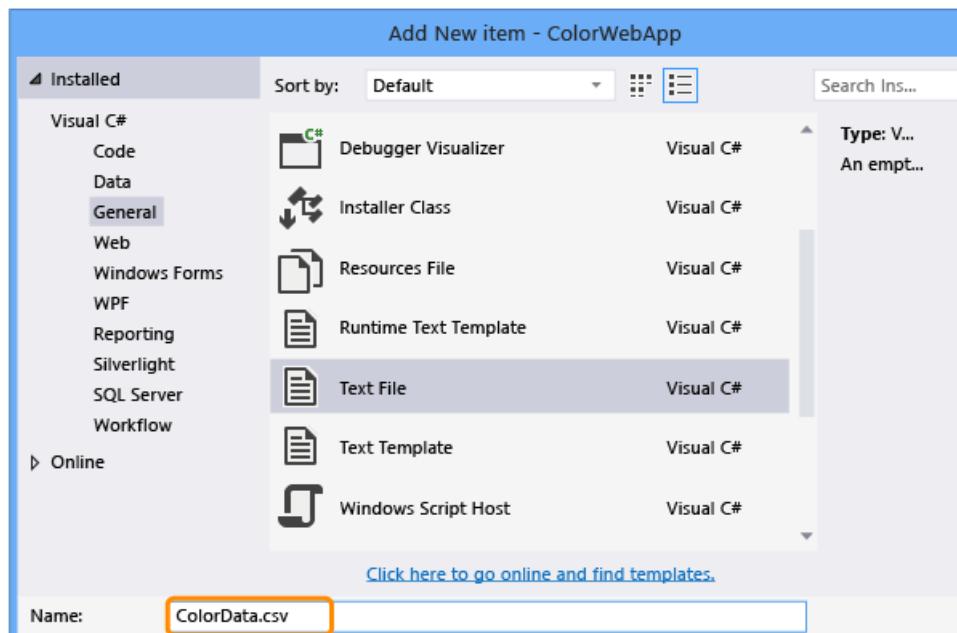
Q: How do I use a comma separated value (CSV) text file as a data source?

A: Here's how:

1. Create a folder to organize your projects database artifacts and add an item.



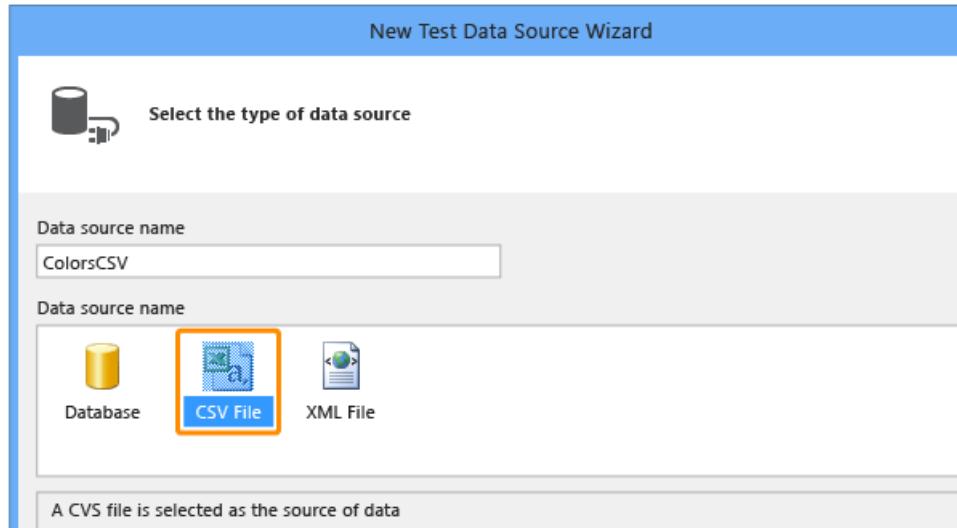
2. Create a text file.



3. Edit the text file and add the following:

```
ColorId, ColorName
0,Red
1,Blue
```

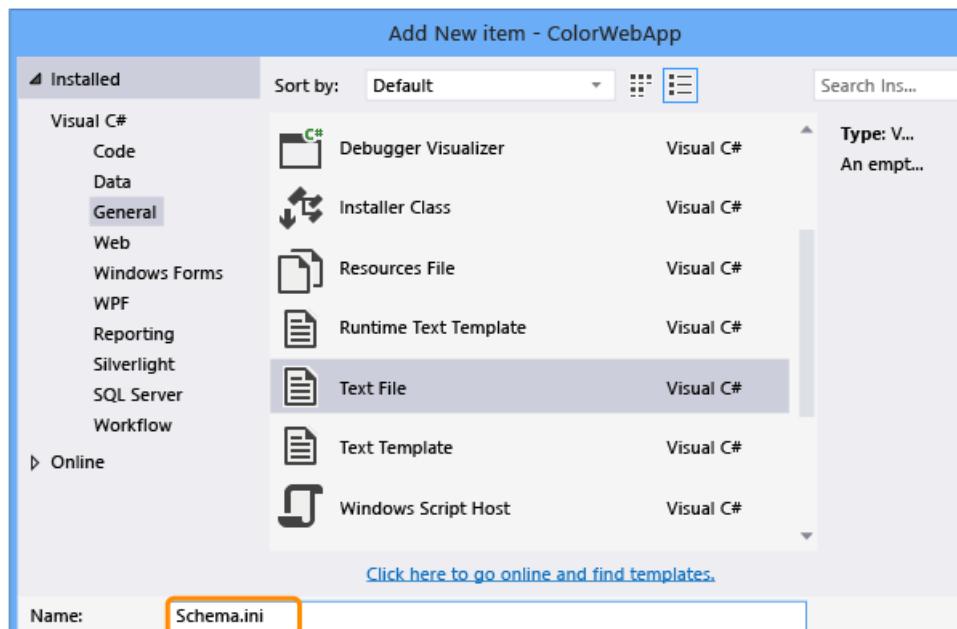
4. Use the steps in [Add the data source](#), but choose CSV file as your data source.



Q: What if my existing CSV file does not contain column headers?

A: If you can't add column headers, you can use a schema description file to treat the CSV file as a database.

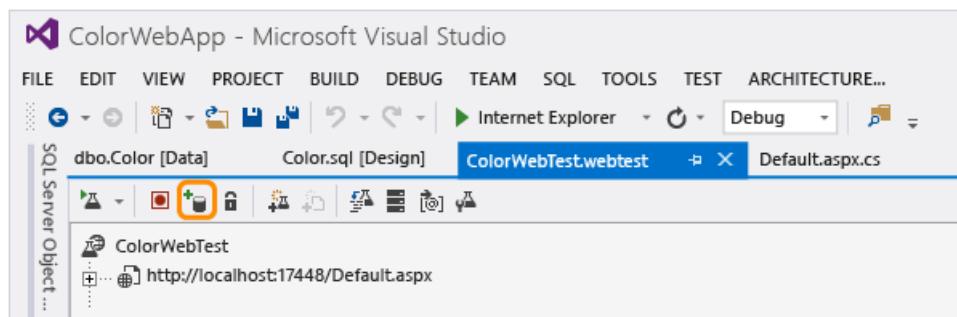
1. Add a new text file named *schema.ini*.



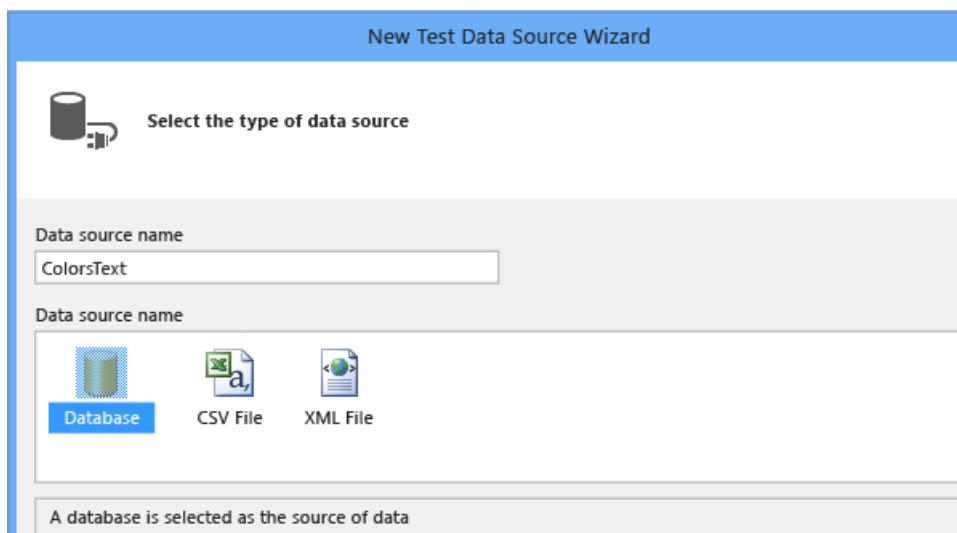
2. Edit the *schema.ini* file to add the information that describes the structure of your data. For example, a schema file describing the CSV file might look like this:

```
[testdata.csv]
ColNameHeader=False
```

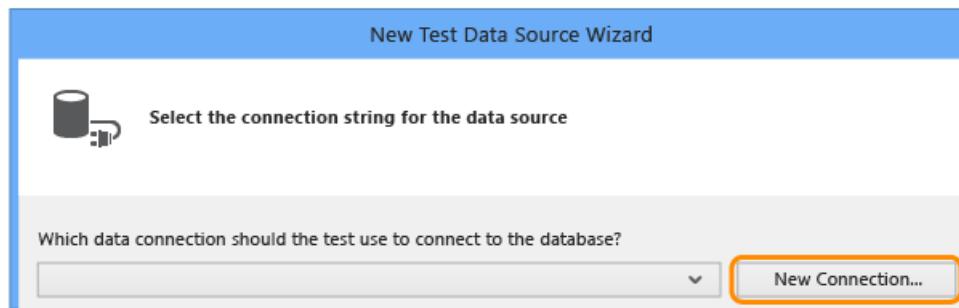
3. Add a data source to the test.



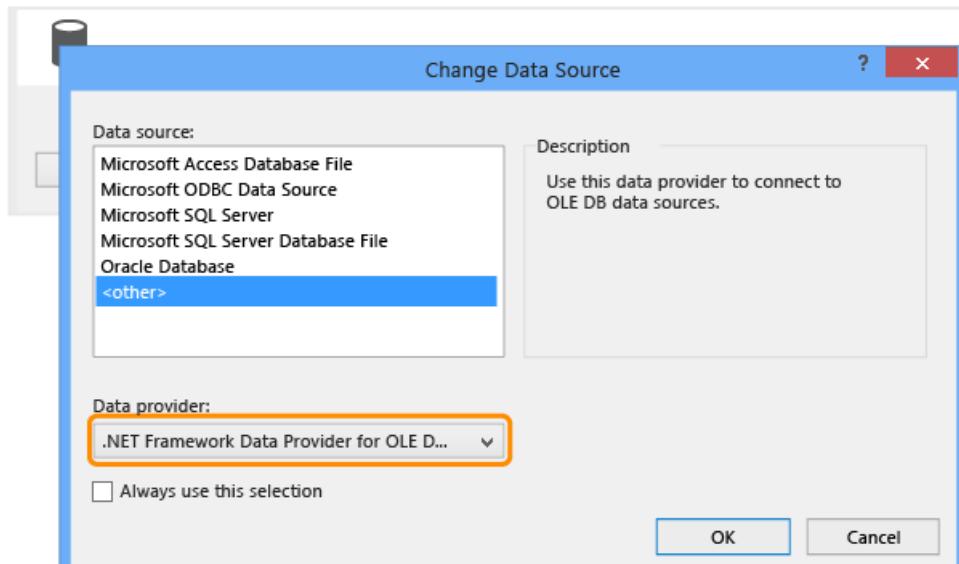
4. If you're using a *schema.ini* file, choose **Database** (not CSV file) as the data source and name it.



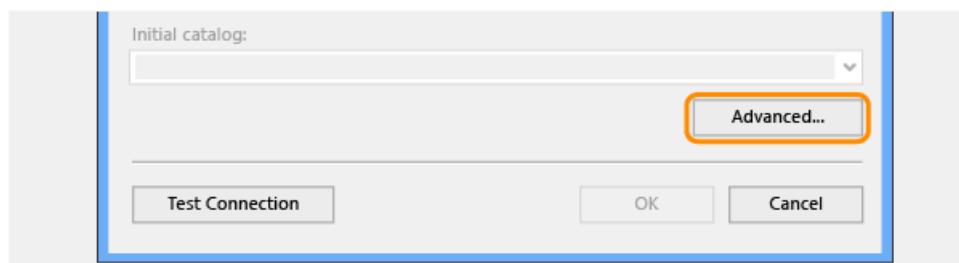
5. Create a new connection.



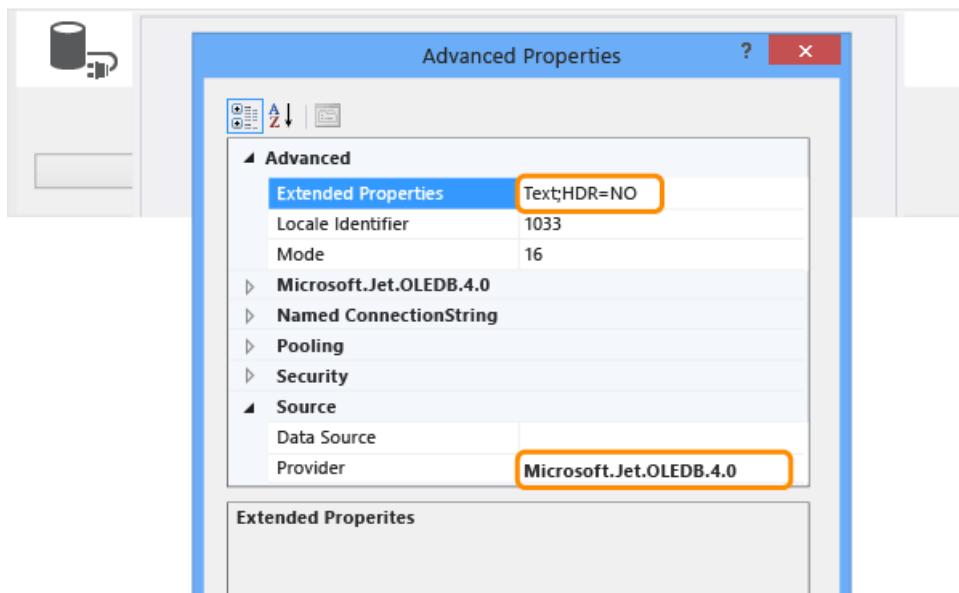
6. Select the .NET Framework Data Provider for OLE DB.



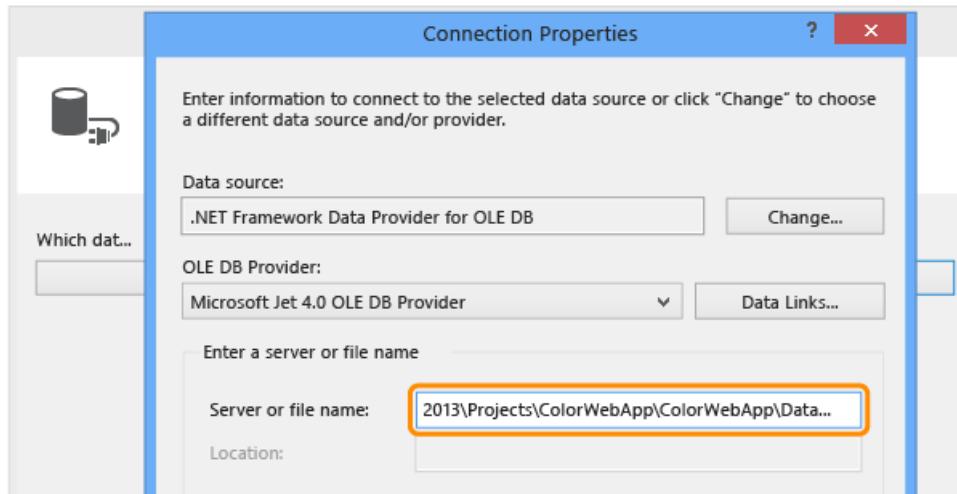
7. Choose **Advanced**.



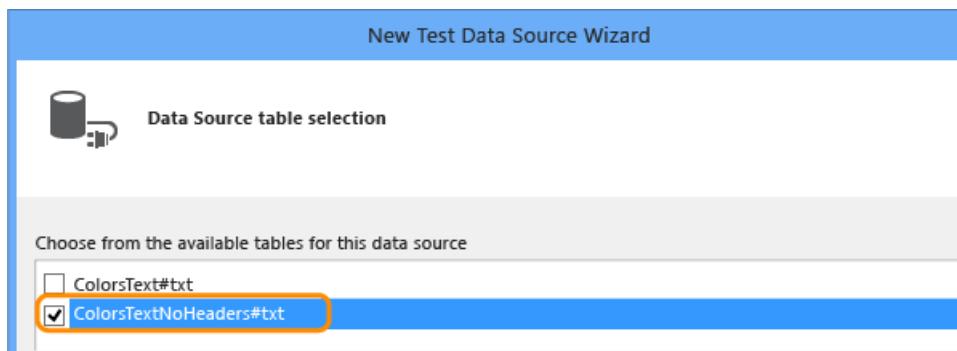
8. For the Provider property, select Microsoft.Jet.OLEDB.4.0, and then set **Extended Properties** to Text;HDR=NO.



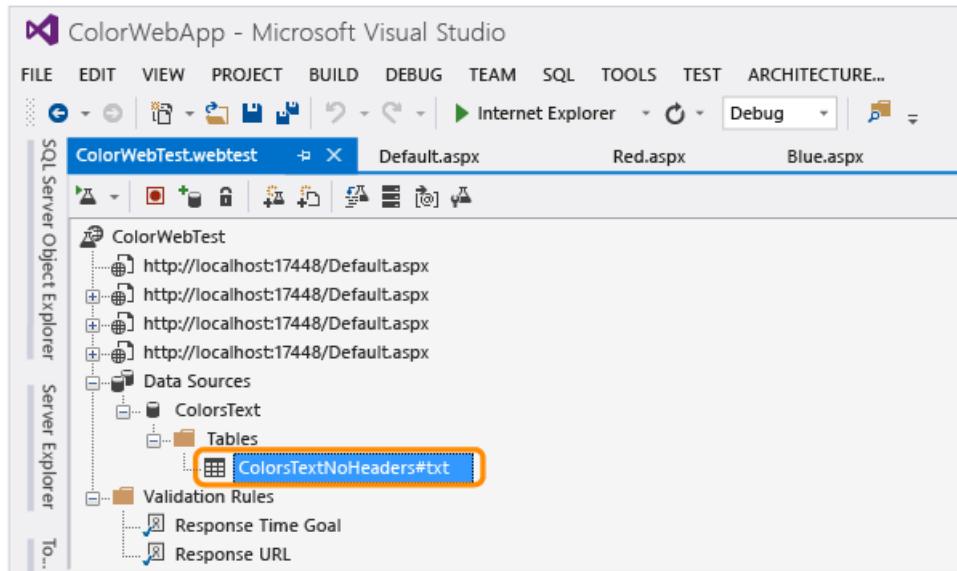
9. Type the name of the folder that contains the schema file and test your connection.



10. Select the CSV file that you want to use.



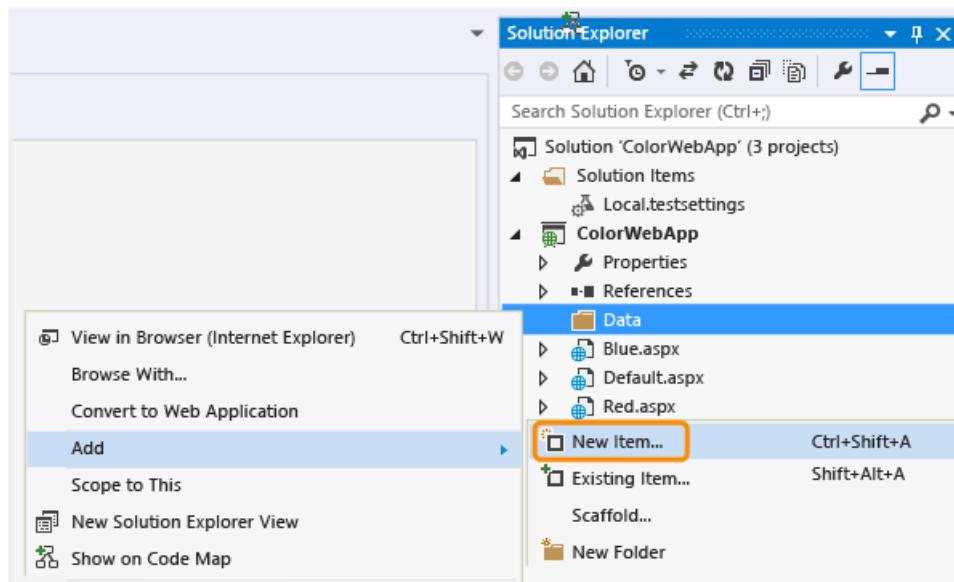
After you finish, the CSV file appears as a table.



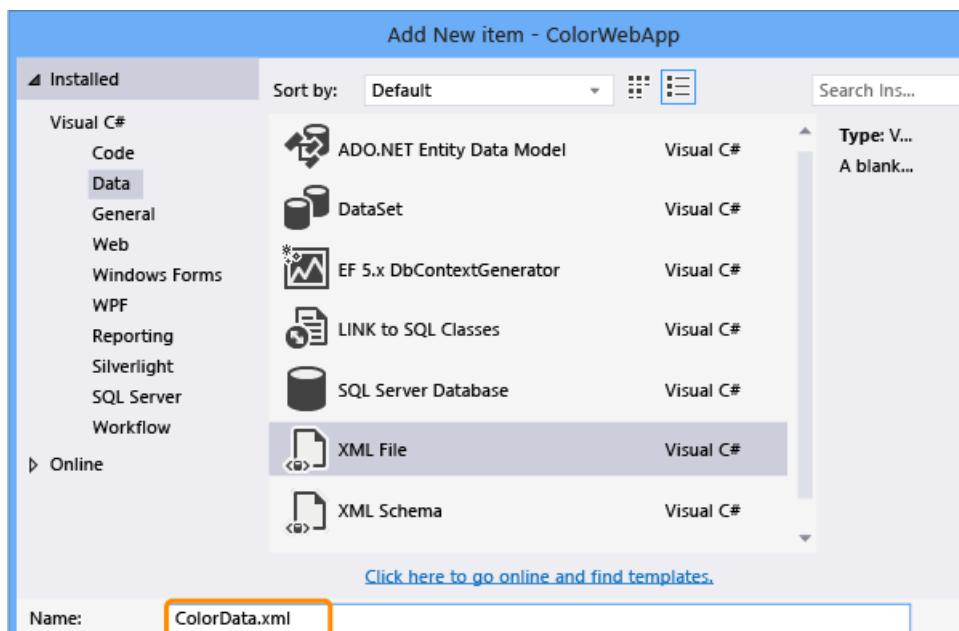
Q: How do I use an XML file as a data source?

A: Yes.

1. Create a folder to organize your projects database artifacts and add an item.



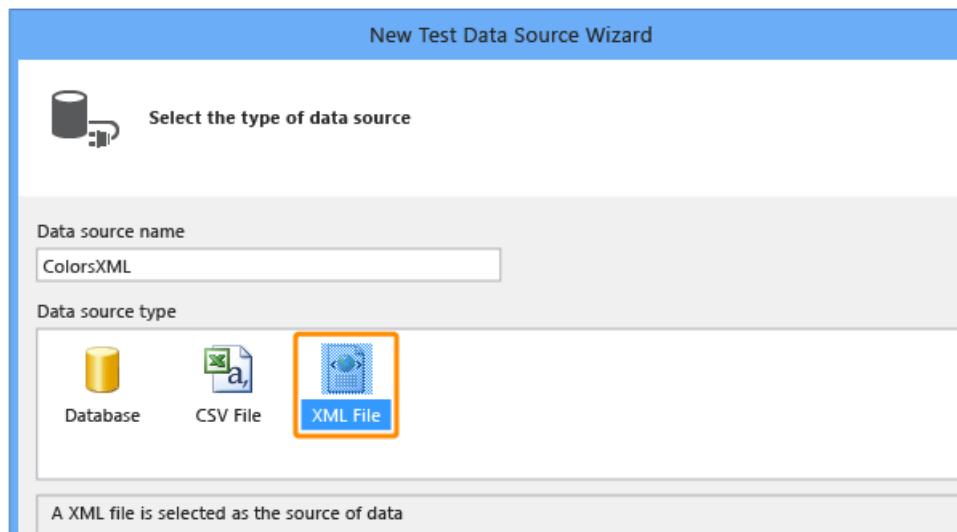
2. Create an XML file.



3. Edit the XML file and add your data:

```
<?xml version="1.0" encoding="utf-8" ?>
<ColorData>
    <Color>
        <ColorId>0</ColorId>
        <ColorName>Red</ColorName>
    </Color>
    <Color>
        <ColorId>1</ColorId>
        <ColorName>Blue</ColorName>
    </Color>
</ColorData>
```

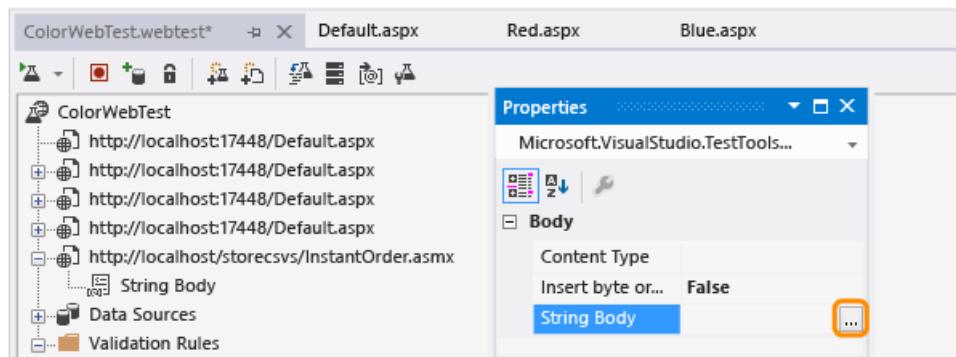
4. Use the steps in [Add the data source](#), but choose XML file as your data source.



Q: Can I add data binding to a web service request that uses SOAP?

A: Yes, you must change the SOAP XML manually.

1. Choose the web service request in the request tree and in the Properties window, choose the ellipsis (...) in the String Body property.



2. Replace values in the SOAP body with data-bound values by using the following syntax:

```
 {{DataSourceName.TableName.ColumnName}}
```

For example, if you have the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <CheckStatus xmlns="http://tempuri.org/">
            <userName>string</userName> <password>string</password> <orderID>int</orderID>
        </CheckStatus>
    </soap:Body>
</soap:Envelope>
```

You can change it to this:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <CheckStatus xmlns="http://tempuri.org/">
            <userName>{{DataSourceName.Users.Name}}</userName> <password>
                {{DataSourceName.Users.Password}}</password> <orderID>{{DataSourceName.Orders.OrderID}}</orderID>
        </CheckStatus>
    </soap:Body>
</soap:Envelope>
```

3. Save the test.

Fix non-detectable dynamic parameters in a web performance test

1/1/2020 • 8 minutes to read • [Edit Online](#)

Some websites use dynamic parameters to process some of their web requests. A dynamic parameter is a parameter whose value is regenerated every time that a user runs the application. An example of a dynamic parameter is a session ID. The session ID usually changes every 5 to 30 minutes. The web performance test recorder and playback engine automatically handles the most common types of dynamic parameters:

- Dynamic parameter values that are set in a cookie value. The web performance test engine automatically handles these during playback.
- Dynamic parameter values that are set in hidden fields on HTML pages, such as ASP.NET view state. These are automatically handled by the recorder, which adds hidden field extraction rules to the test.
- Dynamic parameter values that are set as query string or form post parameters. These are handled through dynamic parameter detection after you record a web performance test.

Some types of dynamic parameters are not detected. An undetected dynamic parameter will cause your web performance test to fail when you run it because the dynamic value will be different every time that the test is run. To handle these parameters correctly, you can add extraction rules to dynamic parameters in your web performance tests manually.

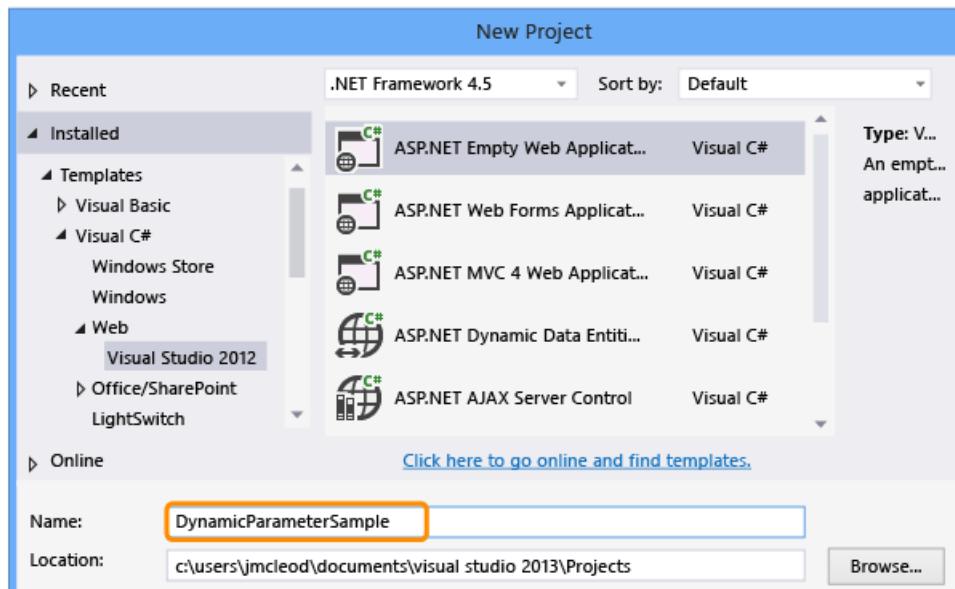
NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

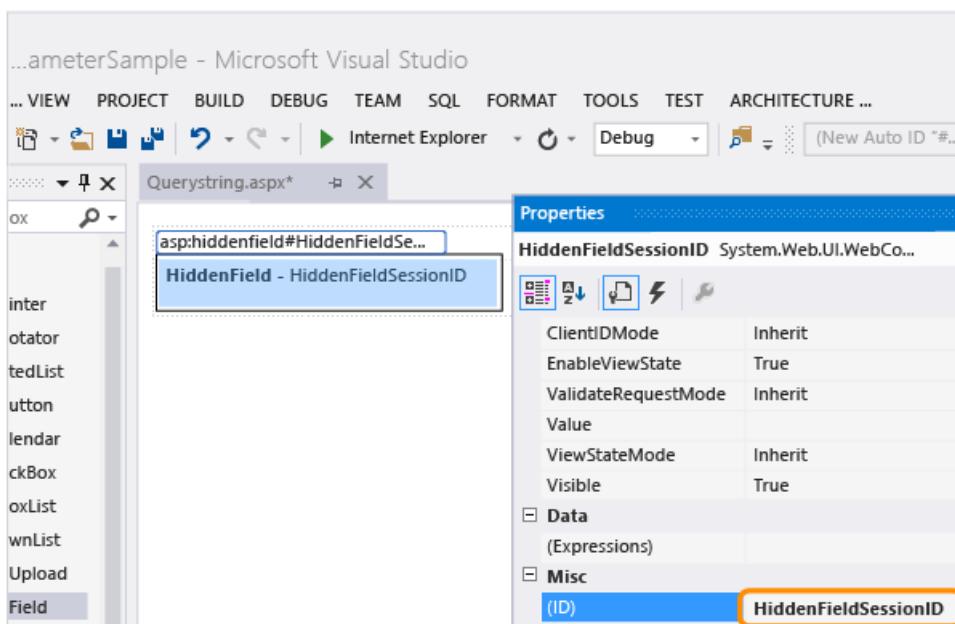
Create and run a web app with dynamic parameters

To demonstrate both a detectable and a non-detectable dynamic parameter, we'll create a simple ASP.NET web application that has three web forms with a few controls and some custom code. We'll then learn how to isolate the dynamic parameters and how to handle them.

1. Create a new ASP.NET project named **DynamicParameterSample**.



2. Add a web form named *Querystring.aspx*.
3. In design view, drag a HiddenField onto the page and in then in change the value for the (ID) property to *HiddenFieldSessionID*.



4. Change to the source view for the Querystring page, and add the following highlighted ASP.NET and JavaScript code used to generate the mock session ID dynamic parameters:

```

<head runat="server">
<title>JavaScript dynamic property correlation sample</title><script type="text/javascript"
language="javascript">      <!--          function jScriptQueryString()          {          var Hidden =
document.getElementById("HiddenFieldSessionID");          var sessionId = Hidden.value;
window.location = 'JScriptQuery.aspx?CustomQueryString=jScriptQueryString__' + sessionId;          }
//--></script>
</head>
<body>
<form id="form1" runat="server">
<div>
<a name="QuerystringHyperlink" href="ASPQuery.aspx?CustomQueryString=ASPQueryString__<%
Session.SessionID %>">Dynamic querystring generated by ASP.net</a>           <br/>           <br/>
<a href="javascript:jScriptQueryString()">Dynamic querystring generated by javascript </a>
</div>
<asp:HiddenField ID="HiddenFieldSessionID" runat="server" />
</form>
</body>
</html>

```

5. Open the *Querystring.aspx.cs* file and add the following highlighted code to the **Page_Load** method:

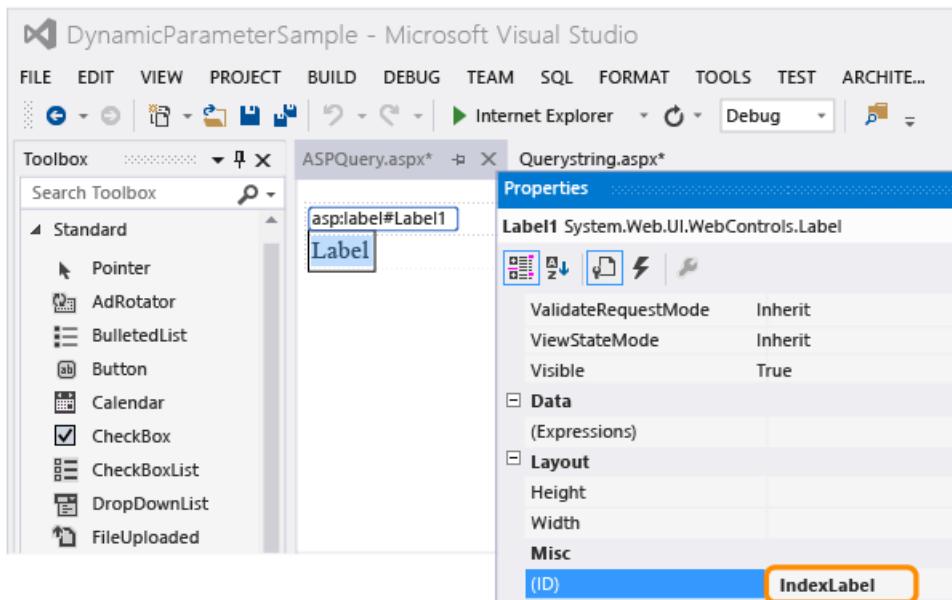
```

public partial class Querystring : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Session.Add("Key", "Value");HiddenFieldSessionID.Value = Session.SessionID;
    }
}

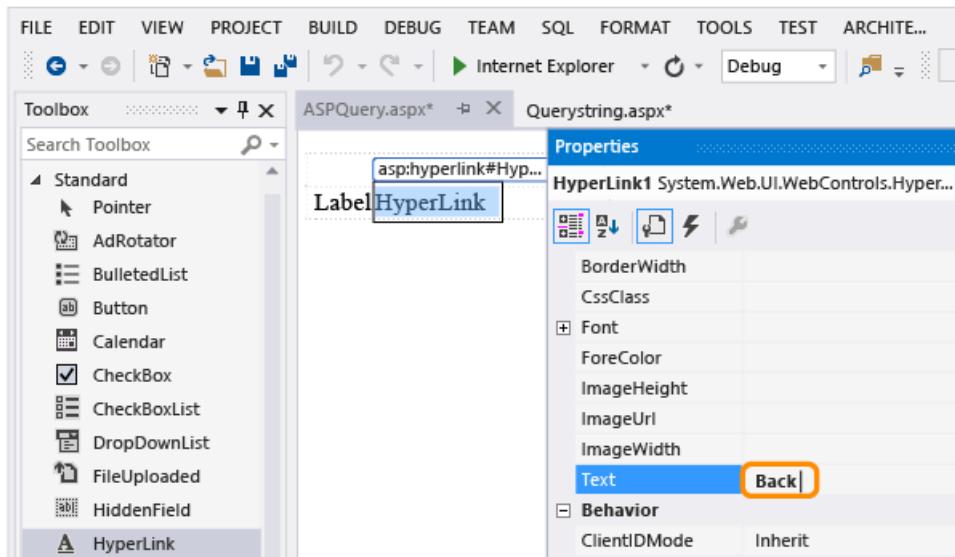
```

6. Add a second web form named *ASPQuery.aspx*.

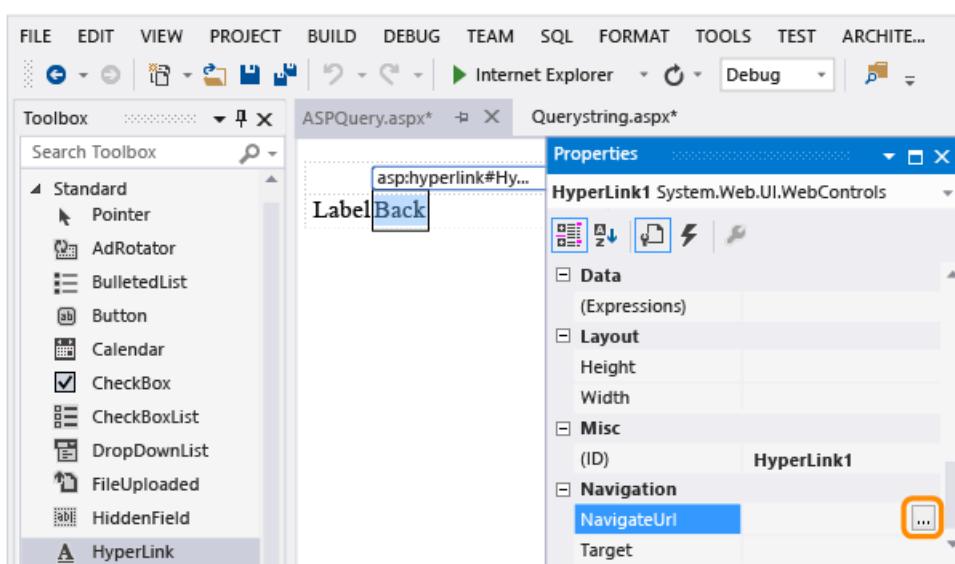
7. In design view, drag a **Label** onto the page and change the value for its **(ID)** property to **IndexLabel**.



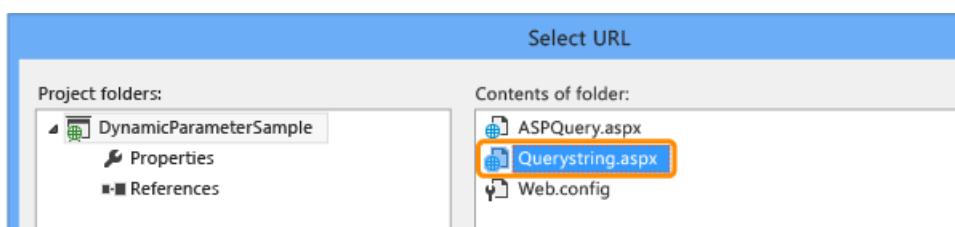
8. Drag a **HyperLink** onto the page and change the vale for its **Text** property to **Back**.



9. Choose (...) for the **NavigationURL** property.



Select *Querystring.aspx*.



10. Open the *ASPQuery.aspx.cs* file, and add the following highlighted code to the *Page_Load* method:

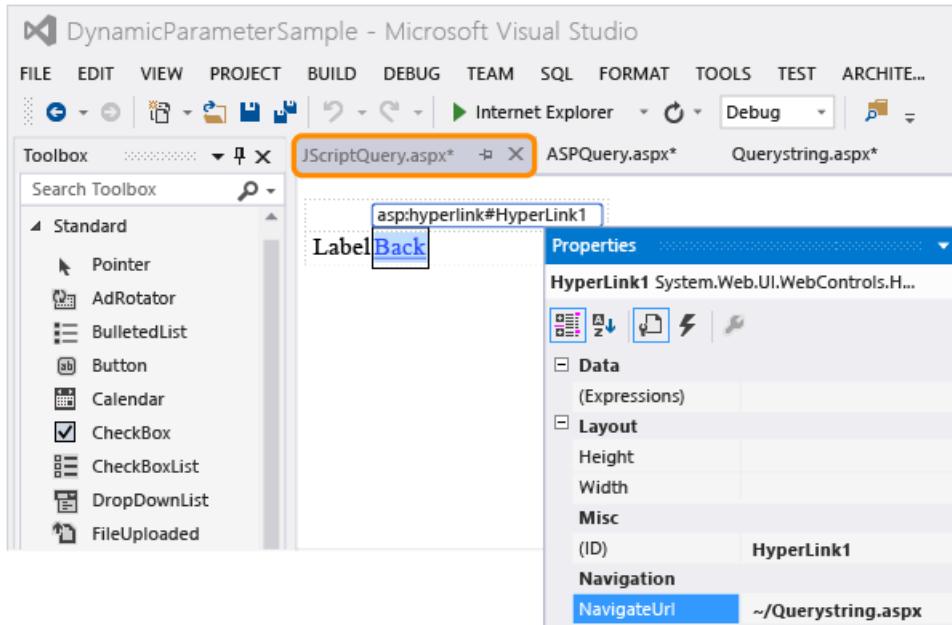
```

protected void Page_Load(object sender, EventArgs e)
{
    int index; string qstring; string dateportion; string
    sessionidportion; qstring = Request.QueryString["CustomQueryString"]; index =
    qstring.IndexOf("__"); dateportion = qstring.Substring(0, index); index += 3;
    sessionidportion = qstring.Substring(index, qstring.Length - index); if (sessionidportion !=
    Session.SessionID) { Response.StatusCode = 401; IndexLabel.Text =
    "Failure! Invalid querystring parameter found."; } else { IndexLabel.Text =
    "Success. Dynamic querystring parameter was found."; }
    IndexLabel.Text += "<br>\r\n";
}

```

11. Add a third web form named *JScriptQuery.aspx*.

Just as we did for the second page, drag a **Label** onto the form, setting its **(ID)** property to **IndexLabel** and drag a **Hyperlink** onto the form, setting its **Text** property to **Back**, and its **NavigationURL** property to **Querystring.aspx**.

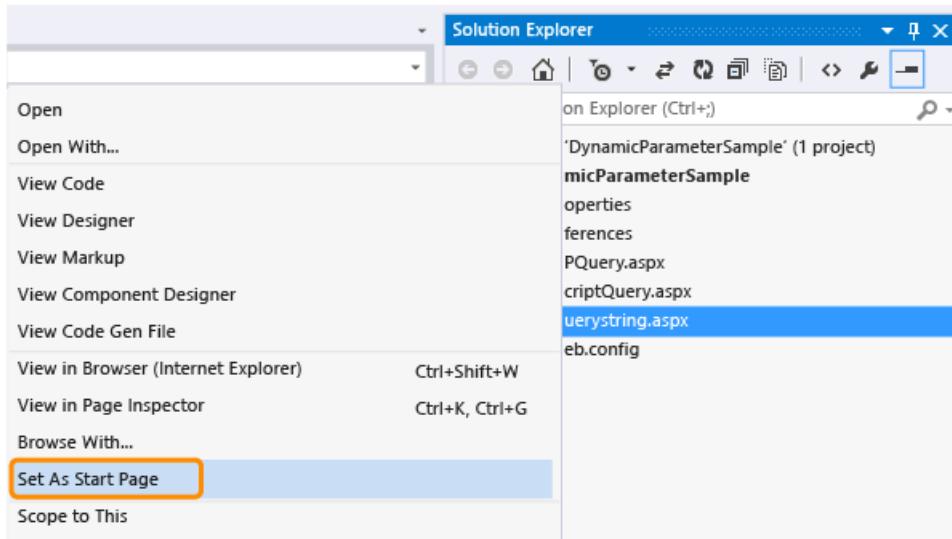


12. Open the *JScriptQuery.aspx.cs* file, and add the following highlighted code to the *Page_Load* method:

```
protected void Page_Load(object sender, EventArgs e)
{
    int index; string qstring; string dateportion; string
    sessionidportion; qstring = Request.QueryString["CustomQueryString"]; index =
    qstring.IndexOf("__"); dateportion = qstring.Substring(0, index); if (sessionidportion != 
    Session.SessionID) { Response.StatusCode = 401; IndexLabel.Text =
    "Failure! Invalid querystring parameter found.";} else { 
    IndexLabel.Text = "Success. Dynamic querystring parameter was found.";
    IndexLabel.Text += "  
\r\n";
}
```

13. Save the project.

14. In **Solution Explorer**, set the *Querystring.aspx* as the start page.



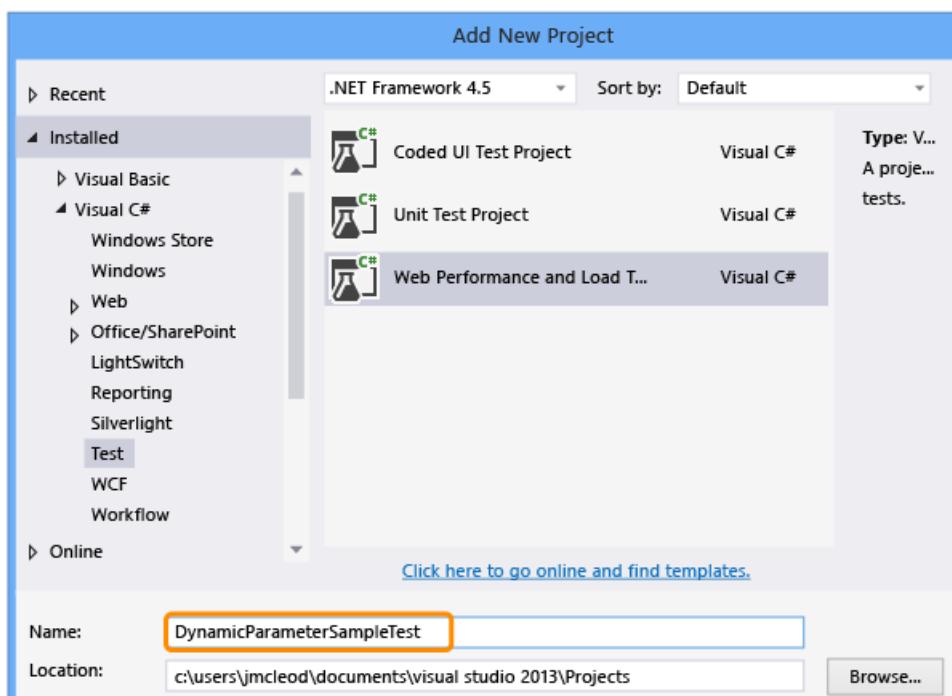
15. Press **Ctrl+F5** to run the web application in the browser. Copy the URL. You will need it when you record your test.

16. Try both links. They should each display the message "Success. Dynamic querystring parameter found."

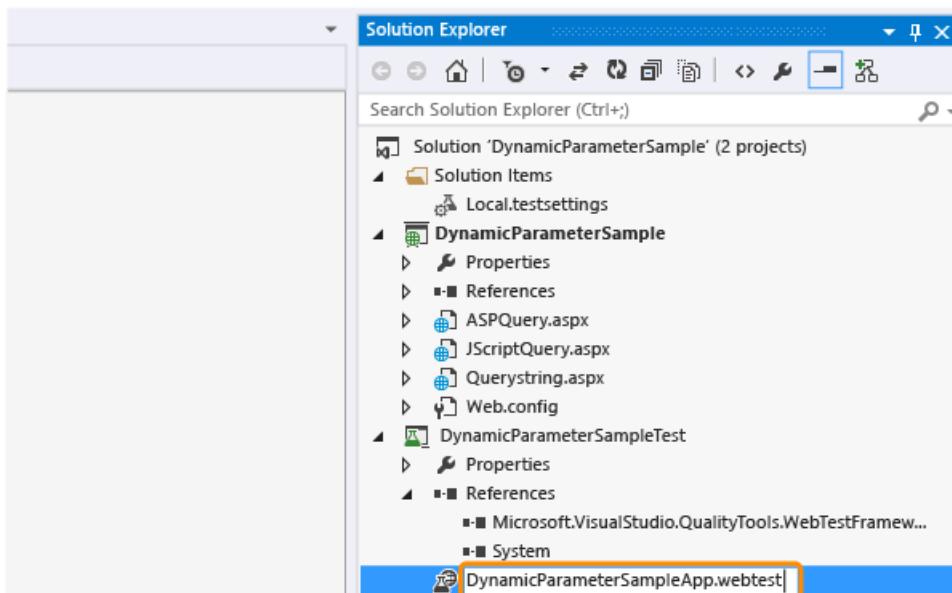


Create a web performance test

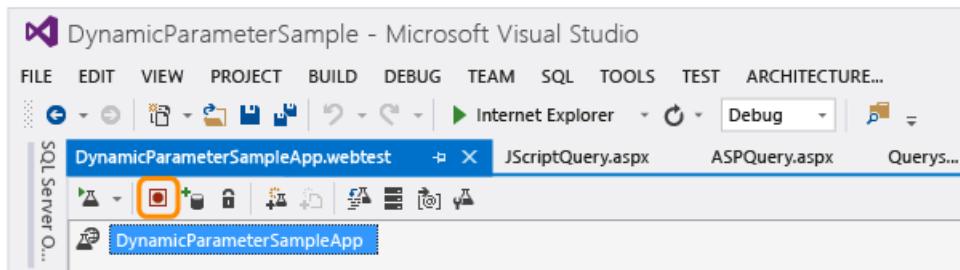
1. Add a web performance and load test project to your solution.



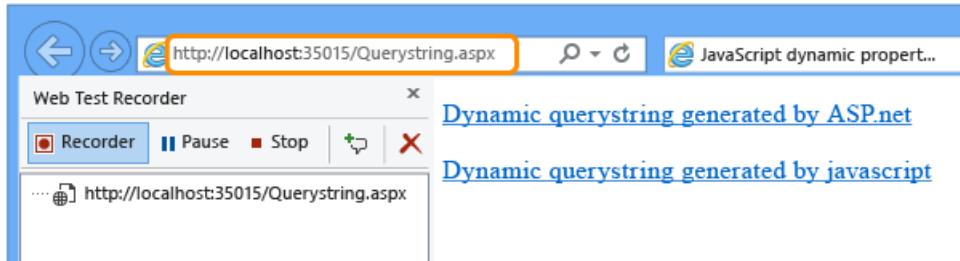
2. Rename WebTest1.webtest to DynamicParameterSampleApp.webtest.



3. Record the test.



4. Copy and paste the URL from the website you're testing into the browser.



5. Browse through the web application. Choose the ASP.NET link, the Back link, and then the javascript link, followed by the back link.

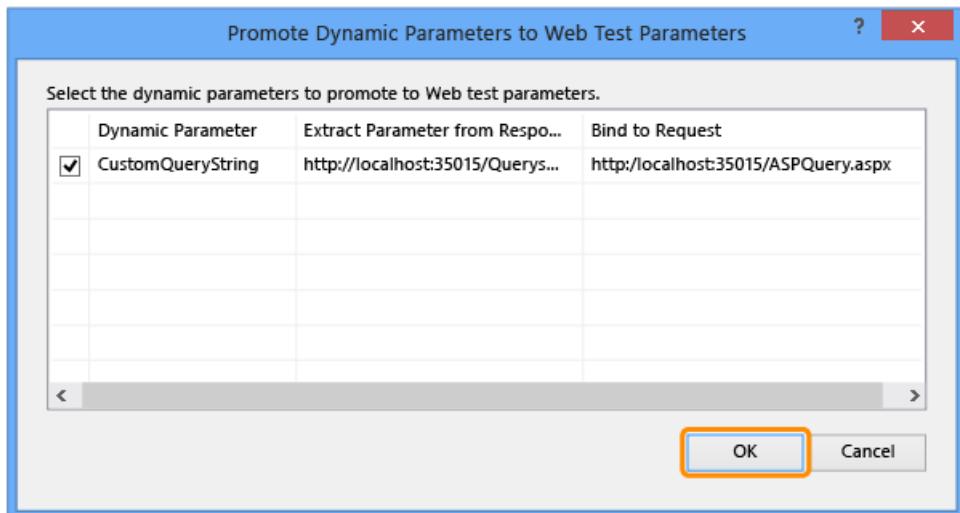
The web test recorder displays the HTTP request and response URLs as you navigate through the web app.

6. Choose the **Stop** button on the test recorder.

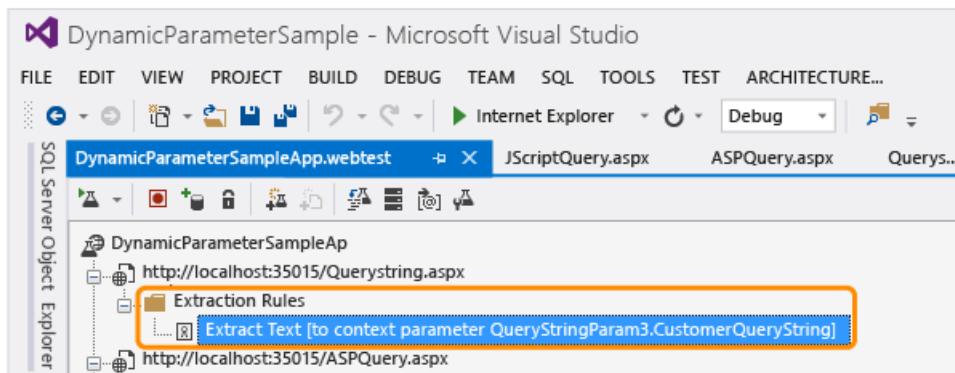
The dialog box for detecting dynamic parameters displays a progress bar that shows the status of parameter detection in the HTTP responses that were received.

7. The dynamic parameter for CustomQueryString in the ASPQuery page is automatically detected. However, The dynamic parameter for CustomQueryString in the JScriptQuery page is not detected.

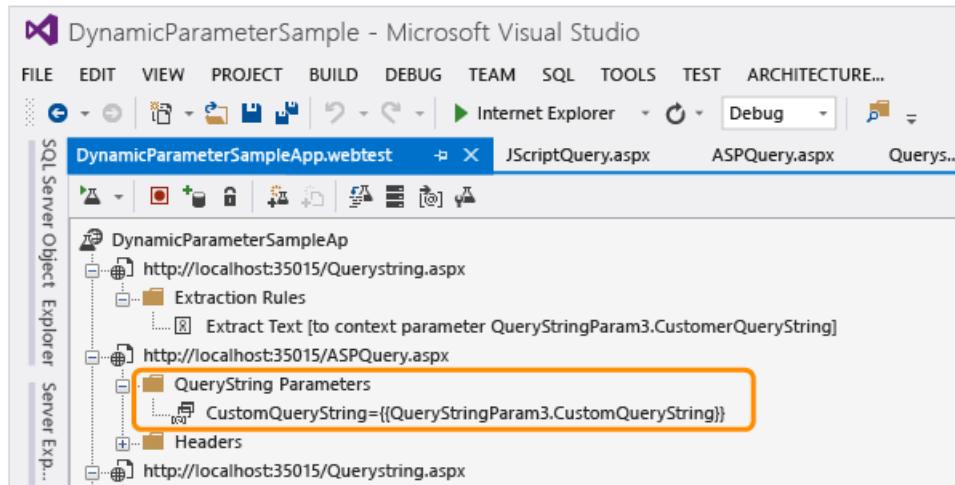
Choose **OK** to add an extraction rule to *QueryString.aspx*, binding it to the ASPQuery page.



The extraction rule is added to the first request for *QueryString.aspx*.



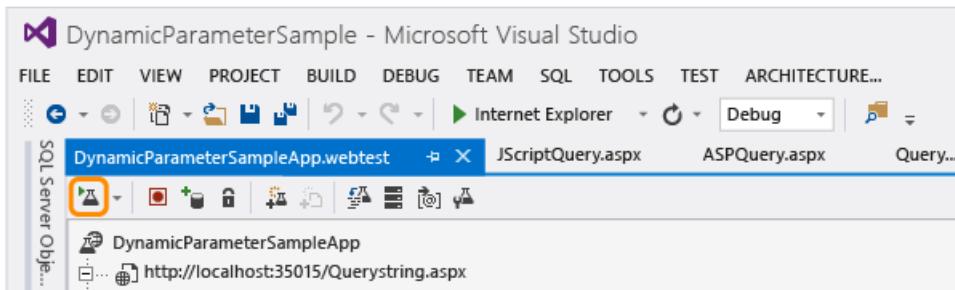
Expand the second request in the request tree for *ASPQuery.aspx* and notice that the *CustomQueryString*'s value has been bound to the extraction rule.



8. Save the test.

Run the test to isolate the non-detected dynamic parameter

1. Run the test.



2. The fourth request for the *JScriptQuery.aspx* page fails. Go to the web test.

The *JScriptQuery.aspx* request node is highlighted in the editor. Expand the node and notice that the "1v0yhyiyr0raa2w4j4pwf5zl" portion of the CustomQueryString appears to be dynamic.

The *JScriptQuery.aspx* request node is highlighted in the editor. Expand the node and notice that the "1v0yhyiyr0raa2w4j4pwf5zl" portion of the CustomQueryString appears to be dynamic.

3. Return to the Web Performance Test Results Viewer and select the *JScriptQuery.aspx* page that failed. Then, choose the request tab, verify that the show raw data check box is cleared, scroll down and choose quick find on the CustomQueryString.

Name	Value
Accept-Encoding	GZIP
Host	localhost:35015
Cookies	
Cookie	ASP.NET_SessionId=5w4v3yrse4...
QueryString Parameters	
CustomQueryString	jScriptQueryString_1v0yhyiyr0...
Form Post Parameters	

4. We know from looking at the test editor, that the *JScriptQuery.aspx* request's CustomQueryString was assigned a value of: `jScriptQueryString_1v0yhyiyr0raa2w4j4pwf5zl`, and that the suspected dynamic portion is "1v0yhyiyr0raa2w4j4pwf5zl". In the find what drop-down list, remove the suspect portion of the search string. The string should be "CustomQueryString=jScriptQueryString__".

Dynamic parameters are assigned their values in one of the requests that precedes the request that has the error. Therefore, select the search up check box and choose find next until you see preceding request for

Querystring.aspx highlighted in the request panel. This should occur after you choose find next three times.

The screenshot shows the Fiddler interface. On the left, a list of sessions is displayed with several items checked. The 'Request' tab is selected in the bottom navigation bar. In the center, the 'Headers' section shows 'HTTP/1.1 200 OK'. Below it, the 'Body' section contains a script block. A red box highlights the URL 'CustomerQueryString=jScriptQueryString__ + sessionId;'. The 'Find' dialog is open on the right, with 'CustomQueryString=jScriptQueryString__' entered in the 'Find what:' field, 'Search up' checked, and 'Current request' unchecked. The 'Find Next' button is visible at the bottom of the dialog.

As shown in the response tab, and in the JavaScript implemented earlier shown below, the query string parameter CustomQueryString is assigned a value of " jScriptQueryString__" and is also concatenated with the returned value from the var sessionId.

```
function jScriptQueryString() { var Hidden = document.getElementById("HiddenFieldSessionID"); var sessionId = Hidden.value; window.location = 'JScriptQuery.aspx?CustomQueryString=jScriptQueryString__' + sessionId; }
```

Now that we know where the error is occurring, and that we need to extract the value for sessionId. However, the extraction value is only text, so we need to further isolate the error by trying to locate a string where the sessionId's actual value is displayed. By looking at the code, you can see that the var sessionId equals the value returned by HiddenFieldSessionID.

5. Use quick find on HiddenFieldSessionID, clearing the search up check box and selecting current request.

The screenshot shows the Fiddler interface. The 'Request' tab is selected. The 'Find' dialog is open on the right, with 'HiddenFieldSessionID' entered in the 'Find what:' field, 'Search up' unchecked, and 'Current request' checked. The 'Find Next' button is visible at the bottom of the dialog. The session list on the left shows several recorded requests, and the body of the current request shows a dynamic hyperlink generated by JavaScript.

Notice that the value returned is not the same string as in the original web performance test recording. For this test run, the value returned is "5w4v3yrse4wa4axrafyqksq" and in the original recording, the value is "1v0hyiyr0raa2w4j4pwf5zl". Because the value does not match that of the original recording, the error is generated.

6. Because we have to fix the dynamic parameter in the original recording, choose recorded result in the

toolbar.

Request	Status	Total Time
✓ D http://localhost:35015/Querystring.aspx	200 OK	0.080 sec
✓ D http://localhost:35015/ASPString.aspx	200 OK	0.041 sec

7. In the recorded results, select the third request, which is the same *Querystringrequest.aspx* request that you isolated in the test run results.

Request
✓ D http://localhost:35015/Querystring.aspx
✓ D http://localhost:35015/ASPQuery.aspx
✓ D http://localhost:35015/Querystring.aspx
✓ D http://localhost:35015/JScriptQuery.aspx
✓ D http://localhost:35015/Querystring.aspx

Choose the response tab, scroll down and choose the original dynamic parameter value of "1v0yhyiyr0raa2w4j4pwf5zl" that you isolated previously and add an extraction rule.

Request Response Context Details

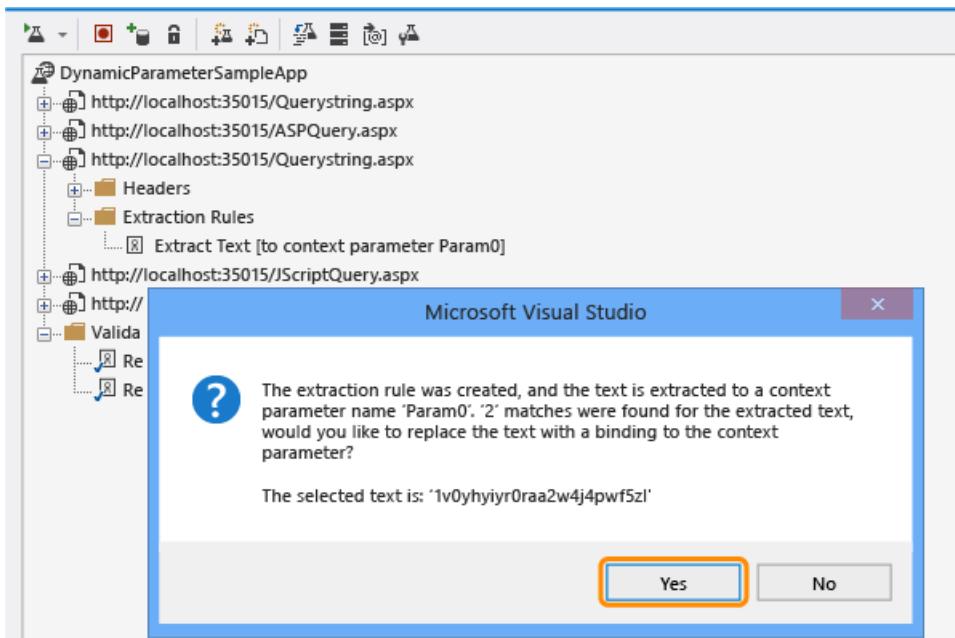
... in HTML editor

```
= "javascript:JScriptQueryString()"> Dynamic querystring generated by javascript </a>
="hidden" name="HiddenFieldSessionID" id="HiddenFieldSessionID" value="1v0yhyiyr0raa2w4j4pwf5zl" />
```

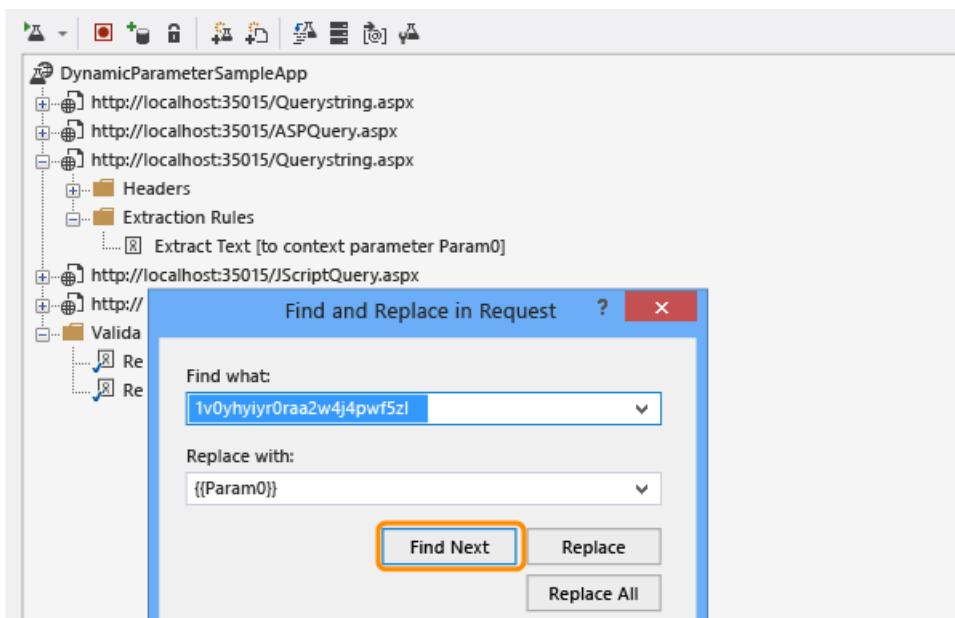
Copy
Select All
Quick Find
Add Extraction Rule

The new extraction rule is added to the *Querystring.aspx* request and is assigned a value of 'Param0'.

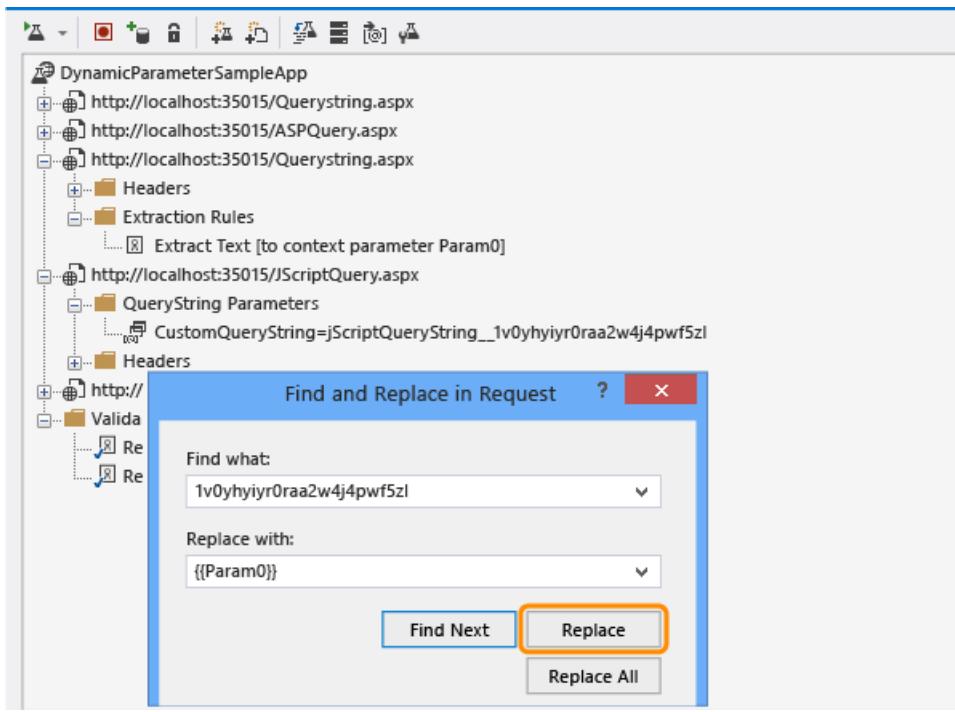
If the dialog box informs us that matches were found for the extracted text to bind the parameter to, choose **Yes**.



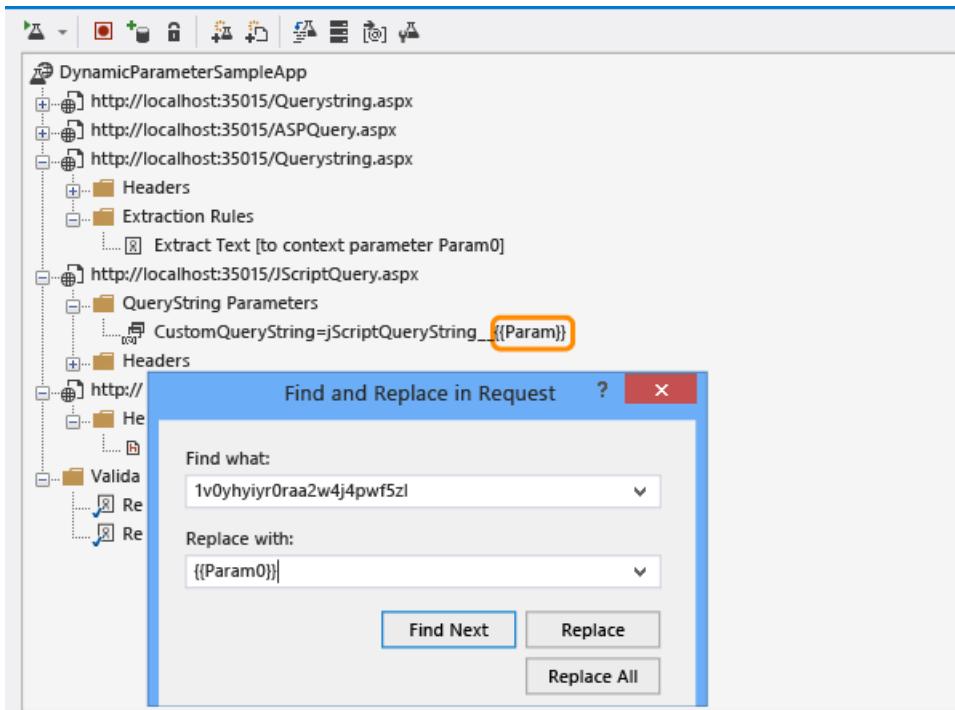
8. Choose **Find Next**. The first match is the one that we need to change, which is the parameter for CustomQueryString in for the JScriptQuery page.



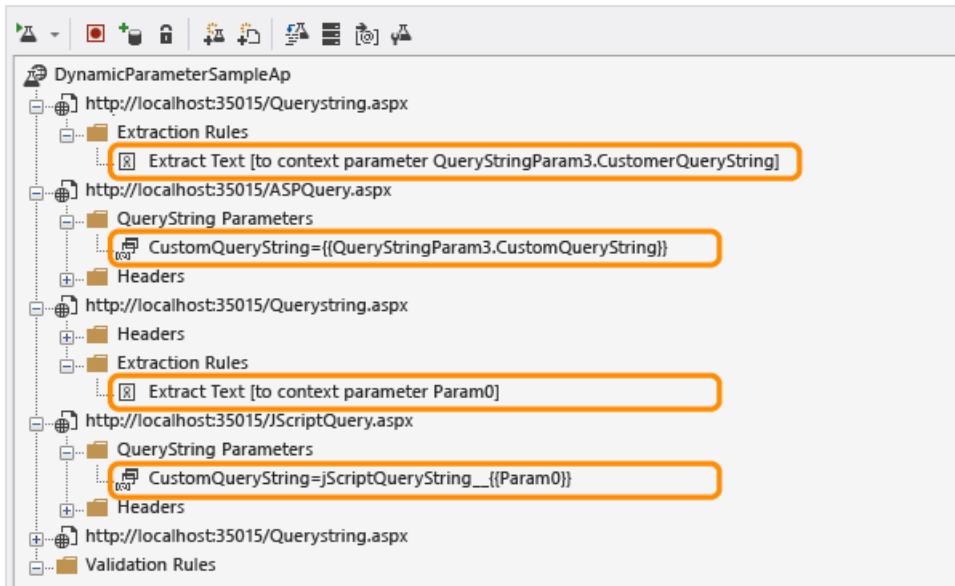
9. Choose **Replace**.



The QueryString parameter under the *JScriptQuery.aspx* request is updated by using the new context parameter: CustomQueryString=jScriptQueryString__{{Param0}}.



10. Close the **Find and Replace** dialog. Notice the similar structure of in the request tree between the detected dynamic parameter and the non-detected dynamic parameter that you correlated.



- Run the test. It now runs without failure.

Q&A

Q: Can I re-run dynamic parameter detection if my web app gets modified?

A: Yes, use the following procedure:

- In the toolbar, choose the **Promote Dynamic Parameters to Web Test Parameters** button.

After the detection process completes, if any dynamic parameters are detected, the **Promote Dynamic Parameters to web test parameters** dialog box appears.

The dynamic parameters are listed under the Dynamic Parameters column. The requests that the dynamic parameter will be extracted from and bound to are listed under the Extract Parameter from Response and Bind to Request columns.

If you choose a dynamic parameter in the **Promote Dynamic Parameters to web test parameters** dialog box, two requests will be highlighted in the Web Performance Test Editor request tree. The first request will be the request that the extraction rule will be added to. The second request is where the extracted value will be bound.

- Select or clear the check box next to the dynamic parameters you would like to automatically correlate. By default, all the dynamic parameters are checked.

Q: Do I need to configure Visual Studio to detect dynamic parameters?

A: The default Visual Studio configuration is to detect dynamic parameters when you record a web performance test. However, if you have Visual Studio options configured not to detect dynamic parameters, or the web application being tested gets modified with additional dynamic parameters; you can still run dynamic parameter detection from the Web Performance Test Editor.

Overview of test agents and test controllers for running load tests

1/1/2020 • 3 minutes to read • [Edit Online](#)

Visual Studio can generate simulated load for your app by using physical or virtual machines. These machines must be set up as a single test controller and one or more test agents. You can use the test controller and test agents to generate more load than a single computer can generate alone.

NOTE

You can also use cloud-based load testing to provide virtual machines that generate the load of many users accessing your website at the same time. However, using the test controller/test agent setup on cloud-hosted virtual machines is not supported. Learn more about cloud-based load testing at [Run load tests using Azure Test Plans](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Load simulation architecture

The load simulation architecture consists of a Visual Studio client, test controller, and test agents.

- The client is used to develop tests, run tests, and view test results.
- The test controller is used to administer the test agents and collect test results.
- The test agents are used to run the tests, and collect data including system information and ASP.NET profiling data defined in the test setting.

This architecture provides the following benefits:

- The ability to scale out load generation by adding additional test agents to a test controller.
- Flexibility for installing the client, test controller, and test agent software on the same or different computers. For example:

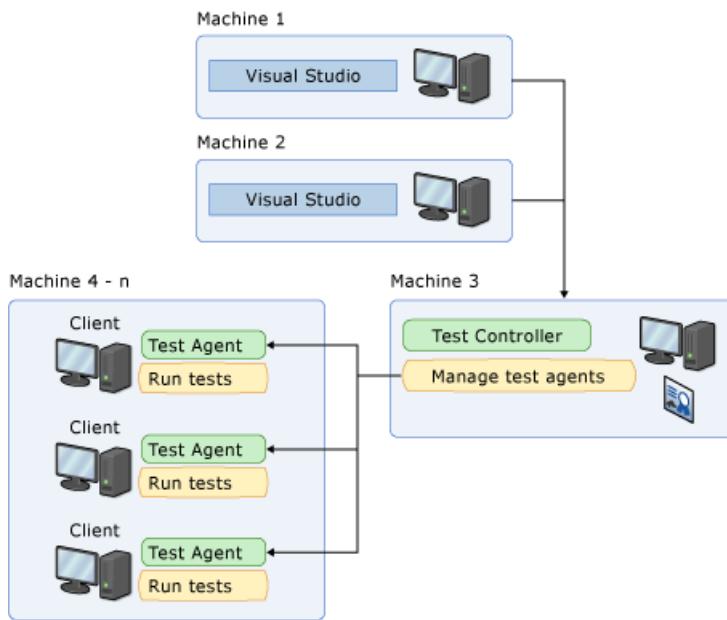
Local configuration:

- Machine1: Visual Studio, controller, agent.



Typical remote configuration:

- Machine1 and 2: Visual Studio (multiple testers can use the same controller).
- Machine3: Controller (can have agents installed, too).
- Machine4-n: Agent or agents all associated with the controller on Machine3.



Even though a test controller typically manages several test agents, an agent can only be associated with a single controller. Each test agent can be shared by a team of developers. This architecture makes it easy to increase the number of test agents, thereby generating larger loads.

Test agent and test controller interaction

The test controller manages a set of test agents to run tests. The test controller communicates with test agents to start tests, stop tests, track test agent status, and collect test results.

Test controller

The test controller provides a general architecture for running tests, and includes special features for running load tests. The test controller sends the load test to all test agents and waits until all the test agents have initialized the test. When all test agents are ready, the test controller sends a message to the test agents to start the test.

Test agent

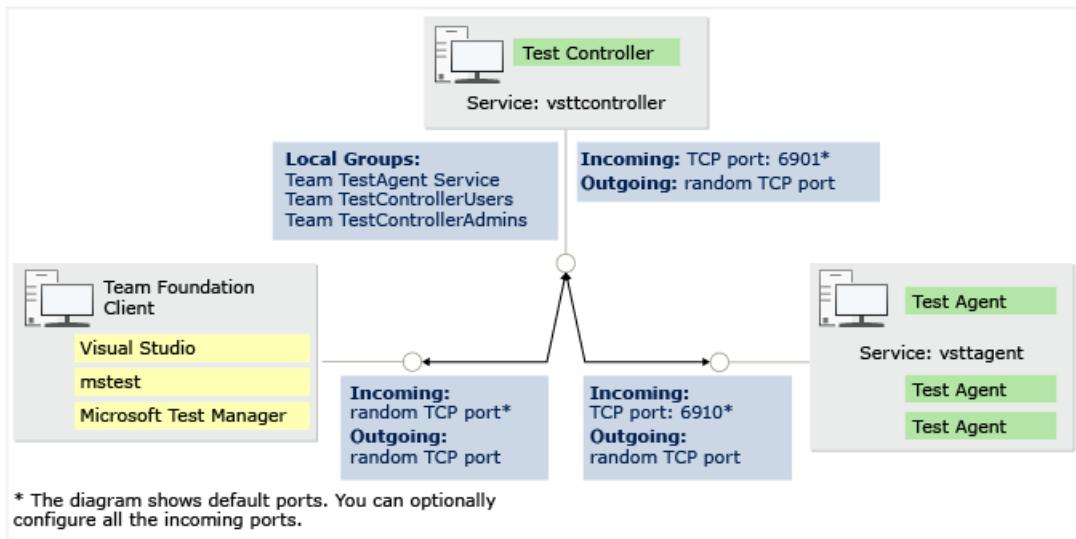
The test agent runs as a service that listens for requests from the test controller to start a new test. When the test agent receives a request, the test agent service starts a process on which to run the tests. Each test agent runs the same load test.

Test agents are assigned a weight by the administrator, and load is distributed according to a test agent's weighting. For example, if test agent 1 has a weighting of 30, and test agent 2 has a weighting of 70, and the load is set to 1000 users, then test agent 1 simulates 300 virtual users whereas test agent 2 simulates 700 virtual users. See [Manage test controllers and test agents with Visual Studio](#).

The test agent takes a set of tests and a set of simulation parameters as input. A key concept is that tests are independent of the computer where they're run.

Test controller and test agent connection points

The following illustration shows the connection points between the test controller, the test agent, and the client. It outlines which ports are used for incoming and outgoing connections as well as security restrictions used on these ports.



For more information see [Configure ports for test controllers and test agents](#).

Test controller and agent installation information

For important information about hardware and software requirements for test controllers and test agents, the procedures for installing them, and configuring your environment for optimal performance, see [Install and configure test agents](#).

Use the test controller and test agent with unit tests

After you have installed a test controller and one or more agents, you can specify whether to use a remote execution with the test controller in the test setting for your load tests. Additionally, you can specify the data and diagnostic adapters to use with the role that is associated with the agents in the test setting.

See also

- [Install and configure test agents](#)

Test controller and test agent requirements for load testing

1/1/2020 • 3 minutes to read • [Edit Online](#)

Several test types including unit, web performance, load, and manual tests are integrated into Visual Studio. Visual Studio enables Visual Studio Application Lifecycle Management users to run tests on remote computers using a test controller and one or more agents. See [Install and configure test agents](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Hardware and software requirements

Both the test controller and test agent computers have specific hardware and software requirements. In addition, if you want to deploy the test controller and test agent computers across multiple languages, you must plan how to support those languages.

Hardware requirements

The following table shows the recommended hardware requirements for deploying a test controller and test agents.

CONFIGURATION	COMPONENT	CPU	HD	MEMORY
< 500 virtual users	Test agent	2.6 GHz	10 GB	2 GB
< 1000 virtual users	Test agent	Dual processor 2.6 GHz	10 GB	2 GB
N x 1000 virtual users	Test agent	Scale out to N agents each with Dual 2.6 Ghz	10GB	2GB
< 30 computers in the test environment. This includes agents and servers under test.	Test Controller	2.6 GHz		
N x 30 computers in the test environment. This includes agents and servers under test.	Test Controller	N 2.6 GHz processors		

NOTE

The number of virtual users will vary widely from test to test. A key cause of this variance is variance in *think times*, or user delays. For more information, see [Edit think times to simulate website human interaction delays](#). In a load test, web tests are generally more efficient and generate more load than unit tests. The numbers in the preceding table are valid for running web tests with 3-5 second think times on a typical web application.

The guidelines presented here are provided as general guidance for hardware planning. Test performance will vary greatly based on the amount of test data and the number of test agents. For test agents, the CPU speed and memory available will limit the test load. Test controllers need greater resources, depending on the number of test agents and the amount of data involved in the tests.

The server that is running Visual Studio should have a reliable network connection with a minimum bandwidth of 1 Mbps and a latency maximum of 350ms. There should be no firewall between the test agents and the test controller. If your test performance does not meet your expectations, consider upgrading your hardware configuration.

Additional hardware considerations

Test agents generate a large amount of data on the test controllers, depending on the duration of the test and the size of the test. Generally, you should plan for an additional 10 GB of hard disk storage for every 24 hours of test data.

In addition to the hardware recommended here, you should consider additional hardware for critical servers, such as redundant power supplies and redundant fans.

Language requirements

To avoid confusion and simplify operation, a test controller and test agents should be configured to use the same language as the computer's operating system and that of Team Foundation Server. If the test agent and test controller are installed on different computers, they must be configured to use the same language. You can, however, install another language version of Visual Studio on an English-language operating system, as long as that language matches that of the Team Foundation Server deployment.

Monitor agent resources

You can monitor agent machines to determine their resource needs by observing the *QTAgent*.exe* processes that execute and scale during tests. The most common bottleneck on the *QTAgent*.exe* processes is CPU utilization. If the CPU utilization is consistently in the high nineties then it is an indication that the agent is being loaded heavily. The next common bottleneck is the memory usage. For demanding tests, monitoring these resources can help determine if you should increase the machines resources, or distribute your tests differently.

See also

- [Install and configure test agents](#)

Manage test controllers and test agents

1/1/2020 • 14 minutes to read • [Edit Online](#)

If you want to use Visual Studio to run tests remotely, distribute tests across multiple machines, or run load tests, you must configure a test controller, test agents, and test settings file. This topic describes how to manage test controllers and test agents after you install and configure them for the first time.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

If you use Microsoft Test Manager to run tests in lab environments, you manage test controllers and their agents by using the **Test Controller Manager** in the **Lab Center** for Microsoft Test Manager. This topic is applicable only if you use Visual Studio to run tests.

For information about how to install and configure test agents and test controllers to run tests in Visual Studio, see [Configure test agents and controllers](#).

To configure and monitor the test controller and any registered agents, you must have a test settings file in your test project that contains the tests you want to run. Open the test settings file, choose **Role** and choose **Manage Test Controllers** from the drop down for the **Controller** field.

For a load test project, you can also choose **Manage Test Controllers** from the **Load Test** menu.

Add a test agent to a test controller

You might want to add a test agent to a different test controller or you might have to add a test agent to a test controller that you have just installed.

To add a test agent to a test controller

1. Choose **Start > Test Agent Configuration Tool**.

The **Configure Test Agent** dialog box is displayed.

NOTE

You must have a test agent already installed to add it to a test controller. For more information about how to install a test agent, see [Install and configure test agents](#).

2. You're presented with two options for how the test agent can be run:

- **Service:** If you do not have to run automated tests that interact with the desktop, such as coded UI tests or creating a video recording when your test runs, under **Run the test agent as**, select **Service**. The test agent will be started as a service. Choose **Next**.

You can now enter the details about the user when the test agent starts as a service.

- a. Enter the name in **User name**.
- b. Enter the password in **Password**.

IMPORTANT USER ACCOUNT INFORMATION

- Null passwords are not supported for user accounts.
- If you want to use the IntelliTrace collector or the network emulation, the user account must be a member of the Administrators group.
- If the agent user name is not in the agent service it will try to add it, which requires permissions on the test controller.
- The user who is trying to use the test controller must be in the test controller's Users account or they will be unable to run the tests against the controller.

- **Interactive Process:** If you want to run automated tests that must interact with the desktop, such as coded UI tests or creating a video recording when your test runs, select **Interactive Process**. The test agent will be started as an interactive process instead of a service.

On the next page, enter the details about the user when the test agent starts as a process, and other options.

- a. Enter the name in **User name**.
- b. Enter the password in **Password**.

NOTE

If you configure the test agent to run as an interactive process with a different user who is not the currently active user, you must restart the computer and log on as this different user to be able to start the agent. In addition, null passwords are not supported for user accounts. If you want to use the IntelliTrace collector or the network emulation, the user account must be a member of the Administrators group.

- c. To make sure that a computer that has a test agent can run tests after it restarts, you can set up the computer to log on automatically as the test agent user. Select **Log on automatically**. This will store the user name and password in an encrypted form in the registry.
- d. To make sure that the screen saver is disabled because this might interfere with any automated tests that must interact with the desktop, select **Ensure screen saver is disabled**.

WARNING

There are security risks if you log on automatically or disable the screen saver. By enabling automatic log on, you enable other users to start this computer and to be able to use the account that automatically logs on. If you disable the screen saver, the computer might not prompt for a user to log on to unlock the computer. This lets anyone access the machine if they have physical access to the computer. If you enable these features on a computer, you should make sure that these computers are physically secure. For example, these computers are located in a physically secure lab. (If you clear **Ensure screen saver is disabled**, this does not enable your screen saver.)

3. To register this agent with a different test controller, select **Register with test controller**. Type the name of your test controller followed by : and the port number that you are using in **Register the test agent with the following test controller**. For example, type **agent1:6901**.

NOTE

The default port number is 6901.

4. To save your changes, choose **Apply Settings**. Close the **Configuration summary** dialog box, and then close the **Test Agent Configuration Tool**.

WARNING

If the agent is currently configured to run on another test controller, you must remove the test agent from that controller.

Remove a test agent from a test controller

A test agent must be set to the offline state before it can be removed.

NOTE

You can't use this procedure to remove agents that are registered to a controller as part of a lab environment. To remove these agents from a controller, you must remove the environment using Microsoft Test Manager.

To remove a test agent from a test controller

1. If the test controller is not registered with a project, follow these steps.
 - a. From Visual Studio open the test settings file for your test project, choose **Role** and choose **Manage Test Controllers** from the drop down for the **Controller** field.
The **Administer Test Controller** dialog box is displayed.
 - b. In the **Controller** drop-down list, type the name of the computer on which you have set up the test controller. If you have administered a specific test controller previously, you can select the name from the list.
 - c. In the **Agents** pane, select the test agent name. If the agent is still online, choose **Offline**. To remove it, choose **Remove**.

NOTE

Removing a test agent just disassociates it from the test controller. To completely uninstall the test agent, use the **Programs and Features** Control Panel on the test agent computer.

2. If the test controller is registered with a project, remove the agent using Microsoft Test Manager.

Change the settings for a test agent

The status of the test agent can be any one of the following values:

STATUS	DESCRIPTION
Running Test	Running tests
Ready	Available to run tests or collect data and diagnostics

STATUS	DESCRIPTION
Offline	Unavailable to run tests or collect data and diagnostics
Disconnected	Test agent is not started

You can change the status and other settings for a test agent using the following procedures.

To change the settings of a test agent

NOTE

If the test agent is registered to a test controller that is registered with a project, change the settings in Microsoft Test Manager.

1. To configure and monitor the test controller and any registered agents for a load test, choose the **Load Test** menu in Visual Studio and then choose **Manage Test Controllers**. For any other tests, open the test settings file for your test project in Visual Studio, choose **Role** and choose **Manage Test Controllers** from the drop down for the **Controller** field.

The **Manage Test Controller** dialog box opens.

2. Select the name of the test controller whose test agents you want to change in the test controller list. If the test controller does not appear in the list, check that the test controller is registered correctly. For more information, see the following procedure about how to configure a test controller.
3. (Optional) In the **Test Agents** pane, select the test agent computer for which you want to change the properties.
4. Choose **Properties**.
5. Change the following test agent properties as required:

TEST AGENT PROPERTY	DESCRIPTION
Weighting	Used to distribute load when you use test agents with different performance levels. For example, a test agent with a weighting of 100 receives two times the load as a test agent with a weighting of 50.

TEST AGENT PROPERTY	DESCRIPTION
IP Switching	<p>Used to configure IP switching. IP switching allows a test agent to send requests to a server by using a range of IP addresses. This simulates calls that come from different client computers.</p> <p>IP Switching is important if your load test is accessing a web farm. Most load balancers establish affinity between a client and a particular web server by using the client's IP address. If all requests seem like they are coming from a single client, the load balancer will not balance the load. To obtain good load balance in the web farm, make sure that requests come from a range of IP addresses. Note: You can specify a network adapter, or use (All unassigned) to automatically select one that is currently not being used.</p> <p>To use the IP switching feature, the Visual Studio Test Agent service must be running as a user in the Administrators group for that agent computer. This user is selected during agent setup, but can be changed by modifying the properties of the service and restarting it.</p> <p>To verify that IP switching is working correctly, enable IIS logging on the web server, use the IIS logging functionality to verify that requests are coming from the IP addresses that you configured.</p>
Attributes	<p>Set of name/value pairs that can be used in test agent selection. For example, a test might require a particular OS. You can add attributes in the Roles tab of your test settings file and they can be used to select a test agent that has matching attributes. If you want to run a test on multiple machines, create an attribute in the test settings role that is configured to run your tests, and then configure a matching attribute on each test agent that you want to use in that role..</p> <p>Note: This setting is only available for test agents that are registered with a test controller that is not registered to a project, because these attributes are only used in test settings for Visual Studio.</p>

Test agent weight and test agent attribute changes go into effect immediately, but do not affect tests that are running. The IP Address Range takes effect after the test controller is restarted.

(Optional) To change the status of a test agent, select the agent in the list and then select the action from the available choices based on the current status of the agent.

NOTE

If your test agent is running as a process, you manage the status of the test agent from the notification area icon that runs on the computer where your test agent is installed. This shows the status of the test agent. You can start, stop or restart the agent if it is running as a process using this tool.

Configure a test controller

To configure a test controller, you must use the **Team Test Controller Configuration Tool**. When you configure your test controller, you can register your test controller with a different project collection, or unregister your test controller from a project collection.

If you want to register your test controller with your Team Foundation Server project collection, the account that

you use for the test controller service must be a member of the Project Collection Test Service Accounts group for the Project Collection, or the account that you use to run the test controller configuration tool must be a Project Collection Administrator.

NOTE

If you unregister a test controller from a project collection that has existing environments in a project collection, the environments are still maintained if you moved that project collection and re-register the test controller to that moved project collection.

To configure a test controller

1. To run the tool to reconfigure your test controller at any time, choose **Start > Test Controller Configuration Tool**.

The **Configure Test Controller** dialog box is displayed.

2. Select the user to use as the logon account for your test controller service.

NOTE

Null passwords are not supported for user accounts.

3. (Optional) If you do not want to use your test controller with a lab environment, but only to run tests from Visual Studio, clear **Register test controller with Team Project Collection**.
4. (Optional) To configure your test controller for load testing, select **Configure test controller for load testing**. Enter the SQL Server instance under **Create load test results database in the following SQL Server instance**.

NOTE

For more trouble shooting information about test controllers, see [Install and configure test agents](#).

Manage your agents when you run your tests with a test controller

When you add roles for your application to your test settings for Visual Studio, you can add agent properties for each of your roles. This determines which test agents are available for this role. When you run your tests using these test settings, the test controller that is selected for the test settings determines the availability of the required agents. These are the following situations that can occur when the agent availability is determined:

- There is no agent available for the role that must run the tests. Your tests cannot be run. You can perform one of the following actions and then rerun your tests:
 - You can wait for an agent to become available for this role to run the tests.
 - If there are any agents that are offline that can be used for this role, you can restart the agent so that it is available.
 - You can add another agent with the correct agent properties for that role to the test controller.
 - You can change the agent properties for this role in the test settings to enable other agents that you want to use.
- There is no agent available for one or more roles that run diagnostic data adapters. Your tests can be run, but the diagnostic data adapter cannot be run. You can run your tests without the diagnostic data adapter, or

you can perform one of the following actions and rerun your tests:

- You can wait for an agent to become available for these roles.
- If there are any agents that are offline that can be used for this role, you must change the state of the agent to online from **Administer Test Controller** on the **Test** menu. In addition, you might have to restart the agent if it has been disconnected from the controller.
- Verify that any agents that you might need for this test run are not busy running tests. You can check the status of any agents from **Administer Test Controller** on the **Test** menu.
- You can add another agent with the correct agent properties for the role to the test controller.
- You can change the agent properties for the role in the test settings to enable other agents that you want to use.

Load tests from delay-signed assemblies

The test controller and test agents can only load test assemblies that are strongly signed assemblies, or unsigned assemblies. Some test assemblies are delay-signed because they need to have access into production assemblies for the application. However, these assemblies are not strongly signed because they are only test assemblies and are not distributed. These assemblies cannot be loaded because they are delay-signed, so you must disable strong name verification for those assemblies on all machines where the assembly will be loaded including the test controller machine. To disable the delay-signed verification, use *sn.exe*. The public key token of the delay-signed assembly for which strong name verification is requested to be skipped may also need to be included.

Use *Sn.exe* (Strong Name tool) to disable the delay-signed verification.

This disables strong-name verification, for the specified assembly only, on the computer on which you run the command. You can do this only if you have sufficient permissions.

After the test run has completed, re-enable the delayed-signing verification by using the *SN.exe* command.

A recommended way to disable and re-enable signing verification is to use the *SN.exe* commands in scripts. You can disable verification in a setup script and re-enable verification in a cleanup script.

See also

- [Install and configure test agents](#)

How to: Specify test agents to use in load test scenarios

1/1/2020 • 2 minutes to read • [Edit Online](#)

After you create your load test by using the **New Load Test Wizard**, you can use the **Load Test Editor** to change the scenarios properties to meet your testing needs and goals.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

NOTE

For a full list of the load test scenario properties and their descriptions, see [Load test scenario properties](#).

The agents are specified by using the **Load Test Editor** to change the **Agents to Use** property in the **Properties** window.

You can specify the agents that you want your scenario to use if you are using controllers and agents to run the load test remotely. For example, you might want to specify a specific set of agents so that you maintain consistency when you analyze performance trends. Also, agents may be geographically distributed, so that an affinity exists between which scripts they run and where the agent is located.

TIP

Rather than physically putting an agent at the remote site, another option is to use network emulation to emulate the slow network. For more information, see [Specify virtual network types](#).

For more information, see [Test controllers and test agents](#).

Another reason is that some, but not all, agents might have software installed on them that is required for a particular scenario.

You can control agent selection for a given test run by using roles in test settings. For more information, see [Collect diagnostic information using test settings](#).

If a test agent machine has more than 75 percent CPU utilization or has less than 10 percent of physical memory available, add more agents to your load test to make sure that the agent machine does not become the bottleneck in your load test.

To specify the agents to use for a scenario

1. Open a load test.

The **Load Test Editor** appears. The load test tree is displayed.

2. In the load test trees **Scenarios** folder, choose the scenario node for which you want to specify the agents to use.

3. On the **View** menu, select **Properties Window**.

The categories and properties of the scenario are displayed in the **Properties** window.

4. In the text box for the **Agents to Use** property, type the list of agents on which the scenario may run.

Agents must be separated by commas, for example "**Agent1, Agent2, Agent3**". Leaving the property blank specifies that the scenario should use all available agents.

NOTE

The **Agents to Use** property is ignored for local runs. For remote runs, if none of the agents specified in **Agents to Use** exist, tests in the scenario will not run.

5. After you change the property, choose **Save** on the **File** menu. You can then run your load test by using the new **Agents to Use** value.

See also

- [Edit load test scenarios](#)
- [Walkthrough: Create and run a load test](#)
- [Test controllers and test agents](#)
- [Load test scenario properties](#)

Assign roles to a test controller and test agent

1/1/2020 • 2 minutes to read • [Edit Online](#)

This article demonstrates how to create and configure a test setting that uses a test controller and test agent to distribute testing across several machines using Visual Studio. It also demonstrates how to add diagnostic and data adapters to the test setting.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Prerequisites

- Create unit tests or coded UI tests to run with the test setting.
- Install a test controller and test agents. For information about how to install a test controller and test agents, see [Install and configure test agents](#).

To create and configure a test setting

1. In **Solution Explorer**, right-click **Solution Items**, point to **Add**, and then choose **New Item**.

The **Add New Item** dialog box appears.

2. In the **Installed Templates** pane, choose **Test Settings**.
3. In the **Name** box, type **TestSettingDistributedTestWalkthrough**.
4. Choose **Add**.

The new test *TestSettingDistributedTestWalkthrough.testsettings* file appears in **Solution Explorer**, under the **Solution Items** folder.

The **Test Settings** dialog box is displayed. The **General** page is selected.

You can now edit and save test settings values.

5. Under **Name**, type the name for the test settings.
6. Under **Description**, type **Distributed test settings**.
7. Leave **Default naming scheme** selected.

To assign roles to a test controller and test agents

1. Choose **Roles**.

The **Roles** page is displayed.

2. To run your test remotely, use the **Test execution method** drop-down list and select **Remote execution**.
3. In the **Controller** drop-down list, type the computer name of [your test controller](#).

NOTE

If this is the first time that you are adding a controller, there are no controllers listed in the drop-down list. The list is populated by previous controllers that you have specified in other test settings.

4. Under **Roles**, choose **Add**.
5. In the highlighted row under the **Name** column, type **Distributed test**.

To assign a diagnostic and data adapter to your test setting

1. Choose **Data and Diagnostics**.

The **Data and Diagnostics** page is displayed.

2. Under **Role**, verify that the **Distributed test** role is selected.
3. Under **Data and Diagnostic for select role**, select the **IntelliTrace** and **System Information** adapters.

For information about these adapters and other adapters that you can use in a test setting, see [Configure unit tests](#).

4. Choose **Hosts**.
5. (Optional) If your machine is running under a 64-bit version of Microsoft Windows, and you compiled your test using the **Any CPU** configuration, use the **Run test in 32 bit or 64 bit process** drop-down list and select **Run tests in 64-bit process on 64-bit machine**.

TIP

For maximum flexibility, you should compile your test projects with the **Any CPU** configuration. Then you can run on both 32-bit and 64-bit agents. There is no advantage to compiling test projects with the **64-bit** configuration.

6. To save the new test settings, choose **Apply**.
7. Choose **Close**.
8. On the **Test** menu, select **Test Settings > Select Test Settings File** and then choose the *TestSettingDistributedTestWalkthrough.testsettings* file.
8. On the **Test** menu, choose **Select Settings File**. Browse to and select the *TestSettingDistributedTestWalkthrough.testsettings* file.
9. Run your test as usual.

When the test controller processes unit tests and coded UI tests, the test controller divides the tests into groups of 100 and sends them to a test agent machine. For example, if you have 250 unit tests and three test agents, the first 100 unit tests will be sent to agent1, the next 100 unit tests will be sent to agent2, and the remaining 50 unit tests will be sent to agent3.

See also

- [Install and configure test agents](#)

Configure ports for test controllers and test agents

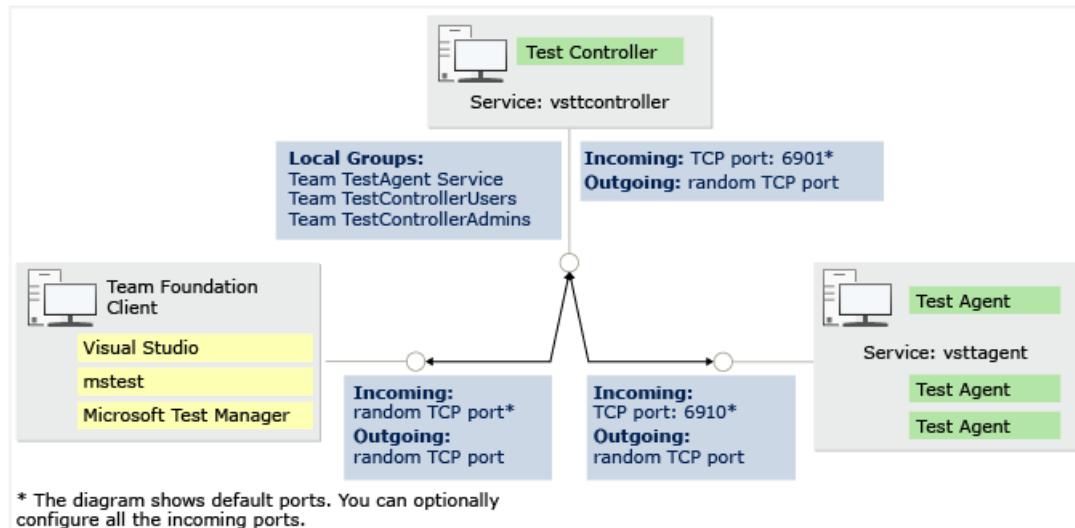
1/1/2020 • 2 minutes to read • [Edit Online](#)

You can change the default incoming ports used by the test controller, the test agent, and the client. This might be necessary if you are trying to use the test controller, the test agent, or the client together with some other software that conflicts with the port settings. Another reason to change the ports is due to the firewall restriction between the test controller and the client. In this case you might want to manually configure the port to accommodate enabling it for a firewall so that the test controller can send results to the client.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

The following illustration shows the connection points between the test controller, test agent and the client. It outlines which ports are used for incoming and outgoing connections as well as security restrictions used on these ports.



Incoming connections

The default port used by the test controller is 6901 and the test agent's default port is 6910. The client uses a random port by default which is used to receive the test results from the test controller. For all incoming connections, the test controller authenticates the calling party and verifies that it belongs to specific security group.

- **Test Controller** Incoming connections are on TCP port 6901. If you need to, you can configure the incoming port. For more information, see [Configure the incoming ports](#).

The test controller needs to be able to make outgoing connection to test agents and to the client.

NOTE

The test controller needs incoming **File and Printer sharing** connection open.

- **Test Agent** Incoming connections are on TCP port 6910. If you need to, you can configure the incoming port. For more information, see [Configure the incoming ports](#).

The test agent needs to be able to make outgoing connection to the test controller.

- **Client** By default, a random TCP port is used for incoming connections. If you need to, you can configure the incoming port. For more information, see [Configure the incoming ports](#).

You might get firewall notifications when the test controller tries to connect to the client the first time.

On Windows Server 2008, the firewall notifications are disabled by default and you must manually add Firewall exceptions for Client programs (*devenv.exe*, *mstest.exe*, *mlm.exe*) so that it can accept incoming connections.

Outgoing connections

Random TCP ports are used for all outgoing connections.

- **Test Controller** The test controller needs to be able to make outgoing connection to Agents and to the Client.
- **Test Agent** The test agent needs to be able to make outgoing connection to Controller.
- **Client** The client needs to be able to make outgoing connection to Controller.

Configure the incoming ports

Follow these directions to configure the ports for a test controller and test agents.

- **Controller Service** Modify the port's value by editing the *%ProgramFiles(x86)%\Microsoft Visual Studio\2017\Enterprise\Common7\IDE\QTCcontroller.exe.config* file:

```
<appSettings>
  <add key="ControllerServicePort" value="6901"/>
</appSettings>
```

- **Agent Service** Modify the port by editing the *%ProgramFiles(x86)%\Microsoft Visual Studio\2017\Enterprise\Common7\IDE\QTAgentService.exe.config* file:

```
<appSettings>
  <add key="AgentServicePort" value="6910"/>
</appSettings>
```

- **Client** Use the registry editor to add the following registry (**DWORD**) values. The client will use one of the ports from the specified range for receiving data from the test controller:

HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\VisualStudio\12.0\EnterpriseTools\QualityTools\ListenPortRange\PortRangeStart

HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\VisualStudio\12.0\EnterpriseTools\QualityTools\ListenPortRange\PortRangeEnd

See also

- [Install and configure test agents](#)

How to: Bind a test controller or test agent to a network adapter

1/1/2020 • 3 minutes to read • [Edit Online](#)

If a computer that has the test controller or the test agent software installed has multiple network adapters, then you must specify the IP address instead of the name of the computer to identify that test controller or test agent.

WARNING

When you try to set up a test agent, you might receive the following error:

Error 8110. Can not connect to the specified controller computer or access the controller object

This error can be caused by installing the test controller on a computer that has more than one network adapter. It is also possible to install agents successfully, and not see this problem until you try to run a test.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Bind a test controller to a specific network adapter

To obtain the IP addresses of the network adapters

1. From Microsoft Windows, choose **Start**, choose in the **Start Search** box, type **cmd**, and then choose **Enter**.
2. Type **ipconfig /all**.

The IP addresses for your network adapters are displayed. Record the IP address of the network adapter that you want to bind your controller to.

To bind a network adapter to a test controller

1. From Microsoft Windows, choose **Start**, choose in the **Start Search** box, type **services.msc**, and then choose **Enter**.

The **Services** dialog box is displayed.

2. In the results pane, under the **Name** column, right-click the **Visual Studio Test Controller** service and then choose **Stop**.

-or-

Open an elevated command prompt and run the following command at a command:

```
net stop vsttcontroller
```

3. Open the *QTController.exe.config* XML configuration file located in %ProgramFiles(x86)%\Microsoft Visual Studio\2017\<edition>\Common7\IDE.
4. locate `<appSettings>` tag.

```
<appSettings>
  <add key="LogSizeLimitInMegs" value="20"/>
  <add key="AgentConnectionTimeoutInSeconds" value="120"/>
  <add key="AgentSyncTimeoutInSeconds" value="300"/>
  <add key="ControllerServicePort" value="6901"/>
  <add key="ControllerUsersGroup" value="TeamTestControllerUsers"/>
  <add key="ControllerAdminsGroup" value="TeamTestControllerAdmins"/>
  <add key="CreateTraceListener" value="no"/>
</appSettings>
```

5. Add the `BindTo` key to specify which network adapter to use in the `<appSettings>` section.

```
  <add key="BindTo" value="<YOUR IP ADDRESS>"/>
</appSettings>
```

6. Start the test controller service. To do this, run the following command at a command prompt:

```
net start vsttcontroller
```

WARNING

You must run the test agent installation again to connect the test agent to the controller. This time, specify the IP address for the controller instead of the controller name.

This applies to the controller, the agent service, and the agent process. The `BindTo` property must be set for each process that is running on a computer that has more than one network adapter. The procedure to set the `BindTo` property is the same for all three processes, as specified earlier in this topic for the test controller.

To bind a network interface card to a test agent

1. From Microsoft Windows, choose **Start**, choose in the **Start Search** box, type **services.msc**, and then choose **Enter**.

The **Services** dialog box is displayed.

2. In the results pane, under the **Name** column, right-click the **Visual Studio Test Agent** service and then choose **Stop**.

-or-

Open an elevated command prompt and run the following command at a command:

```
net stop vsttagent
```

3. Open the *QTAgentService.exe.config* XML configuration file located in %ProgramFiles(x86)%\Microsoft Visual Studio\2017\<edition>\Common7\IDE.

4. locate `<appSettings>` tag.

```
<appSettings>
  <appSettings>
    <add key="LogSizeLimitInMegs" value="20"/>
    <add key="AgentServicePort" value="6910"/>
    <add key="ControllerConnectionPeriodInSeconds" value="30"/>
    <add key="StopTestRunCallTimeoutInSeconds" value="120"/>
    <add key="CreateTraceListener" value="no"/>
    <add key="GetCollectorDataTimeout" value="300"/>
  </appSettings>  </appSettings>
```

5. Add the `BindTo` key to specify which network adapter to use in the `<appSettings>` section.

```
  <add key="BindTo" value="<YOUR IP ADDRESS>"/>
</appSettings>
```

6. Start the test agent service. To do this, run the following command at a command prompt:

```
net start vsttagent
```

See also

- [Install and configure test agents](#)
- [Modify load test logging settings](#)
- [Configure ports for test controllers and test agents](#)
- [How to: Specify timeout periods for test controllers and test agents](#)

How to: Set up your test agent to run tests that interact with the desktop

1/1/2020 • 5 minutes to read • [Edit Online](#)

If you want to run automated tests that interact with the desktop, you must set up your agent to run as a process instead of a service. For example, if you want to run a coded UI test remotely using a test controller and test agent, or you want to run a test and capture a video recording when you run it, you must set up your agent to run as a process. When you assign agents to roles in your test settings using Visual Studio, or you assign agents to roles in your environment by using Microsoft Test Manager, you must change the setup for any agents assigned to roles that have to interact with the desktop.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

WARNING

If you use Microsoft Test Manager to set up a lab environment, it installs the test agent. You can specify in the [Environment Creation Wizard](#) that you want to configure one of the roles to run coded UI tests.

IMPORTANT

The computer that is running an agent on which you want to run coded UI tests cannot be locked or have an active screen saver.

If you are running coded UI tests that start a browser, the service account for the test agent is used to start that browser. This service account must be the same as the user account that is the active user on this computer. If it is not the same user account, the browser will not start.

IMPORTANT

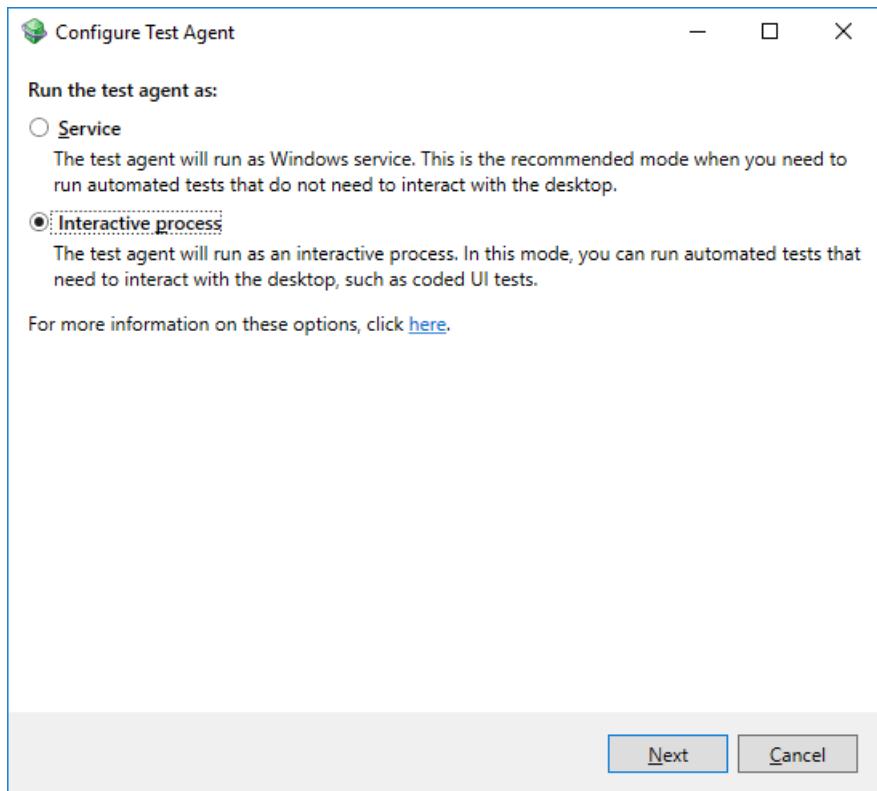
If you are running a coded UI test that starts a browser as part of a build definition, the service account for the build service is used to start that browser. This service account must be the same as the user account that is the active user on this computer. If it is not the same user account, the browser will not start.

Use the following procedure to set up any agents that are assigned to a role that performs a task that needs to interact with the desktop.

To set up an agent to run as a process

1. To configure the test agent you have installed to run as a process, go to **Start > Test Agent Configuration Tool**.

The **Configure Test Agent** dialog box is displayed.



2. Select **Interactive Process**. The test agent will be started as a process instead of a service. Choose **Next**.
3. Enter the user name and password for the user that will run the test agent process.

NOTE

- The user who you add to start the process must also be added as a member of the TeamTestAgentService group on the computer for the test controller for this agent. If this user is the current user, when you add this user to the test controller computer you must log off or reboot.
- Null passwords are not supported for user accounts.
- If you want to use the IntelliTrace or the network emulation data and diagnostic adapter, the user account must be a member of the Administrators group. If the machine that is running the test agent is running an OS that has Least-Privileged User Account, you have to run it as an administrator also (elevated). If the agent user name is not in the agent service it will try to add it, which requires permissions on the test controller.
- The user trying to use the test controller must be in the test controller's Users account or they won't be able to run the tests against the controller.

4. To make sure that a computer that has a test agent can run tests after rebooting, you can set up the computer to sign in automatically as the test agent user. Select **Log on automatically**. This will store the user name and password in an encrypted form in the registry.

NOTE

When you are connected to the lab environment using a remote desktop or guest-based connection, you might experience frequent, unexpected disconnects. One possible cause of the loss of the connection is that the machine is configured to automatically log onto the network.

5. To make sure that the screen saver is disabled because this might interfere with any automated tests that must interact with the desktop, select **Ensure screen saver is disabled**.

WARNING

There are security risks if you log on automatically or disable the screen saver. By enabling automatic log on, you enable other users to start this computer and to be able to use the account that automatically logs on. If you disable the screen saver, the computer might not prompt for a user to log on to unlock the computer. This enables anyone to access the computer if they have physical access to the computer. If you enable these features on a computer, you should make sure that these computers are physically secure. For example, these computers are located in a physically secure lab. If you clear **Ensure screen saver is disabled**, this does not enable your screen saver.

To change the agent back to run as a service, you can use this tool and select **Service**.

6. To apply your changes, choose **Apply Settings**.

A **Configuration summary** dialog box is displayed that shows the status of each of the steps to configure your test agent.

7. To close the **Configuration summary** dialog box, choose **Close**. Then choose **Close** again to close the **Test Agent Configuration Tool**.

NOTE

There is a notification area icon that runs on the computer for a test agent that is running as a process. It shows the status of the test agent. You can start, stop or restart the agent if it is running as a process using this tool. To start the test agent as a process if it is not running, choose **Start > Visual Studio > Microsoft Visual Studio Test Agent**.

If the test controller for this test agent is registered with Team Foundation Server, the status of a test agent that is running as an interactive process is displayed in the **Controllers** view in the **Lab Center** for Microsoft Test Manager. It is listed with a preceding asterisk symbol to denote that it is running as an interactive process. To restart this test agent, you must use the tool that runs on the computer for the test agent and not the **Controllers** view.

See also

- [Install and configure test agents](#)

How to: Specify timeout periods for test controllers and test agents

1/1/2020 • 3 minutes to read • [Edit Online](#)

Both the test controller and the test agent have several timeout settings that specify how long they should wait for responses from one another, or from a data source before failing with an error. Under certain circumstances, it might be necessary to edit the timeout values to meet the needs of your topology or other environment issues. To edit the timeout values, edit the XML configuration file that is associated with either the test controller or test agent, as covered in the following procedures.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To edit a test controller or a test agent's various timeout settings, modify the following configuration files using the key names and values in the tables:

- Test controller: *QTController.exe.config*

KEY NAME	DESCRIPTION	VALUE
AgentConnectionTimeoutInSeconds	Number of seconds to wait for agent ping request before connection is considered lost.	"n" seconds.
AgentSyncTimeoutInSeconds	When you start a synchronizing test run, number of seconds to wait for all agents to sync before aborting the run.	"n" seconds.
AgentInitializeTimeout	Number of seconds to wait for all agents and their data collectors to initialize at the beginning of a test run, before aborting the test run. This value should be reasonably large if using data collectors.	"n" seconds. Default: "120" (two minutes).
AgentCleanupTimeout	Number of seconds to wait for all agents and their data collectors to clean up, before completing the test run. This value should be reasonably large if using data collectors.	"n" seconds. Default: "120" (two minutes).

- Test Agent: *QTAgentService.exe.config*

KEY NAME	DESCRIPTION	VALUE
ControllerConnectionPeriodInSeconds	Number of seconds between attempts to connect to the controller.	"n" seconds. Default: "30" (thirty seconds).

Key Name	Description	Value
RemotingTimeoutSeconds	Maximum time a remoting call can last in seconds.	"n" seconds. Default: "600" (ten minutes).
StopTestRunCallTimeoutInSeconds	Number of seconds to wait for call to stop the test run.	"n" seconds. Default: "120" (two minutes).
GetCollectorDataTimeout	Number of seconds to wait for the data collector.	"n" seconds. Default: "300" (five minutes).

To specify agent timeout options for a test controller

1. Open the *QTCcontroller.exe.config* XML configuration file located in %ProgramFiles(x86)%\Microsoft Visual Studio\2017\Enterprise\Common7\IDE.
2. Locate `<appSettings>` tag.

```
<appSettings>
<add key="LogSizeLimitInMegs" value="20"/>
<add key="AgentConnectionTimeoutInSeconds" value="120"/>
<add key="AgentSyncTimeoutInSeconds" value="300"/>
<add key="ControllerServicePort" value="6901"/>
<add key="ControllerUsersGroup" value="TeamTestControllerUsers"/>
<add key="ControllerAdminsGroup" value="TeamTestControllerAdmins"/>
<add key="CreateTraceListener" value="no"/>
</appSettings>
```

3. Edit an existing value for one of the test controller's timeout keys. For example, you can change the default value for the key `AgentConnectionTimeoutInSeconds` from two minutes to three minutes:

```
<add key="AgentConnectionTimeoutInSeconds" value="180"/>
```

-or-

Add an additional key and specify a timeout value. For example, you can add the `AgentInitializeTimeout` key in the `<appSettings>` section and specify a value of five minutes:

```
<appSettings>
<add key="AgentInitializeTimeout" value="300"/>
</appSettings>
```

To specify agent timeout options for a test agent

1. Open the *QTAgentService.exe.config* XML configuration file located in %ProgramFiles(x86)%\Microsoft Visual Studio\2017\Enterprise\Common7\IDE.
2. Locate `<appSettings>` tag.

```
<appSettings>
  <appSettings>
    <add key="LogSizeLimitInMegs" value="20"/>
    <add key="AgentServicePort" value="6910"/>
    <add key="ControllerConnectionPeriodInSeconds" value="30"/>
    <add key="StopTestRunCallTimeoutInSeconds" value="120"/>
    <add key="CreateTraceListener" value="no"/>
    <add key="GetCollectorDataTimeout" value="300"/>
  </appSettings>  </appSettings>
```

3. Edit an existing value for one of the test agent's timeout keys. For example, you can change the default value for the key `ControllerConnectionPeriodInSeconds` from thirty seconds to one minute:

```
<add key="ControllerConnectionPeriodInSeconds" value="60"/>
```

-or-

Add an additional key and specify a timeout value. For example, you can add the `RemotingTimeoutSeconds` key in the `<appSettings>` section and specify a value of fifteen minutes:

```
<appSettings>
  <add key=" RemotingTimeoutSeconds " value="900"/>
</appSettings>
```

See also

- [Install and configure test agents](#)
- [Modify load test logging settings](#)
- [Configure ports for test controllers and test agents](#)
- [How to: Bind a test controller or test agent to a network adapter](#)

Strategies for troubleshooting test controllers and test agents in load tests

1/1/2020 • 4 minutes to read • [Edit Online](#)

This article covers some common problems you might encounter when you work with test controllers and test agents in Visual Studio.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Unable to collect performance counters on test agent computer

When you run a load test, you might receive errors when you try to connect to a test agent computer and collect performance counters. The Remote Registry service is the service responsible for providing performance counter data to a remote computer. On some operating systems, the Remote Registry service does not start automatically. To fix this problem, manually start the Remote Registry service.

NOTE

You can access the Remote Registry service in **Control Panel**. Choose **Administrative Tools** and then choose **Services**.

Another cause of this problem is that you do not have sufficient permissions to read performance counters. For local test runs, the account of the user who is running the test must be a member of the Power Users group or higher, or be a member of the Performance Monitor Users group. For remote test runs, the account that the controller is configured to run as must be a member of the Power Users group or higher, or be a member of the Performance Monitor Users group.

Set the logging level on a test controller computer

You can control the level of logging on a test controller computer. This is useful when you are trying to diagnose a problem when you are running a load test on an environment.

To set the logging level on a test controller computer

1. Stop the test controller service. At a command prompt, type `net stop vsttcontroller`.
2. Open the file `QTController.exe.config`. This file is located in the controller installation directory.
3. Edit the entry for the `EqtTraceLevel1` switch in the system diagnostics section of the file. Your code should resemble this:

```

<system.diagnostics>
  <trace autoflush="true" indentsize="4">
    <listeners>
      <add name="myListener" type="System.Diagnostics.TextWriterTraceListener"
initializeData="d:\VSTestHost.log" />
    </listeners>
  </trace>
  <switches>
    <!-- You must use integral values for "value": 
        0 = off,
        1 = error,
        2 = warn,
        3 = info,
        4 = verbose. -->
    <add name="EqtTraceLevel" value="4" />
  </switches>
</system.diagnostics>

```

4. Save the file.

5. Start the controller service. At a command prompt, type `net start vsttcontroller`.

This applies to the test controller, the test agent service, and the test agent process. When diagnosing problems, it is helpful to enable logging on all three processes. The procedure to set the logging level is the same for all three processes, as specified earlier for the test controller. To set the logging levels for the test agent service and the agent process, use the following configuration files:

- *QTController.exe.config* Controller service
- *QTAgentService.exe.config* Agent service
- *QTDAgent(32).exe.config* Agent data adapter process for 32-bit architecture.
- *QTDAgent(64).exe.config* Agent data adapter process for 64-bit architecture.
- *QTAgent(32).exe.config* Agent test process for 32-bit architecture.
- *QTAgent(64).exe.config* Agent test process for 64-bit architecture.

Bind a test controller to a network adapter

When you try to set up a test agent, you might receive the following error:

Error 8110. Cannot connect to the specified controller computer or access the controller object.

This error can be caused by installing the test controller on a computer that has more than one network adapter.

NOTE

It is also possible to install test agents successfully, and not see this problem until you try to run a test.

To fix this error, you must bind the test controller to one of the network adapters. You have to set the `BindTo` property on the test controller, and then change the test agent to refer to the test controller by IP address instead of by name. The steps are provided in the following procedures.

To obtain the IP address of the network adapter

1. Choose **Start**, and then choose **Run**.

The **Run** dialog box is displayed.

2. Type `cmd` and then choose **OK**.

A command prompt opens.

3. Type `ipconfig /all`.

The IP addresses for your network adapters are displayed. Record the IP address of the network adapter that you want to bind your controller to.

To bind a test controller to a network adapter

1. Stop the test controller service. At a command prompt, type `net stop vsttcontroller`.
2. Open the file `QTController.exe.config`. This file is located in `%ProgramFiles(x86)%\Microsoft Visual Studio\2017\Enterprise\Common7\IDE`.
3. Add an entry for the `BindTo` property to the application settings. Specify the IP address of the network adapter that you want to bind the controller to. Your code should resemble this:

```
<appSettings>
    <add key="LogSizeLimitInMegs" value="20" />
    <add key="AgentSyncTimeoutInSeconds" value="120" />
    <add key="ControllerServicePort" value="6901" />
    <add key="ControllerUsersGroup" value="TeamTestControllerUsers" />
    <add key="ControllerAdminsGroup" value="TeamTestControllerAdmins" />
    <add key="CreateTraceListener" value="no" />
    <add key="BindTo" value="<YOUR IP ADDRESS>" />
</appSettings>
```

4. Save the file.
5. Start the test controller service. At a command prompt, type `net start vsttcontroller`.

To connect a test agent to a bound controller

- Run the test agent installation again. This time, specify the IP address for the test controller instead of the test controller name.

This applies to the test controller, the test agent service, and the test agent process. The `BindTo` property must be set for each process that is running on a computer that has more than one network adapter. The procedure to set the `BindTo` property is the same for all three processes, as specified earlier for the test controller. To set the logging levels for the test agent service and the test agent process, use the configuration files that are listed in [Set the logging level on a test controller computer](#).

See also

- [Test controllers and test agents](#)

How to: Use the load test API

1/1/2020 • 2 minutes to read • [Edit Online](#)

Visual Studio supports load test plug-ins which can control or enhance a load test. Load test plug-ins are user defined classes which implement the [ILoadTestPlugin](#) interface found in the [Microsoft.VisualStudio.TestTools.LoadTesting](#) namespace. Load test plug-ins allow for custom load test control, such as, aborting a load test when a counter or error threshold is met. Use the properties on the [LoadTest](#) class to get or set load test parameters from user defined code. Use the events on the [LoadTest](#) class to attach delegates for notifications when the load test is running.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

TIP

Use the object browser to examine the [Microsoft.VisualStudio.TestTools.LoadTesting](#) namespace. Both the Visual C# and Visual Basic editors offer IntelliSense support for coding with the classes in the namespace.

You can also create plug-ins for web performance tests. For more information, see [How to: Create a web performance test plug-in](#) and [How to: Create a request-level plug-in](#).

To use the LoadTesting namespace

1. Open a web performance and load test project that contains a load test.
2. Add a Visual C# or a Visual Basic class library project to your test solution.
3. Add a reference in the web performance and load test project to the class library project.
4. Add a reference to the [Microsoft.VisualStudio.QualityTools.LoadTestFramework](#) DLL in the Class Library project.
5. In the class file located in the class library project, add a `using` statement for the [Microsoft.VisualStudio.TestTools.LoadTesting](#) namespace.
6. Create a public class that implements the [ILoadTestPlugin](#) interface.
7. Build the project.
8. Add the new load test plug-in using the Load Test Editor:
 - a. Right-click the root node of the load test and then choose **Add Load Test Plug-in**.
 - b. The **Add Load Test Plug-in** dialog box is displayed.
 - c. In the **Properties for selected plug-in** pane, set the initial values for the plug-in to use at run time.

NOTE

You can expose as many properties as you want from your plug-ins. Just make them public, settable, and of a base type such as Integer, Boolean, or String. You can also edit the load test plug-in properties later using the **Properties** window.

9. Run your Load test.

For an implementation of [ILoadTestPlugin](#), see [How to: Create a load test plug-in](#).

See also

- [Microsoft.VisualStudio.TestTools.LoadTesting](#)
- [Create custom code and plug-ins for load tests](#)
- [How to: Use the web performance test API](#)
- [How to: Create a load test plug-in](#)

How to: Use the web performance test API

1/1/2020 • 2 minutes to read • [Edit Online](#)

You can write code for your web performance tests. The web performance test API is used to create coded web performance tests, web performance test plug-ins, request plug-ins, requests, extraction rules, and validation rules. The classes that make up these types are the core classes in this API. The other types in this API are used to support creating [WebTest](#), [WebTestPlugin](#), [WebTestRequestPlugin](#), [WebTestRequest](#), [ExtractionRule](#), and [ValidationRule](#) objects. You use the [Microsoft.VisualStudio.TestTools.WebTesting](#) namespace to create customized web performance tests.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

You can also use the web performance test API to programmatically create and save declarative web performance tests. To do this, use the [DeclarativeWebTest](#) and [DeclarativeWebTestSerializer](#) classes.

TIP

Use the object browser to examine the [Microsoft.VisualStudio.TestTools.WebTesting](#) namespace. Both the Visual C# and Visual Basic editors offer IntelliSense support for coding with the classes in the namespace.

You can also create plug-ins for load tests. For more information, see [How to: Use the load test API](#) and [How to: Create a load test plug-in](#).

To use the WebTesting namespace

1. Open a web performance and load test project that contains a web performance test.
2. Add a Visual C# or a Visual Basic class library project to your test solution.
3. Add a reference in the web performance and load test project to the class library project.
4. Add a reference to the [Microsoft.VisualStudio.QualityTools.WebTestFramework](#) DLL in the class library project.
5. In the class file that is located in the class library project, add a `using` statement for the [Microsoft.VisualStudio.TestTools.WebTesting](#) namespace.
6. Create a class that implements the [WebTestPlugin](#) interface.
7. Build the project.
8. Add the new web performance test plug-in by using the Web Performance Test Editor:
 - a. Choose **Add Web Test Plug-in** on the toolbar.
The **Add Web Test Plug-in** dialog box is displayed.
 - b. Under **Select a plug-in**, select your web performance test plug-in class.
 - c. In the **Properties for selected plug-in** pane, set the initial values for the plug-in to use at run time.

NOTE

You can expose as many properties as you want from your plug-ins; just make them public, settable, and of a base type such as Integer, Boolean, or String. You can also edit the web performance test plug-in properties later by using the Properties window.

- d. Choose **OK**.
9. Run your web performance test.

For an example implementation of [WebTestPlugin](#), see [How to: Create a web performance test plug-in](#).

See also

- [Microsoft.VisualStudio.TestTools.WebTesting](#)
- [Create custom code and plug-ins for load tests](#)
- [How to: Use the load test API](#)
- [How to: Create a web performance test plug-in](#)

Code a custom extraction rule for a web performance test

1/1/2020 • 5 minutes to read • [Edit Online](#)

You can create your own extraction rules. To do this, you derive your own rules from an extraction rule class. Extraction rules derive from the [ExtractionRule](#) base class.

NOTE

You can also create custom validation rules. For more information, see [Create custom code and plug-ins for load tests](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To create a custom extraction rule

1. Open a Test project that contains a web performance test.
2. (Optional) Create a separate Class library project in which to store your extraction rule.

IMPORTANT

You can create the class in the same project that your tests are in. However, if you want to reuse the rule, it is better to create a separate Class library project in which to store your rule. If you create a separate project, you must complete the optional steps in this procedure.

3. (Optional) In the Class library project, add a reference to the Microsoft.VisualStudio.QualityTools.WebTestFramework.dll.
4. Create a class that derives from the [ExtractionRule](#) class. Implement the [Extract](#) and [RuleName](#) members.
5. (Optional) Build the new Class library project.
6. (Optional) In the Test project, add a reference to the Class library project that contains the custom extraction rule.
7. In the Test project, open a web performance test in the **Web Performance Test Editor**.
8. To add the custom extraction rule, right-click a web performance test request and select **Add Extraction Rule**.

The **Add Extraction Rule** dialog box appears. You will see your custom validation rule in the **Select a rule** list, together with the predefined validation rules. Select your custom extraction rule and then choose **OK**.

9. Run your web performance test.

Example

The following code shows an implementation of a custom extraction rule. This extraction rule extracts the value

from a specified input field. Use this example as a starting point for your own custom extraction rules.

```
using System;
using System.Collections.Generic;
using Microsoft.VisualStudio.TestTools.WebTesting;
using System.Globalization;

namespace ClassLibrary2
{
    //-----
    // This class creates a custom extraction rule named "Custom Extract Input"
    // The user of the rule specifies the name of an input field, and the
    // rule attempts to extract the value of that input field.
    //-----
    public class CustomExtractInput : ExtractionRule
    {
        /// Specify a name for use in the user interface.
        /// The user sees this name in the Add Extraction dialog box.
        //-----
        public override string RuleName
        {
            get { return "Custom Extract Input"; }
        }

        /// Specify a description for use in the user interface.
        /// The user sees this description in the Add Extraction dialog box.
        //-----
        public override string RuleDescription
        {
            get { return "Extracts the value from a specified input field"; }
        }

        // The name of the desired input field
        private string NameValue;
        public string Name
        {
            get { return NameValue; }
            set { NameValue = value; }
        }

        // The Extract method. The parameter e contains the web performance test context.
        //-----
        public override void Extract(object sender, ExtractionEventArgs e)
        {
            if (e.Response.HtmlDocument != null)
            {
                foreach (HtmlTag tag in e.Response.HtmlDocument.GetFilteredHtmlTags(new string[] { "input" }))
                {
                    if (String.Equals(tag.GetAttributeValueAsString("name"), Name,
StringComparison.InvariantCultureIgnoreCase))
                    {
                        string formFieldValue = tag.GetAttributeValueAsString("value");
                        if (formFieldValue == null)
                        {
                            formFieldValue = String.Empty;
                        }

                        // add the extracted value to the web performance test context
                        e.WebTest.Context.Add(this.ContextParameterName, formFieldValue);
                        e.Success = true;
                        return;
                    }
                }
            }
            // If the extraction fails, set the error text that the user sees
            e.Success = false;
            e.Message = String.Format(CultureInfo.CurrentCulture, "Not Found: {0}", Name);
        }
    }
}
```

```
    }
}
```

```
Imports System
Imports System.Collections.Generic
Imports Microsoft.VisualStudio.TestTools.WebTesting
Imports System.Globalization

Namespace ClassLibrary2

' -----
' This class creates a custom extraction rule named "Custom Extract Input"
' The user of the rule specifies the name of an input field, and the
' rule attempts to extract the value of that input field.
' -----
Public Class CustomExtractInput
    Inherits ExtractionRule

    ' Specify a name for use in the user interface.
    ' The user sees this name in the Add Extraction dialog box.
    ' -----
    Public Overrides ReadOnly Property RuleName() As String
        Get
            Return "Custom Extract Input"
        End Get
    End Property

    ' Specify a description for use in the user interface.
    ' The user sees this description in the Add Extraction dialog box.
    ' -----
    Public Overrides ReadOnly Property RuleDescription() As String
        Get
            Return "Extracts the value from a specified input field"
        End Get
    End Property

    ' The name of the desired input field
    Private NameValue As String
    Public Property Name() As String
        Get
            Return NameValue
        End Get
        Set(ByVal value As String)
            NameValue = value
        End Set
    End Property

    ' The Extract method. The parameter e contains the web performance test context.
    ' -----
    Public Overrides Sub Extract(ByVal sender As Object, ByVal e As ExtractionEventArgs)

        If Not e.Response.HtmlDocument Is Nothing Then

            For Each tag As HtmlTag In e.Response.HtmlDocument.GetFilteredHtmlTags(New String() {"input"})

                If String.Equals(tag.GetAttributeValueAsString("name"), Name,
StringComparison.InvariantCultureIgnoreCase) Then

                    Dim formFieldValue As String = tag.GetAttributeValueAsString("value")
                    If formFieldValue Is Nothing Then

                        formFieldValue = String.Empty
                    End If

                    ' add the extracted value to the web performance test context
                    e.WebTest.Context.Add(Me.ContextParameterName, formFieldValue)
                    e.Success = True
                End If
            Next
        End If
    End Sub
End Class
```

```
        Return
    End If
    Next
End If
' If the extraction fails, set the error text that the user sees
e.Success = False
e.Message = String.Format(CultureInfo.CurrentCulture, "Not Found: {0}", Name)
End Sub
End Class
End Namespace
```

The [Extract](#) method contains the core functionality of an extraction rule. The [Extract](#) method in the previous example takes an [ExtractionEventArgs](#) that provides the response generated by the request this extraction rule covers. The response contains an [HtmlDocument](#) which contains all the tags in the response. Input tags are filtered out of the [HtmlDocument](#). Each input tag is examined for an attribute called `name` whose value equals the user supplied value of the `Name` property. If a tag with this matching attribute is found, an attempt is made to extract a value that is contained by the `value` attribute, if a value attribute exists. If it exists, the name and value of the tag are extracted and added to the web performance test context. The extraction rule passes.

See also

- [ExtractionRule](#)
- [Microsoft.VisualStudio.TestTools.WebTesting.Rules](#)
- [ExtractAttributeValue](#)
- [ExtractFormField](#)
- [ExtractHTTPHeader](#)
- [ExtractRegularExpression](#)
- [ExtractText](#)
- [ExtractHiddenFields](#)
- [Coding a custom validation rule for a web performance test](#)

Code a custom validation rule for a web performance test

1/1/2020 • 5 minutes to read • [Edit Online](#)

You can create your own validation rules. To do this, you derive your own rule class from a validation rule class. Validation rules derive from the [ValidationRule](#) base class.

NOTE

You can also create custom extraction rules. For more information, see [Create custom code and plug-ins for load tests](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To create custom validation rules

1. Open a Test Project that contains a web performance test.
2. (Optional) Create a separate Class library project in which to store your validation rule.

IMPORTANT

You can create the class in the same project that your tests are in. However, if you want to reuse the rule, it is better to create a separate Class library project in which to store your rule. If you create a separate project, you must complete the optional steps in this procedure.
3. (Optional) In the Class library project, add a reference to the Microsoft.VisualStudio.QualityTools.WebTestFramework DLL.
4. Create a class that derives from the [ValidationRule](#) class. Implement the [Validate](#) and [RuleName](#) members.
5. (Optional) Build the new Class library project.
6. (Optional) In the Test Project, add a reference to the Class library project that contains the custom validation rule.
7. In the Test Project, open a web performance test in the **Web Performance Test Editor**.
8. To add the custom validation rule to a web performance test request, right-click a request and select **Add Validation Rule**.

The **Add Validation Rule** dialog box appears. You will see your custom validation rule in the **Select a rule** list, together with the predefined validation rules. Select your custom validation rule and then choose **OK**.

9. Run your web performance test.

Example

The following code shows an implementation of a custom validation rule. This validation rule mimics the behavior

of the predefined Required Tag validation rule. Use this example as a starting point for your own custom validation rules.

WARNING

Public properties in the code for a custom validator cannot have null values.

```
using System;
using System.Diagnostics;
using System.Globalization;
using Microsoft.VisualStudio.TestTools.WebTesting;

namespace SampleWebTestRules
{
    //-----
    // This class creates a custom validation rule named "Custom Validate Tag"
    // The custom validation rule is used to check that an HTML tag with a
    // particular name is found one or more times in the HTML response.
    // The user of the rule can specify the HTML tag to look for, and the
    // number of times that it must appear in the response.
    //-----
    public class CustomValidateTag : ValidationRule
    {
        /// Specify a name for use in the user interface.
        /// The user sees this name in the Add Validation dialog box.
        //-----
        public override string RuleName
        {
            get { return "Custom Validate Tag"; }
        }

        /// Specify a description for use in the user interface.
        /// The user sees this description in the Add Validation dialog box.
        //-----
        public override string RuleDescription
        {
            get { return "Validates that the specified tag exists on the page."; }
        }

        // The name of the required tag
        private string RequiredTagNameValue;
        public string RequiredTagName
        {
            get { return RequiredTagNameValue; }
            set { RequiredTagNameValue = value; }
        }

        // The minimum number of times the tag must appear in the response
        private int MinOccurrencesValue;
        public int MinOccurrences
        {
            get { return MinOccurrencesValue; }
            set { MinOccurrencesValue = value; }
        }

        // Validate is called with the test case Context and the request context.
        // These allow the rule to examine both the request and the response.
        //-----
        public override void Validate(object sender, ValidationEventArgs e)
        {
            bool validated = false;
            int numTagsFound = 0;

            foreach (HtmlTag tag in e.Response.HtmlDocument.GetFilteredHtmlTags(RequiredTagName))
            {

```

```
        Debug.Assert(string.Equals(tag.Name, RequiredTagName,
StringComparison.InvariantCultureIgnoreCase));

        if (++numTagsFound >= MinOccurrences)
        {
            validated = true;
            break;
        }
    }

    e.IsValid = validated;

    // If the validation fails, set the error text that the user sees
    if (!validated)
    {
        if (numTagsFound > 0)
        {
            e.Message = String.Format("Only found {0} occurrences of the tag", numTagsFound);
        }
        else
        {
            e.Message = String.Format("Did not find any occurrences of tag '{0}'", RequiredTagName);
        }
    }
}
```

```
Imports System
Imports System.Diagnostics
Imports System.Globalization
Imports Microsoft.VisualStudio.TestTools.WebTesting

Namespace SampleWebTestRules

    ' This class creates a custom validation rule named "Custom Validate Tag".
    ' The custom validation rule is used to check that an HTML tag with a
    ' particular name is found one or more times in the HTML response.
    ' The user of the rule can specify the HTML tag to look for, and the
    ' number of times that it must appear in the response.
    '

Public Class CustomValidateTag
    Inherits Microsoft.VisualStudio.TestTools.WebTesting.ValidationRule

        ' Specify a name for use in the user interface.
        ' The user sees this name in the Add Validation dialog box.
        '

    Public Overrides ReadOnly Property RuleName() As String
        Get
            Return "Custom Validate Tag"
        End Get
    End Property

        ' Specify a description for use in the user interface.
        ' The user sees this description in the Add Validation dialog box.
        '

    Public Overrides ReadOnly Property RuleDescription() As String
        Get
            Return "Validates that the specified tag exists on the page."
        End Get
    End Property

        ' The name of the required tag
    Private RequiredTagNameValue As String
    Public Property RequiredTagName() As String
        Get

```

```

        Return RequiredTagNameValue
    End Get
    Set(ByVal value As String)
        RequiredTagNameValue = value
    End Set
End Property

' The minimum number of times the tag must appear in the response
Private MinOccurrencesValue As Integer
Public Property MinOccurrences() As Integer
    Get
        Return MinOccurrencesValue
    End Get
    Set(ByVal value As Integer)
        MinOccurrencesValue = value
    End Set
End Property

' Validate is called with the test case Context and the request context.
' These allow the rule to examine both the request and the response.
'-----
Public Overrides Sub Validate(ByVal sender As Object, ByVal e As ValidationEventArgs)

    Dim validated As Boolean = False
    Dim numTagsFound As Integer = 0

    For Each tag As HtmlTag In e.Response.HtmlDocument.GetFilteredHtmlTags(RequiredTagName)

        Debug.Assert(String.Equals(tag.Name, RequiredTagName,
StringComparison.InvariantCultureIgnoreCase))

        numTagsFound += 1
        If numTagsFound >= MinOccurrences Then

            validated = True
            Exit For
        End If
    Next

    e.IsValid = validated

    ' If the validation fails, set the error text that the user sees
    If Not (validated) Then
        If numTagsFound > 0 Then
            e.Message = String.Format("Only found {0} occurrences of the tag", numTagsFound)
        Else
            e.Message = String.Format("Did not find any occurrences of tag '{0}'", RequiredTagName)
        End If
    End If
End Sub
End Class
End Namespace

```

See also

- [ValidationRule](#)
- [Microsoft.VisualStudio.TestTools.WebTesting.Rules](#)
- [ValidateFormField](#)
- [ValidationRuleFindText](#)
- [ValidationRuleRequestTime](#)
- [ValidationRuleRequiredAttributeValue](#)
- [ValidationRuleRequiredTag](#)
- [Coding a custom extraction rule for a web performance test](#)

Create custom code and plug-ins for load tests

1/1/2020 • 2 minutes to read • [Edit Online](#)

A custom plug-in uses code that you write and attach to a load test or a web performance test. You can use the load test API and the web performance test API to create custom plug-ins for tests to expand to the built-in functionality. You can add multiple plug-ins to your load test.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Tasks

TASKS	ASSOCIATED TOPICS
Create a custom plug-in for your load test: You can use load test API to create a custom plug-in to add more testing functionality to your load test.	- How to: Use the load test API - How to: Create a load test plug-in
Create a custom plug-in for your Web Performance test: You can use web performance test API to create a custom plug-in to add more testing functionality to your web performance test, including at the request level. You can also create a web service test. Additionally, you can create a web recorder plug-in that can modify a web performance test after it is recorded, but before it appears in the Web Performance Test Result Viewer.	- How to: Use the web performance test API - How to: Create a web performance test plug-in - How to: Create a request-level plug-in - How to: Create a web service test - How to: Create a recorder plug-in
Add UI features to Web Performance Test Results Viewer: You can add more UI features to the Web Performance Test Results Viewer using a Visual Studio add-in.	- How to: Create a Visual Studio add-in for the web performance test results viewer
Create a custom HTTP body editor: You can create a custom editor to edit binary or string http XML responses from a web service.	- How to: Create a custom HTTP body editor for the web performance test editor

Reference

[WebTestPlugin](#)

[WebTestRequestPlugin](#)

[ILoadTestPlugin](#)

[WebTestRecorderPlugin](#)

[Microsoft.VisualStudio.TestTools.LoadTesting](#)

See also

- [Analyze load test results](#)

- Generate and run a coded web performance test

How to: Create a web performance test plug-in

1/1/2020 • 4 minutes to read • [Edit Online](#)

Web performance tests plug-ins enable you to isolate and reuse code outside the main declarative statements in your web performance test. A customized web performance test plug-in offers you a way to call some code as the web performance test is run. The web performance test plug-in is run one time for every test iteration. In addition, if you override the PreRequest or PostRequest methods in the test plug-in, those request plug-ins will run before or after each request, respectively.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

You can create a customized web performance test plug-in by deriving your own class from the [WebTestPlugin](#) base class.

You can use customized web performance test plug-ins with the web performance tests you have recorded, which enables you to write a minimal amount of code to obtain a greater level of control over your web performance tests. However, you can also use them with coded web performance tests. For more information, see [Generate and run a coded web performance test](#).

NOTE

You can also create load test plug-ins. See [How to: Create a load test plug-in](#).

To create a custom web performance test plug-in

1. Open a web performance and load test project that contains a web performance test.
2. In **Solution Explorer**, right-click on the solution and select **Add** and then choose **New Project**.
3. Create a new **Class Library** project.

The new class library project is added to **Solution Explorer** and the new class appears in the **Code Editor**.

4. In **Solution Explorer**, right-click the **References** folder in the new class library and select **Add Reference**.

The **Add Reference** dialog box is displayed.

5. Choose the **.NET** tab, scroll down, and select **Microsoft.VisualStudio.QualityTools.WebTestFramework**
6. Choose **OK**.

The reference to **Microsoft.VisualStudio.QualityTools.WebTestFramework** is added to the **Reference** folder in **Solution Explorer**.

7. In **Solution Explorer**, right-click on the top node of the web performance and load test project that contains the load test to which you want to add the web performance test plug-in and select **Add**

Reference.

8. The **Add Reference dialog box** is displayed.
9. Choose the **Projects** tab and select the **Class Library Project**.
10. Choose **OK**.
11. In the **Code Editor**, write the code of your plug-in. First, create a new public class that derives from [WebTestPlugin](#).
12. Implement code inside one or more of the event handlers. See the following Example section for a sample implementation.
 - [PostWebTestRecordingEventArgs](#)
 - [PostWebTestEventArgs](#)
 - [PreRequestEventArgs](#)
 - [PostRequestEventArgs](#)
 - [PrePageEventArgs](#)
 - [PostPageEventArgs](#)
 - [PreTransactionEventArgs](#)
 - [PostTransactionEventArgs](#)
13. After you have written the code, build the new project.
14. Open a web performance test.
15. To add the web performance test plug-in, choose **Add Web Test Plug-in** on the toolbar.

The **Add Web Test Plug-in** dialog box is displayed.

16. Under **Select a plug-in**, select your web performance test plug-in class.
17. In the **Properties for selected plug-in** pane, set the initial values for the plug-in to use at run time.

NOTE

You can expose as many properties as you want from your plug-ins; just make them public, settable, and of a base type such as Integer, Boolean, or String. You can also change the web performance test plug-in properties later by using the Properties window.

18. Choose **OK**.

The plug-in is added to the **Web Test Plug-ins** folder.

WARNING

You might get an error similar to the following when you run a web performance test or load test that uses your plug-in:

Request failed: Exception in <plug-in> event: Could not load file or assembly '<"Plug-in name".dll file>, Version=<n.n.n.n>, Culture=neutral, PublicKeyToken=null' or one of its dependencies. The system cannot find the file specified.

This is caused if you make code changes to any of your plug-ins and create a new DLL version (**Version=0.0.0.0**), but the plug-in is still referencing the original plug-in version. To correct this problem, follow these steps:

1. In your web performance and load test project, you will see a warning in references. Remove and re-add the reference to your plug-in DLL.
2. Remove the plug-in from your test or the appropriate location and then add it back.

Example

The following code creates a customized web performance test plug-in that adds an item to the [WebTestContext](#) that represents the test iteration.

After running the web performance test, by using this plug-in you can see the added item that is named **TestIterationNumber** in the **Context** tab in the [Web Performance Results Viewer](#).

```

using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
using Microsoft.VisualStudio.TestTools.WebTesting;

namespace SampleRules
{
    [Description("This plugin can be used to set the ParseDependentsRequests property for each request")]
    public class SampleWebTestPlugin : WebTestPlugin
    {
        private bool m_parseDependents = true;

        public override void PreWebTest(object sender, PreWebTestEventArgs e)
        {
            // TODO: Add code to execute before the test.
        }

        public override void PostWebTest(object sender, PostWebTestEventArgs e)
        {
            // TODO: Add code to execute after the test.
        }

        public override void PreRequest(object sender, PreRequestEventArgs e)
        {
            // Code to execute before each request.
            // Set the ParseDependentsRequests value on the request
            e.Request.ParseDependentRequests = m_parseDependents;
        }

        // Properties for the plugin.
        [DefaultValue(true)]
        [Description("All requests will have their ParseDependentsRequests property set to this value")]
        public bool ParseDependents
        {
            get
            {
                return m_parseDependents;
            }
            set
            {
                m_parseDependents = value;
            }
        }
    }
}

```

See also

- [WebTestRequestPlugin](#)
- [Create custom code and plug-ins for load tests](#)
- [How to: Create a request-level plug-in](#)
- [Code a custom extraction rule for a web performance test](#)
- [Code a custom validation rule for a web performance test](#)
- [How to: Create a load test plug-in](#)
- [Generate and run a coded web performance test](#)

How to: Create a request-level plug-in

1/1/2020 • 4 minutes to read • [Edit Online](#)

Requests are the declarative statements that constitute web performance tests. Web performance test plug-ins enable you to isolate and reuse code outside the main declarative statements in your web performance test. You can create plug-ins and add them to an individual request as well as to the web performance test that contains it. A customized *request plug-in* offers you a way to call code as a particular request is run in a web performance test.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Every web performance test request plug-in has a `PreRequest` method and a `PostRequest` method. After you attach a request plug-in to a particular http request, the `PreRequest` event will be fired before the request is issued and the `PostRequest` fired after the response is received.

You can create a customized web performance test request plug-in by deriving your own class from the [WebTestRequestPlugin](#) base class.

You can use customized web performance test request plug-ins with the web performance tests you have recorded. Customized web performance test request plug-ins enable you to write a minimal amount of code to attain a greater level of control over your web performance tests. However, you can also use them with coded web performance tests. See [Generate and run a coded web performance test](#).

To create a request-level plug-in

1. In **Solution Explorer**, right-click the solution, select **Add** and then choose **New Project**.
2. Create a new **Class Library** project.
3. In **Solution Explorer**, right-click the **References** folder in the new class library and select **Add Reference**.

The **Add Reference** dialog box is displayed.

4. Choose the **.NET** tab, scroll down, select **Microsoft.VisualStudio.QualityTools.WebTestFramework** and then choose **OK**

The reference to **Microsoft.VisualStudio.QualityTools.WebTestFramework** is added to the **Reference** folder in **Solution Explorer**.

5. In **Solution Explorer**, right-click the top node of the web performance and load test project that contains the load test to which you want to add the web performance test request test plug-in. Select **Add Reference**.

The **Add Reference dialog box is displayed**.

6. Choose the **Projects** tab, select the **Class Library Project** and then choose **OK**.
7. In the **Code Editor**, write the code of your plug-in. First, create a new public class that derives from [WebTestRequestPlugin](#).
8. Implement code inside one or both of the [PreRequest](#) and [PostRequest](#) event handlers. See the following Example section for a sample implementation.

9. After you have written the code, build the new project.
 10. Open the web performance test to which you want to add the request plug-in.
 11. Right-click the request to which you want to add the request plug-in, and select **Add Request Plug-in**.
- The **Add Web Test Request Plug-in** dialog box is displayed.
12. Under **Select a plug-in**, select your new plug-in.
 13. In the **Properties for selected plug-in** pane, set the initial values for the plug-in to use at run time.

NOTE

You can expose as many properties as you want from your plug-ins; just make them public, settable, and of a base type such as Integer, Boolean, or String. You can also change the web performance test plug-in properties later by using the Properties window.

14. Choose **OK**.

The plug-in is added to the **Request Plug-ins** folder, which is a child folder of the HTTP request.

WARNING

You might get an error similar to the following when you run a web performance test or load test that uses your plug-in:

Request failed: Exception in <plug-in> event: Could not load file or assembly '<"Plug-in name".dll file>, Version=<n.n.n.n>, Culture=neutral, PublicKeyToken=null' or one of its dependencies. The system cannot find the file specified.

This is caused if you make code changes to any of your plug-ins and create a new DLL version (**Version=0.0.0.0**), but the plug-in is still referencing the original plug-in version. To correct this problem, follow these steps:

1. In your web performance and load test project, you will see a warning in references. Remove and re-add the reference to your plug-in DLL.
2. Remove the plug-in from your test or the appropriate location and then add it back.

Example

You can use the following code to create a customized web performance test plug-in that displays two dialog boxes. One dialog box displays the URL that is associated with the request to which you attach the request add-in. The second dialog box displays the computer name for the agent.

NOTE

The following code requires that you add a reference to System.Windows.Forms.

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using Microsoft.VisualStudio.TestTools.WebTesting;

namespace RequestPluginNamespace
{
    public class MyWebRequestPlugin : WebTestRequestPlugin
    {
        public override void PostRequest(object sender, PostRequestEventArgs e)
        {
            MessageBox.Show(e.WebTest.Context.AgentName);
        }
        public override void PreRequest(object sender, PreRequestEventArgs e)
        {
            MessageBox.Show(e.Request.Url);
        }
    }
}
```

See also

- [WebTestRequestPlugin](#)
- [Create custom code and plug-ins for load tests](#)
- [Code a custom extraction rule for a web performance test](#)
- [Code a custom validation rule for a web performance test](#)
- [How to: Create a load test plug-in](#)
- [Generate and run a coded web performance test](#)

How to: Create a load test plug-in

1/1/2020 • 4 minutes to read • [Edit Online](#)

You can create a load test plug-in to run code at different times while the load test is running. You create a plug-in to expand upon or modify the built in functionality of the load test. For example, you can code a load test plug-in to set or modify the load test pattern while the load test is running. To do this, you must create a class that inherits the [ILoadTestPlugin](#) interface. This class must implement the [Initialize](#) method of this interface. For more information, see [ILoadTestPlugin](#).

TIP

You can also create plug-ins for web performance tests. For more information, see [How to: Create a web performance test plug-in](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To create a load test plug-in in C#

1. Open a web performance and load test project that contains a web performance test.
2. Add a load test to the test project and configure it to run a web performance test.

For more information, see [Quickstart: Create a load test project](#).

3. Add a new **Class Library** project to the solution. (In **Solution Explorer**, right-click on the solution and select **Add** and then choose **New Project**.)
4. In **Solution Explorer**, right-click the **References** folder in the new class library and select **Add Reference**.

The **Add Reference** dialog box is displayed.

5. Choose the **.NET** tab, scroll down, and then select **Microsoft.VisualStudio.QualityTools.LoadTestFramework**.
6. Choose **OK**.

The reference to **Microsoft.VisualStudio.QualityTools.LoadTestFramework** is added to the **Reference** folder in **Solution Explorer**.

7. In **Solution Explorer**, right-click the top node of the web performance and load test project that contains the load test to which you want to add the load test plug-in and select **Add Reference**.

The **Add Reference dialog box is displayed**.

8. Choose the **Projects** tab and select the Class Library Project.
9. Choose **OK**.
10. In the **Code Editor**, add a `using` statement for the **Microsoft.VisualStudio.TestTools.LoadTesting**

namespace.

11. Implement the [ILoadTestPlugin](#) interface for the class that was created in the Class Library project. See the following Example section for a sample implementation.
 12. After you have written the code, build the new project.
 13. Right-click on the top node of the load test and then choose **Add Load Test Plug-in**.
- The **Add Load Test Plug-in** dialog box is displayed.
14. Under **Select a plug-in**, select your load test plug-in class.
 15. In the **Properties for selected plug-in** pane, set the initial values for the plug-in to use at run time.

NOTE

You can expose as many properties as you want from your plug-ins; just make them public, settable, and of a base type such as Integer, Boolean, or String. You can also change the web performance test plug-in properties later by using the **Properties** window.

16. Choose **OK**.

The plug-in is added to the **Load Test Plug-ins** folder.

WARNING

You might get an error similar to the following when you run a web performance test or load test that uses your plug-in:

Request failed: Exception in <plug-in> event: Could not load file or assembly '<"Plug-in name".dll file>, Version=<n.n.n.n>, Culture=neutral, PublicKeyToken=null' or one of its dependencies. The system cannot find the file specified.

This is caused if you make code changes to any of your plug-ins and create a new DLL version (**Version=0.0.0.0**), but the plug-in is still referencing the original plug-in version. To correct this problem, follow these steps:

1. In your web performance and load test project, you will see a warning in references. Remove and re-add the reference to your plug-in DLL.
2. Remove the plug-in from your test or the appropriate location and then add it back.

Example

The following code shows a load test plug-in that runs custom code after a LoadTestFinished event occurs. If this code is run on a test agent on a remote machine and the test agent does not have a localhost SMTP service, the load test will remain in the "In progress" state because a message box will be open.

NOTE

The following code requires that you add a reference to System.Windows.Forms.

```

using System;
using Microsoft.VisualStudio.TestTools.LoadTesting;
using System.Net.Mail;
using System.Windows.Forms;

namespace LoadTestPluginTest
{
    public class MyLoadTestPlugin : ILoadTestPlugin
    {
        LoadTest myLoadTest;

        public void Initialize(LoadTest loadTest)
        {
            myLoadTest = loadTest;
            myLoadTest.LoadTestFinished += new
                EventHandler(myLoadTest_LoadTestFinished);
        }

        void myLoadTest_LoadTestFinished(object sender, EventArgs e)
        {
            try
            {
                // place custom code here
                MailAddress MyAddress = new MailAddress("someone@example.com");
                MailMessage MyMail = new MailMessage(MyAddress, MyAddress);
                MyMail.Subject = "Load Test Finished -- Admin Email";
                MyMail.Body = myLoadTest..Name + " has finished.";

                SmtpClient MySmtpClient = new SmtpClient("localhost");
                MySmtpClient.Send(MyMail);
            }

            catch (SmtpException ex)
            {
                MessageBox.Show(ex.InnerException.Message +
                    ".\r\nMake sure you have a valid SMTP.", "LoadTestPlugin", MessageBoxButtons.OK,
                MessageBoxIcon.Warning, MessageBoxDefaultButton.Button1);
            }
        }
    }
}

```

Eight events are associated with a load test that can be handled in the load test plug-in to run custom code with the load test. The following is a list of the events that provide access to different periods of the load test run:

- [LoadTestStarting](#)
- [LoadTestFinished](#)
- [LoadTestWarmupComplete](#)
- [TestStarting](#)
- [TestFinished](#)
- [ThresholdExceeded](#)
- [Heartbeat](#)
- [LoadTestAborted](#)

See also

- [ILoadTestPlugin](#)

- [Create custom code and plug-ins for load tests](#)
- [How to: Create a Web performance test plug-in](#)

How to: Create a recorder plug-in

1/1/2020 • 6 minutes to read • [Edit Online](#)

The [WebTestRecorderPlugin](#) lets you modify a recorded web performance test. The modification occurs after you choose **Stop** in the **Web Performance Test Recorder** toolbar but prior to the test being saved and presented in the Web Performance Test Editor.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

A recorder plug-in enables you to perform your own custom correlation on dynamic parameters. With the built-in correlation functionality, web performance tests detect the dynamic parameters in the web recording upon completion, or when you use the **Promote Dynamic Parameters to Web Test Parameters** on the **Web Performance Test Editor** toolbar. However, the built in detection functionality does not always find all the dynamic parameters. For example, it does not find a session ID, which usually gets its value changed between 5 to 30 minutes. Therefore, you have to manually perform the correlation process.

The [WebTestRecorderPlugin](#) lets you write code for your own custom plug-in. This plug-in can perform correlation or modify the web performance test in many ways prior to it being saved and presented in the Web Performance Test Editor. Therefore, if you determine that a specific dynamic variable has to be correlated for a lot of your recordings, you can automate the process.

Some other ways that a recorder plug-in can be used is for adding extraction and validation rules, adding context parameters, or converting comments to transactions in a web performance test.

The following procedures describe how to create the rudimentary code for a recorder plug-in, deploy the plug-in and execute the plug-in. The sample code following the procedures demonstrates how to use Visual C# to create a custom dynamic parameter correlation recorder plug-in.

Create a recorder plug-in

To create a recorder plug-in

1. Open a solution that contains the web performance and load test project with the web performance test for which you want to create a recorder plug-in.
2. Add a new **Class Library** project to the solution.
3. In **Solution Explorer**, in the new class library project folder, right-click the **References** folder and select **Add Reference**.

TIP

An example of a new class library project folder is **RecorderPlugins**.

The **Add Reference** dialog box is displayed.

4. Select the **.NET** tab.
5. Scroll down and select **Microsoft.VisualStudio.QualityTools.WebTestFramework** and then choose **OK**.

The **Microsoft.VisualStudio.QualityTools.WebTestFramework** is added in the **References** folder in **Solution Explorer**.

6. Write the code for your recorder plug-in. First, create a new public class that derives from [WebTestRecorderPlugin](#).
7. Override the [PostWebTestRecording](#) method.

```
public class Class1 : WebTestRecorderPlugin
{
    public override void PostWebTestRecording(object sender, PostWebTestRecordingEventArgs e)
    {
        base.PostWebTestRecording(sender, e);
    }
}
```

The event arguments will give you two objects to work with: the recorded result and the recorded web performance test. This will allow you to iterate through the result looking for certain values and then jump to the same request in the web performance test to make modifications. You can also just modify the web performance test if you wanted to add a context parameter or parameterize parts of the URL.

NOTE

If you do modify the web performance test, you will also need to set the [RecordedWebTestModified](#) property to true:

```
e.RecordedWebTestModified = true;
```

8. Add more code according to what you want the recorder plug-in to execute after the web recording occurs. For example, you can add code to handle custom correlation as shown in the sample below. You can also create a recorder plug-in for such things as converting comments to transactions or adding validation rules to the web performance test.
9. On the **Build** menu, choose **Build <class library project name>**.

Next, deploy the recorder plug-in in order for it to register with Visual Studio.

Deploy the recorder plug-in

After you compile the recorder plug-in, place the resulting DLL in one of two locations:

- *%ProgramFiles(x86)%\Microsoft Visual Studio\[version]\[edition]\Common7\IDE\PrivateAssemblies\WebTestPlugins*
- *%USERPROFILE%\Documents\Visual Studio [version]\WebTestPlugins*

WARNING

After you copy the recorder plug-in to one of the two locations, you must restart Visual Studio for the recorder plug-in to be registered.

Execute the recorder plug-in

1. Create a new web performance test.

The **Enable WebTestRecordPlugins** dialog box displays.

2. Select the check box for the recorder plug-in and choose **OK**.

After the web performance test completes recording, the new recorder plug-in will be executed.

WARNING

You might get an error similar to the following when you run a web performance test or load test that uses your plug-in:

Request failed: Exception in <plug-in> event: Could not load file or assembly '<"Plug-in name".dll file>, Version=<n.n.n.n>, Culture=neutral, PublicKeyToken=null' or one of its dependencies. The system cannot find the file specified.

This is caused if you make code changes to any of your plug-ins and create a new DLL version (**Version=0.0.0.0**), but the plug-in is still referencing the original plug-in version. To correct this problem, follow these steps:

1. In your web performance and load test project, you will see a warning in references. Remove and re-add the reference to your plug-in DLL.
2. Remove the plug-in from your test or the appropriate location and then add it back.

Example

This sample demonstrates how to create a customized web performance test recorder plug-in to perform custom dynamic parameter correlation.

NOTE

A complete listing of the sample code is located at the bottom of this topic.

Iterate through the result to find first page with ReportSession

This part of the code sample iterates through each recorded object and searches the response body for ReportSession.

```
foreach (WebTestResultUnit unit in e.RecordedWebTestResult.Children)
{
    WebTestResultPage page = unit as WebTestResultPage;
    if (page != null)
    {
        if (!foundId)
        {
            int indexOfReportSession = page.RequestResult.Response.BodyString.IndexOf("ReportSession");
            if (indexOfReportSession > -1)
            {

```

Add an extraction rule

Now that a response has been found, you need to add an extraction rule. This part of the code sample creates the extraction rule using the [ExtractionRuleReference](#) class and then finds the correct request in the web performance test to add the extraction rule to. Each result object has a new property added called DeclarativeWebTestId which is what is being used in the code to get correct request from the web performance test.

```

ExtractionRuleReference ruleReference = new ExtractionRuleReference();
ruleReference.Type = typeof(ExtractText);
ruleReference.ContextParameterName = "SessionId";
ruleReference.Properties.Add(new PluginOrRuleProperty("EndsWith", "&ControlID="));
ruleReference.Properties.Add(new PluginOrRuleProperty("HtmlDecode", "True"));
ruleReference.Properties.Add(new PluginOrRuleProperty("IgnoreCase", "True"));
ruleReference.Properties.Add(new PluginOrRuleProperty("Index", "0"));
ruleReference.Properties.Add(new PluginOrRuleProperty("Required", "True"));
ruleReference.Properties.Add(new PluginOrRuleProperty("StartsWith", "ReportSession="));
ruleReference.Properties.Add(new PluginOrRuleProperty("UseRegularExpression", "False"));

WebTestRequest requestInWebTest = e.RecordedWebTest.GetItem(page.DeclarativeWebTestItemId) as WebTestRequest;
if (requestInWebTest != null)
{
    requestInWebTest.ExtractionRuleReferences.Add(ruleReference);
    e.RecordedWebTestModified = true;
}

```

Replace query string parameters

Now the code finds all the query string parameters that have ReportSession as name and change the value to {{SessionId}} as shown in this part of the code sample:

```

WebTestRequest requestInWebTest = e.RecordedWebTest.GetItem(page.DeclarativeWebTestItemId) as WebTestRequest;
if (requestInWebTest != null)
{
    foreach (QueryStringParameter param in requestInWebTest.QueryStringParameters)
    {
        if (param.Name.Equals("ReportSession"))
        {
            param.Value = "{{SessionId}}";
        }
    }
}

```

```

using System.ComponentModel;
using Microsoft.VisualStudio.TestTools.WebTesting;
using Microsoft.VisualStudio.TestTools.WebTesting.Rules;

namespace RecorderPlugin
{
    [DisplayName("Correlate ReportSession")]
    [Description("Adds extraction rule for Report Session and binds this to querystring parameters that use ReportSession")]
    public class CorrelateSessionId : WebTestRecorderPlugin
    {
        public override void PostWebTestRecording(object sender, PostWebTestRecordingEventArgs e)
        {
            //first find the session id
            bool foundId = false;
            foreach (WebTestResultUnit unit in e.RecordedWebTestResult.Children)
            {
                WebTestResultPage page = unit as WebTestResultPage;
                if (page != null)
                {
                    if (!foundId)
                    {
                        int indexOfReportSession =
page.RequestResult.Response.BodyString.IndexOf("ReportSession");
                        if (indexOfReportSession > -1)
                        {
                            //add an extraction rule to this request
                            // Get the corresponding request in the Declarative Web performance test

```

See also

- [WebTestRequestPlugin](#)
 - [PostWebTestRecording](#)
 - [ExtractionRuleReference](#)
 - [PostWebTestRecording](#)
 - Create custom code and plug-ins for load tests
 - Generate and run a coded web performance test

How to: Create a custom HTTP body editor for the Web Performance Test Editor

1/1/2020 • 7 minutes to read • [Edit Online](#)

You can create a custom content editor that enables you to edit the string body content or the binary body content of a web service request, for example, SOAP, REST, asmx, wcf, RIA, and other web service request types.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

You can implement these kinds of editors:

- **String content editor** This is implemented using the [IStringHttpBodyEditorPlugin](#) interface.
- **Binary content editor** This is implemented using the [IBinaryHttpBodyEditorPlugin](#) interface.

These interfaces are contained in the [Microsoft.VisualStudio.TestTools.WebTesting](#) namespace.

Create a Windows Control Library project

1. In Visual Studio, create a new **Windows Forms Control Library** project. Name the project **MessageEditors**.

The project is added to the new solution and a [UserControl](#) named *UserControl1.cs* is presented in the Designer.

2. From the **Toolbox**, under the **Common Controls** category, drag a [RichTextBox](#) onto the surface of *UserControl1*.
3. Choose the action tag glyph (□) on the upper-right corner of the [RichTextBox](#) control, and then select and **Dock in Parent Container**.
4. In **Solution Explorer**, right-click the Windows Forms Library project and select **Properties**.
5. In the **Properties**, select the **Application** tab.
6. In the **Target framework** drop-down list, select .NET Framework 4 (or later).
7. The **Target Framework Change** dialog box is displayed.
8. Choose **Yes**.
9. In **Solution Explorer**, right-click the **References** node and select **Add Reference**.
10. The **Add Reference** dialog box is displayed.
11. Choose the **.NET** tab, scroll down, and select **Microsoft.VisualStudio.QualityTools.WebTestFramework** and then choose **OK**.
12. If **View Designer** is not still open, in **Solution Explorer**, right-click **UserControl1.cs** and then select **View Designer**.
13. On the design surface, right-click and select **View Code**.

14. (Optional) Change the name of the class and the constructor from UserControl1 to a meaningful name, for example, MessageEditorControl:

NOTE

The sample uses MessageEditorControl.

```
namespace MessageEditors
{
    public partial class MessageEditorControl : UserControl
    {
        public MessageEditorControl()
        {
            InitializeComponent();
        }
    }
}
```

15. Add the following properties to enable getting and setting the text in RichTextBox1. The [IStringHttpBodyEditorPlugin](#) interface will use EditString and the [IBinaryHttpBodyEditorPlugin](#) will use EditByteArray:

```
public String EditString
{
    get
    {
        return this.richTextBox1.Text;
    }
    set
    {
        this.richTextBox1.Text = value;
    }
}

public byte[] EditByteArray
{
    get
    {
        return System.Convert.FromBase64String(this.richTextBox1.Text);
    }
    set
    {
        this.richTextBox1.Text = System.Convert.ToBase64String(value, 0, value.Length);
    }
}
```

Add a class to the Windows Control Library project

Add a class to the project. It will be used to implement the [IStringHttpBodyEditorPlugin](#) and [IBinaryHttpBodyEditorPlugin](#) interfaces.

Overview of the code in this procedure

The MessageEditorControl [UserControl](#) that was created in the previous procedure is instantiated as messageEditorControl:

```
private MessageEditorControl messageEditorControl
```

The messageEditorControl instance is hosted within the plug-in dialog that is created by the [CreateEditor](#) method. Additionally, the messageEditorControl's [RichTextBox](#) is populated with the contents in the [IHttpBody](#). However, the creation of the plug-in cannot occur unless [SupportsContentType](#) returns `true`. In the case of this editor, [SupportsContentType](#) returns `true` if the [ContentType](#) in the [IHttpBody](#) contains "xml".

When editing of the string body is completed and the user clicks **OK** in the plug-in dialog box, [GetNewValue](#) is called to get the edited text as a string and update the **String Body** in the request in the Web Test Performance Editor.

Create a class and implement the [IStringHttpBodyEditorPlugin](#) interface

1. In **Solution Explorer**, right-click the Windows Forms Control Library project and select **Add New Item**.

The **Add New Item** dialog box is displayed.

2. Select **Class**.

3. In the **Name** text box, type a meaningful name for the class, for example, `MessageEditorPlugins`.

4. Choose **Add**.

Class1 is added to the project and presented in the Code Editor.

5. In the code editor, add the following `using` statement:

```
using Microsoft.VisualStudio.TestTools.WebTesting;
```

6. Paste in the following code to implement the interface:

```

/// <summary>
/// Editor for generic text based hierarchical messages such as XML and JSON.
/// </summary>
public class XmlMessageEditor : IStringHttpBodyEditorPlugin
{
    public XmlMessageEditor()
    {
    }

    /// <summary>
    /// Determine if this plugin supports the content type.
    /// </summary>
    /// <param name="contentType">The content type to test.</param>
    /// <returns>Returns true if the plugin supports the specified content type.</returns>
    public bool SupportsContentType(string contentType)
    {
        return contentType.ToLower().Contains("xml");
    }

    /// <summary>
    /// Create a UserControl to edit the specified bytes.
    /// This control will be hosted in the
    /// plugin dialog which provides OK and Cancel buttons.
    /// </summary>
    /// <param name="contentType">The content type of the BinaryHttpBody.</param>
    /// <param name="initialValue">The bytes to edit. The bytes are the payload of a BinaryHttpBody.
    </param>
    /// <returns>A UserControl capable of displaying and editing the byte array value of the specified
    content type.</returns>
    public object CreateEditor(string contentType, string initialValue)
    {
        messageEditorControl = new MessageEditorControl();
        messageEditorControl.EditValue = initialValue;
        return this.messageEditorControl;
    }

    /// <summary>
    /// Gets the edited bytes after the OK button is clicked on the plugin dialog.
    /// </summary>
    public string GetnewValue()
    {
        return messageEditorControl.EditValue;
    }

    private MessageEditorControl messageEditorControl;
}

```

Add a IBinaryHttpBodyEditorPlugin to the class

Implement the [IBinaryHttpBodyEditorPlugin](#) interface.

Overview of the code in this procedure

The code implementation for the [IBinaryHttpBodyEditorPlugin](#) interface is similar to the [IStringHttpBodyEditorPlugin](#) covered in the previous procedure. However, the binary version uses an array of bytes to handle the binary data instead of a string.

The MessageEditorControl [UserControl](#) created in the first procedure is instantiated as messageEditorControl:

```
private MessageEditorControl messageEditorControl
```

The messageEditorControl instance is hosted within the plug-in dialog that is created by the [CreateEditor](#) method.

Additionally, the messageEditorControl's [RichTextBox](#) is populated with the contents in the [IHttpBody](#). However, the creation of the plug-in cannot occur unless [SupportsContentType](#) returns `true`. In the case of this editor, [SupportsContentType](#) returns `true` if the [ContentType](#) in the [IHttpBody](#) contains "msbin1".

When editing of the string body is completed and the user clicks **OK** in the plug-in dialog box, [GetnewValue](#) is called to get the edited text as a string and update the [BinaryHttpBody.Data](#) in the request in the Web Test Performance Editor.

To add the [IBinaryHttpBodyEditorPlugin](#) to the class

- Write or copy the following code under the [XmlMessageEditor](#) class added in the previous procedure to instantiate the [Msbin1MessageEditor](#) class from [IBinaryHttpBodyEditorPlugin](#) interface and implement the required methods:

```
/// <summary>
/// Editor for MSBin1 content type (WCF messages)
/// </summary>
public class Msbin1MessageEditor : IBinaryHttpBodyEditorPlugin
{
    /// <summary>
    ///
    /// </summary>
    public Msbin1MessageEditor()
    {
    }

    /// <summary>
    /// Determine if this plugin supports a content type.
    /// </summary>
    /// <param name="contentType">The content type to test.</param>
    /// <returns>Returns true if the plugin supports the specified content type.</returns>
    public bool SupportsContentType(string contentType)
    {
        return contentType.ToLower().Contains("msbin1");
    }

    /// <summary>
    /// Create a UserControl to edit the specified bytes. This control will be hosted in the
    /// plugin dialog which provides OK and Cancel buttons.
    /// </summary>
    /// <param name="contentType">The content type of the BinaryHttpBody.</param>
    /// <param name="initialValue">The bytes to edit. The bytes are the payload of a BinaryHttpBody.
    </param>
    /// <returns>A UserControl capable of displaying and editing the byte array value of the
    // specified content type.</returns>
    public object CreateEditor(string contentType, byte[] initialValue)
    {
        messageEditorControl = new MessageEditorControl();
        messageEditorControl.EditByteArray = initialValue;
        return messageEditorControl;
    }

    /// <summary>
    /// Gets the edited bytes after the OK button is clicked on the plugin dialog.
    /// </summary>
    public byte[] GetnewValue()
    {
        return messageEditorControl.EditByteArray;
    }

    private MessageEditorControl messageEditorControl;
    private object originalMessage;
}
```

Build and deploy the plug-ins

1. On the **Build** menu, choose **Build <Windows Form Control Library project name>**.
2. Close all instances of Visual Studio.

NOTE

Closing Visual Studio makes sure that the *.dll* file isn't locked before you try to copy it.

3. Copy the resulting *.dll* file from your project's *bin\debug* folder (for example, *MessageEditors.dll*) to
%ProgramFiles%\Microsoft Visual Studio\2017\
<edition>\Common7\IDE\PrivateAssemblies\WebTestPlugins.
4. Open Visual Studio.

The *.dll* is now registered with Visual Studio.

Verify the plug-ins using a Web Performance Test

1. Create a test project.
2. Create a web performance test and enter a URL in the browser to a web service.
3. When you finish the recording, in the Web Performance Test Editor, expand the request for the web service and select either a **String Body** or a **Binary Body**.
4. In the **Properties** window, select either String Body or Binary Body and choose the ellipsis (...).

The **Edit HTTP Body Data** dialog box is displayed.

5. You can now edit the data and choose **OK**. This invokes the applicable *GetNewValue* method to update the contents in the **IHttpBody**.

Compile the code

Verify that the Targeted framework for the Windows Control Library project is .NET Framework 4.5. By default, Windows Control Library projects target the .NET Framework 4.5 Client framework, which will not allow the inclusion of the Microsoft.VisualStudio.QualityTools.WebTestFramework reference.

For more information, see [Application page, project designer \(C#\)](#).

See also

- [IStringHttpBodyEditorPlugin](#)
- [IBinaryHttpBodyEditorPlugin](#)
- [IHttpBody](#)
- [UserControl](#)
- [RichTextBox](#)
- [Create custom code and plug-ins for load tests](#)
- [How to: Create a request-level plug-in](#)
- [Code a custom extraction rule for a web performance test](#)
- [Code a custom validation rule for a web performance test](#)
- [How to: Create a load test plug-in](#)
- [Generate and run a coded web performance test](#)

- How to: Create a Visual Studio add-in for the Web Performance Test Results Viewer

Create a diagnostic data adapter to collect custom data or affect a test machine

1/1/2020 • 2 minutes to read • [Edit Online](#)

You might want to create your own diagnostic data adapter to collect data when you run a test, or you might want to affect the test machine as part of your test. For example, you might want to collect log files that are created by your application under test and attach them to your test results, or you might want to run your tests when there is limited disk space left on your computer. Using APIs provided within Visual Studio Enterprise, you can write code to perform tasks at specific points in your test run. For example, you can perform tasks when a test run starts, before and after each individual test is run, and when the test run finishes.

You can provide default input to your custom diagnostic data adapter using a configuration settings file. For example, you can provide information about the location of the file you want to collect and attach to your test results, or how much disk space you want to be left on the system. This data can be configured for each test settings that you create. It can be displayed and edited using the default editor provided with Microsoft Test Manager or you can create your own user control to use as an editor. Any changes that are made to the adapter configuration in your editor are stored with your test settings.

If you are running your tests from Visual Studio, you must set these test settings to be active. For more information about test settings, see [Collect diagnostic information using test settings](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Tasks

Use the following topics to help you create Diagnostic Data Adapters:

TASKS	ASSOCIATED TOPICS
Creating a Diagnostic Data Adapter: You create a diagnostic data adapter by creating a class library, and then use the diagnostic data adapter APIs to collect information that you want or impact a test system that you are using to run your tests.	- How to: Create a diagnostic data adapter
Selecting a Custom Diagnostic Data Adapter to Use When Tests are Run: You can select which diagnostic data adapter to use for your test settings, so that the adapter is used when you run your tests.	- Collect diagnostic data while testing (Azure Test Plans) - Collect diagnostic data in manual tests (Azure Test Plans)

See also

- [Collect diagnostic information using test settings](#)

How to: Create a diagnostic data adapter

1/1/2020 • 8 minutes to read • [Edit Online](#)

To create a *diagnostic data adapter*, you create a class library using Visual Studio, and then add the Diagnostic Data Adapter APIs provided by Visual Studio Enterprise to your class library. Send any information that you want as a stream or a file to the [DataCollectionSink](#) provided by the framework, when handling the events that are raised during the test run. The streams or files sent to the [DataCollectionSink](#) are stored as attachments to the test results when your test finishes. If you create a bug from these test results or when you use Test Runner, the files are also linked to the bug.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

You can create a diagnostic data adapter that affects the machine where your tests are run, or a machine that is part of the environment you are using to run your application under test. For example, collecting files on your test machine where the tests are run, or collecting files on the machine serving in the web server role for your application.

You can give your diagnostic data adapter a friendly name that displays when you create your test settings using Microsoft Test Manager or using Visual Studio. Test settings enable you to define which machine role will run specific diagnostic data adapters in your environment when you run your tests. You can also configure your diagnostic data adapters when you create your test settings. For example, you may create a diagnostic data adapter that collects custom logs from your web server. When you create your test settings, you can select to run this diagnostic data adapter on the machine or machines that are performing this web server role and you can modify the configuration for your test settings to collect only the last three logs that were created. For more information about test settings, see [Collect diagnostic information using test settings](#).

Events are raised when you run your tests so that your diagnostic data adapter can perform tasks at that point in the test.

IMPORTANT

These events may be raised on different threads, especially when you have tests running on multiple machines. Therefore, you must be aware of possible threading issues and not inadvertently corrupt the internal data of the custom adapter. Make sure your diagnostic data adapter is thread safe.

The following is a partial list of key events that you can use when you create your diagnostic data adapter. For a complete list of diagnostic data adapter events, see the abstract [DataCollectionEvents](#) class.

EVENT	DESCRIPTION
SessionStart	Start of your test run
SessionEnd	End of your test run
TestCaseStart	Start of each test in the test run

EVENT	DESCRIPTION
TestCaseEnd	End of each test in the test run
TestStepStart	Start of each test step in a test
TestStepEnd	End of each test step in a test

NOTE

When a manual test is completed, no more data collection events are sent to the diagnostic data adapter. When a test is rerun, it will have a new test case identifier. If a user resets a test during a test (which raises the [TestCaseReset](#) event), or changes a test step outcome, no data collection event is sent to the diagnostic data adapter, but the test case identifier remains the same. To determine whether a test case has been reset, you must track the test case identifier in your diagnostic data adapter.

Use the following procedure to create diagnostic data adapter that collects a data file that is based on information that you configure when you create your test settings.

For a complete example diagnostic data adapter project, including a custom configuration editor, see [Sample project for creating a diagnostic data adapter](#).

Create and install a diagnostic data adapter

1. Create a new **Class Library** project.
2. Add the assembly **Microsoft.VisualStudio.QualityTools.ExecutionCommon**.
 - a. In **Solution Explorer**, right-click **References** and choose the **Add Reference** command.
 - b. Choose **.NET** and locate **Microsoft.VisualStudio.QualityTools.ExecutionCommon.dll**.
 - c. Choose **OK**.
3. Add the assembly **Microsoft.VisualStudio.QualityTools.Common**.
 - a. In **Solution Explorer**, right-click **References** and select the **Add Reference** command.
 - b. Choose .NET, locate **Microsoft.VisualStudio.QualityTools.Common.dll**.
 - c. Choose **OK**.
4. Add the following `using` directives to your class file:

```
using Microsoft.VisualStudio.TestTools.Common;
using Microsoft.VisualStudio.TestTools.Execution;
using System.Linq;
using System.Text;
using System.Xml;
using System;
```

5. Add the [DataCollectorTypeUriAttribute](#) to the class for your diagnostic data adapter to identify it as a diagnostic data adapter, replacing **Company**, **Product**, and **Version** with the appropriate information for your Diagnostic Data Adapter:

```
[DataCollectorTypeUri("datacollector://Company/Product/Version")]
```

6. Add the [DataCollectorFriendlyNameAttribute](#) attribute to the class, replacing the parameters with the appropriate information for your Diagnostic Data Adapter:

```
[DataCollectorFriendlyName("Collect Log Files", false)]
```

This friendly name is displayed in the test settings activity.

NOTE

You can also add the [DataCollectorConfigurationEditorAttribute](#) to specify the [Type](#) of your custom configuration editor for this data adapter, and to optionally specify the help file to use for the editor.

You can also apply the [DataCollectorEnabledByDefaultAttribute](#) to specify that it should always be enabled.

7. Your diagnostic data adapter class must inherit from the [DataCollector](#) class as follows:

```
public class MyDiagnosticDataAdapter : DataCollector
```

8. Add the local variables as follows:

```
private DataCollectionEvents dataEvents;  
private DataCollectionLogger dataLogger;  
private DataCollectionSink dataSink;  
private XElement configurationSettings;
```

9. Add the [Initialize](#) method and a [Dispose](#) method. In the [Initialize](#) method, you initialize the data sink, any configuration data from test settings, and register the event handlers you want to use as follows:

```

public override void Initialize(
    XElement configurationElement,
    DataCollectionEvents events,
    DataCollectionSink sink,
    DataCollectionLogger logger,
    DataCollectionEnvironmentContext environmentContext)
{
    dataEvents = events; // The test events
    dataLogger = logger; // The error and warning log
    dataSink = sink; // Saves collected data
    // Configuration from the test settings
    configurationSettings = configurationElement;

    // Register common events for the data collector
    // Not all of the events are used in this class
    dataEvents.SessionStart +=  

        new EventHandler<SessionStartEventArgs>(OnSessionStart);
    dataEvents.SessionEnd +=  

        new EventHandler<SessionEndEventArgs>(OnSessionEnd);
    dataEvents.TestCaseStart +=  

        new EventHandler<TestCaseStartEventArgs>(OnTestCaseStart);
    dataEvents.TestCaseEnd +=  

        new EventHandler<TestCaseEndEventArgs>(OnTestCaseEnd);
}

public override void Dispose(bool disposing)
{
    if (disposing)
    {
        // Unregister the registered events
        dataEvents.SessionStart -=  

            new EventHandler<SessionStartEventArgs>(OnSessionStart);
        dataEvents.SessionEnd -=  

            new EventHandler<SessionEndEventArgs>(OnSessionEnd);
        dataEvents.TestCaseStart -=  

            new EventHandler<TestCaseStartEventArgs>(OnTestCaseStart);
        dataEvents.TestCaseEnd -=  

            new EventHandler<TestCaseEndEventArgs>(OnTestCaseEnd);
    }
}

```

10. Use the following event handler code and private method to collect the log file generated during the test:

```

public void OnTestCaseEnd(sender, TestCaseEndEventArgs e)
{
    // Get any files to be collected that are
    // configured in your test settings
    List<string> files = getFilesToCollect();

    // For each of the files, send the file to the data sink
    // which will attach it to the test results or to a bug
    foreach (string file in files)
    {
        dataSink.SendFileAsync(e.Context, file, false);
    }
}

// A private method that returns the file names
private List<string> getFilesToCollect()
{
    // Get a namespace manager with our namespace
    XmlNamespaceManager nsmgr =
        new XmlNamespaceManager(
            configurationSettings.OwnerDocument.NameTable);
    nsmgr.AddNamespace("ns",
        "http://MyCompany/schemas/MyDataCollector/1.0");

    // Find all of the "File" elements under our configuration
    XmlNodeList files =
        configurationSettings.SelectNodes(
            "//ns:MyDataCollector/ns:File");

    // Build the list of files to collect from the
    // "FullPath" attributes of the "File" nodes.
    List<string> result = new List<string>();
    foreach (XmlNode fileNode in files)
    {
        XmlAttribute pathAttribute =
            fileNode.Attributes["FullPath"];
        if (pathAttribute != null &&
            !String.IsNullOrEmpty(pathAttribute.Value))
        {
            result.Add(pathAttribute.Value);
        }
    }

    return result;
}

```

These files are attached to the test results. If you create a bug from these test results or when you use Test Runner, the files are also attached to the bug.

If you want to use your own editor to collect data to use in your test settings, see [How to: Create a custom editor for data for your diagnostic data adapter](#).

11. To collect a log file when a test finishes based on what the user configured in test settings, you must create an *App.config* file and add it to your solution. This file has the following format and must contain the URI for your diagnostic data adapter to identify it. Substitute real values for the "Company/ProductName/Version".

NOTE

If you do not need to configure any information for your diagnostic data adapter, then you do not need to create a configuration file.

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="DataCollectorConfiguration"
      type="Microsoft.VisualStudio.TestTools.Execution.DataCollectorConfigurationSection,
      Microsoft.VisualStudio.QualityTools.ExecutionCommon, Version=4.0.0.0, Culture=neutral,
      PublicKeyToken=b03f5f7f11d50a3a "/>
  </configSections>
  <DataCollectorConfiguration xmlns="http://microsoft.com/schemas/VisualStudio/TeamTest/2010">
    <DataCollector typeUri="datacollector://MyCompany/MyProduct/1.0">
      <DefaultConfiguration>
        <!-- Your default config settings -->
        <Binaries>
          <BinaryFullPath="C:\Example\Example.dll"/>
          <BinaryFullPath="\\Server2\Example2.dll"/>
        </Binaries>
        <Symbols>
          <SymbolFullPath="\\ExampleServer\ExampleSymbol.pdb"/>
        </Symbols>
      </DefaultConfiguration>
    </DataCollector>
  </DataCollectorConfiguration>
</configuration>

```

NOTE

The default configuration element can contain any data that you require. If the user does not configure the diagnostic data adapter in test settings, then the default data will be passed to your diagnostic data adapter when it is executed. Because the XML you add to the `<DefaultConfigurations>` section is not likely to be part of the declared schema, you can ignore any XML errors it generates.

There are other examples of configuration files in the following path based on your installation directory: *Program Files\Microsoft Visual Studio 10.0\Common7\IDE\PrivateAssemblies\DataCollectors*.

For more information about how to configure your test settings to use an environment when you run your tests, see [Collect diagnostic data in manual tests \(Azure Test Plans\)](#).

For more information about installing the configuration file, see [How to: Install a custom diagnostic data adapter](#)

12. Build your solution to create your diagnostic data adapter assembly.
13. For information about installing your custom editor, see [How to: Install a custom diagnostic data adapter](#).
14. For more information about how to configure your test settings to use an environment when you run your tests, see [Collect diagnostic data in manual tests \(Azure Test Plans\)](#).
15. To select your diagnostic data adapter, you must first select an existing test settings or create a new one from Microsoft Test Manager or Visual Studio. The adapter is displayed on the **Data and Diagnostics** tab of your test settings with the friendly name that you assigned to the class.
16. Set these test settings to be active. For more information about test settings, see [Collect diagnostic information using test settings](#).
17. Run your tests using the test settings with your diagnostic data adapter selected.

The data file that you specified is attached to your test results.

See also

- [DataCollectorConfigurationEditorAttribute](#)
- [DataCollectionEvents](#)
- [DataCollector](#)
- [DataCollectionSink](#)
- [DataCollectorTypeUriAttribute](#)
- [DataCollectorFriendlyNameAttribute](#)
- [DataCollectorEnabledByDefaultAttribute](#)
- [Collect diagnostic information using test settings](#)
- [Collect diagnostic data in manual tests \(Azure Test Plans\)](#)
- [Collect diagnostic data while testing \(Azure Test Plans\)](#)
- [How to: Create a custom editor for data for your diagnostic data adapter](#)

Collect diagnostic information using test settings

1/1/2020 • 6 minutes to read • [Edit Online](#)

You can use *Test settings* in Visual Studio to collect extra data when you run your tests. For example, you might want to make a video recording as you run your test. There are diagnostic data adapters to:

- Collect each UI action step in text format
- Record each UI action for playing back
- Collect system information
- Collect event log data
- Collect IntelliTrace data to help isolate non-reproducible bugs

Diagnostic data adapters can also be used to change the behavior of a test machine. For example, with a test setting in Visual Studio, you can emulate various network topology bottlenecks to evaluate the performance of your team's application.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Use test settings with Visual Studio

To run your unit, coded UI, web performance, or load tests by using Visual Studio, you can add, configure and select the test settings to use when you run your tests. To run your tests, collect data, or affect a test machine remotely, you must specify a test controller to use in your test settings. The test controller has agents that can be used for each role in your test settings.

Diagnostic data adapter details

The following table provides an overview of the various ways that the diagnostic data adapters can be configured for use with local or remote machine roles.

DIAGNOSTIC DATA ADAPTER THAT IS USED IN TEST SETTING	MANUAL TESTS ON LOCAL MACHINE	AUTOMATED TESTS	MANUAL TESTS: COLLECTING DATA BY USING A SET OF ROLES AND AN ENVIRONMENT	NOTES
ASP.NET Client Proxy for IntelliTrace and Test Impact: This proxy lets you collect information about the http calls from a client to a web server for the IntelliTrace and Test Impact diagnostic data adapters.	Yes	Yes	Yes	- Use this only when either the IntelliTrace or Test Impact diagnostic data adapters are selected for a client role.
ASP.NET profiler: You can create a test setting that includes ASP.NET profiling, which collects performance data on ASP.NET web applications.	No	Yes (See Notes)	No	- This diagnostic data adapter is supported only when you run load tests from Visual Studio.

DIAGNOSTIC DATA ADAPTER THAT IS USED IN TEST SETTING	MANUAL TESTS ON LOCAL MACHINE	AUTOMATED TESTS	MANUAL TESTS: COLLECTING DATA BY USING A SET OF ROLES AND AN ENVIRONMENT	NOTES
<p>Code coverage: You can create a test setting that includes code coverage information that is used to investigate how much of your code is covered by tests.</p>	No	Yes (See Notes)	No	<ul style="list-style-type: none"> - You can use code coverage only when you run an automated test from Visual Studio or <code>mstest.exe</code>, and only from the machine that runs the test. Remote collection is not supported. - Collecting code coverage data does not work if you also have the test setting configured to collect IntelliTrace information. <p>Note: This diagnostic data adapter is only applicable to Visual Studio test settings. It is not used for test settings in Microsoft Test Manager. Additionally, this adapter is for compatibility with Visual Studio 2010 test projects.</p> <p>Note: For compatibility, the code coverage applies when automated tests are run from Microsoft Test Manager or on a remote Test agent from Visual Studio using the legacy MSTest runner.</p>
<p>Event log: You can configure a test setting to include event log collecting, which is included in the test results.</p>	Yes	Yes	Yes	

DIAGNOSTIC DATA ADAPTER THAT IS USED IN TEST SETTING	MANUAL TESTS ON LOCAL MACHINE	AUTOMATED TESTS	MANUAL TESTS: COLLECTING DATA BY USING A SET OF ROLES AND AN ENVIRONMENT	NOTES
<p>IntelliTrace: You can configure the diagnostic data adapter for <i>IntelliTrace</i> to collect specific diagnostic trace information to help isolate bugs that are difficult to reproduce. This creates an IntelliTrace file that contains this information. An IntelliTrace file has an extension of <i>.iTrace</i>. When a test fails, you can create a bug. The IntelliTrace file that is saved together with the test results is automatically linked to this bug. The data that is collected in the IntelliTrace file increases debugging productivity by reducing the time that is required to reproduce and diagnose an error in the code. From this IntelliTrace file the local session can be simulated on another computer. This reduces the risk of a bug being non-reproducible.</p>	Yes	Yes	Yes	<ul style="list-style-type: none"> - If you enable the collection of IntelliTrace data, collection of code coverage data does not work. - If you use IntelliTrace for a web client role, you must also select the ASP.NET Client Proxy for IntelliTrace and Test Impact diagnostic data adapter. - Only the following versions of IIS are supported: IIS 7.0, IIS 7.5 and IIS 8.0.
<p>Network emulation: You can specify that you want to place an artificial network load on your test by using a test setting. Network emulation affects the communication to and from the machine by emulating a particular network connection speed, such as dial-up.</p>	No	Yes (See Notes)	No	<p>You can use the network emulation diagnostic data adapter for a client or server role. You do not have to use the adapter on both these roles that communicate with each other. Note: This diagnostic data adapter is only applicable to Visual Studio test settings. It is not used for test settings in Microsoft Test Manager. Note: Network emulation cannot be used to</p>

DIAGNOSTIC DATA ADAPTER THAT IS USED IN TEST SETTING	MANUAL TESTS ON LOCAL MACHINE	AUTOMATED TESTS	MANUAL TESTS: COLLECTING DATA BY USING A SET OF ROLES AND AN ENVIRONMENT	increase the network connection speed. Warning: If you include the network NOTES emulation diagnostic data adapter in the test settings and you intend to use it on your local machine, then you must also bind the network emulation driver to one of your machine's network adapters. The network emulation driver is required for the network emulation diagnostic data adapter to function. The network emulation driver is installed and bound to your adapter in two ways: <ul style="list-style-type: none">• Network emulation driver installed with Microsoft Visual Studio Test Agent: The Visual Studio Test Agent can be used on both remote machines and your local machine. When you install a Visual Studio Test Agent, the installation process includes a configuration step that binds the network emulation driver to your network card. For more information, see Install and configure test agents.• Network emulation driver installed with Microsoft Visual Studio

DIAGNOSTIC DATA ADAPTER THAT IS USED IN TEST SETTING	MANUAL TESTS ON LOCAL MACHINE	AUTOMATED TESTS	MANUAL TESTS: COLLECTING DATA BY USING A SET OF ROLES AND AN ENVIRONMENT	Test Professional: NOTES: When you use network emulation for the first time, you are prompted to bind the network emulation driver to a network card.
				<p>You can also install the network emulation driver from the command line on your local machine without installing the Visual Studio test agent by using the following command:</p> <p>VSTestConfig NETWORKEMULATION ON /install</p> <p>Warning: The Network Emulation adapter is ignored by load tests. Instead, load tests use the settings that are specified in the network mix of the load test scenario.</p>
System information: A test setting can be set up to include the system information about the machine on which the test is run.	Yes	Yes	Yes	
Test impact: You can collect information about which methods of your applications code were used when a test case was run. This can be used together with changes to the application code that was made by developers to determine which tests were affected by those development changes.	Yes	Yes	Yes	<ul style="list-style-type: none"> - If you are collecting test impact data for a web client role, you must also select the ASP.NET Client Proxy for IntelliTrace and Test Impact diagnostic data adapter. - Only the following versions of IIS are supported: IIS 7.0, IIS 7.5 and IIS 8.0.

DIAGNOSTIC DATA ADAPTER THAT IS USED IN TEST SETTING	MANUAL TESTS ON LOCAL MACHINE	AUTOMATED TESTS	MANUAL TESTS: COLLECTING DATA BY USING A SET OF ROLES AND AN ENVIRONMENT	NOTES
<p>Video Recorder: You can create a video recording of your desktop session when you run a test. The video can help other team members isolate application issues that are difficult to reproduce.</p>	Yes	Yes (See Notes)	Yes	<ul style="list-style-type: none"> - If you enable the test agent software to run as a process instead of a service, you can create a video recording when you run automated tests.

How to: Create a test settings file for a distributed load test

1/1/2020 • 11 minutes to read • [Edit Online](#)

Configure *test settings* for your load tests so you can distribute those tests across multiple machines using test agents and test controllers. You can also configure test settings to use *diagnostic data adapters*, which specify the kinds of data that you want to collect or how to affect the test machines when you run your load tests from Visual Studio.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

For example, you can use the ASP.NET Profiler diagnostic data adapter to collect the performance breakdown of the code. Additionally, diagnostic data adapters can be used to simulate potential bottlenecks on the test machine or reduce the available system memory.

Test settings for Visual Studio are stored in a file. The test settings define the following information about each role:

- The set of roles that are required for your application under test
- The role to use to run your tests
- The diagnostic data adapters to use for each role

When you run your tests, you select the test settings to use as the active test settings depending on what you require for that specific test run. The test settings file is stored as part of your solution. The file name has the extension `.testsettings`.

When you add a web performance and load test project to a solution, a `Default.testsettings` file is created. The file is added automatically to the solution under the **Solution Items** folder. This file runs your tests locally without any diagnostic data adapters. You can add another `.testsettings` file, or edit a `.testsettings` file to specify diagnostic data adapters and test controllers.

The test controller will have agents that can be used for each role in your test settings. For more information about test controllers and test agents, see [Manage test controllers and test agents with Visual Studio](#).

Follow these steps to create and remove test settings in your solution for load tests that you plan to run from Visual Studio.

Create a test settings file

1. In **Solution Explorer**, right-click **Solution Items**, point to **Add**, and then choose **New Item**.

The **Add New Item** dialog box appears.

2. In the **Installed Templates** pane, choose **Test Settings**.

3. (Optional) In the **Name** box, change the name of the test settings file.

4. Choose **Add**.

The new test settings file appears in **Solution Explorer**, under the **Solution Items** folder.

5. The **Test Settings** dialog box is displayed. The **General** page is selected.

You can now edit and save test settings values.

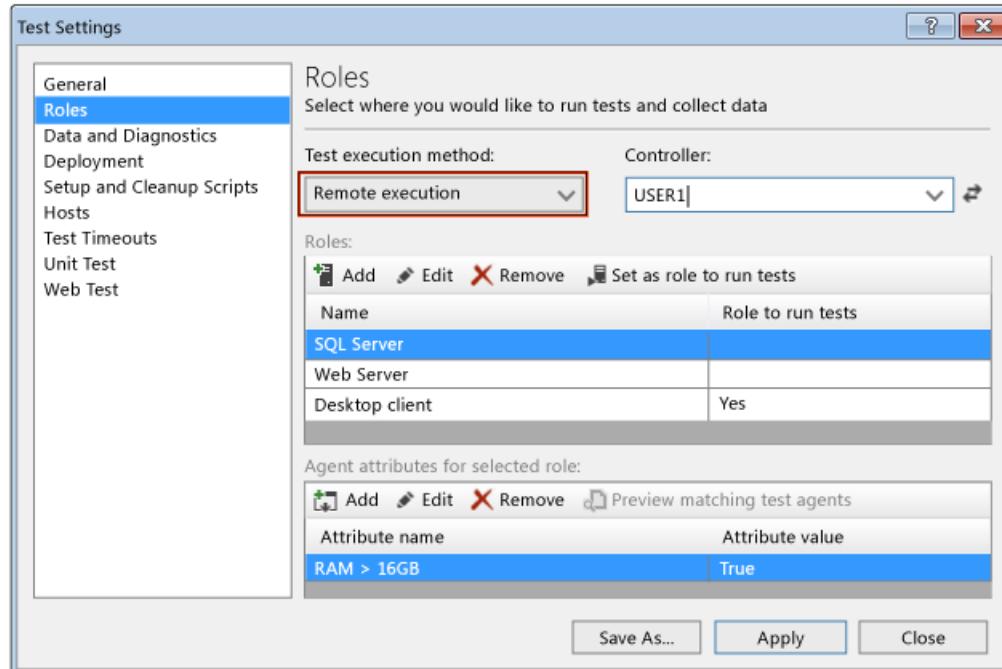
6. Under **Name**, type the name for the test settings.

7. (Optional) Under **Description**, type a description for the test setting so other team members know what it is intended for.

8. (Optional) To select the default naming scheme for your test runs, select **Default naming scheme**. To define your own naming scheme, select **User-defined scheme** and then type the text that you want in **Prefix text**. To append the date and time stamp to the test run name, select **Append date-time stamp**.

9. Choose **Roles**.

The **Roles** page is displayed.



10. To run your tests remotely, or to run your tests remotely and collect data remotely, use the **Test execution method** drop-down and select **Remote execution**.

11. Use the **Controller** drop-down to select a test controller for the test agents from **Controller** that will be used to run your tests or collect data.

NOTE

If this is the first time that you are adding a controller, no controllers will be listed in the drop-down list. The list is populated by previous controllers that you have specified in other test settings. You must type the name of the controller in the box (for example, **TestControllerMachine1**).

12. To add the roles that you want to use to run tests and collect data, Under **Roles**, choose **Add**.

13. Type a name for the role in the **Name** column. For example, the role might be "Web Server".

14. Repeat steps 12 and 13 to add all roles that you require.

Each role uses a test agent that is managed by the test controller.

15. Select the role that you want to run your tests, and then choose **Set as role to run tests**.

IMPORTANT

The other roles that you create and define will not run tests, but will be used only to collect data according to the data and diagnostic adapters that you specify for the roles in the **Data and Diagnostic** page.

16. To limit the agents that can be used for a role, select the role and then choose **Add** in the toolbar under **Agent attributes for selected role**.

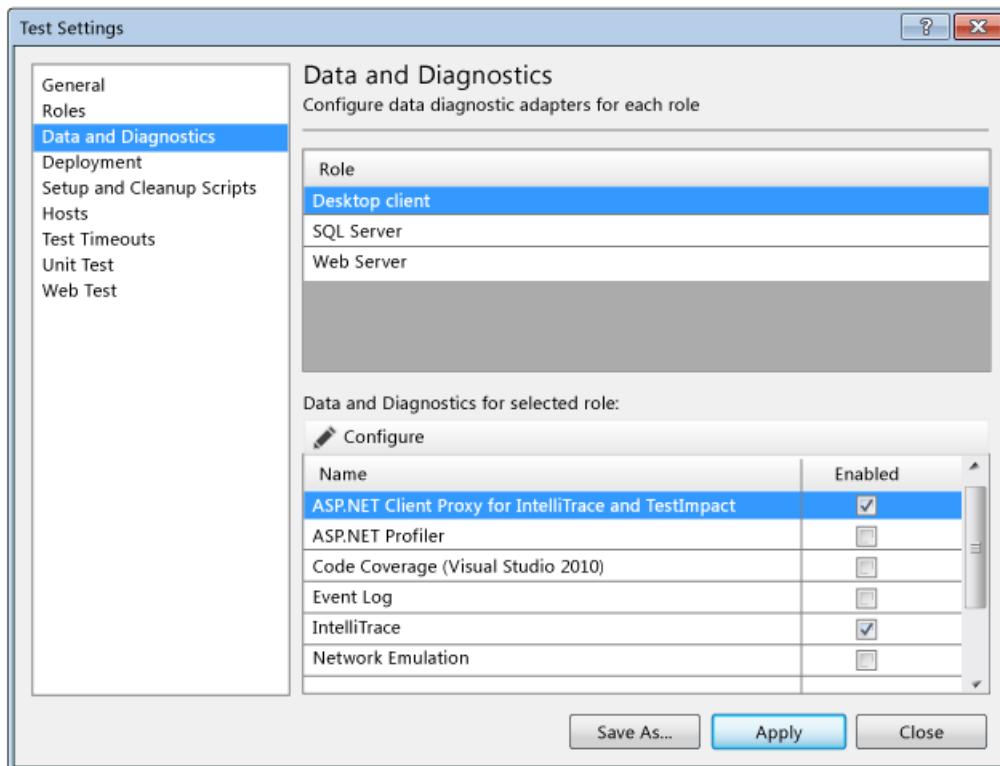
The **Agent Selection Rule** dialog box is displayed.

Type the name in **Attribute Name** and the value in **Attribute Value**, and then choose **OK**. Add as many attributes as you require.

For example, you could add an attribute that is named "RAM > 16GB" that has a value of "True" or "False" to filter on test agent machines that have more than 16GB of memory. To apply the same attribute to one or more test agents, you use the **Manage Test Controller** dialog box. For more information, see [Manage test controllers and test agents with Visual Studio](#).

17. Choose **Data and Diagnostics**.

The **Data and Diagnostics** page is displayed.



18. In the **Data and Diagnostic** page, you define what the role does by selecting the *diagnostic data adapters* that the role will use to collect data. Therefore, if one or more data and diagnostic adapters are enabled for the role, the test controller will select an available test agent machine to collect data for the specified data and diagnostic data adapters based on the attributes that you defined for the role. To select the data and diagnostic data adapters that you want to collect for each role, select the role. For each role, select the diagnostic data adapters according to the needs of the tests. To configure each diagnostic data adapter that you have selected for each role, choose **Configure**.

Example of roles and diagnostic data adapters:

For example, you could create a client role that is named "Desktop Client" that has an attribute of "Uses SQL" set to "True" and a server role that is named "SQL Server" that has an attribute set to "RAM > 16GB". If you specify that the "Desktop Client" will run the tests by choosing **Set as role to run tests** in the **Roles**

page, the test controller will select machines that have test agents that include the attribute of "Uses SQL" set to "True" on which to run the tests. The test controller will also select SQL server machines that have test agents that include the attribute "RAM > 16GB" only to collect data that is defined by the data and diagnostic adapters that are included in the role. The "Desktop Client" tests agent can also collect data for the machines on which it is run if you select data and diagnostic adapters for that role, too.

For details about each diagnostic data adapter and how to configure it, you can view the associated topic in the following table.

For more information about diagnostic data adapters, see [Collect diagnostic information using test settings](#).

Diagnostic Data Adapters for Load Tests

DIAGNOSTIC DATA ADAPTER	USING IN LOAD TESTS	ASSOCIATED TOPIC
ASP.NET Client Proxy for IntelliTrace and Test Impact: This proxy lets you collect information about the http calls from a client to a web server for the IntelliTrace and Test Impact diagnostic data adapters.	 Unless you have a specific need to collect system information for the test agent machines, do not include this adapter. Caution: We do not recommend the use of the IntelliTrace adapter in load tests because of problems that occur because of the large amount of data that is collected. Test impact data is not collected by using load tests.	
IntelliTrace: You can configure specific diagnostic trace information that is stored in a log file. A log file has an extension of <code>.tdlog</code> . When you run your test and a test step fails, you can create a bug. The log file that contains the diagnostic trace is automatically attached to this bug. The data that is collected in the log file increases debugging productivity by reducing the time that is required to reproduce and diagnose an error in the code. From this log file the local session can be recreated on another computer. This reduces the risk that a bug cannot be reproduced. For more information, see Collect IntelliTrace data .	 We do not recommend the use of the IntelliTrace adapter in load tests because of problems that occur because of the large amount of data that is collected and logged. You should attempt to use the IntelliTrace adapter only in load tests that do not run long and do not use many test agents.	How to: Collect IntelliTrace data to help debug difficult issues

DIAGNOSTIC DATA ADAPTER	USING IN LOAD TESTS	ASSOCIATED TOPIC
ASP.NET Profiler: You can create a test setting that includes ASP.NET profiling, which collects performance data on ASP.NET web applications.	The ASP.NET profiler diagnostic data adapter profiles the Internet Information Services (IIS) process, so it will not work against a development web server. To profile the website in your load test, you have to install a test agent on the machine that the IIS is running on. The test agent will not be generating load, but it will be a collection only agent. For more information, see Install and configure test agents .	How to: Configure ASP.NET profiler for load tests using test settings
Event log: You can configure a test setting to include event log collecting, which will be included in the test results.		How to: Configure event log collection using test settings
Network Emulation: You can specify that you want to put an artificial network load on your test by using a test setting. Network emulation affects the communication to and from the machine by emulating a particular network connection speed, such as dial-up. Note: Network emulation cannot be used to increase the network connection speed.	The Network Emulation adapter is ignored by load tests. Instead, load tests use the settings that are specified in the network mix of the load test scenario. For more information, see Specify virtual network types .	
System Information: A test setting can be set up to include the system information about the machines on which the System Information diagnostic and data collector is run. The system information is specified in the test results by using a test setting.	 You can collect system information from both the load agents and the system under test.	No configuration is required to collect this information.
Test Impact: You can collect information about which methods of your applications code were used when a test case was run. This can be used together with changes to the application code that are made by developers to determine which tests were affected by those development changes.	Test impact data is not collected with load tests.	

DIAGNOSTIC DATA ADAPTER	USING IN LOAD TESTS	ASSOCIATED TOPIC
<p>Video Recorder: You can create a video recording of your desktop session when you run an automated test. This can be useful to view the user actions for a coded UI test. The video can help other team members isolate application issues that are difficult to reproduce. Note: When running tests remotely the video recorder will not work unless the agent is running in interactive process mode.</p>	 Warning: We do not recommend the use of the Video Recorder adapter for load tests.	How to: Include recordings of the screen and voice during tests using test settings

19. Choose **Deployment**.

The **Deployment** page is displayed.

20. To create a separate directory for deployment every time that you run your tests, select **Enable deployment**.

NOTE

If you do this, you can continue to build your application when you run your tests.

21. To add a file to the directory you are using to run your tests, choose **Add file**, and then select the file that you want to add.

NOTE

When you run a load tests, plug-in assemblies, data files, and uploaded files are automatically deployed.

22. To add a directory to the directory that you are using to run your tests, choose **Add directory** and then select the directory that you want to add.

23. To run scripts before and after your tests, choose **Setup and Cleanup Scripts**.

The **Setup and Cleanup Scripts** page is displayed.

- Type the location of the script file in **Setup script** or choose the ellipsis (...) to locate the setup script.
- Type the location of the script file in **Cleanup script** or choose the ellipsis (...) to locate the cleanup script.

24. To run your tests by using a different host, choose **Hosts**.

- In **Host Type**, verify that the **Default** is selected.

NOTE

The **ASP.NET** in **Host type** is not supported in load tests.

- Use the **Run test in 32-bit or 64-bit** process drop-down to select whether you want the web performance and unit tests in your load test to run as 32-bit or 64-bit processes.

NOTE

For maximum flexibility, you should compile your web performance and load test projects by using the **Any CPU** configuration. Then you can run on both 32-bit and 64-bit agents. Compiling web performance and load test projects by using the **64-bit** configuration offers no advantage.

25. (Optional) To limit the time for each test run and individual tests, choose **Test Timeouts**.
 - a. To abort a test run when a time limit is exceeded, select **Abort a test run if the total time exceeds** and then type a value for this limit.
 - b. To fail an individual test when a time limit is exceeded, select **Mark an individual test as failed if its execution time exceeds**, and type a value for this limit.
26. Skip **Unit Test**. Load tests do not use these settings.
27. Skip **Web Test**. Load tests do not use these settings.
28. To save the test settings, choose **Save As**. Type the name of the file that you want in **Object name**.

Remove a test settings file from your solution

Under the **Solution Items** folder in **Solution Explorer**, right-click the test settings that you want to remove, and then choose **Remove**.

The test settings file is removed from your solution.

See also

- [Test controllers and test agents](#)
- [Collect diagnostic information using test settings](#)

How to: Configure ASP.NET profiler for load tests using test settings in Visual Studio

10/18/2019 • 2 minutes to read • [Edit Online](#)

You can use the ASP.NET profiler diagnostic data adapter to collect ASP.NET profiler information. This diagnostic data adapter collects performance data for ASP.NET applications.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

NOTE

This diagnostic data adapter cannot be used for tests that are run using Microsoft Test Manager. You can use the ASP.NET Profiler diagnostic adapter with load tests using websites only, which requires Visual Studio Enterprise.

The ASP.NET profiler diagnostic data adapter lets you collect ASP.NET profiler data from the application tier when you run a load test. You should not run the profiler for long load tests, for example, load tests that run longer than one hour. This is because the profiler file can become large, perhaps hundreds of megabytes. Instead, run shorter load tests by using the ASP.NET profiler, which will still give you the benefit of deep diagnosis of performance problems.

NOTE

The ASP.NET profiler diagnostic data adapter profiles the Internet Information Services (IIS) process. Therefore, it will not work against a development web server. To profile the website in your load test, you have to install a test agent on the machine on which the IIS is running. The test agent will not generate load, but will be an agent for collection only. For more information, see [Install and configure test agents](#).

For more information, see [How to: Create a test setting for a distributed load test](#).

Configure the ASP.NET profiler for your test settings

Before you perform the steps in this procedure, you must open your test settings from Visual Studio and select the **Data and Diagnostics** page.

1. Select the role to use to collect the ASP.NET profiler data.

WARNING

This role must be a web server.

2. Select **ASP.NET Profiler** to enable collecting ASP.NET profiling data, and then choose **Configure**.

The dialog box to configure ASP.NET profiling data collection is displayed.

3. In **Profiler Sampling interval**, type a value that indicates how many non-halted CPU clock cycles to wait

between taking ASP.NET profiling samples.

4. To enable tier interaction profiling, select **Enable Tier Interaction Profiling**.

Tier interaction profiling counts the number of requests that are sent to the web server for each artifact (for example, *MyPage.aspx* or *CompanyLogo.gif*) and the time it took to service each request. Additionally, tier interaction profiling collects which ADO.NET connections were used as a part of the page request, and how many queries and stored procedure calls were executed as a part of servicing that request.

Two different sets of timing information are collected:

- The timing information (Min, Max, Average, and Total) for servicing each web request.
- The timing information (Min, Max, Average and Total) of executing each query.

With the ASP.NET profiler diagnostic data adapter configured in your test setting, you can now collect ASP.NET profiling data on your ASP.NET web application.

See also

- [Collect diagnostic information using test settings](#)
- [How to: Create a test setting for a distributed load test](#)
- [Test controllers and test agents](#)

How to: Configure network emulation using test settings in Visual Studio

1/1/2020 • 4 minutes to read • [Edit Online](#)

You can configure the diagnostic data adapter to test your application under various network environments from Visual Studio. It can also be configured to test an artificial network load, or bottleneck, when you run your tests.

WARNING

If you run your tests on a real network that is a slower type than the network you are emulating, the test will still run at the slower network speed. The emulation can only slow down the network environment, not speed it up.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

The following procedure describes how to configure network emulation from the configuration editor. These steps apply to both the configuration editor in Microsoft Test Manager and Visual Studio.

NOTE

The network emulation diagnostic data adapter is only applicable to Visual Studio test settings. It is not used for test settings in Microsoft Test Manager.

An account that has administrator privileges must be used for network emulation. If you have selected network emulation for a local role that runs manual tests, you must start Microsoft Test Manager by using administrator privileges. If you have selected network emulation for any other role, you must verify that the test agent on the machine for that role uses a user account that is a member of the administrators group. For more information about how to set up the account for your test agent, see [Install and configure test agents](#).

NOTE

The Network Service account, which is the default account for the test agent, is not a member of the administrators group.

True Network Emulation

Visual Studio uses software-based true network emulation for all test types. This includes load tests. True network emulation simulates network conditions by direct manipulation of the network packets. The true network emulator can emulate the behavior of both wired and wireless networks by using a reliable physical link, such as an Ethernet. The following network attributes are incorporated into true network emulation:

- Round-trip time across the network (latency)
- The amount of available bandwidth
- Queuing behavior
- Packet loss

- Reordering of packets
- Error propagations.

True network emulation also provides flexibility in filtering network packets based on IP addresses or protocols such as TCP, UDP, and ICMP.

True network emulation can be used by network-based developers and testers to emulate a desired test environment, assess performance, predict the effect of change, or make decisions about technology optimization. When compared to hardware test beds, true network emulation is a much cheaper and more flexible solution.

Configure network emulation for your test settings

Before you perform the steps in this procedure, you must open your test settings from Visual Studio and then select the **Data and Diagnostics** page.

To configure network emulation for your test settings

1. Select the role to use to emulate a specific network.

NOTE

You have to configure the Network Emulation adapter only on either the client role or the server role. You do not have to use the adapter on both roles. The adapter emulates network noise that affects communication between both roles, so that you do not have to use it on both. Unless it is necessary, you should pick a client role for the Network Emulation adapter to avoid extra overhead on the server role.

2. Select **Network Emulation** and then choose **Configure**.

The dialog box to configure network emulation is displayed.

3. Choose the arrow next to **Select the network profile to use**, and select the network type that you want to emulate when you run a test (for example, **Cable-DSL 768Kps**).

WARNING

If you run your tests on a real network that is a slower type than the network that you are emulating, the test will still run at the slower network speed. The emulation can only slow down the network environment, not speed it up.

4. If you include the network emulation diagnostic data adapter in the test settings and you intend to use it on your local machine, then you must also bind the network emulation driver to one of your machine's network adapters. The network emulation driver is required for the network emulation diagnostic data adapter to function. The network emulation driver is installed and bound to your adapter in two ways:

- **Network emulation driver installed with Microsoft Visual Studio Test Agent:** The Microsoft Visual Studio Test Agent can be used on both remote machines and your local machine. When you install a Visual Studio Test Agent, the installation process includes a configuration step that binds the network emulation driver to your network card. For more information, see [Install and configure test agents](#).
- **Network emulation driver installed with Microsoft Visual Studio Test Professional:** When you use network emulation for the first time, you are prompted to bind the network emulation driver to a network card.

TIP

You can also install the network emulation driver from the command line on your local machine without installing the Visual Studio test agent by using the following command: **VSTestConfig NETWORKEMULATION /install**

See also

- [Collect diagnostic information using test settings](#)
- [Run manual tests \(Azure Test Plans\)](#)

How to: Collect IntelliTrace data to help debug difficult issues

1/1/2020 • 5 minutes to read • [Edit Online](#)

You can configure the diagnostic data adapter for IntelliTrace to collect specific diagnostic trace information in Visual Studio. Tests can use this adapter, the test can collect significant diagnostic events for the application that a developer can use later to trace through the code to find the cause of a bug. The diagnostic data adapter for IntelliTrace can be used for either manual or automated tests.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

NOTE

IntelliTrace works only on an application that is written by using managed code. If you are testing a web application that uses a browser as a client, you should not enable IntelliTrace for the client in your test settings because no managed code is available to trace. In this case, you may want to set up an environment and collect IntelliTrace data remotely on your web server.

The IntelliTrace data is stored in a file that has an extension of *.iTrace*. When you run your test and a test step fails, you can create a bug. The IntelliTrace file that contains the diagnostic information is automatically attached to this bug.

NOTE

The diagnostic data adapter for IntelliTrace does not create an IntelliTrace file when a test pass is successful. It saves a file only on a failed test case or when you submit a bug.

The data that is collected in the IntelliTrace file increases debugging productivity by reducing the time that is required to reproduce and diagnose an error in your code. Additionally, because you can share the IntelliTrace file with another individual who can replicate your local session on their computer, it reduces the probability that a bug will be non-reproducible.

NOTE

If you enable IntelliTrace in your test settings, collecting code coverage data will not work.

WARNING

The diagnostic data adapter for IntelliTrace works by instrumenting a managed process, which must be performed after the tests for the test run are loaded. If the process that you want to monitor has already started, no IntelliTrace files will be collected because the process is already running. To circumvent this, make sure that the process is stopped before the tests are loaded. Then start the process after the tests are loaded or the first test is started.

The following procedure describes how to configure the IntelliTrace data that you want to collect. These steps

apply to both the configuration editor in Microsoft Test Manager and Test Settings dialog box in Visual Studio.

NOTE

The user account for the test agent that is used to collect IntelliTrace data must be a member of the administrators group. For more information, see [Install and configure test agents](#).

Configure the data to collect with the IntelliTrace diagnostic data adapter

Before you perform the steps in this procedure, you must open your test settings from either Microsoft Test Manager or Visual Studio and select the **Data and Diagnostics** page.

To configure the data to collect with the IntelliTrace diagnostic data adapter

1. Select the role to use to collect IntelliTrace data.
2. Select **IntelliTrace**.
3. If you are adding IntelliTrace for a web client role or for an ASP.NET web application, you must also select **ASP.NET Client Proxy for IntelliTrace and Test Impact**.

This proxy enables you to collect information about the http calls from a client to a web server for the IntelliTrace and Test Impact diagnostic data adapters.

WARNING

If you decide to use a custom account for the identity that is being used for the application pool on the Internet Information Server (IIS) where you intend to collect IntelliTrace data, you must create the local user profile on the IIS machine for the custom account that is being used. You can create the local profile for the custom account either by logging on to the IIS machine locally one time or by running the following command line by using the custom account credentials:

`runas /user:domain\name /profile cmd.exe`

4. Choose **Configure** for **IntelliTrace** to modify default IntelliTrace settings.

The dialog box to configure the data that will be collected is displayed.

WARNING

If you enable collecting IntelliTrace data, collecting code coverage data will not work.

5. Choose the **General** tab. Select **IntelliTrace events only** to record significant diagnostic events that have minimal impact on performance when you test.

-or-

Select **IntelliTrace events and call information** to record diagnostic events and method level tracing that shows call information. This level of tracing might have performance impact when you run your tests.

6. To collect data from your ASP.NET application that is running on Internet Information Services, select **Collect data from ASP.NET applications that are running on Internet Information Services**. Set up and configure your test agent on the web server role. See [Install and configure test agents](#).
7. Choose the **Modules** tab. Select either **Collect data from all modules except for the following** and use **Add** to add to the list of modules and **Remove** to remove a module. This option lets you include all the

modules that are running on the system except the modules that you specify.

-or-

Select **Collect data from only the following modules** and use **Add** to add to the list of modules and **Remove** to remove a module. This option lets you specify exactly which modules you want.

NOTE

If possible, select the specific processes that you want to monitor. This is recommended for optimum performance.

8. Choose the **Processes** tab. Select **Collect data from all processes except for the following** and use **Add** to add to the list of processes and **Remove** to remove a process. This option lets you include all the processes that are running on the system except the processes that you specify.

-or-

Select **Collect data from specified processes only** and use **Add** to add to the list of processes and **Remove** to remove a process. This option lets you specify exactly which processes you want.

9. (Optional) Choose the **IntelliTrace Events** tab. Select or clear each IntelliTrace event category that you want to include or exclude when you collect diagnostic events.
10. (Optional) Expand each IntelliTrace event category and select or clear each specific event that you want to include or exclude in the IntelliTrace events.
11. (Optional) Choose the **Advanced** tab. Next, choose the arrow next to **Maximum amount of disk space for recording** and select the maximum size that you want to enable for the IntelliTrace file to use.

NOTE

If you increase the size of the recording, a time-out issue might occur when you save this recording together with your test results.

12. If you are using Microsoft Test Manager, choose **Save**. If you are using Visual Studio, choose **OK**. The IntelliTrace settings are now configured and saved for your test settings.

NOTE

To reset the configuration for this diagnostic data adapter, choose **Reset to default configuration** for Visual Studio or **Reset to default** for Microsoft Test Manager.

See also

- [Collect diagnostic data while testing \(Azure Test Plans\)](#)
- [Collect diagnostic data in manual tests \(Azure Test Plans\)](#)
- [Collect diagnostic information using test settings](#)
- [Collect IntelliTrace data](#)

How to: Include recordings of the screen and voice during tests using test settings

1/1/2020 • 2 minutes to read • [Edit Online](#)

From the configuration editor in Visual Studio, you can configure the diagnostic data adapter that records the screen and voice of the user who's running the test. This diagnostic data adapter saves a screen and voice recording of the desktop session during the test. The recording is saved with the test result or it can be attached to a bug. Other team members can use the recording to isolate application defects that are difficult to reproduce.

WARNING

The screen and voice recordings do not support multiple monitor configurations.

The screen and voice recorder can be used with either manual or automated tests. For example, if you run a coded UI test remotely you might want to record the desktop to see the coded UI test as it runs. For more information about how to capture a screen and voice recording remotely, see [How to: Set up your test agent to run tests that interact with the desktop](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To configure screen and voice recording for your test settings

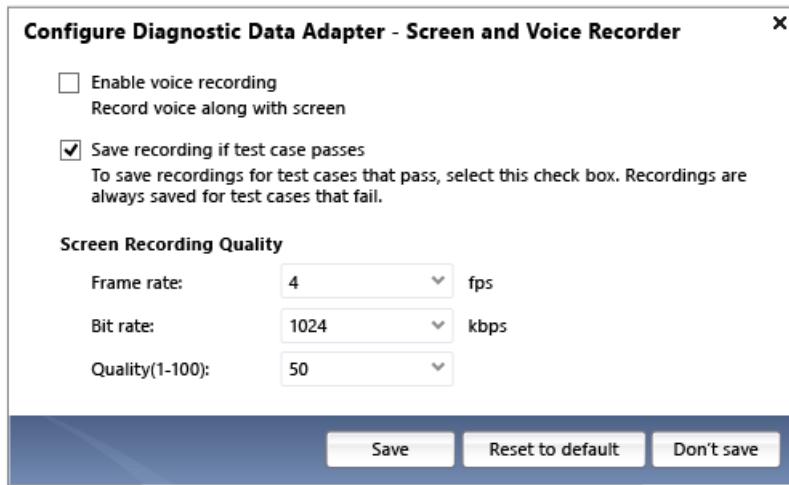
1. Open the test settings that you want to configure for recording the screen and voice. For more information, see [Collect diagnostic data while testing \(Azure Test Plans\)](#) or [Collect diagnostic information using test settings](#).
2. In the test settings, select the **Role** to use to record the screen and voice.

NOTE

For manual tests and automated tests this would be the machine that runs the tests.

3. Select **Screen and Voice Recorder** and then choose **Configure**.

The **Configure Diagnostic Data Adapter – Screen and Voice Recorder** dialog box is displayed.



4. (Optional) Select **Enable voice recording** to capture audio content in your recording.
5. (Optional) Select the check box next to **Save recording if test case passes** to specify saving screen and voice recordings for both failed and passed tests.

WARNING

If you select **save recording if test case passes**, the recording is stored with the test results, which uses storage space on the server. You can use the **Test Attachment Cleaner** tool to clean up these attachments.

6. Under **Screen Recording Quality**, configure the following drop-down list options:
 - a. **Frame rate:** Specify how many frames per second you want to use in the screen and voice recording. The default value is 4 frames per second. Values between 2 and 20 can be specified.
 - b. **Bit rate:** Specify how many kilobytes per second to use in the screen and voice recording. The default value is 512. Values between 512 and 10,000 can be specified.
 - c. **Quality(1-100):** You can specify the quality of the screen and voice recording by selecting a range between 1 and 100. The default is 50 (mid-range).
7. Choose **OK**. The diagnostic trace collector settings are now configured and saved for your test settings.

TIP

To reset the configuration for this diagnostic data adapter, choose **Reset to default configuration** for Visual Studio and **Reset to default** for Microsoft Test Manager.

See also

- [Collect diagnostic data while testing \(Azure Test Plans\)](#)
- [Collect diagnostic data in manual tests \(Azure Test Plans\)](#)
- [Collect diagnostic information using test settings](#)
- [Run manual tests \(Azure Test Plans\)](#)

Analyze load test results using the Load Test Analyzer

1/1/2020 • 3 minutes to read • [Edit Online](#)

Find bottlenecks, identify errors, and measure improvements in your app when you use the **Load Test Analyzer**.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Analyze load test results in these ways:

- Monitor a load test when it is running.
- Analyze a load test after it has completed.
- View results from a previous load test.

You can also create reports that compare two or more reports for trend analysis to share with stakeholders. See [Reporting load tests results for test comparisons or trend analysis](#).

You can complete these tasks whether you run your load test from Visual Studio Enterprise or from the command line, and whether you run your load test on a single computer or on a remote machine.

Differences between analyzing a running and a completed load test

When you run a load test, the **Load Test Analyzer** displays in a separate tab, together with the name of your load test and the time that the test was started (for example, **LoadTest1 [12:40 PM]**). When a load test runs, a smaller set of the performance counter data is maintained in memory. You can monitor this set of data when your load test runs. After a load test has completed, you can analyze the full set of data from the database.

Differences exist in what data is displayed when a load test runs and what data that you can see after a load test has completed. For example, 90 percent and 95 percent response time data is not calculated until the load test has completed. Some differences also occur in the functionality of the tools that are available to analyze the data.

When you run the load test, two views are available: The **Graphs** view and the **Tables** view. The **Graphs** view allows you to graph performance counters that are collected. The **Tables** view gives you information about each of the tests, pages, transactions, and requests that are collected. You also get a table that lists the errors.

By default, when the load test run has completed, the **Summary** view is displayed. You can switch between the **Summary**, **Graphs**, **Tables**, and **Details** views by using the toolbar. The **Load Test Analyzer** can be docked or set to float by using the usual Visual Studio window manipulation techniques. When you analyze completed load test runs, you can have multiple **Load Test Analyzers** open at the same time to compare the different load test runs.

Tasks

TASKS	ASSOCIATED TOPICS
<p>Accessing the results of your load test: When you run a load test from the Load Test Editor, the load test results open automatically and the running load test is displayed in the Load Test Analyzer.</p>	<ul style="list-style-type: none"> - How to: Access load test results for analysis
<p>Add analysis notes to your load test: You can add comments to your load test when you conduct your analysis. The comments are stored permanently, together with the load test result. The description that you enter also displays in the Description column that is associated with the load test in the Open and Manage Test Results dialog box in the Load Test Editor.</p> <p>For more information, see How to: Access load test results for analysis.</p> <p>Additionally, the comments are displayed when you create an Excel report for the load test results.</p> <p>For more information, see Reporting load tests results for test comparisons or trend analysis.</p>	
<p>Analyzing the results of your load test: After you access the load test run data, you can analyze the resulting data. You can view the Load Test Summary to understand the results quickly. The load test summary shows the key results in a compact and easily read format.</p> <p>You can print the load test summary. This makes it convenient to use when you communicate results to stakeholders.</p> <p>You can analyze the details of your load test results by using the graphs and tables in the results. These include Errors, Pages, Requests, SQL Trace, Tests, Thresholds, and Transactions.</p>	<ul style="list-style-type: none"> - Load test results summary overview - How to: View web page response - Analyzing threshold rule violations - Analyze load test results in the Graphs view - Analyze load test results and errors in the Tables view
<p>Analyzing the virtual user activity in your load test results to isolate performance issues: You can use the Virtual User Activity Chart to visualize what virtual users are doing during a load test. This can help you isolate spikes in a CPU or drops in requests/sec, and determine which tests or pages are running during these spikes and drops.</p>	<ul style="list-style-type: none"> - Analyzing virtual user activity in the Details view

How to: Access load test results for analysis

1/1/2020 • 2 minutes to read • [Edit Online](#)

When you run a load test from the Load Test Editor, the load test results open automatically and the running load test is displayed in the **Load Test Analyzer**. When you run a load test from the command line, you must access the load test results manually.

The load test result for the completed load test contains performance counter samples and error information that was collected periodically from the computers-under-test. A large number of performance counter samples can be collected over the course of a load test run. The amount of performance data that is collected depends on the length of the test run, the sampling interval, the number of computers under test, the number of counters being collected, the data collectors that are configured, and the logging levels. For a large load test, the amount of performance data that is collected can easily be several gigabytes. For more information, see [Test controllers and test agents](#).

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To access a load test result

1. From a web performance and load test project, open a load test.
2. In the Load Test Editor's toolbar, choose the **Open and Manage Results** button.

The **Open and Manage Results** dialog box appears.

3. In **Enter a controller name to find load test results**, select a controller. Select **<local> - No controller** to access results stored locally.
4. In **Show results for the following load test**, select the load test whose results you want to view. Select **<Show results for all tests>** to see all results for all tests.

If there are load test results available, they appear in the **Load test results** list. The columns are **Time**, **Duration**, **User**, **Outcome**, **Test**, and **Description**. **Test** contains the name of the test, and **Description** contains the optional description that is added before the test is run.

NOTE

The results appear with the most recent results at the top of the list.

5. In the **Load test results** list, select the load test results you want to analyze and choose **Open**.
6. The **Load Test Analyzer** appears. The selected load test result is displayed in the Summary view. For more information, see [Load test results summary overview](#).

You can manage other aspects of load test results in the **Open and Manage Results** dialog box including importing, exporting, and removing load test results. For more information, see [Manage load test results in the load test results repository](#).

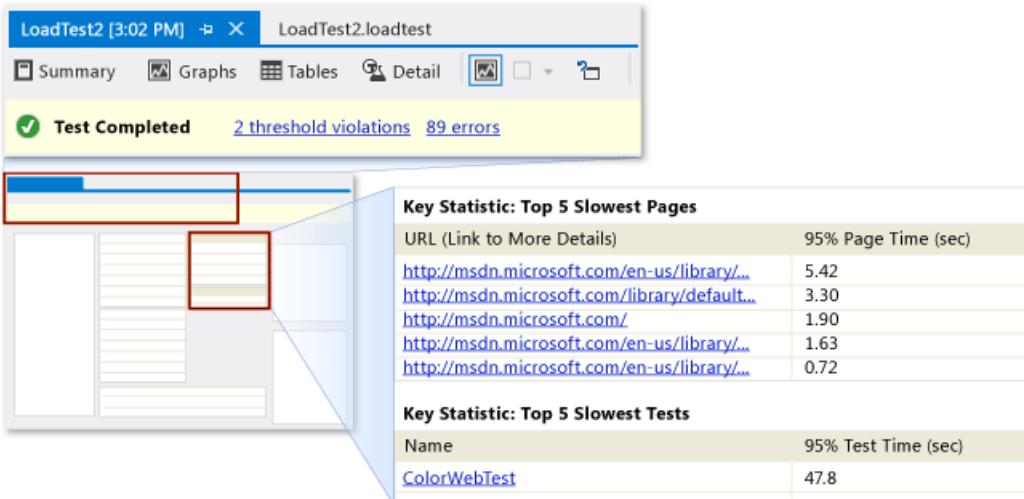
See also

- [Analyze load test results](#)

Load test results summary overview

1/1/2020 • 6 minutes to read • [Edit Online](#)

After you run a load test, you can view the load test summary to understand the results quickly. The load test summary provides the key results in a compact and easy to read format. You can also print the load test summary. This makes it convenient to use when you communicate results to stakeholders. The load test summary is also the default view when you open a load test result from a previously run load test. For more information, see [How to: Access load test results for analysis](#).



NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

The load test summary

The load test summary is divided into sections. The initial sections appear at the top of the summary, and are always visible. When you view the load test summary, the following items are first:

- Test Run Information
- Overall Results
- Key Statistic: Top 5 Slowest Pages
- Key Statistic: Top 5 Slowest Tests
- Key Statistic: Top 5 Slowest SQL Operations

NOTE

The SQL Operations section is displayed only if SQL tracing is enabled in the load test.

The closing sections appear at the end of the summary, and can be collapsed to save space. The following items appear at the end of the load test summary:

- Test Results

- Page Results
- Transaction Results
- System Under Test Resources
- Controller and Agent Resources
- Errors

Test run information

The test run information section contains general information about the run including the name of the test, the start and end times, and the controller that ran the test. This section also contains the optional description of the run that you add when you run the load test.

Overall results

The overall results section contains summary results of the test including the number of requests per second, the total number of failed requests, the average response time, and the average page time.

Key statistic: Top 5 slowest pages

The slowest pages section contains the top 5 slowest pages in the load test. The URL and the average page load time are displayed for each page. The pages are listed in descending order. You can choose the URL of a page to open the **Pages** table and inspect more details for that page. For more information, see [How to: View web page response](#).

The percentile value for **95% Page Time (sec)** report that 95% of the pages completed in less than this time in seconds.

Key statistic: Top 5 slowest tests

The slowest tests section contains the top 5 slowest tests in the load test. The name of the test and the average test time are displayed for each test. The tests are listed in descending order. You can choose the name of a test to open the **Tests** table and inspect more details for that test. For more information, see [Analyze load test results and errors in the Tables view](#).

The percentile value for **95% Test Time (sec)** report that 95% of the tests completed in less than this time in seconds.

Key statistic: Top 5 slowest SQL operations

If SQL tracing is enabled in the load test, the slowest queries section contains the top 5 slowest queries in the load test. The name of the operation and the duration are displayed for each test. The duration is displayed in microseconds (SQL Server 2005) or milliseconds (SQL Server 2000 and earlier). The tests are listed in descending order by duration. You can choose the name of an operation to open the **SQL Trace** table and inspect more details for that operation. For more information, see [The SQL Trace data table](#).

Test results

The test results section contains a list of all the tests and scenarios in the load test. The name of the test, the scenario, the number of times it ran, the number of times it failed, and the average test time are displayed. You can choose the name of a test to open the **Tests** table and inspect more details for that test. For more information, see [Analyze load test results and errors in the Tables view](#).

NOTE

You can collapse and expand this section by choosing the arrow to the left of the section title.

Page results

The page results section contains a list of all the web pages in the load test. The URL, the scenario, the name of the test, the average page time, and the count are displayed. You can choose the URL of a page to open the **Pages** table and inspect more details for that page. For more information, see [How to: View web page response](#).

NOTE

You can collapse and expand this section by choosing the arrow to the left of the section title.

Transaction results

The transaction results section contains a list of all the transactions in the load test. The name of the transaction, the scenario, the test, the response time, the elapsed time, and the count are displayed. You can choose the name of a transaction to open the **Transactions** table and inspect more details for that transaction. For more information, see [Analyze load test results and errors in the Tables view](#).

NOTE

You can collapse and expand this section by choosing the arrow to the left of the section title.

The percentile values report the following transaction information:

- 90% of the total transactions were completed in less than <time> seconds.
- 95% of the total transactions were completed in less than <time> seconds.

System under test resources

The system under test resources section contains a list of computers that are the set of target computers for which load is being generated. This includes any computer from which you collect counter sets other than Agent or Controller. The computer name, % processor time, and available memory are displayed. You can choose a computer name to open the **System under Test** graph and see the resource usage over time. For more information, see [Analyze load test results in the Graphs view](#).

NOTE

You can collapse and expand this section by choosing the arrow to the left of the section title.

Controller and agent resources

The controller and agent resources section contains a list of the computers that are used to run the test. The computer name, % processor time, and available memory are displayed. You can choose a computer name to open the **Controller and Agents** graph and see the resource usage over time. For more information, see [Analyze load test results in the Graphs view](#).

NOTE

You can collapse and expand this section by choosing the arrow to the left of the section title.

Errors

The errors section contains a list of all the errors that occurred during the load test. The type and subtype of the error, the count, and the last message are displayed. You can choose an error to open the **Errors** table and inspect more details for that error. For more information, see [Analyze load test results and errors in the Tables view](#).

NOTE

You can collapse and expand this section by choosing the arrow to the left of the section title.

Print a summary

You can print the load test summary by choosing **Print** on the shortcut menu on the summary. You can preview the print first by choosing **Print Preview** on the shortcut menu on the summary. You can also print directly from the preview screen.

See also

- [Analyze threshold rule violations](#)
- [Analyze load test results](#)

Analyze load test results and errors in the Tables view of the Load Test Analyzer

1/1/2020 • 13 minutes to read • [Edit Online](#)

When you view the results of a load test run, you can display different panes that provide you with different ways to analyze the data. You can view the data as a graph, to see how it changes over time, or you can view the data as detailed tables.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To switch to table view, choose **Tables** on the **load test** toolbar. To switch between the different tables, use the **Table** drop-down list on the toolbar above the table grid. In table view, you can view up to four tables at a time. For more information, see [Tile load test tables](#) in this topic.

Most numeric values displayed in a table for performance counters are cumulative over the whole load test run. Columns named **Last** are an exception, and represent the value from the most recent sampling interval.

NOTE

Columns named **Last** are available only while a load test is executing. After a load test is completed, these columns are not available.

You can sort most tables by choosing the title of the column that you want to sort on. By default, some tables do not display all available columns. You can add columns to tables, if columns are available. To add columns, right-click the table and then choose **Add/Remove Columns**.

NOTE

You can copy the data from a table into other applications such as Excel for additional analysis.

The load test tables

The following table lists the tables that are available to analyze load test runs.

TABLE NAME	DESCRIPTION
Errors	Displays a list of errors that occurred during the load test run. For more information, see The Errors table in this topic, and Analyze load test results .
Pages	Displays a list of pages accessed during a load test run. Some data in this table is available only after a load test has completed. For more information, see How to: View web page response .

TABLE NAME	DESCRIPTION
Requests	Displays details for individual requests issued during a load test. This includes all HTTP requests, and dependent requests such as images. For more information, see The Requests table in this topic.
SQL Trace	Displays the results of SQL tracing. This table is available only after a load test has completed, and only if SQL tracing was used during the test. For more information, see The SQL Trace data table in this topic.
Tests	Displays details for individual tests run during a load test. For more information, see The Tests table in this topic.
Thresholds	Displays a list of threshold rule violations that occurred during the load test run. For more information, see Analyzing threshold rule violations .
Transactions	Displays a list of transactions that occurred during a load test run. For more information, see The Transactions table in this topic.
Agents	Displays only if your load test is using a test controller and test agents. Displays a list of the agents that were used during the load test run. The Agents table includes how many requests the agent tested and of those requests, how many failed. Additionally, the Agents table includes the number of tests in the load tests test mix that the agent tested and of those, how many failed.
Test Details	Displays details for the tests included in the test mix for the load test. The details include the name of the test, the scenario that the test was in, the time that the test started, the length of time it took the test to run, and the test outcome indicating if the test passed or failed. If the test failed, a link is present in the Details column. You can choose the link which will take you to the Web Performance Test Editor with the failed request highlighted.

Collect percentile data

Some load test tables can contain additional columns, which include percentile data and response times broken into groups based on network emulation. By default, this data is not collected. Percentile data is only available when you save results to a database, and not when you save locally. For more information, see [Managing load test results in the Load Test Results Repository](#). Additionally, to collect this data, in the **Load Test Editor**, under the **Run Settings** node, select the specific run setting node to change. In the **Properties** window, for the **Timing Details Storage** property, select **StatisticsOnly** or **AllIndividualDetails**. For more information, see [How to: View web page response](#).

The Requests table

The **Requests** table displays details for individual requests issued during a load test. This includes all HTTP requests, and dependent requests such as images. The table lists requests by test and scenario, because one request can be included in many tests and scenarios.

The following table lists the columns in the **Requests** table:

COLUMN	DESCRIPTION	VISIBLE BY DEFAULT
Request	The URL of the request. For example, <i>home.html</i> , or <i>orange-arrow.gif</i> .	Yes
Scenario	The name of the scenario.	Yes
Test	The name of the test.	Yes
Total	The total number of this web performance test request issued during the load test run. The total includes passed and failed requests, but does not include cached requests, because they are not issued to the web server.	Yes
Passed	The number of times the request was issued and passed.	No
Failed	The number of times the request was issued and failed. The entries in this column appear as hyperlinks. You can choose any hyperlink to view a list of the individual errors in the Load Test Errors dialog box. For more information, see Analyze load test results .	Yes
Cached	The total number of times the request was already cached.	No
Requests/Sec	The rate per second of the request during the load test run.	No
Passed/Sec	The rate per second of this request during the load test run, for the instances of this request that passed.	No
Failed/Sec	The rate per second of this request during the load test run, for the instances of this request that failed.	No
First Byte Time	The average time to receive the first byte of the response, measured from the time the request was sent to the web server. The units are seconds.	No
Response Time	The average time to receive the entire response to a request, measured from the time the request was sent to the web server. The units are seconds.	Yes
Content Length	The average length of the content of the response to the request. The units are bytes.	Yes

The Tests table

The **Tests** table displays details for individual tests run during a load test. The table lists tests by test and scenario, because one test can be included in many scenarios.

The following table lists the columns in the **Tests** table.

COLUMN	DESCRIPTION	VISIBLE BY DEFAULT
Test	The name of the test.	Yes
Scenario	The name of the scenario.	Yes
Total	The total number of times the test was run in the scenario. This includes the number of times the test passed and failed.	Yes
Passed	The number of times the test was run in the scenario and passed.	Yes
Failed	The number of times the test was run in the scenario and failed. The entries in this column appear as hyperlinks. You can choose any hyperlink to view a list of the individual errors in the Load Test Errors dialog box. For more information, see Analyze load test results .	Yes
Tests/Sec	The rate per second of the test during the load test run.	Yes
Passed/Sec	The rate per second of this test during the load test run, for the instances of this test that passed.	No
Failed/Sec	The rate per second of this test during the load test run, for the instances of this test that failed.	No
Test Time	The average time to execute the test during the load test run. The units are seconds.	Yes
90% Test Time	The 90th percentile value for Test Time.	No
95% Test Time	The 95th percentile value for Test Time.	Yes
Requests/Test	The average number of requests in the test if it is a web performance test.	No

The Transactions table

The **Transactions** table displays a list of transactions that occurred during a load test run. Transactions refer to either transactions defined in a web performance test, or timers defined in a unit test. Transaction does not refer to database transactions.

The following table lists the columns in the **Transactions** table.

NOTE

To view all columns, you must enable the Timing Details Storage property that is associated with the active run setting. For more information, see [How to: Specify the Timing Details Storage property](#).

COLUMN	DESCRIPTION	VISIBLE WITHOUT TIMING DETAILS
Transaction	The name of the transaction.	Yes
Scenario	The name of the scenario.	Yes
Test	The name of the test.	Yes
Total	The total number of transactions issued during the load test run.	Yes
Transaction Time	The time to execute the transaction during a load test run. For web performance tests, think time is included in the calculation. The units are seconds.	No
Response Time	The response time for the web performance test transaction in a load test run. Response Time is different from Transaction Time in that Response Time does not include any think time that occurred during the transaction. The units are seconds.	No
Ave. Transaction Time	The average transaction time. This time includes think times. For example, if you have three requests and each has a think time, this time will include those think times and the actual time to execute requests.	No
Ave. Response Time	The average response time for a web performance test transaction in a load test run. Response Time is different from Transaction Time in that Response Time does not include any think time that occurred during the transaction. The units are seconds.	No
Min Response Time	This does not include think times.	No
Max Response Time	This does not include think times.	No
Median Response Time	This does not include think times.	No

COLUMN	DESCRIPTION	VISIBLE WITHOUT TIMING DETAILS
90% Response Time	The 90th percentile value for Transaction Time. This does not include think times. Note: This is different from Visual Studio Team System 2008 Test Load Agent, which used the 90% Transaction Time value.	No
95% Response Time	The 95th percentile value for Transaction Time. This does not include think times. Note: This is different from Visual Studio Team System 2008 Test Load Agent, which used the 95% Transaction Time value.	No
99% Response Time	The 99th percentile value for Transaction Time. This does not include think times.	No
Std Dev Response Time	This does not include think times.	No

The Errors table

When you run a load test, you can analyze errors that occur. Analyzing errors and adjusting your tests are an important part of the load test process. If any errors occurred, an **errors** hyperlink appears on the load test status bar and specifies the number of errors that occurred. To display the errors table, you choose the hyperlink.

The errors table groups the errors that occurred during a load test by the type and subtype of the error. There is also a **total** line in the table that specifies the total count of all the errors that occurred.

The errors table contains the following columns:

COLUMN	DESCRIPTION	VISIBLE BY DEFAULT
Type	The type of the error. For example, <code>HttpError</code> .	Yes
SubType	The subtype of the error. For example, <code>LoadTestException</code> .	Yes
Count	The number of errors of this type that occurred during the load test. The entries in this column appear as hyperlinks. You can choose any hyperlink to view a list of the individual errors.	Yes
Last Message	A message that describes the error. For example, <code>404 - NotFound</code> .	Yes

For more information, see [Working with load test tables](#).

Drill down to the error list

The errors table groups the errors by the type and subtype of the error. To view a table of the individual errors, you display the **Load Test Errors** dialog box. To display the dialog box, choose a hyperlink in the **Count** column of the errors table. You can also display the dialog box by right-clicking a row in the errors table that is

populated, and choosing **Errors**.

NOTE

Only the first 1,000 instances of any error type and subtype combination are collected. When you display the **Load Test Errors** dialog box, you will see at most the first 1,000 instances that error.

The **Load Test Errors** table contains the following columns:

COLUMN	DESCRIPTION
Time	The time during the load test at which the error occurred.
Agent	The name of the agent computer on which the error occurred. This is important when you run load tests using test controllers and test agents. For more information, see Install and configure test agents .
Test	The name of the web performance test in which the error occurred.
Scenario	The name of the scenario in which the error occurred.
Request	The URL of the request in which the error occurred.
Type	The type of the error. For example, <code>HttpError</code> .
SubType	The subtype of the error. For example, <code>LoadTestException</code> .
Text	The text of the error message. For example, <code>404 - NotFound</code> .
Stack	The entries in this column are either empty, or the word Stack is formatted as a hyperlink. You can choose the hyperlink to view a stack trace of the error.
Details	The entries in this column are either empty, or the word TestLog is formatted as a hyperlink. This link can help you isolate errors in the load test. For example, choosing the TestLog link on a web performance test request error will open up the results for the web performance test in the Web Performance Test Results Viewer and highlight the request error.

NOTE

You can sort the table by choosing the column headers.

The SQL Trace data table

You can collect SQL trace data during a load test run to analyze later. Collecting trace data lets you identify the slowest running queries and stored procedures in the SQL Server database being tested.

If SQL tracing is enabled, a file is created during the load test run that contains the trace data. This data is automatically saved in the Load Test Results Store at the end of the test run and the trace file is deleted. You analyze the trace data in the **SQL Trace** table after your load test has completed.

To view SQL trace data

1. In the Load Test Analyzer, choose **Tables** on the toolbar to make sure that the table grid is displayed.
2. In the **Table** drop-down list box, select **SQL Trace**.
3. The trace data that was collected during the run is displayed in the grid. The table lists the slowest running SQL operations sorted by duration, with the slowest at the top. Typically, the **Duration** column is the first column to examine. The data is displayed in milliseconds.

The columns displayed are as follows:

- **Event Class**
- **Duration**
- **CPU**
- **Reads**
- **Writes**
- **TextData**
- **StartTime**
- **EndTime**

If you want to trace SQL events other than the data identified in these columns, you can set up your own custom SQL tracing using the SQL Profiler tool, separate from Visual Studio.

Tile load test tables

When you view the results of a load test run, you can view the data as detailed tables. To switch to table view, choose **Tables** on the **load test** toolbar. The tables that are available are **Errors**, **Pages**, **Requests**, **SQL Trace**, **Tests**, **Thresholds**, and **Transactions**. For more information, see [Working with load test tables](#).

In table view, you can view up to four tables at a time without the tables overlapping.

To tile tables

1. On the **Load Test Analyzer** toolbar, choose **Tables**.

Table view opens. The default layout is two horizontal panels.

2. On the **Load Test Analyzer** toolbar, choose the **layout** button and then choose one of the following options:
 - **One Panel**
 - **Two Horizontal Panels**
 - **Three Horizontal Panels**
 - **Four Horizontal Panels**
3. To switch between the different tables, use the drop-down list above the table grid in each panel.

NOTE

You cannot display the same table in more than one panel. If you change the table displayed in one panel to a table already displayed in another panel, the tables switch panels.

See also

- [Analyze load test results](#)
- [How to: Access load test results for analysis](#)
- [Analyze load test results in the Graphs view](#)
- [Analyze threshold rule violations](#)
- [Manage load test results in the Load Test Results Repository](#)
- [Load test results summary overview](#)

How to: View web page response time in a load test using the Load Test Analyzer

1/1/2020 • 5 minutes to read • [Edit Online](#)

The time it takes for each web page to load is known as *response time*. When you create a web performance test, you can set a response time goal for each web page request in your web performance test.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

If you run your web performance test under stress in a load test, you will be able to analyze the following information for each page:

- The average response time for the page.
- The percent of test iterations that meet the response time goal for the page.
- You can analyze web page response times by using the Tables view or the Graphs view in the **Load Test Analyzer**.
- Analyzing web page response times in the tables view
- Analyzing web page response times in the graphs view

View response time data in a table

1. In the **Load Test Analyzer**, choose **Tables** on the toolbar to make sure that the table grid is displayed.
2. In the **Table** drop-down list box, select **Pages**.
3. The data for each page is displayed in the grid. The following columns are ordinarily displayed.

COLUMN HEADING	DESCRIPTION
Page	The name of the web page.
Scenario	The name of the scenario. Important if you have more than one scenario in your web performance test.
Test	The name of the web performance test. Important if you have more than one web performance test in your load test.
Network	<p>The network type.</p> <p>By default, this data is not collected. To collect this data, in the Load Test Editor, under the Run Settings node, select the run setting node to change. In the Properties window, for the Timing Details Storage property, select AllIndividualDetails.</p>

COLUMN HEADING	DESCRIPTION
Total	The total number of requests that were made for the web page. This is the total for all iterations in the load test.
Ave	<p>Average page response time.</p> <p>By default, this data is not collected. To collect this data, in the Load Test Editor, under the Run Settings node, select the run setting node to change. In the Properties window, for the Timing Details Storage property, select AllIndividualDetails.</p>
Min	<p>The minimum page response time.</p> <p>By default, this data is not collected. To collect this data, in the Load Test Editor, under the Run Settings node, select the run setting node to change. In the Properties window, for the Timing Details Storage property, select AllIndividualDetails.</p>
Median	<p>The median page response time.</p> <p>By default, this data is not collected. To collect this data, in the Load Test Editor, under the Run Settings node, select the run setting node to change. In the Properties window, for the Timing Details Storage property, select AllIndividualDetails.</p>
90%	<p>The 90th percentile for the response time. This indicates that 90% of the pages responded faster than this number, and 10% of the pages responded more slowly.</p> <p>By default, this data is not collected. To collect this data, in the Load Test Editor, under the Run Settings node, select the run setting node to change. In the Properties window, for the Timing Details Storage property, select AllIndividualDetails.</p>
95%	<p>The 95th percentile for the response time. This indicates that 95% of the pages responded faster than this number, and 5% of the pages responded more slowly.</p>
99%	<p>The 99th percentile for the response time. This indicates that 99% of the pages responded faster than this number, and 1% of the pages responded more slowly.</p> <p>By default, this data is not collected. To collect this data, in the Load Test Editor, under the Run Settings node, select the run setting node to change. In the Properties window, for the Timing Details Storage property, select AllIndividualDetails.</p>
Max	<p>The maximum page response time.</p> <p>By default, this data is not collected. To collect this data, in the Load Test Editor, under the Run Settings node, select the run setting node to change. In the Properties window, for the Timing Details Storage property, select AllIndividualDetails.</p>

COLUMN HEADING	DESCRIPTION
Std Dev	By default, the standard deviation data is not collected. To collect this data, in the Load Test Editor , under the Run Settings node, select the run setting node to change. In the Properties window, for the Timing Details Storage property, select AllIndividualDetails .
Page Time	The average response time for all requests that were made for the web page.
Goal	The page time goal. This is a constant value for the page. Note: Page Time Goal is displayed only when the goal has been defined for the request in the web performance test.
% Meeting Goal	The percent of the requests that were made for the web page that met the response time goal.

For more information, see [Analyze load test results and errors in the Tables view](#).

View response time data in a graph

You can also view response time data in a graph to see how it changes over time during your load test. This is especially useful if your load pattern increases as the test runs (for example, if you use the step load pattern). For more information, see [Edit load patterns to model virtual user activities](#).

To view response time data in a graph:

1. In the **Load Test Analyzer**, choose **Graphs** on the toolbar to make sure that the graph is displayed.
2. In the **Counters** window, expand the node of the scenario in which you are interested (for example, **Scenario1**).
3. Expand the node of the web performance test in which you are interested.
4. Expand the node **Pages**.
5. Expand the node of the page in which you are interested.
6. Right-click **% Pages Meeting Goal** and then choose **Show Counter on Graph**.

The data is added to the graph.

7. (Optional) Repeat the previous step for **Avg. Page Time**, **Page Response Time Goal**, and **Total Pages**.

NOTE

Page Response Time Goal is constant.

For more information, see [Analyze load test results in the Graphs view](#).

See also

- [Analyze load test results and errors in the Tables view](#)
- [How to: Access load test results for analysis](#)
- [Analyze load test results](#)

Analyze load test results in the Graphs view of the Load Test Analyzer

1/1/2020 • 4 minutes to read • [Edit Online](#)

The results of a load test are displayed as data in several different panes.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To display test results as graphs, choose **Graphs** on the **load test** toolbar. Each individual graph is displayed in a panel with the graph name displayed at the top in a drop-down list. To display a different graph in the panel, choose a different graph name from the list.

Up to four graph panels can be displayed at a time. You can switch between different panel layouts by using the **panel layout** toolbar button.

Several built-in graphs are provided. You can use the built-in graphs as is or you can customize them. Additionally, you can create your own graphs. For more information, see [How to: Add and delete counters on graphs](#) and [How to: Create custom graphs](#).

Built-in graphs

The following table lists the built-in graphs that are available to analyze load test results.

GRAPH NAME	DESCRIPTION
Key Indicators	Counters that describe basic aspects of test performance such as user load, throughput, and response time.
Test Response Time	Data about the amount of time tests take to run.
Page Response Time	The average response time for web pages that are accessed during the load test.
System under Test	Information about the computers on which the application being tested runs. This includes data about memory use, the processor, the physical disk, processes. By default, Only the Available Mbytes and Processor Time counters are collected.
Controller and Agents	Information about the computers on which the load tests run. This includes data about memory use, the processor, the physical disk, processes. By default Only the Available Mbytes and Processor Time counters are collected.

GRAPH NAME	DESCRIPTION
Transaction Response Time	The average response time for transactions that occur during the load test.

You can display different counters on the graph both at run time and after a test has run.

NOTE

Only response time performance counters can be added to an automatically generated response time graph.

The counter information displays both in the graph and in the legend underneath the graphs. You can also zoom in on a section of the graph. For more information, see [How to: Zoom in on a region of the graph](#).

Counters displayed in graphs

Graphs display *counters*. Counters refer to the data gathered during a load test, such as tests per second or average test time. For more information about counters, see [Specifying the counter sets and threshold rules for computers in a load test](#).

The legend for the counters that are displayed in the graphs shows several columns of useful data about the load test run. To turn off the display of any data in the graph, clear the check box in the row in the legend.

The legend contains the following columns:

COUNTER	THE NAME OF THE COUNTER
Instance	The name of the counter instance.
Category	The name of the counter category.
Computer	The name of the computer to which the counter is collected.
Color	The color of the line in the graph.
Range	Indicates the number that is represented by 100 on the graph for that counter. For example, for a range whose upper value is 10,000, the 100 label at the top of the graph represents 10,000.
Min	Indicates the minimum value for the counter in milliseconds.
Max	Indicates the maximum value for the counter in milliseconds.
Avg	Indicates the average value for the counter in milliseconds.
Last	Shows the value of the counter during the most recent sampling interval in milliseconds.

Tasks

TASKS	ASSOCIATED TOPICS
<p>Customize the graphs by using the legend: The Graphs view legend displays information for each performance counter that is associated with a graph. You can use the legend to remove performance counters, highlight performance counters in the graph, and customize the plotting options.</p>	<ul style="list-style-type: none"> - Using the Graphs view legend to analyze load tests
<p>Display counters on graphs: You can add different kinds of data to a load test results graph by placing counters on the graph.</p>	<ul style="list-style-type: none"> - How to: Add and delete counters on graphs
<p>Zoom in on graphs: After a load test has finished, you can use zoom bars to zoom in and scroll to a region of the graph. By zooming in, you can examine the data that was generated during a load test run in finer detail.</p>	<ul style="list-style-type: none"> - How to: Zoom in on a region of the graph
<p>Tile graphs: You can arrange load test results graphs in any of several patterns. You can tile up to four graphs.</p>	
<p>Create custom graphs: You can design graphs that display specific information about load test results. You design a custom graph by specifying the load test counters that the graph will display.</p>	<ul style="list-style-type: none"> - How to: Create custom graphs
<p>Export the performance counters data in the graph: You can export the graph data to Microsoft Excel by using the Export Graph Data to Excel button on the Load Test Analyzer toolbar while you are in the Graphs view.</p>	

Related tasks

[Analyze load test results and errors in the tables view](#)

[How to: Access load test results for analysis](#)

[Analyze load test results](#)

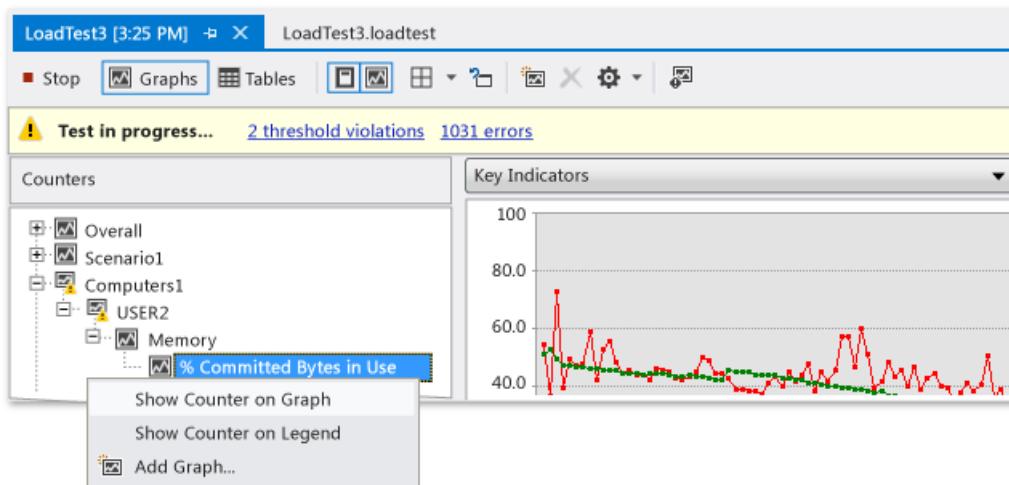
See also

- [How to: Add and delete counters on graphs](#)
- [How to: Create custom graphs](#)
- [How to: Zoom in on a region of the graph](#)

How to: Add and Delete Counters on Graphs in Load Test Results

1/16/2020 • 4 minutes to read • [Edit Online](#)

You can use the **Counters** panel to add performance counters to a graph.



NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Performance Counter Sampling Interval Considerations

Choose a value for the **Sample Rate** property in the load test run settings based on the length of your load test. A smaller sample rate, such as the default value of five seconds, requires more space in the load test results database. For longer load tests, increasing the sample rate reduces the amount of data that you collect. For more information, see [How to: Specify the sample rate](#).

Here are some guidelines for sample rates:

LOAD TEST DURATION	RECOMMENDED SAMPLE RATE
< 1 Hour	5 seconds
1 - 8 Hours	15 seconds
8 - 24 Hours	30 seconds
> 24 Hours	60 seconds

Considerations for including Timing Details to Collect Percentile Data

There is a property in the run settings in the Load Test Editor named **Timing Details Storage**. If the **Timing Details Storage** property is enabled, then the time to execute each individual test, transaction, and page during the load test will be stored in the load test results repository. This allows for 90th and 95th percentile data to be shown in the **Load Test Analyzer** in the Tests, Transactions, and Pages tables.

There are two choices for enabling the **Timing Details Storage** property in the run settings properties named **StatisticsOnly** and **AllIndividualDetails**. With either option, all the individual tests, pages, and transactions are timed, and percentile data is calculated from the individual timing data. The difference is that with the **StatisticsOnly** option, as soon as the percentile data has been calculated, the individual timing data is deleted from the repository. This reduces the amount of space that is required in the repository when you use timing details. However, advanced users might want to process the timing detail data in other ways, by using SQL tools. If this is the case, the **AllIndividualDetails** option should be used so that the timing detail data is available for that processing. Additionally, if you set the property to **AllIndividualDetails**, then you can analyze the virtual user activity using the **Virtual User Activity** chart in the **Load Test Analyzer** after the load test completes running. For more information, see [Analyze virtual user activity in the Details view](#).

The amount of space that is required in the load test results repository to store the timing details data could be very large, especially for longer running load tests. Also, the time to store this data in the load test results repository at the end of the load test is longer because this data is stored on the load test agents until the load test has finished executing. When the load test finishes, the data is stored into the repository. By default, the **Timing Details Storage** property is enabled. If this is an issue for your testing environment, you might want to set the **Timing Details Storage** to **None**.

For more information, see [How to: Specify the timing details storage property](#).

To display a particular performance counter on a load test graph

1. After a load test is finished, or after you load a test result, in the Load Test Analyzer's toolbar, choose **Graphs**.

The **Counters** panel is displayed in the Graphs view.

NOTE

If the **Counters** panel is not visible, choose **Show Counters Panel** on the toolbar.

2. In the **Counters** panel, expand nodes in the hierarchy until you find the performance counter that you want to see displayed graphically.

For example, to display the available memory on a computer where tests are running, expand **Computers**, expand the node for the computer, and then expand **Memory**. You will see the **Available MBytes** counter.

3. Choose the graph on which you want to display the performance counter.
4. Right-click the performance counter in the **Counters** panel and select **Show Counter on Graph**.

TIP

To temporarily stop displaying the performance counter's data on the graph, clear the check box for the performance counter in the Legend. This allows the min, max and average statistics to still be analyzed without viewing the trend line on the graph. This can be useful if the graph contains several overlapping performance counter plots while you are analyzing issues. For more information, see [Use the Graphs view legend to analyze load tests](#).

5. To remove the performance counter data from the graph, right-click the performance counter in the **Counter** column of the legend and select **Delete**.

- or -

Right-click the data line in the graph and select **Delete**.

- or -

Choose the performance counter in the **Counter** column of the legend or the data line in the graph, and then press the **Delete** key.

NOTE

You can also choose to place a performance counter on the legend but not on the graph by using the **Add Counter on Legend** command.

See also

- [Analyze load test results in the Graphs view](#)
- [How to: Create custom graphs](#)

How to: Create custom graphs in load test results

1/16/2020 • 2 minutes to read • [Edit Online](#)

You can design graphs that display specific information about load test results. You design a custom graph by specifying the load test counters that the graph will display.

You can perform the following procedure either while a load test is running or after it has finished running.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To create a custom load test results graph

1. On the **Load Test** toolbar, choose **Add New Graph**.

- or -

On the **Load Test Analyzer**, right-click in the **Counters** panel or in a graph, and then select **Add Graph**.

The **Enter Graph Name** dialog box is displayed.

2. Under **Graph name**, type a name for the graph, and choose **OK**.

The new graph appears in the **Load Test Analyzer**. It appears in the currently selected graph panel; it replaces the graph that was displayed in that panel.

3. Customize the new graph by adding counters. For more information, see [How to: Add and delete counters on graphs](#).

See also

- [Analyze load test results in the Graphs view](#)
- [How to: Add and delete counters on graphs](#)

Use the Graphs view legend to analyze load tests

1/1/2020 • 2 minutes to read • [Edit Online](#)

The Load Test Analyzer's Graphs view includes a legend panel that displays information for each performance counter that is associated with the currently selected graph.

Show on graph		Range of Y-axis			Statistics				
Counter	Instance	Category	Computer	Color	Range	Min	Max	Avg.	Last
Key Indicators									
<input checked="" type="checkbox"/> User Load _Total LoatTest... USER1									
				Red	100	25	25	25	25
<input checked="" type="checkbox"/> Errors/Sec _Total LoatTest... USER1									
				Green	100	0	8.80	4.04	1.80
<input checked="" type="checkbox"/> Threshold Vi... _Total LoatTest... USER1									
				Blue	10	0	1.20	0.024	0
Test Response Time									
System under Test									
Controller and Agents									
Performance counter									
Color and line style									

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

The following information is contained within the legend:

- Show on graph:** Use the check boxes to specify whether the line for a particular counter, such as **User load** or **Errors/Sec**, is plotted on the graph. Select a check box if you want the line to be plotted on the graph. Clear a check box to remove the plot line from the graph. When a plot line is removed, the statistics for the counter continue to display in the legend.
- Range:** This column displays the performance counter's y-axis range. By default, this value will automatically adjust as the range of sample data changes. An automatically adjusted range will always be the next power of 10 greater than Max value; this includes negative powers of ten. A graph can contain a variety of counters, each with a different range. Therefore, the y-axis is not labeled with any specific range, but is instead labeled with values from 0-100 that represent a percentage of the total range for each counter. For example, for a counter with a range of 1000, a data point of 60 on the y-axis would correspond to a value of 600 for the counter.

NOTE

You can turn off the automatic range value adjustment by locking the range to a specific value. When the range is locked, any values exceeding the range are displayed as the maximum value you specified at the top of the graph. Use the **Plot Options** dialog box to lock the range at a specific value.

- Counter:** The four columns named **Counter**, **Instance**, **Category**, and **Computer** together uniquely identify the performance counter.
- Color:** The **Color** column shows the color and line style of the plotted line for the performance counter. Use the **Plot Options** dialog box to change the color or line style of a performance counter on the graph. The **Plot Options** dialog box is available from the legend's shortcut menu.
- Statistics:** The **Min**, **Max**, **Avg** and **Last** columns show the respective statistics for the performance counter. These values correspond to the data that is displayed on the visible region of the graph. For

example, if you zoom into a region of a run, the legend statistics will reflect values for only the zoomed area. The "Last" column is the value of the performance counter on the most recently completed sampling interval.

NOTE

The Last column only displays in the Load Test Analyzer's Legend while the load test is running.

For more information, see [How to: Zoom in on a region of the graph](#).

Selecting an item in the legend does the following:

- Allows the item to be removed from both the legend and the graph. Either right-click the item and select **Delete**, or press the **Delete** key.
- Highlights the plotted line on the graph.
- Causes the data grid to display data for the selected item.
- Lets you access the **Plot Options** dialog box for the counter.

TIP

You can use **Graph Options drop-down** button in the **Load Test Analyzer** toolbar and select **Show Legend** to show or hide the **Legend** panel that is associated with the graph view.

See also

- [How to: Zoom in on a region of the graph](#)
- [Analyze load test results in the Graphs view](#)

How to: Zoom in on a region of the graph in load test results

1/1/2020 • 5 minutes to read • [Edit Online](#)

After a load test has finished, you can use zoom bars to zoom in and scroll to a region of the graph. By zooming in, you can examine the data that was generated during a load test run in finer detail.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Zoom in is available only when you are analyzing the result of a completed load test, not while you are observing the results of a running test.

The zoom control is visible only in the **Load Test Analyzer** when you view a load test result in zooming mode. Zooming mode is established in the Graph view when either a load test has completed or a load test that has previously run is loaded. You can show or hide the zoom controls on the graphs by using **Show Zoom Controls** on the toolbar.

The **horizontal x-axis zoom** can be adjusted to analyze specific time periods during the load test. The **vertical y-axis zoom** can be adjusted to analyze specific value ranges for the counters that are included in the graph.

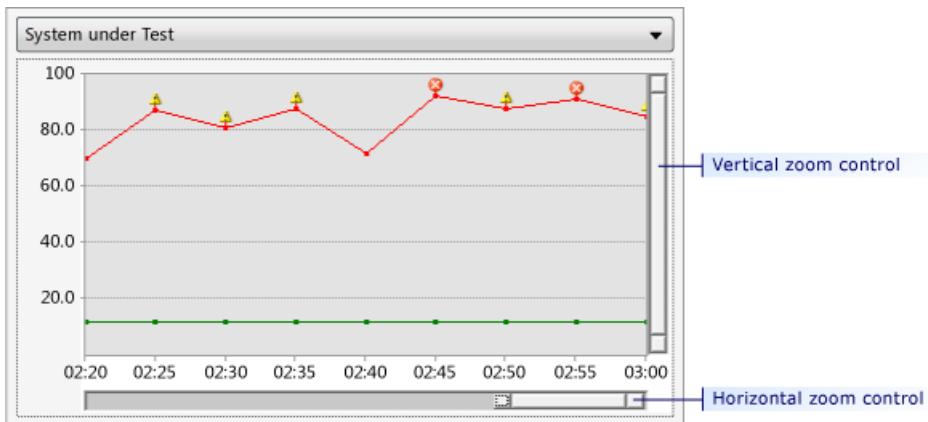
Both the **horizontal timeline** and the **vertical value range** zoom controls can be adjusted by using the mouse. The **horizontal timeline control** can also be adjusted by using the left and right arrow keys. By using the arrow keys to adjust the zoom control, you can adjust the windows range by 1 sampling interval at a time. Using the **Shift** and arrow keys makes adjustments of 10 sampling intervals.

To adjust the zoom control by using the arrow key, first set the focus on the zoom control by using the **Tab** key. When the left slider has the focus, the arrow keys will move the starting boundary of the zoom window by 1 interval left or right. When the focus is on the center slider, you can use the arrow keys to scroll the zoom window left or right 1 sampling interval without changing the size of the zoom window. And finally, the right side slider moves, extending or reducing the range of the end of the zoom window by 1 sampling interval.

To return the horizontal and vertical zoom controls to show the full timeline and value ranges, you can use the **Zoom Out Horizontal** option, the **Zoom Out Vertical** option, or the **Zoom Out Both** option in the pop-up menu on the graph.

TIP

You can use **Synchronize Horizontal Zoom Controls** in the toolbar to switch on or off automatic horizontal zoom synchronization. With synchronization on, any zooming you apply to a graph will also be applied to any other graphs on the Graphs view.



In the previous illustration, the **System under Test** graph has been zoomed in to investigate threshold issues. The threshold violations have been enabled by using **Show Threshold Violations On Graph** from the **Graph Options** drop-down in the toolbar.

For more information, see [Analyze load test results in the Graphs view](#).

Display graphs

Before you change the display of a graph by zooming in or out or by scrolling, follow this procedure to display graphs.

To display graphs:

1. Run a load test until it is completed.
 2. At the end of the load test run, choose **Yes** in the dialog box that asks about viewing results from the load test results store.
- or -

View the details of a previously run load test. For more information, see [How to: Access load test results for analysis](#).

3. Choose **Graphs** if your graphs are not displayed.
4. If zoom bars are not displayed, choose **Show Zoom Controls**.

Two zoom bars are available for each graph. The zoom bar that controls vertical scale appears to the left of the graph. The zoom bar that controls horizontal scale appears under the graph.

Each zoom bar has two handles. A handle is a rectangular area at each end of the zoom bar.

Zoom and scroll

When you have multiple graphs displayed, you can keep them synchronized so that they display the same portion of the load test run.

To synchronize zooming and scrolling

1. On the **Load Test Analyzer**, choose **Synchronize Horizontal Zoom Controls**.

When the **Synchronize Horizontal Zoom Controls** button is selected, zooming and scrolling the time scale of an individual graph also zooms and scrolls the time scale of the other graphs.

2. Again, choose **Synchronize Horizontal Zoom Controls**.

When the **Synchronize Horizontal Zoom Controls** button is not selected, zooming and scrolling the time scale of an individual graph affects that graph only.

To zoom and scroll to a region of the graph

1. On the zoom bar under a graph, drag the left-side handle to the right.

This zooms in on the latter part of the test run. Similarly, dragging the right-side handle to the left zooms in on earlier parts of the test run.

2. To zoom in on a particular area, slide both handles toward the center of a graph.

The closer the two handles are to each other, the more you zoom in to display shorter, finer segments of the load test.

Choose the center section of the zoom bar and then drag it to scroll to a particular point in the load test.

To zoom to a region of the graph by choosing and dragging

1. Choose a graph at one end of the zoom area.
2. Drag the mouse pointer to the other end of the zoom area.
3. Release the mouse button.

This magnifies the area that you defined by choosing and dragging.

The following procedure describes how to quickly zoom out without having to adjust the ends of the zoom bar.

To zoom out

1. Right-click a zoomed-in graph.
2. On the shortcut menu, select **Zoom Out Horizontal**.

This zooms out to show the entire duration of the load test run.

See also

- [Analyze load test results in the Graphs view](#)
- [Analyze load test results](#)
- [How to: Add and delete counters on graphs](#)

Analyzing load test virtual user activity in the Details view of the Load Test Analyzer

1/1/2020 • 2 minutes to read • [Edit Online](#)

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life blog post](#).

Virtual User Activity Chart

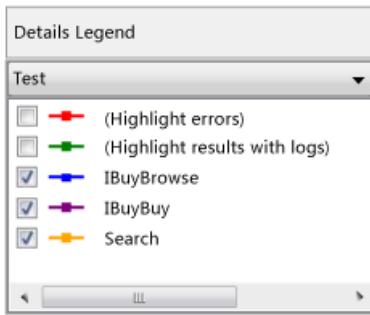


The **Details** view displays the **Virtual User Activity Chart**, which is used to visually analyze what the individual virtual users did during the load test. **Virtual User Activity Chart** lets you see patterns of user activity, load patterns, correlate failed or slow tests, and see requests with other virtual user activity. The **Virtual User Activity Chart** can also help you determine spikes in CPU usage, drops in requests per second, and what tests or pages were running during the spikes and drops.

NOTE

Before you run the load test for which you want to use the **Virtual User Activity Details Chart**, you must verify that the **Timing Details Storage** property is set to the **AllIndividualDetails** option by using the Load Performance Test Editor.

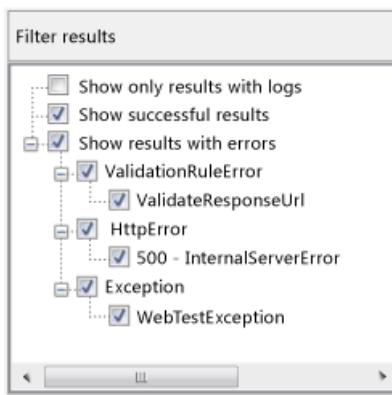
Details Legend Panel



The details legend panel is visible in the **Virtual User Activity Chart**. The details legend pane lets you filter out tests, pages and transactions based on several different criteria. For example, you can remove certain tests from the view, or remove all successful tests, or remove tests which failed with certain failures. You can also remove all tests that do not have logs.

You can highlight tests which failed, which displays all failed tests colored in red. You can also highlight tests that have test logs. Tests with logs will be colored in green.

Filter results Panel



The Filter results panel is visible in the **Virtual User Activity Chart**. The Filter results panel can filter on the following:

- **Show only results with logs** Displays only test results that have test logs associated with them.
- **Show successful results** Displays successful results.
- **Show results with errors** Displays results with errors that can help in debugging.

Tasks

TASKS	ASSOCIATED TOPICS
Run your load test: After you have created a load test and configured it to enable virtual user activity data collecting, you must run the test until it is complete in order to view the Virtual User Activity Chart .	
View the load test results that contain the virtual user activity data: After your load test has been created, configured, and has completed running, you can view the virtual user activity data by using the Virtual User Activity Chart .	<ul style="list-style-type: none"> - Analyze load test results - How to: Analyze what virtual users are doing during a load test
Isolate performance issues in load tests: You can use the Virtual User Activity Chart to help isolate performance issues in your load test.	<ul style="list-style-type: none"> - Walkthrough: Using the virtual user activity chart to isolate issues

See also

- [Analyze load test results](#)
- [Analyze load test results and errors in the Tables view](#)

How to: Analyze what virtual users are doing during a load test using the virtual user activity chart

10/18/2019 • 3 minutes to read • [Edit Online](#)

View the virtual user activity that's associated with your load test by using the **Virtual User Activity Chart**. Each row in the chart represents an individual virtual user. The **Virtual User Activity Chart** shows you exactly what each virtual user was executing during the test. You can see patterns of user activity, load patterns, correlate failed or slow tests, and see requests with other virtual user activity. The **Virtual User Activity Chart** is available only after the load test has finished running.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

The procedures below demonstrate how to view the **Virtual User Activity Chart**, how to investigate a specific user's activity, and how to use filtering.

To view the Virtual User Activity Chart in your load test results

1. To view the virtual user data, you must first configure the **All Individual Details** setting for the **Timing Details Storage** property that is associated with your load test. Then run the load test.
2. After your load test runs, the test results summary page is displayed. Choose the **User Detail** button on the toolbar.

-or-

Open the Graphs view by choosing the **Graphs** button on the toolbar. Right-click a graph and then select **Go to user detail**.

If you use this option, the **Virtual User Activity Chart** will auto-zoom to the part of the test that you right-clicked. For example, if your pointer is located on approximately the 30 second mark, the detail view will display approximately on the 30 second mark in the **Zoom to time period** tool at the bottom of the **Virtual User Activity Chart**.

Next, you can use investigate a specific user's activity details in the **Virtual User Activity Chart**.

To investigate a specific user's activity in the Virtual User Activity Chart

1. Use the Zoom to time period tool at the bottom of the **Virtual User Activity Chart** to select an area on the chart where you want to investigate details on a specific user.
2. Hover your pointer over a detail in the graph. Notice that the following information is displayed in the tool tip:
 - **User Id**
 - **Scenario**
 - **Test**

- **URL** (Does not display in a test or transaction)
- **Outcome**
- **Browser** (Does not display in a test or transaction)
- **Network**
- **Start Time**
- **Duration**
- **Agent**
- **Test log** (Link to the test log)

NOTE

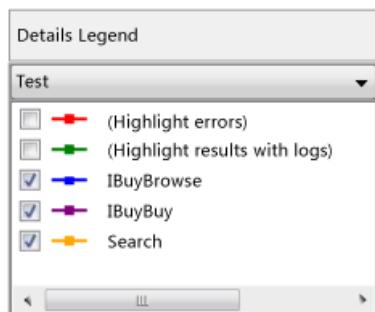
To assist in debugging your application, if you choose the **Test log** link, the web test result or unit test result associated with the log open.

Next, you can use the filtering and highlighting operations available in the **Virtual User Activity Chart**.

To use filtering options in the Virtual User Activity Chart

1. In the **Details Legend**, use the drop-down list to select either **Test**, **Page**, or **Transaction**.

Details Legend panel



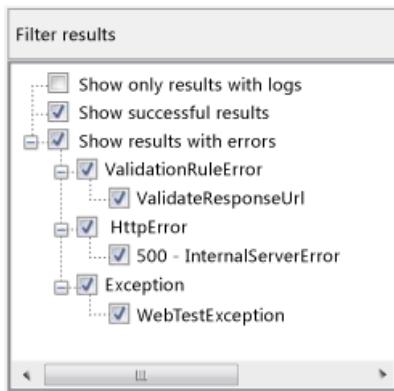
2. Select or clear the check boxes for the errors, logs, tests, search, and aspx pages that are associated with the load test.

The **Virtual User Activity Chart** updates accordingly.

The **Virtual User Activity Chart** provides the ability to filter out tests, pages, and transactions based on several different criteria. You can remove certain tests from the view, or remove all successful tests, or remove tests that failed with certain failures. You can also remove all tests that do not have logs.

For example, you can select the **(Highlight errors)** option, which displays all errors on the chart colored in red. You can also select the **(Highlight results with logs)** option, which displays all the test results that have logs colored in green in the chart.

Filter results panel



3. In the **Filter results**, select or clear the check boxes for the following filter options:

- **Show only results with logs** Displays only test results that have test logs associated with them.
- **Show successful results** Displays successful results.
- **Show results with errors** Displays results with errors that can assist in debugging.

NOTE

The list of error types that are listed under the **Show results with errors** node can be further investigated by choosing the **Tables** button in the **Web Performance Test Results Viewer** toolbar. For more information, see [Analyze load test results and errors in the Tables view](#).

The **Virtual User Activity Chart** updates accordingly.

See also

- [Analyze virtual user activity in the Details view](#)
- [Walkthrough: Using the virtual user activity chart to isolate issues](#)

Walkthrough: Using the Virtual User Activity Chart to isolate issues

10/18/2019 • 3 minutes to read • [Edit Online](#)

In this walkthrough, you'll learn how to use the Virtual User Activity Chart to isolate errors that occurred for individual virtual users that ran your load test.

The Virtual User Activity Chart lets you visualize the virtual user activity that is associated with your load test. Each row in the chart represents an individual virtual user. The Virtual User Activity Chart shows you exactly what each virtual user was executing during the test. This lets you isolate performance issues by seeing patterns of user activity, load patterns, correlate failed or slow tests, and see requests with other virtual user activity. The Virtual User Activity Chart is available only after the load after has finished running.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Prerequisites

- Visual Studio Enterprise
- Complete these procedures:
 - [Record and run a web performance test](#).
 - [Create and run a load test](#)

Open the ColorWebApp solution created in the previous walkthroughs

1. Open Visual Studio.
2. Open the **ColorWebApp** solution that contains the *LoadTest1.loadtest*. This load test results from conducting the steps in the three walkthroughs that are listed at the beginning of this topic in the prerequisites section.

The remaining steps in this walkthrough assume a web application named ColorWebApp, a web performance test named *ColorWebAppTest.webtest* and a load test named *LoadTest1.loadtest*.

Run the load test

Run your load test to collect virtual user activity data.

- In the **Load Test Editor**, choose the **Run** button on the toolbar. LoadTest1 starts to run.

Isolate issues in the Virtual User Activity Chart

After you have run your load test and collected the virtual user activity data, you can view the data in the load test results by using the **Load Test Analyzer** Details view in the **Virtual User Activity Chart**. Additionally, you can use the **Virtual User Activity Chart** to help isolate performance issues in your load test.

To use the Virtual User Activity Chart in your load test results

1. After the load test is finished running, the **Summary** page for the load test results is displayed in the **Load Test Analyzer**. Choose the **Graphs** button on the toolbar.

The Graphs view is displayed.

2. On the **Page Response Time** graph, right-click near one of the threshold violation icons and select **Go to user detail**.

NOTE

You can use the **Details** button in the **Load Test Editor** toolbar to open the User Activity chart too. However, if you use the **Go to user detail** option, the **Virtual User Activity Chart** will automatically zoom in on the part of the test that you right clicked in the graph.

The Details view is displayed with the **Virtual User Activity Chart** focused on the time period when the threshold violations occurred.

On the y-axis, the horizontal plots represent individual virtual users. The x-axis displays time line for the load test run.

3. In the **Zoom to time period** tool below the **Virtual User Activity Chart**, adjust the left and right sliders until both are close to the threshold violation icon. This changes the time scale in the **Virtual User Activity Chart**
4. In the **Details Legend**, select the check box for **(Highlight errors)**. Notice that the virtual user who caused the threshold violation is highlighted.
5. In the **Filter results** panel, clear the check boxes for **Show successful results** and **HttpError** but leave the **ValidationRuleError** check box selected.

The **Virtual User Activity Chart** displays only the virtual users that spent more than 3 seconds on the *Red.aspx* page as specified by the threshold violation configured in the previous walkthrough.

6. Rest the mouse pointer over the horizontal line that represents the virtual user with the validation rule error for the threshold violation.
7. A tool tip is displayed with the following information:

- **User ID**
- **Scenario**
- **Test**
- **Outcome**
- **Network**
- **Start Time**
- **Duration**
- **Agent**
- **Test log**

8. Notice that **Test log** is a link. Choose the **Test log** link.
9. The ColorWebTest web performance test that is associated with the log opens in the **Web Performance Test Results Viewer**. This lets you isolate where the threshold violations occurred.

You can use various settings in both the **Details Legend** and **Filter results** panels to help in isolating performance issues, and errors in your load tests. Experiment with these settings and the **Zoom to time period** tool to see how the virtual user data is presented in the **Virtual User Activity Chart**.

See also

- [Analyze virtual user activity in the Details view](#)
- [Test controllers and test agents](#)
- [How to: Create a test setting for a distributed load test](#)
- [Install and configure test agents](#)
- [Collect diagnostic information using test settings](#)

Analyzing threshold rule violations in load tests

Using the Load Test Analyzer

1/1/2020 • 2 minutes to read • [Edit Online](#)

Threshold rules are associated with specific performance counters, and violations indicate that a performance counter exceeded or fell below a set value. When you run a load test, you can analyze violations that occur for the threshold rules you set up previously.

If any violations occurred, a **threshold violations** hyperlink appears on the **Load Test Analyzer** status bar and specifies the number of violations that occurred. You choose the hyperlink to display the threshold violations table. You can also view threshold violations in the **Counters** window, and on the graph.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

View threshold violations in the table

The threshold violations table displays the first 1,000 violations. The following table contains these columns:

COLUMN	DESCRIPTION	VISIBLE BY DEFAULT
Time	The time during the load test at which the violation occurred.	Yes
Computer	The name of the computer under test on which the violation occurred. Note: This is important when you run load tests on rigs.	Yes
Category	The category of the performance counter on which the violation occurred.	Yes
Counter	The name of the performance counter on which the violation occurred.	Yes
Instance	The performance counter instance on which the violation occurred.	Yes
Message	A message that describes the threshold violation. For example, The value 5 exceeds the critical threshold value of 0.	Yes

NOTE

You can sort the table by choosing the column headers.

For more information, see [Analyze load test results and errors in the Tables view](#).

View threshold violations in the Counters panel

You can view threshold violations in the **Counters** panel, in the tree that lists the performance counters for your load test. Icons in the **Counters** panel communicate threshold violations. The icon will be one of the following:

The icon will be one of the following:

- No threshold violation.
- A critical threshold violation occurred on the last interval.
- A critical threshold violation occurred on a prior interval.
- A warning threshold violation occurred on the last interval.
- A warning threshold violation occurred on a prior interval.

Optionally, threshold violations can be shown on the graph also. The threshold icon appears on the graph next to the data point where the threshold violation occurred.

In the counter tree, the icon for a threshold violation is propagated from the specific counter node, up to the root node. This alerts you to a violation on a counter that may not be visible in the tree because the tree has not been expanded.

View threshold violations on the graph

You can view threshold violations on the graph. Similar to the **Counters** panel, icons communicate threshold violations on the graph. The icons appear on the graph next to the data point where the threshold violation occurred. If a threshold violation occurs on a counter that does not appear on the graph, you can add it to the graph by dragging it from the **Counters** panel to the graph.

For more information, see [Analyze load test results in the Graphs view](#).

See also

- [Specifying the counter sets and threshold rules for computers in a load test](#)
- [Analyze load test results](#)
- [Analyze load test results and errors in the Tables view](#)

Report load tests results for test comparisons or trend analysis

1/1/2020 • 2 minutes to read • [Edit Online](#)

You can generate Microsoft Excel load test reports that are based on two or more test results.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Two types of load test reports are available:

- Run comparison—This report is actually two reports that display side-by-side comparison data using tables and bar charts.
- Trend—You can generate trend analysis on two or more reports. The results are displayed using line charts.

Either report can be used to share performance data with stakeholders and convey whether the overall performance and health of the system is getting better or worse.

Report definitions are stored in the load test database. When a report is saved, the definition for the report is saved in the database and can be reused later.

Also, the spreadsheet file can be shared with stakeholders so that stakeholders do not have to connect to the database to see the report.

NOTE

If you add comments to a load test, they appear in the Excel report.

Tasks

TASKS	ASSOCIATED TOPICS
Create a performance and stress report: You can create reports on your load and web performance tests, using Microsoft Excel.	- How to: Create load test performance reports using Microsoft Excel
Manually create a performance and stress report by using Microsoft Word: You can create reports on your load and web performance tests manually by copying and pasting summary, table, and graph data to a Microsoft Word document.	- How to: Manually create a load test performance report using Microsoft Word

See also

- [Analyze load test results](#)

How to: Create load test performance reports using Microsoft Excel

1/1/2020 • 7 minutes to read • [Edit Online](#)

You can generate Microsoft Excel load test reports that are based on two or more test results.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Two types of load test reports are available:

- **Run comparison** This creates a set of reports that compares the data from two load test results using tables and bar charts.
- **Trend** You can generate trend analysis on two or more load test results. The results are displayed using line charts, but the data is available in pivot tables.

TIP

You can also manually create Microsoft Word reports by copying and pasting data from the summary view, graphs view, and tables view. See [How to: Manually create a load test performance report using Microsoft Word](#).

Either report can be used to share performance data with stakeholders and convey whether the overall performance and health of the system is getting better or worse.

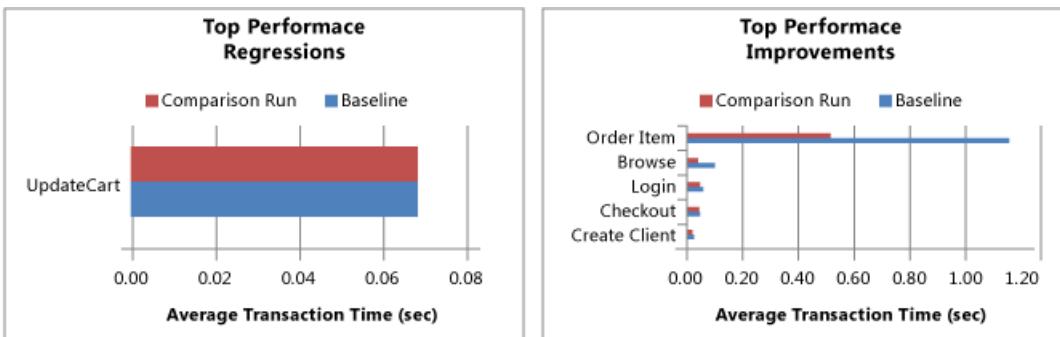
Report definitions are stored in the load test database. When a report is saved, the definition for the report is saved in the database and can be re-used later.

Also, the Excel workbook can be shared with stakeholders so that stakeholders do not have to connect to the database to see the report.

NOTE

You can share the Excel workbook; however, only users who have Visual Studio installed on their machine will be able to modify any of the spreadsheets. Other users will not see the **Load Test Report** option in the **Office** ribbon, but they will be able to view the workbook.

The following illustration is an example of a report that shows a correlation between a decline in transaction (Update Cart) speed and the degeneration of the (% Processor) counter. This points to a potential problem in the application code, instead of the database or network, and is a good candidate to diagnose by using the ASP.NET Profiler.



Excel reports can either be generated in the **Load Test Analyzer**, by using the **Create Excel Report** button in the toolbar, or from Excel by using the **Load Test Report** option in the **Load Test** tab of the **Office** ribbon.

NOTE

If you add comments to a load test, they appear in the Excel report.

To generate load test comparison reports using Excel

1. Before you generate a report, you must first run a load test.
2. You can create Excel load test reports in two ways:
 - After you complete a load test, in the **Load Test Results** page, choose the **Create Excel Report** button in the toolbar.

NOTE

If the **Create Excel Report** button is disabled in the **Web Performance Test Results Viewer** toolbar, you may need to run Microsoft Excel one time before it is enabled. When Visual Studio Enterprise is installed, the Visual Studio Enterprise load test add-in is copied to your computer for Microsoft Excel; however, Microsoft Excel must be run to complete the installation process for the add-in.

Microsoft Excel opens with the **Generate a Load Test Report Wizard**.

OR

- a. Open Microsoft Excel, select the **Load Test** tab in the **Office** ribbon and then choose **Load Test Report**.
The **Generate a Load Test Report Wizard** appears.
- b. In the **Select database which contains load tests** page, under **Server name**, type the name of the server containing the load test results.
- c. In the **Database name** drop-down list, select the database containing the load test results.
3. In the **How do you want to generate your report** page, verify that **Create a report** is selected and choose **Next**.

4. In the **What type of report do you want to generate** page, verify that **Run comparison** is selected and choose **Next**.
5. In the **Enter load test report details** page, type a name for your report in **Report Name**.
6. Select the load test you want to generate the report for and choose **Next**.
7. In the **Select the runs for your report** page, under **Select one or more runs to add to the report**, select two load test results that you want to compare in the report and choose **Next**.

NOTE

You can only generate a comparison report on two load test results. If you select either one load test result or more than two load test results, a warning message will appear.

8. In the **Select the counters for your report** page, under **Select one or more counters to add to the report** an expandable list of counters is available to customize your report. Select the counters that you want to compare from the two selected test runs in the report and choose **Finish**.
9. The Excel workbook report is generated with the following spreadsheet tabs:
 - **Table of Contents** - Displays the load test report name and provides a table of contents with links to the various tabs in the report.
 - **Runs** - Provides details on which two runs are being compared in the report.
 - **Test Comparison** - Provides bar graph details on performance regressions and improvements between the two runs being compared.
 - **Page Comparison** - Provides bar graph and percentage performance comparison data between the two runs on the various pages in the test runs.
 - **Machine Comparison** - Provides comparison data between the two runs based on the machines that were used.
 - **Error Comparison** - Compares the error types encountered between the two runs and the number of occurrences.

TIP

For better reports several properties are available in load tests and web performance tests that enable richer reports. The page request has two properties that are presented in the reports: Goal and Reporting Name. Page response times will be reported against goal, and the reporting name will be used instead of the URL in the reports. In a load test Run Settings, under Manage Counter Sets, the Computer Tags property is presented in the report machine names. This is very useful to describe the role of a particular machine in the report.

To generate load test trend reports using Excel

1. Before you generate a report, you must run a load test.
2. You can create Excel load test reports in two ways:
 - After you complete a load test, in the **Load Test Results** page, choose the **Create Excel Report** button in the toolbar.

NOTE

If the **Create Excel Report** button is disabled in the **Web Performance Test Results Viewer** toolbar, you may need to run Microsoft Excel one time before it is enabled. When Visual Studio Enterprise is installed, the Visual Studio Enterprise load test add-in is copied to your computer for Microsoft Excel; however, Microsoft Excel must be run to complete the installation process for the add-in.

Microsoft Excel opens with the **Generate a Load Test Report Wizard**.

OR

- a. Open Microsoft Excel, select the **Load Test** tab in the **Office** ribbon and then choose **Load Test Report**.

The **Generate a Load Test Report Wizard** appears.

- b. In the **Select database which contains load tests** page, under **Server name**, type the name of the server containing the load test results.
- c. In the **Database name** drop-down list, select the database containing the load test results.
3. In the **How do you want to generate your report** page, verify that **Create a report** is selected and choose **Next**.
4. In the **What type of report do you want to generate** page, verify that **Trend** is selected and choose **Next**.
5. In the **Enter load test report details** page, type a name for your report in **Report Name**.
6. Select the load test you want to generate the report for and choose **Next**.
7. In the **Select the runs for your report** page, under **Select one or more runs to add to the report**, select the load test results that you want to compare in the report and choose **Next**.
8. In the **Select the counters for your report** page, under **Select one or more counters to add to the report**, an expandable list of counters is available to customize your report. Select the counters that you want to compare for trend analysis and choose **Finish**.
9. The report is generated with a table of contents that has links to the various Excel workbook tabs generated in the report. The links are based on the counters selected for the trend report. For example, if you left the default counters selected in step 7, then the report will generate data which is presented in separate tabs in Excel for each counter listed in step 7. The data that is generated for each counter is presented in trend-style graphs.

TIP

For better reports several properties are available in load tests and web performance tests that enable richer reports. The page request has two properties that are presented in the reports: Goal and Reporting Name. Page response times will be reported against goal, and the reporting name will be used instead of the URL in the reports. In a load test Run Settings, under Manage Counter Sets, the Computer Tags property is presented in the report machine names. This is very useful to describe the role of a particular machine in the report.

.NET security

Load test results and reports contain potentially sensitive information that might be used to build an attack against your computer or your network. Load test results and reports contain computer names and connection strings. You should be aware of this when you share load test reports with other people.

See also

- [Report load tests results for test comparisons or trend analysis](#)

How to: Manually create a load test performance report using Microsoft Word

1/16/2020 • 2 minutes to read • [Edit Online](#)

You can manually create Microsoft Word load test reports by copying and pasting data from the Load Test Results summary view and graphs view. The data that is presented in the summary view and graphs view is applied in HTML format when it is copied.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

TIP

You can copy plain text from the tables view and screenshots from the details view to Microsoft Word, but it is not applied in HTML format and will require additional formatting and editing.

TIP

You can also generate organized Microsoft Excel reports automatically. For more information, see [How to: Create load test performance reports using Using Microsoft Excel](#).

Copy summary view data

1. In the **Load Test Results**, if the summary view is not currently displayed, click **Summary** in the toolbar.
2. In the summary view, right-click and select **Select All**.
3. In the summary view, right-click and select **Copy**. This renders the summary view data as HTML format to the clipboard.
4. In Microsoft Word, paste the summary view data in the desired location.
5. You can now modify, format, and delete aspects of the copied content to meet your reporting needs.

Copy graph view data

1. In the **Load Test Results**, if the graphs view is not currently displayed, choose **Graphs** in the toolbar.
2. (Optional) Zoom in on the specific graph that you want to copy to your Microsoft Word document, as shown in the following illustration. For more information, see [How to: Zoom in on a region of the graph](#).



3. On the graph that you want to copy to your Microsoft Word document, right-click and select **Copy**.
4. In Microsoft Word, paste the graph and associated table data in the desired location.

WARNING

You cannot copy the graph from a remote desktop and paste it to another machine, because only the table information that is associated with the graph will be copied and not the graph image. The graph image is stored in the temporary directory on the machine from which it was copied, and the second machine cannot dereference that directory.

See also

- [Report load tests results for test comparisons or trend analysis](#)
- [How to: Create load test performance reports using Microsoft Excel](#)

Manage load test results in the Load Test Results Repository

1/1/2020 • 3 minutes to read • [Edit Online](#)

When you run your load tests, any information gathered during a load test run may be stored in the *Load Test Results Repository*, which is a SQL database. The Load Test Results Repository contains performance counter data and any information about recorded errors. The Results Repository database is created by setup for controllers, or created automatically on the first local run of a load test. For a local run, the database will be created automatically if the load test schema is not present.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

If you modify the controller's results repository connection string to use a different server, the new server must have the *loadtestresultsrepository.sql* script run to create the schema.

Visual Studio Enterprise provides named counter sets which collect common performance counters based on a technology. These sets are useful when you are analyzing an IIS server, an ASP.NET server, or a SQL server. All of the data collected with counter sets is stored in the Load Test Results Repository.

IMPORTANT

There is a difference between a counter set and the performance counter data. A counter set is metadata. It defines a group of performance counters that should be collected from a computer that is performing a particular role such as IIS or SQL Server. The counter set is part of the load test definition. Performance counter data is collected based on the counter sets, the mapping of the counter set to a specific computer, and the sample rate.

SQL Server versions

To use load tests, you can use SQL Server Express LocalDB, which is installed with Visual Studio. It is the default database server for load tests (including Microsoft Excel integration). SQL Server Express LocalDB is an execution mode of SQL Server Express that is targeted to program developers. SQL Server Express LocalDB installation copies a minimal set of files necessary to start the SQL Server Database Engine.

If your team expects heavy database needs, or your projects outgrow SQL Server Express LocalDB, you should consider upgrading to either SQL Express or full SQL Server to provide further scaling potential. If you upgrade SQL Server, the MDF and LDF files for the SQL Server Express LocalDB are stored in the user profile folder. These files can be used to import the load test database to SQL Server Express or SQL Server.

Load test results store considerations

When Visual Studio Enterprise is installed, the load test results store is set up to use an instance of SQL Express that is installed on the computer. SQL Express is limited to using a maximum of 4 GB of disk space. If you will run many load tests over a long period of time, you should consider configuring the load test results store to use an instance of the full SQL Server product if available.

Load Test Analyzer tasks

TASKS	ASSOCIATED TOPICS
<p>Set up a load test results repository: You can set up a load test results repository on a SQL database. Note: A load test repository can also be created when you install a test controller. For more information, see Install and configure test agents.</p>	
<p>Selecting and viewing a results repository: You can select a specific results repository. You are not limited to a local results store. Frequently, load tests are run on a remote set of Agent computers. Test results from your agents or your local computer can be saved to any SQL server on which you have created a load test results store. In either case, you must identify where to store your load test results by using the Administer Test Controllers window.</p>	<ul style="list-style-type: none">- How to: Select a load test results repository- How to: Access load test results for analysis
<p>Deleting a load test result from the repository: You can remove a load test result from the Load Test Editor by using the Open and Manage Load Test Results dialog box.</p>	<ul style="list-style-type: none">- How to: Delete load test results from a repository
<p>Import and export results into a repository: You can import and export load test results from the Load Test Editor.</p>	<ul style="list-style-type: none">- How to: Import load test results into a repository- How to: Export load test results from a repository

Related tasks

Analyze load test results

You can view the results of both a running load test and a completed load test by using the **Load Test Analyzer**.

See also

- [Analyze load test results](#)
- [How to: Access load test results for analysis](#)

How to: Select a load test results repository

1/1/2020 • 2 minutes to read • [Edit Online](#)

You are not limited to a local results store. Frequently, load tests are run on a remote set of Agent computers. Agents, together with a controller, can generate more simulated load than any single computer. For more information, see [Test controllers and test agents](#).

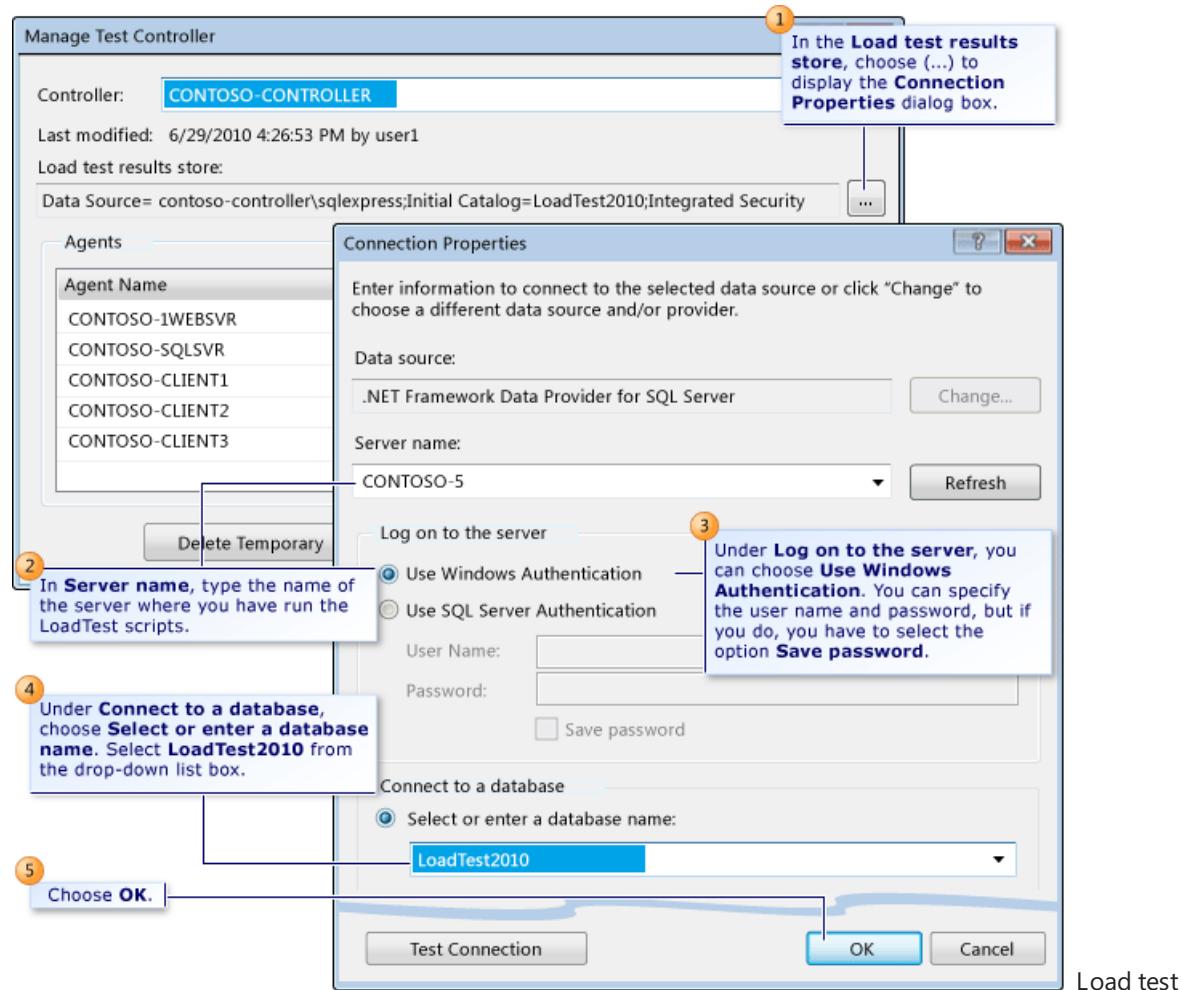
Test results from your agents or your local computer can be saved to any SQL server on which you have created a load test results store. In either case, you must identify where you want to store your load test results by using the **Administer Test Controllers** window.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Identify a results store for load test data

1. In **Solution Explorer**, open your load test file.
2. From the **Load Test** toolbar, choose **Manage Test Controllers**. The **Manage Test Controller** dialog box is displayed. If you are using an agent remotely, you must select a controller.



results store connection properties

3. In the **Load test results store**, click (...) to display the **Connection Properties** dialog box.

4. In **Server Name**, type the name of the server where you have run the `LoadTest` scripts.

TIP

If you are using SQL Express on your local machine for the load test store, enter <computername>\sqlexpress (for example, **MyComputer\sqlexpress**).

5. Under **Log on to the server**, you can choose **Use Windows Authentication**. You can specify the username and password, but if you do, you have to select the option **Save my password**.
6. Under **Connect to a database**, choose **Select or enter a database name**. Select **LoadTest** from the drop-down list box.
7. Choose **OK**. You can test the connection by choosing **Test Connection**.
8. Choose **Close** in the **Manage Test Controller** dialog box.

See also

- [Manage load test results in the Load Test Results Repository](#)
- [Test controllers and test agents](#)

How to: Import load test results into a repository

1/1/2020 • 2 minutes to read • [Edit Online](#)

When you run a load test, information gathered during the run is stored in the Load Test Results Repository. The Load Test Results Repository contains performance counter data and information about any errors. For more information, see [Manage load test results in the Load Test Results Repository](#).

You can manage load test results from the Load Test Editor by using the **Open and Manage Load Test Results** dialog box. You can open, import, export, and remove load test results.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To import results into a repository

1. From a web performance and load test project, open a load test.
2. On the embedded toolbar, choose **Open and Manage Results**.

The **Open and Manage Load Test Results** dialog box is displayed.

3. In **Enter a controller name to find load test results**, select a controller. Select **<local>** to access results stored locally.

If there are load test results available, they appear in the **Load test results** list. The columns are **Time**, **Duration**, **User**, **Outcome**, **Test**, and **Description**. **Test** contains the name of the test, and **Description** contains the optional description that is added before the test is run.

4. Choose **Import**.

The **Import Load Test Results** dialog box appears.

5. In the **File name** box, type the name of an archived test results file, and then choose **Open**.

- or -

Browse to the file, and then choose **Open**.

NOTE

An archived test results file that you specify in this step must have been created by performing the Export operation.

The results are imported and appear in the **Load test results** list.

See also

- [Manage load test results in the Load Test Results Repository](#)
- [Analyze load test results](#)
- [How to: Export load test results from a repository](#)

How to: Export load test results from a repository

1/1/2020 • 2 minutes to read • [Edit Online](#)

When you run a load test, information gathered during the run is stored in the Load Test Results Repository. The Load Test Results Repository contains performance counter data and information about any errors. For more information, see [Manage load test results in the Load Test Results Repository](#).

You can manage load test results from the Load Test Editor by using the **Open and Manage Load Test Results** dialog box. You can open, import, export, and remove load test results.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To export results from a repository

1. From a web performance and load test project, open a load test.
2. On the embedded toolbar, choose **Open and Manage Results**.

The **Open and Manage Load Test Results** dialog box is displayed.

3. In **Enter a controller name to find load test results**, select a controller. Select **<Local - No controller>** to access results stored locally.
4. In **Show results for the following load test**, select the load test whose results you want to view. Select **<Show results for all tests>** to see all results for all tests.

If load test results are available, they appear in the **Load test results** list. The columns are **Time**, **Duration**, **User**, **Outcome**, **Test**, and **Description**. **Test** contains the name of the test, and **Description** contains the optional description that is added before the test is run. The **Description** column displays the short descriptions that were entered in the **Analysis Comments** for this test result.

5. In the **Load test results** list, choose a result. You can use the **Shift** key, the **Ctrl** key, or both to select more than one result, and export them to a single file.
6. Choose **Export**.

The **Export Load Test Results** dialog box appears.

7. In the **File name** box, type a name, and then choose **Save**.

The results are exported to an archive file.

NOTE

The **Open and Manage Load Test Results** dialog box remains open after the results appear.

See also

- [Manage load test results in the Load Test Results Repository](#)
- [How to: Delete load test results from a repository](#)

- [Analyze load test results](#)
- [How to: Import load test results into a repository](#)

How to: Delete load test results from a repository

1/1/2020 • 2 minutes to read • [Edit Online](#)

When you run a load test, information that was gathered during the run is stored in the Load Test Results Repository. The Load Test Results Repository contains performance counter data and information about any errors. For more information, see [Manage load test results in the Load Test Results Repository](#).

You can manage load test results from the Load Test Editor by using the **Open and Manage Load Test Results** dialog box. You can open, import, export, and remove load test results.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

To delete results from a repository

1. From a web performance and load test project, open a load test.
2. On the embedded toolbar, choose **Open and Manage Results**.

The **Open and Manage Load Test Results** dialog box is displayed.

3. In **Enter a controller name to find load test results**, select a controller. Select **<Local - No controller>** to access results that are stored locally.
4. In **Show results for the following load test**, select the load test whose results you want to view. Select **<Show results for all tests>** to see all results for all tests.

If load test results are available, they appear in the **Load test results** list. The columns are **Time**, **Duration**, **User**, **Outcome**, **Test**, and **Description**. **Test** contains the name of the test, and **Description** contains the optional description that is added before the test is run. The **Description** column displays the short descriptions that were entered in the **Analysis Comments** for this test result.

5. In the **Load test results** list, choose a result. You can use the **Shift** key, the **Ctrl** key, or both to select more than one result.
6. Choose **Remove**.

The results are removed from the repository.

NOTE

The **Open and Manage Load Test Results** dialog box remains open after the results are removed.

See also

- [How to: Export load test results from a repository](#)
- [Manage load test results in the Load Test Results Repository](#)
- [Analyze load test results](#)
- [How to: Import load test results into a repository](#)

How to: Create an add-in for the Web Performance Test Results Viewer

1/1/2020 • 11 minutes to read • [Edit Online](#)

You can extend the UI for the **Web Performance Test Results Viewer** by using the following namespaces:

- [Microsoft.VisualStudio.TestTools.LoadTesting](#)
- [Microsoft.VisualStudio.TestTools.WebTesting](#)

Additionally, you need to add a reference to LoadTestPackage DLL, which is located in the `%ProgramFiles(x86)%\Microsoft Visual Studio\<version>\Enterprise\Common7\IDE\PrivateAssemblies` folder.

To extend the **Web Performance Test Results Viewer**'s UI, you must create a Visual Studio add-in and a user control. The following procedures explain how to create the add-in, the user control, and how to implement the classes necessary to extend the **Web Performance Test Results Viewer**'s UI.

NOTE

Web performance and load test functionality is deprecated. Visual Studio 2019 is the last version where web performance and load testing will be available. For more information, see the [Cloud-based load testing service end of life](#) blog post.

Create or open a solution that contains an ASP.NET web application and a web performance and load test project

To prepare for extending the Web Performance Test Results Viewer

Either create or open a non-production solution that you can experiment with which contains an ASP.NET web application and a web performance and load test project with one or more web performance tests for the ASP.NET web application.

NOTE

You can create an ASP.NET web application and web performance and load test project that contains web performance tests by following the procedures in [How to: Create a web service test](#) and [Generate and run a coded web performance test](#).

Create a Visual Studio add-in

An add-in is a compiled DLL that runs in the Visual Studio integrated development environment (IDE). Compilation helps protect your intellectual property and improves performance. Although you can create add-ins manually, you may find it easier to use the **Add-In Wizard**. This wizard creates a functional but basic add-in that you can run immediately after you create it. After the **Add-In Wizard** generates the basic program, you can add code to it and customize it.

The **Add-In Wizard** lets you supply a display name and description for your add-in. Both will appear in **Add-In Manager**. Optionally, you can have the wizard generate code that adds to the **Tools** menu a command to open the add-in. You can also choose to display a custom **About** dialog box for your add-in. When the wizard is finished, you have a new project that has just one class that implements the add-in. That class is named Connect.

You will use the **Add-In Manager** at the end of this article.

To create an add-in by using the Add-In Wizard

1. In **Solution Explorer**, right-click the solution, choose **Add**, and then select **New Project**.

2. Create a new **Visual Studio Add-in** project.

The Visual Studio **Add-In Wizard** starts.

3. Choose **Next**.

4. On the **Select a Programming Language** page, select the programming language that you want to use to write the add-in.

NOTE

This topic uses Visual C# for the sample code.

5. On the **Select An Application Host** page, select **Visual Studio** and clear **Visual Studio Macros**.

6. Choose **Next**.

7. Type a name and description for your add-in on the **Enter a Name and Description** page.

After the add-in is created, its name and description are displayed in the **Available Add-Ins** list in **Add-In Manager**. Add enough detail to the description of your add-in so that users can learn what your add-in does, how it works, and so on.

8. Choose **Next**.

9. On the **Choose Add-In Options** page, select **I would like my Add-in to load when the host application starts**.

10. Clear the remaining check boxes.

11. On the **Choosing 'Help About' Information** page, you can specify whether you want information about your add-in to be displayed in an **About** dialog box. If you do want the information to be displayed, select the **Yes, I would like my Add-in to offer 'About' box information** check box.

Information that can be added to the Visual Studio **About** dialog box includes version number, support details, licensing data, and so forth.

12. Choose **Next**.

13. The options that you selected are displayed on the **Summary** page for you to review. If you are satisfied, choose **Finish** to create the add-in. If you want to change something, choose the **Back** button.

The new solution and project are created and the *Connect.cs* file for the new add-in is displayed in the **Code Editor**.

You will add code to the *Connect.cs* file after the following procedure, which creates a user control that will be referenced by this *WebPerfTestResultsViewerAddin* project.

After an add-in is created, you must register it with Visual Studio before it can be activated in **Add-In Manager**. You do this by using an XML file that has an *.addin* file name extension.

The *.addin* file describes the information that Visual Studio requires to display the add-in in **Add-In Manager**. When Visual Studio starts, it looks in the *.addin* file location for any available *.addin* files. If it finds any, it reads the XML file and gives **Add-In Manager** the information that it requires to start the add-in when it is clicked.

The *.addin* file is created automatically when you create an add-in by using the **Add-In Wizard**.

Add-in file locations

Two copies of the `.addin` files are automatically created by the **Add-In Wizard**, as follows:

.ADDIN FILE LOCATION	DESCRIPTION
Root project folder	Used for deployment of the add-in project. Included in the project for ease of editing and has the local path for XCopy-style deployment.
Add-in folder	Used for running the add-in in the debugging environment. Should always point to the output path of the current build configuration.

Create a Windows Form Control Library project

The Visual Studio add-in created in the previous procedure references a Windows Forms Control Library project to create an instance of a [UserControl](#) class.

To create a control to be used in the Web Test Results Viewer

1. In **Solution Explorer**, right-click the solution, choose **Add**, and then select **New Project**.
2. Create a new **Windows Forms Control Library** project.
3. From the **Toolbox**, drag a [DataGridView](#) onto the surface of `userControl1`.
4. Click the action tag glyph (□) on the upper-right corner of the [DataGridView](#) and follow these steps:
 - a. Choose **Dock in Parent Container**.
 - b. Clear the check boxes for **Enable Adding**, **Enable Editing**, **Enable Deleting** and **Enable Column Reordering**.
 - c. Choose **Add Column**.
The **Add Column** dialog box is displayed.
 - d. In the **Type** drop-down list, select **DataGridViewTextBoxColumn**.
 - e. Clear the text "Column1" in **Header text**.
 - f. Choose **Add**.
 - g. Choose **Close**.
5. In the **Properties** window, change the **(Name)** property of the [DataGridView](#) to **resultControlDataGridView**.
6. Right-click the design surface and select **View Code**.
The `UserControl1.cs` file is displayed in the **Code Editor**.
7. Change the name of the instantiated [UserControl](#) class from `UserContro1` to `resultControl`:

```

namespace WebPerfTestResultsViewerControl
{
    public partial class resultControl : UserControl
    {
        public resultControl()
        {
            InitializeComponent();
        }
    }
}

```

In the next procedure, you will add code to the WebPerfTestResultsViewerAddin project's *Connect.cs* file, which will reference the *resultControl* class.

You will be adding some additional code to the *Connect.cs* file later.

Add code to the WebPerfTestResultsViewerAddin

1. In **Solution Explorer**, right-click the **References** node in the WebPerfTestResultsViewerAddin project and select **Add Reference**.
2. In the **Add Reference** dialog box, choose the **.NET** tab.
3. Scroll down and select **Microsoft.VisualStudio.QualityTools.WebTestFramework** and **System.Windows.Forms**.
4. Choose **OK**.
5. Right-click the **References** node again, and select **Add Reference**.
6. In the **Add Reference** dialog box, choose the **Browse** tab.
7. Choose the drop-down for **Look in** and navigate to %ProgramFiles(x86)%\Microsoft Visual Studio\2017\Enterprise\Common7\IDE\PrivateAssemblies and select the *Microsoft.VisualStudio.QualityTools.LoadTestPackage.dll* file.
8. Choose **OK**.
9. Right-click the WebPerfTestResultsViewerAddin project node, and select **Add Reference**.
10. In the **Add Reference** dialog box, choose the **Projects** tab.
11. Under **Project Name**, select the **WebPerfTestResultsViewerControl** project and choose **OK**.
12. If the *Connect.cs* file is not still open, in **Solution Explorer**, right-click the **Connect.cs** file in the WebPerfTestResultsViewerAddin project and select **View Code**.
13. In the *Connect.cs* file, add the following Using statements:

```

using System.IO;
using System.Windows.Forms;
using System.Collections.Generic;
using Microsoft.VisualStudio.TestTools.LoadTesting;
using Microsoft.VisualStudio.TestTools.WebTesting;
using WebPerfTestResultsViewerControl;

```

14. Scroll down to the bottom of the *Connect.cs* file. You need to add a list of GUIDs for the **UserControl** in case more than one instance of the **Web Performance Test Results Viewer** is open. You will add code later that uses this list.

A second List of string is used in the *OnDisconnection* method, which you will code later.

```

private DTE2 _applicationObject;
private AddIn _addInInstance;

private Dictionary<Guid, List<UserControl>> m_controls = new Dictionary<Guid, List<UserControl>>();
private List<string> temporaryFilePaths = new List<string>();

```

15. The *Connect.cs* file instantiates a class named *Connect* from the [IDTExtensibility2](#) class and also includes some methods for implementing the Visual Studio add-in. One of the methods is the *OnConnection* method, which receives notification that the add-in is being loaded. In the *OnConnection* method, you will use the *LoadTestPackageExt* class to create your extensibility package for the **Web Performance Test Results Viewer**. Add the following code to the *OnConnection* method:

```

public void OnConnection(object application, ext_ConnectMode connectMode, object addInInst, ref Array
custom)
{
    _applicationObject = (DTE2)application;
    _addInInstance = (AddIn)addInInst;

    // Create a load test packge extensibility class.          LoadTestPackageExt
loadTestPackageExt =
_applicationObject.GetObject("Microsoft.VisualStudio.TestTools.LoadTesting.LoadTestPackageExt") as
LoadTestPackageExt;           // Process open windows.      foreach (WebTestResultViewer
webTestResultViewer in loadTestPackageExt.WebTestResultViewerExt.ResultWindows)           {
WindowCreated(webTestResultViewer);           }           // Create event handlers.
loadTestPackageExt.WebTestResultViewerExt.WindowCreated += new
EventHandler<WebTestResultViewerExt.WindowEventArgs>(WebTestResultViewerExt_WindowCreated);
loadTestPackageExt.WebTestResultViewerExt.WindowClosed += new
EventHandler<WebTestResultViewerExt.WindowClosedEventArgs>(WebTestResultViewerExt_WindowClosed);
loadTestPackageExt.WebTestResultViewerExt.SelectionChanged += new
EventHandler<WebTestResultViewerExt.SelectionChangedEventArgs>(WebTestResultViewerExt_SelectionChanged);
}

```

16. Add the following code to the *connect* class to create the *WebTestResultViewerExt.WindowCreated* method for the *loadTestPackageExt.WebTestResultViewerExt.WindowCreated* event handler you added in the *OnConnection* method and for the *WindowCreated* method that the *WebTestResultViewerExt.WindowCreated* method calls.

```

void WebTestResultViewerExt_WindowCreated(object sender,
WebTestResultViewerExt.WindowEventArgs e)
{
    // New control added to new result viewer window.
    WindowCreated(e.WebTestResultViewer);
}

private void WindowCreated(WebTestResultViewer viewer)           {           // Instantiate an instance
of the resultControl referenced in the WebPerfTestResultsViewerControl project.
    resultControl resultControl = new resultControl();           // Add to the dictionary of
open playback windows.
System.Diagnostics.Debug.Assert(!m_controls.ContainsKey(viewer.TestResultId));
List<UserControl> userControls = new List<UserControl>();           userControls.Add(resultControl);
// Add Guid to the m_control List to manage Result viewers and controls.
m_controls.Add(viewer.TestResultId, userControls);           // Add tabs to the playback control.
resultControl.Dock = DockStyle.Fill;           viewer.AddResultPage(new Guid(), "Sample",
resultControl);           }

```

17. Add the following code to the *connect* class to create the *WebTestResultViewer.SelectedChanged* method for the *loadTestPackageExt.WebTestResultViewerExt.SelectionChanged* event handler you added in the *OnConnection* method:

```

void WebTestResultViewer_SelectedChanged(object sender,
WebTestResultViewerExt.SelectionChangedEventArgs e)
{
    foreach (UserControl userControl in m_controls[e.TestResultId])           {           //
Update the userControl in each result viewer.           resultControl = userControl
as resultControl;           if (resultControl != null)           // Call the
resultControl's Update method (This will be added in the next procedure).
resultControl.Update(e.WebTestRequestResult);           }
}

```

18. Add the following code to the connect class to create the WebTesResultViewer_WindowClosed method for the event handler for the loadTestPackageExt.WebTestResultViewerExt.WindowClosed you added in the OnConnection method:

```

void WebTesResultViewer_WindowClosed(object sender, WebTestResultViewerExt.WindowClosedEventArgs e)
{
    if (m_controls.ContainsKey(e.WebTestResultViewer.TestResultId))
    {
        m_controls.Remove(e.WebTestResultViewer.TestResultId);
    }
}

```

Now that the code has been completed for the Visual Studio add-in, you need to add the Update method to the resultControl in the WebPerfTestResultsViewerControl project.

Add Code to the WebPerfTestResultsViewerControl

1. In **Solution Explorer**, right-click the WebPerfTestResultsViewerControl project node and select **Properties**.
2. Select the **Application** tab and then choose the **Target framework** drop-down list and select **.NET Framework 4** (or later). Close the **Properties** window.
This is required in order to support the DLL references that are needed for extending the **Web Performance Test Results Viewer**.
3. In **Solution Explorer**, in the WebPerfTestResultsViewerControl project, right-click the **References** node and select **Add Reference**.
4. In the **Add Reference** dialog box, click the **.NET** tab.
5. Scroll down and select **Microsoft.VisualStudio.QualityTools.WebTestFramework**.
6. Choose **OK**.
7. In the *UserControl1.cs* file, add the following Using statements:

```

using Microsoft.VisualStudio.TestTools.WebTesting;
using Microsoft.VisualStudio.TestTools.WebTesting.Rules;

```

8. Add the Update method that is called and passed a WebTestRequestResult from the WebPerfTestResultsViewerAddin WebTestResultViewer_SelectedChanged method in the *Connect.cs* file. The Update method populates the DataGridView with various properties passed to it in the WebTestRequestResult.

```

public void Update(WebTestRequestResult WebTestResults)
{
    // Clear the DataGridView when a request is selected.
    resultControlDataGridView.Rows.Clear();
    // Populate the DataGridControl with properties from the WebTestResults.
    this.resultControlDataGridView.Rows.Add("Request: " +
WebTestResults.Request.Url.ToString());
    this.resultControlDataGridView.Rows.Add("Response: " +
WebTestResults.Response.ResponseUri.ToString());
    foreach (RuleResult ruleResult in WebTestResults.ExtractionRuleResults)
    {
        this.resultControlDataGridView.Rows.Add("Extraction rule results: " +
ruleResult.Message.ToString());
    }
    foreach (RuleResult ruleResult in WebTestResults.ValidationRuleResults)
    {
        this.resultControlDataGridView.Rows.Add("Validation rule results: " +
ruleResult.Message.ToString());
    }
    foreach (WebTestError webTestError in WebTestResults.Errors)
    {
        this.resultControlDataGridView.Rows.Add("Error: " + webTestError.ErrorType.ToString() +
" " + webTestError.ErrorSubtype.ToString() + " " + webTestError.ExceptionText.ToString());
    }
}

```

Build the solution

- On the **Build** menu, select **Build Solution**.

Register the WebPerfTestResultsViewerAddin add-in

1. On the **Tools** menu, select **Add-in Manager**.
2. The **Add-in Manager** dialog box is displayed.
3. Select the check box for the WebPerfTestResultsViewerAddin add-in in the **Available Add-ins** column and clear the check boxes underneath the **Startup** and **Command Line** columns.
4. Choose **OK**.

Run the web performance test using the Web Test Results Viewer

1. Run your web performance test and you will see the WebPerfTestResultsViewerAddin add-in's new tab titled Sample displayed in the **Web Performance Test Results Viewer**.
2. Choose the tab to see the properties presented in the DataGridView.

.NET security

To improve security by preventing malicious add-ins from automatically activating, Visual Studio provides settings in a **Tools Options** page named **Add-in/Macros Security**.

In addition, this options page allows you to specify the folders in which Visual Studio searches for *.AddIn* registration files. This improves security by allowing you to limit the locations where *.AddIn* registration files can be read. This helps prevent malicious *.AddIn* files from unintentionally being used.

Add-In Security Settings

The settings in the options page for add-in security are as follows:

- **Allow Add-in components to load.** Selected by default. When selected, add-ins are allowed to load in Visual Studio. When not selected, add-ins are prohibited from loading in Visual Studio.
- **Allow Add-in components to load from a URL.** Not selected by default. When selected, add-ins can be loaded from external websites. When not selected, remote add-ins are prohibited from loading in Visual Studio. If an add-in cannot load for some reason, then it cannot be loaded from the Web. This setting controls only the loading the add-in DLL. The *.Addin* registration files must always be located on the local system.

See also

- [UserControl](#)
- [Microsoft.VisualStudio.TestTools.LoadTesting](#)
- [Microsoft.VisualStudio.TestTools.WebTesting](#)
- [Microsoft.VisualStudio.TestTools.WebTesting.Rules](#)
- [UserControl](#)
- [DataGrid](#)
- [Create custom code and plug-ins for load tests](#)

Use Coded UI test to test your code

1/1/2020 • 19 minutes to read • [Edit Online](#)

Coded UI tests (CUITs) drive your application through its user interface (UI). These tests include functional testing of the UI controls. They let you verify that the whole application, including its user interface, is functioning correctly. Coded UI tests are useful where there is validation or other logic in the user interface, for example in a web page. They are also frequently used to automate an existing manual test.

Creating a Coded UI test in Visual Studio is easy. You simply perform the test manually while **Coded UI Test Builder** runs in the background. You can also specify what values should appear in specific fields. **Coded UI Test Builder** records your actions and generates code from them. After the test is created, you can edit it in a specialized editor that lets you modify the sequence of actions.

The specialized **Coded UI Test Builder** and editor make it easy to create and edit Coded UI tests, even if your main skills are concentrated in testing rather than coding. But if you are a developer and you want to extend the test in a more advanced way, the code is structured so that it is straightforward to copy and adapt. For example, you might record a test to buy something at a website, and then edit the generated code to add a loop that buys many items.

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Requirements

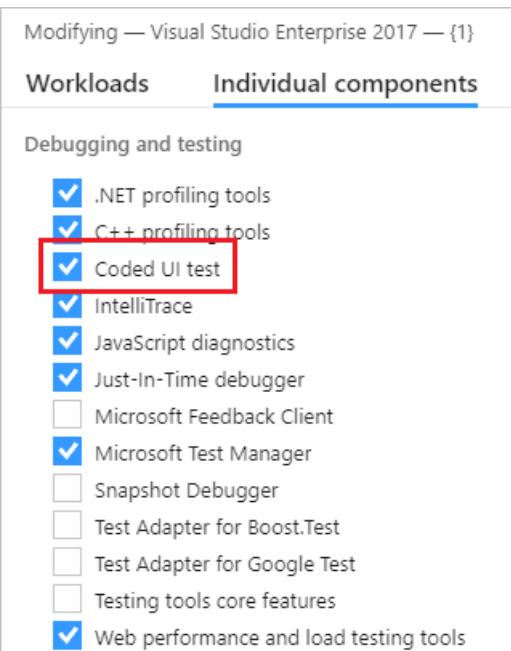
- Visual Studio Enterprise
- Coded UI test component

For more information about which platforms and configurations are supported by Coded UI tests, see [Supported platforms](#).

Install the Coded UI test component

To access the Coded UI test tools and templates, install the **Coded UI test** component of Visual Studio.

1. Launch **Visual Studio Installer** by choosing **Tools > Get Tools and Features**.
2. In **Visual Studio Installer**, choose the **Individual components** tab, and then scroll down to the **Debugging and testing** section. Select the **Coded UI test** component.

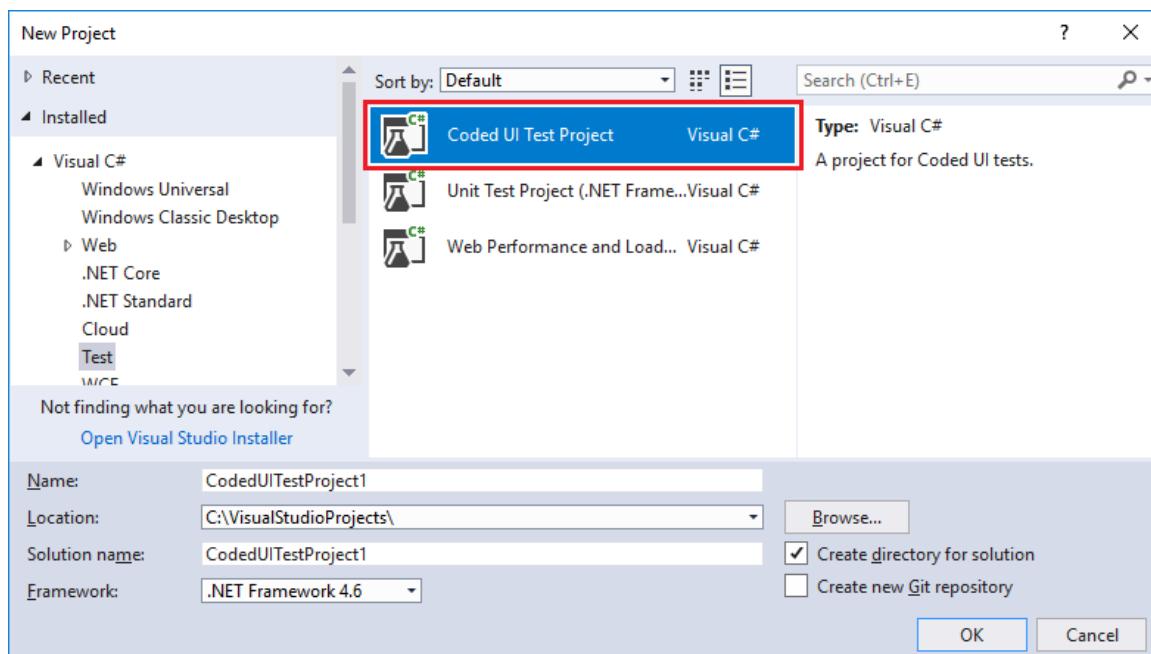


3. Select **Modify**.

Create a Coded UI test

1. Create a Coded UI test project.

Coded UI tests must be contained in a Coded UI test project. If you don't already have a Coded UI test project, create one. Choose **File > New > Project**. Search for and select the **Coded UI Test Project** project template.



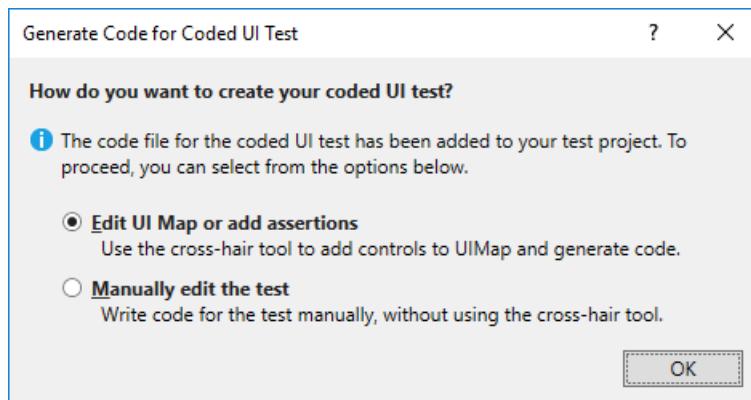
NOTE

If you don't see the **Coded UI Test Project** template, you need to [install the Coded UI test component](#).

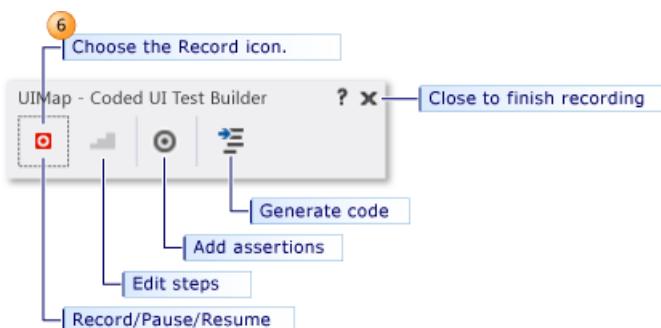
2. Add a Coded UI test file.

If you just created a Coded UI project, the first CUIT file is added automatically. To add another test file, open the shortcut menu on the Coded UI test project in **Solution Explorer**, and then choose **Add > Coded UI Test**.

In the **Generate Code for Coded UI Test** dialog box, choose **Record actions > Edit UI map or add assertions**.



The **Coded UI Test Builder** appears.



3. Record a sequence of actions.

To start recording, choose the **Record** icon. Perform the actions that you want to test in your application, including starting the application if that is required. For example, if you are testing a web application, you might start a browser, navigate to the website, and sign in to the application.

To pause recording, for example if you have to deal with incoming mail, choose **Pause**.

WARNING

All actions performed on the desktop will be recorded. Pause the recording if you are performing actions that may lead to sensitive data being included in the recording.

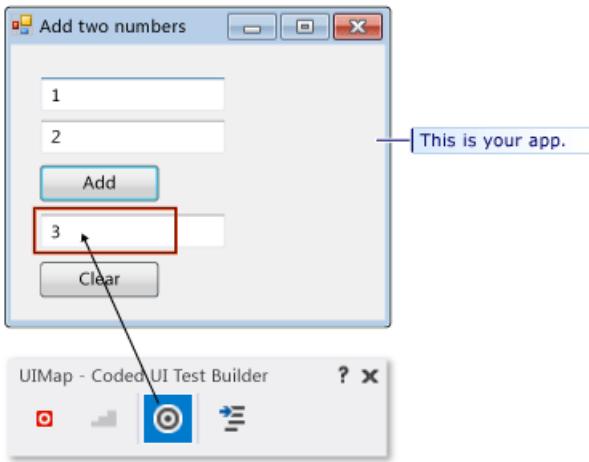
To delete actions that you recorded by mistake, choose **Edit Steps**.

To generate code that will replicate your actions, choose the **Generate Code** icon and type a name and description for your Coded UI test method.

4. Verify the values in UI fields such as text boxes.

Choose **Add Assertions** in the **Coded UI Test Builder**, and then choose a UI control in your running application. In the list of properties that appears, select a property, for example, **Text** in a text box. On the shortcut menu, choose **Add Assertion**. In the dialog box, select the comparison operator, the comparison value, and the error message.

Close the assertion window and choose **Generate Code**.



TIP

Alternate between recording actions and verifying values. Generate code at the end of each sequence of actions or verifications. If you want, you will be able to insert new actions and verifications later.

For more details, see [Validate properties of controls](#).

5. View the generated test code.

To view the generated code, close the UI Test Builder window. In the code, you can see the names that you gave to each step. The code is in the CUIT file that you created:

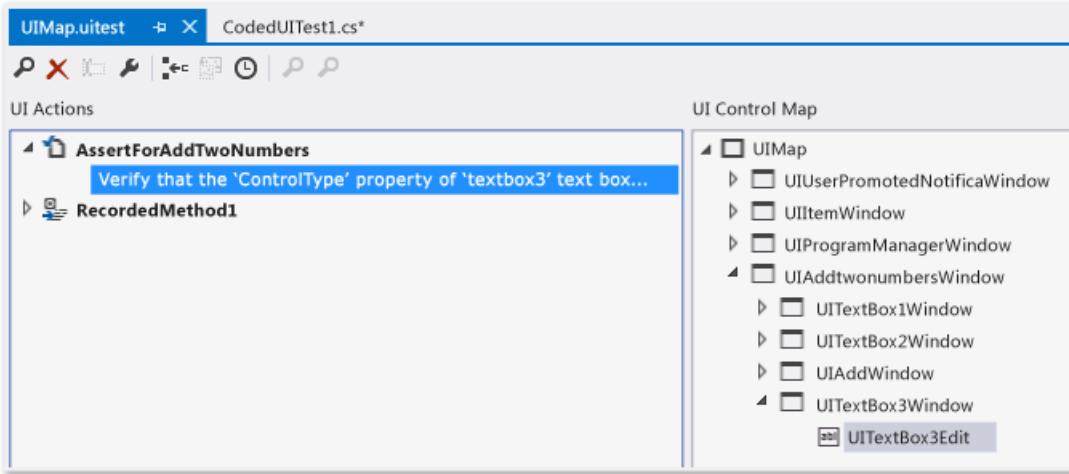
```
[CodedUITest]
public class CodedUITest1
{
    ...
    [TestMethod]
    public void CodedUITestMethod1()
    {
        this.UIMap.AddTwoNumbers();
        this.UIMap.VerifyResultValue();
        // To generate more code for this test, select
        // "Generate Code" from the shortcut menu.
    }
}
```

6. Add more actions and assertions.

Place the cursor at the appropriate point in the test method and then, on the shortcut menu, choose **Generate Code for Coded UI Test**. New code will be inserted at that point.

7. Edit the detail of the test actions and the assertions.

Open *UIMap.uitest*. This file opens in the **Coded UI Test Editor**, where you can edit any sequence of actions that you recorded as well as edit your assertions.



For more information, see [Edit Coded UI tests using the Coded UI Test editor](#).

8. Run the test.

Use Test Explorer, or open the shortcut menu in the test method, and then choose **Run Tests**. For more information about how to run tests, see [Run unit tests with Test Explorer](#) and [Additional options for running Coded UI tests](#) in the [What's next?](#) section at the end of this topic.

The remaining sections in this topic provide more detail about the steps in this procedure.

For a more detailed example, see [Walkthrough: Creating, editing, and maintaining a Coded UI test](#). In the walkthrough, you will create a simple Windows Presentation Foundation (WPF) application to demonstrate how to create, edit, and maintain a Coded UI test. The walkthrough provides solutions for correcting tests that have been broken by various timing issues and control refactoring.

Start and stop the application under test

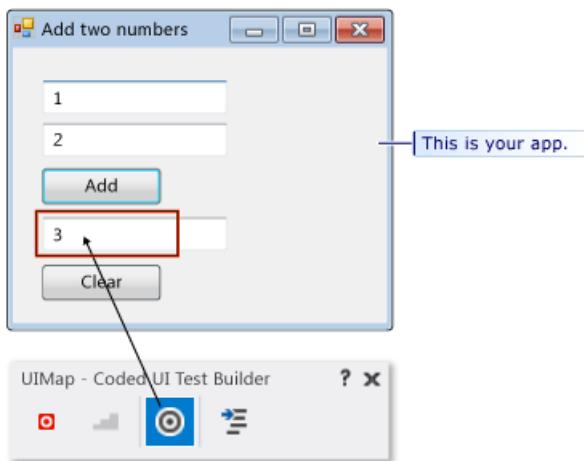
If you don't want to start and stop the application, browser, or database separately for each test, do one of the following:

- If you do not want to record the actions to start your application under test, you must start your application before you choose the **Record** icon.
- At the end of a test, the process in which the test runs is terminated. If you started your application in the test, the application usually closes. If you do not want the test to close your application when it exits, add a `.runsettings` file to your solution, and use the `KeepExecutorAliveAfterLegacyRun` option. For more information, see [Configure unit tests by using a .runsettings file](#).
- Add a test initialize method, identified by a `[TestInitialize]` attribute, which runs code at the start of each test method. For example, you could start the application from the `TestInitialize` method.
- Add a test cleanup method, identified by a `[TestCleanup]` attribute, that runs code at the end of each test method. For example, the method to close the application could be called from the `TestCleanup` method.

Validate the properties of UI controls

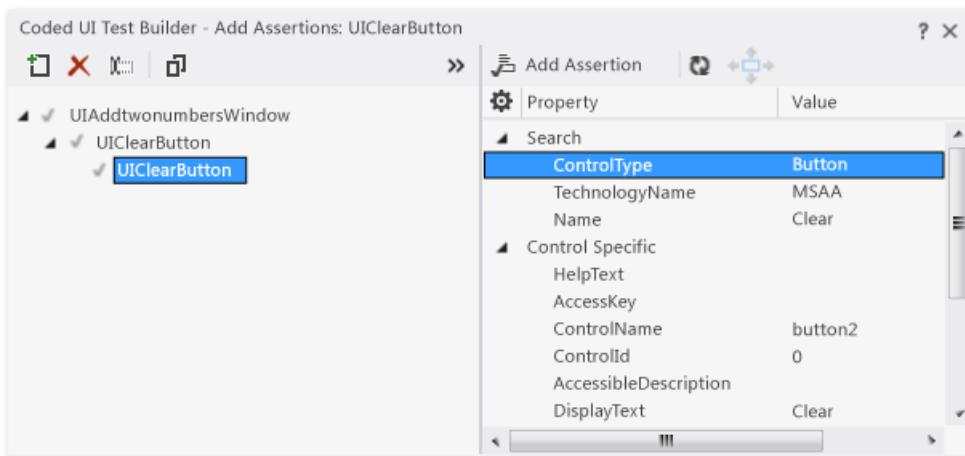
You can use the **Coded UI Test Builder** to add a user interface (UI) control to the `UIMap` for your test, or to generate code for a validation method that uses an assertion for a UI control.

To generate assertions for your UI controls, choose the **Add Assertions** tool in the **Coded UI Test Builder** and drag it to the control on the application under test that you want to verify is correct. When the box outlines your control, release the mouse. The control class code is immediately created in the `UIMap.Designer.cs` file.



The properties for this control are now listed in the **Add Assertions** dialog box.

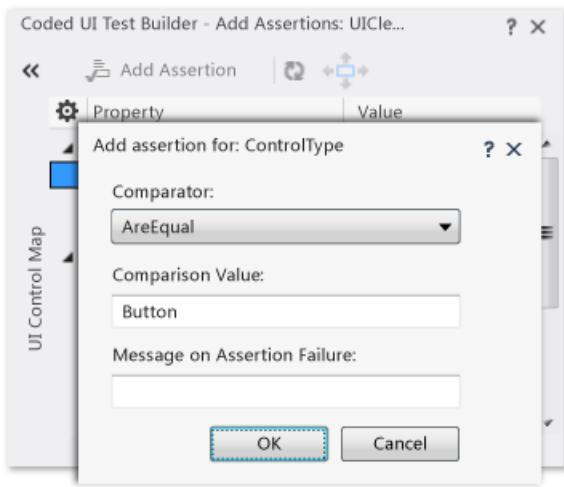
Another way of navigating to a particular control is to choose the arrow (<<) to expand the view for the **UI Control Map**. To find a parent, sibling, or child control, you can click anywhere on the map and use the arrow keys to move around the tree.



TIP

If you don't see any properties when you select a control in your application, or you don't see the control in the UI Control Map, verify that the control has a unique ID in the application code. The unique ID can be an HTML ID attribute or a WPF UId.

Next, open the shortcut menu on the property for the UI control that you want to verify, and then point to **Add Assertion**. In the **Add Assertion** dialog box, select the **Comparator** for your assertion, for example [AreEqual](#), and type the value for your assertion in **Comparison Value**.



When you have added all your assertions for your test, choose **OK**.

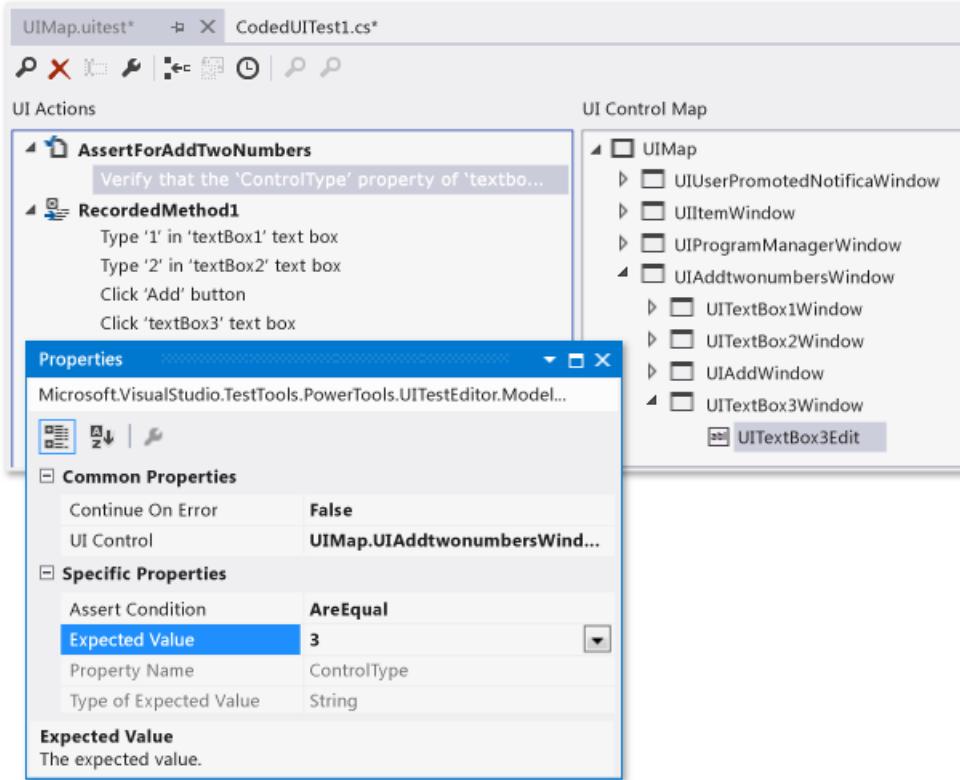
To generate the code for your assertions and add the control to the UI map, choose the **Generate Code** icon. Type a name for your Coded UI test method and a description for the method, which will be added as comments for the method. Choose **Add and Generate**. Next, choose the **Close** icon to close the **Coded UI Test Builder**. This generates code similar to the following code. For example, if the name you entered is `AssertForAddTwoNumbers`, the code will look like this example:

- Adds a call to the assert method `AssertForAddTwoNumbers` to the test method in your Coded UI test file:

```
[TestMethod]
public void CodedUITestMethod1()
{
    this.UIMap.AddTwoNumbers();
    this.UIMap.AssertForAddTwoNumbers();
}
```

You can edit this file to change the order of the steps and assertions, or to create new test methods. To add more code, place the cursor on the test method and on the shortcut menu choose **Generate Code for Coded UI Test**.

- Adds a method called `AssertForAddTwoNumbers` to your UI map (`UIMap.uitest`). This file opens in the **Coded UI Test Editor**, where you can edit the assertions.



For more information, see [Edit Coded UI tests using the Coded UI test editor](#).

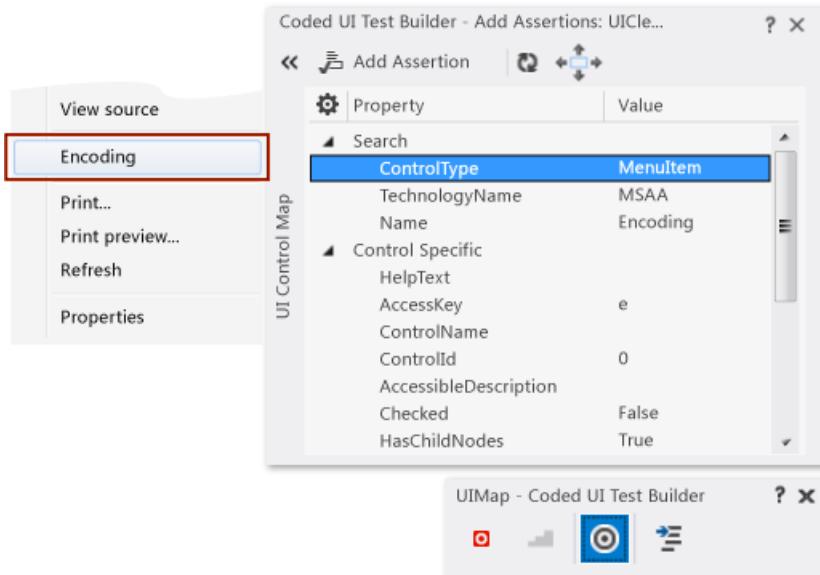
You can also view the generated code of the assertion method in *UIMap.Designer.cs*. However, you should not edit this file. If you want to make an adapted version of the code, copy the methods to another file such as *UIMap.cs*, rename the methods, and edit them there.

```
public void AssertForAddTwoNumbers()
{
    ...
}
```

Select a hidden control using the keyboard

If the control you want to select loses focus and disappears when you select the **Add Assertions** tool from the **Coded UI Test Builder**:

Sometimes, when you add controls and verify their properties, you might have to use the keyboard. For example, when you try to record a Coded UI test that uses a right-click menu control, the list of menu items in the control will lose focus and disappear when you try to select the **Add Assertions** tool from the **Coded UI Test Builder**. This is demonstrated in the following illustration, where the right-click menu in Internet Explorer loses focus and disappears if you try to select it with the **Add Assertions** tool.



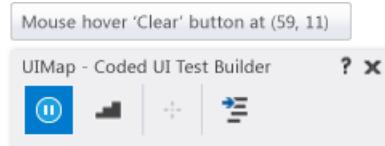
To use the keyboard to select a UI control, hover over the control with the mouse. Then hold down the **Ctrl** key and the **I** key at the same time. Release the keys. The control is recorded by the **Coded UI Test Builder**.

Manually record mouse hovers

If you can't record a mouse hover on a control:

Under some circumstances, a particular control that's being used in a Coded UI test might require you to use the keyboard to manually record mouse hover events. For example, when you test a Windows Form or a Windows Presentation Foundation (WPF) application, there might be custom code. Or, there might be special behavior defined for hovering over a control, such as a tree node expanding when a user hovers over it. To test circumstances like these, you have to manually notify the **Coded UI Test Builder** that you are hovering over the control by pressing predefined keyboard keys.

When you perform your Coded UI test, hover over the control. Then press and hold **Ctrl**, while you press and hold the **Shift** and **R** keys on your keyboard. Release the keys. A mouse hover event is recorded by the **Coded UI Test Builder**.



After you generate the test method, code similar to the following example will be added to the *UIMap.Designer.cs* file:

```
// Mouse hover '1' label at (87, 9)
Mouse.Hover(uIItem1Text, new Point(87, 9));
```

Configure mouse-hover keyboard assignments

If the key assignment for capturing mouse hover events is being used elsewhere in my environment:

If necessary, the default keyboard assignment of **Ctrl+Shift+R** that is used to apply mouse hover events in your Coded UI tests can be configured to use different keys.

WARNING

You should not have to change the keyboard assignments for mouse hover events under ordinary circumstances. Use caution when reassigning the keyboard assignment. Your choice might already be in use elsewhere within Visual Studio or the application being tested.

To change the keyboard assignments, modify the following configuration file:

`%ProgramFiles(x86)%\Microsoft Visual Studio\2017\Enterprise\Common7\IDE\CodedUITestBuilder.exe.config`

In the configuration file, change the values for the `HoverKeyModifier` and `HoverKey` keys to modify the keyboard assignments:

```
<!-- Begin : Background Recorder Settings -->
<!-- HoverKey to use. -->
<add key="HoverKeyModifier" value="Control, Shift"/>
<add key="HoverKey" value="R"/>
```

Set implicit mouse hovers for the web browser

If you're having issues recording mouse hovers on a website:

In many websites, when you hover over a particular control, it expands to show additional details. Generally, these look like menus in desktop applications. Because this is a common pattern, Coded UI tests enable implicit hovers for web browsing. For example, if you record hovers in Internet Explorer, an event is fired. These events can lead to redundant hovers getting recorded. Because of this, implicit hovers are recorded with

`ContinueOnError` set to `true` in the UI test configuration file. This allows playback to continue if a hover event fails.

To enable the recording of implicit hovers in a web browser, open the configuration file:

`%ProgramFiles(x86)%\Microsoft Visual Studio\2017\Enterprise\Common7\IDE\CodedUITestBuilder.exe.config`

Verify that the configuration file has the key `RecordImplicitHovers` set to a value of `true` as shown in the following sample:

```
<!--Use this to enable/disable recording of implicit hovers.-->
<add key="RecordImplicitHover" value="true"/>
```

Customize the Coded UI test

After you've created your Coded UI test, you can edit it by using any of the following tools in Visual Studio:

- Use **Coded UI Test Builder** to add additional controls and validation to your tests. See the section [Add controls and validate their properties](#) in this topic.
- **Coded UI Test Editor** lets you easily modify your Coded UI tests. Using **Coded UI Test Editor**, you can locate, view, and edit your test methods. You can also edit UI actions and their associated controls in the UI control map. For more information, see [Edit Coded UI tests using the Coded UI test editor](#).
- **Code Editor:**
 - Manually add code for the controls in your test as described in the [Coded UI control actions and properties](#) section in this topic.
 - After you create a Coded UI test, you can modify it to be data-driven. For more information, see [Create a data-driven Coded UI test](#).

- In a Coded UI test playback, you can instruct the test to wait for certain events to occur, such as a window to appear, the progress bar to disappear, and so on. To do this, add the appropriate `UITestControl.WaitForControlXXX()` method. For a complete list of the available methods, see [Make coded UI tests wait for specific events during playback](#). For an example of a Coded UI test that waits for a control to be enabled using the `WaitForControlEnabled` method, see [Walkthrough: Creating, editing and maintaining a coded UI test](#).
- Coded UI tests include support for some of the HTML5 controls that are included in Internet Explorer 9 and Internet Explorer 10. For more information, see [Using HTML5 controls in coded UI tests](#).
- Coded UI test coding guidance:
 - [Anatomy of a coded UI test](#)
 - [Best practices for coded UI tests](#)
 - [Test a large application with multiple UI Maps](#)
 - [Supported configurations and platforms for coded UI tests and action recordings](#)

The generated code

When you choose **Generate Code**, several pieces of code are created:

- A line in the test method.

```
[CodedUITest]
public class CodedUITest1
{
    ...
    [TestMethod]
    public void CodedUITestMethod1()
    {
        this.UIMap.AddTwoNumbers();
        // To generate more code for this test, select
        // "Generate Code" from the shortcut menu.
    }
}
```

You can right-click in this method to add more recorded actions and verifications. You can also edit it manually to extend or modify the code. For example, you could enclose some of the code in a loop.

You can also add new test methods and add code to them in the same way. Each test method must have the `[TestMethod]` attribute.

- A method in `UIMap.uitest`.

This method includes the detail of the actions you recorded or the value that you verified. You can edit this code by opening `UIMap.uitest`. It opens in a specialized editor in which you can delete or refactor the recorded actions.

You can also view the generated method in `UIMap.Designer.cs`. This method performs the actions that you recorded when you run the test.

```
// File: UIMap.Designer.cs
public partial class UIMap
{
    /// <summary>
    /// Add two numbers
    /// </summary>
    public void AddTwoNumbers()
    { ... }
}
```

WARNING

You should not edit this file, because it will be regenerated when you create more tests.

You can make adapted versions of these methods by copying them to *UIMap.cs*. For example, you could make a parameterized version that you could call from a test method:

```
// File: UIMap.cs
public partial class UIMap // Same partial class
{
    /// <summary>
    /// Add two numbers - parameterized version
    /// </summary>
    public void AddTwoNumbers(int firstNumber, int secondNumber)
    { ... // Code modified to use parameters.
    }
}
```

- Declarations in *UIMap.uitest*.

These declarations represent the UI controls of the application that are used by your test. They are used by the generated code to operate the controls and access their properties.

You can also use them if you write your own code. For example, you can have your test method choose a hyperlink in a web application, type a value in a text box, or branch off and take different testing actions based on a value in a field.

You can add multiple Coded UI tests and multiple UI map objects and files to facilitate testing a large application. For more information, see [Test a large application with multiple UI Maps](#).

For more information about the generated code, see [Anatomy of a coded UI test](#).

Coded UI control actions and properties

When you work with UI test controls in Coded UI tests they are separated into two parts: actions and properties.

- The first part consists of actions that you can perform on UI test controls. For example, Coded UI tests can simulate mouse clicks on a UI test control, or simulate keys typed on the keyboard to affect a UI test control.
- The second part consists of enabling you to get and set properties on a UI test control. For example, Coded UI tests can get the count of items in a `ListBox`, or set a `CheckBox` to the selected state.

Accessing Actions of UI Test Control

To perform actions on UI test controls, such as mouse clicks or keyboard actions, use the methods in the [Mouse](#) and [Keyboard](#) classes:

- To perform a mouse-oriented action, such as a mouse click, on a UI test control, use [Click](#).

```
Mouse.Click(buttonCancel);
```

- To perform a keyboard-oriented action, such as typing into an edit control, use [SendKeys](#).

```
Keyboard.SendKeys(textBoxDestination, @"C:\Temp\Output.txt");
```

Accessing Properties of UI Test Control

To get and set UI control specific property values, you can directly get or set the values the properties of a control, or you can use the [UITestControl.GetProperty](#) and [UITestControl SetProperty](#) methods with the name of the specific property that you want you get or set.

[GetProperty](#) returns an object, which can then be cast to the appropriate [Type](#). [SetProperty](#) accepts an object for the value of the property.

To get or set properties from UI test controls directly

With controls that derive from [UITestControl](#), such as [HtmlList](#) or [WinComboBox](#), you can get or set their property values directly. The following code shows some examples:

```
int i = myHtmlList.ItemCount;
myWinCheckBox.Checked = true;
```

To get properties from UI test controls

- To get a property value from a control, use [GetProperty](#).
- To specify the property of the control to get, use the appropriate string from the [PropertyNames](#) class in each control as the parameter to [GetProperty](#).
- [GetProperty](#) returns the appropriate data type, but this return value is cast as an [Object](#). The return [Object](#) must then be cast as the appropriate type.

Example:

```
int i = (int)GetProperty(myHtmlList.PropertyNames.ItemCount);
```

To set properties for UI test controls

- To set a property in a control, use [SetProperty](#).
- To specify the property of the control to set, use the appropriate string from the [PropertyNames](#) class as the first parameter to [SetProperty](#), with the property value as the second parameter.

Example:

```
SetProperty(myWinCheckBox.PropertyNames.Checked, true);
```

Debug

You can analyze Coded UI tests using Coded UI test logs. Coded UI test logs filter and record important information about your Coded UI test runs. The format of the logs lets you debug issues quickly. For more information, see [Analyze coded UI tests using coded UI test logs](#).

What's next?

Additional options for running Coded UI tests: You can run Coded UI tests directly from Visual Studio, as described earlier in this topic. Additionally, you can run automated UI tests from Microsoft Test Manager, or using Azure Pipelines. When Coded UI tests are automated, they have to interact with the desktop when you

run them, unlike other automated tests.

- [Run unit tests with Test Explorer](#)
- [Run tests in your build process](#)
- [How to: Set up your test agent to run tests that interact with the desktop](#)

Adding support for custom controls: The Coded UI testing framework does not support every possible UI and might not support the UI you want to test. For example, you cannot immediately create a Coded UI test of the UI for Microsoft Excel. However, you can create an extension to the Coded UI testing framework that supports a custom control.

- [Enable coded UI testing of your controls](#)
- [Extend coded UI tests and action recordings](#)

Coded UI tests are often used to automate manual tests. For more information about manual tests, see [Run manual tests with Microsoft Test Manager](#). For more information about automated tests, see [Test tools in Visual Studio](#).

See also

- [Record and play back manual tests](#)
- [Xamarin.UITest](#)
- [Assert](#)
- [Walkthrough: Create, edit, and maintain a Coded UI test](#)
- [Create a Coded UI test to test a UWP app](#)
- [Anatomy of a Coded UI test](#)
- [Best practices for Coded UI tests](#)
- [Test a large application with multiple UI Maps](#)

Walkthrough: Create, edit, and maintain a coded UI test

1/1/2020 • 10 minutes to read • [Edit Online](#)

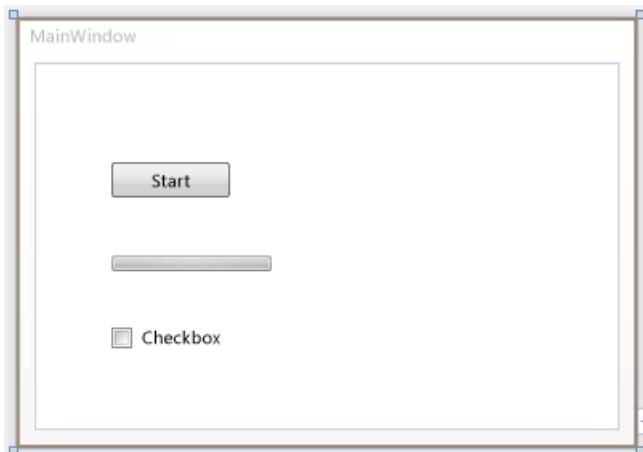
In this walkthrough, you'll learn how to create, edit, and maintain a coded UI test to test a Windows Presentation Framework (WPF) app. The walkthrough provides solutions for correcting tests that have been broken by various timing issues and refactoring of controls.

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Create a WPF app

1. Create a new **WPF App (.NET Framework)** project and name it **SimpleWPFApp**.
The **WPF Designer** opens and displays MainWindow of the project.
2. If the toolbox is not currently open, open it. Choose the **View** menu, and then choose **Toolbox**.
3. Under the **All WPF Controls** section, drag a **Button**, **CheckBox** and **ProgressBar** control onto the MainWindow in the design surface.
4. Select the **Button** control. In the **Properties** window, change the value for the **Name** property from <No Name> to button1. Then change the value for the **Content** property from Button to Start.
5. Select the **ProgressBar** control. In the **Properties** window, change the value for the **Name** property from <No Name> to progressBar1. Then change the value for the **Maximum** property from **100** to **10000**.
6. Select the **Checkbox** control. In the **Properties** window, change the value for the **Name** property from <No Name> to checkBox1 and clear the **Enabled** property.



7. Double-click the button control to add a click event handler.

The *MainWindow.xaml.cs* is displayed in the Code Editor with the cursor in the new `button1_Click` method.

8. At the top of the `MainWindow` class, add a delegate. The delegate will be used for the progress bar. To add

the delegate, add the following code:

```
public partial class MainWindow : Window
{
    private delegate void ProgressBarDelegate(System.Windows.DependencyProperty dp, Object value);

    public MainWindow()
    {

        InitializeComponent();
    }
}
```

9. In the button1_Click method, add the following code:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    double progress = 0;

    ProgressBarDelegate updatePbDelegate =
        new ProgressBarDelegate(progressBar1.SetValue);

    do
    {
        progress++;

        Dispatcher.Invoke(updatePbDelegate,
            System.Windows.Threading.DispatcherPriority.Background,
            new object[] { ProgressBar.ValueProperty, progress });
        progressBar1.Value = progress;
    }
    while (progressBar1.Value != progressBar1.Maximum);

    checkBox1.IsEnabled = true;
}
```

10. Save the file.

Run the WPF app

1. On the **Debug** menu, select **Start Debugging** or press **F5**.

2. Notice that the check box control is disabled. Choose **Start**.

In a few seconds, the progress bar should be 100% complete.

3. You can now select the check box control.

4. Close SimpleWPFApp.

Create a shortcut to the WPF app

1. Locate the SimpleWPFApp application that you created earlier.

2. Create a desktop shortcut to the SimpleWPFApp application. Right-click *SimpleWPFApp.exe* and choose **Copy**. On your desktop, right-click and choose **Paste shortcut**.

TIP

A shortcut to the application makes it easier to add or modify coded UI tests for your application because it lets you start the application quickly.

Create a coded UI test for SimpleWPFApp

1. In **Solution Explorer**, right-click the solution and choose **Add > New Project**.
2. Search for and select the **Coded UI Test Project** project template, and continue through the steps until the project is created.

NOTE

If you don't see the **Coded UI Test Project** template, you need to [install the coded UI test component](#).

The new coded UI test project named **CodedUITestProject1** is added to your solution and the **Generate Code for Coded UI Test** dialog box appears.

3. Select the **Record actions, edit UI map or add assertions** option and choose **OK**.

The **UIMap - Coded UI Test Builder** dialog appears, and the Visual Studio window is minimized.

For more information about the options in the dialog box, see [Create coded UI tests](#).

4. Choose **Start Recording** on the **UIMap - Coded UI Test Builder** dialog.



You can pause the recording if needed, for example if you have to deal with incoming mail.



WARNING

All actions performed on the desktop will be recorded. Pause the recording if you are performing actions that may lead to sensitive data being included in the recording.

5. Launch the SimpleWPFApp using the desktop shortcut.

As before, notice that the check box control is disabled.

6. On the SimpleWPFApp, choose **Start**.

In a few seconds, the progress bar should be 100% complete.

7. Check the check box control which is now enabled.

8. Close the SimpleWPFApp application.

9. On the **UIMap - Coded UI Test Builder** dialog, choose **Generate Code**.

10. In the **Method Name** box, type **SimpleAppTest** and choose **Add and Generate**. In a few seconds, the coded UI test appears and is added to the solution.

11. Close **UIMap - Coded UI Test Builder**.

The *CodedUITest1.cs* file appears in the code editor.

12. Save your project.

Run the test

1. From the **Test** menu, choose **Windows** and then choose **Test Explorer**.
2. From the **Build** menu, choose **Build Solution**.
3. In the *CodedUITest1.cs* file, locate the **CodedUITestMethod** method, right-click and select **Run Tests**, or run the test from **Test Explorer**.

While the coded UI test runs, the SimpleWPFApp is visible. It conducts the steps that you did in the previous procedure. However, when the test tries to select the check box for the check box control, the **Test Results** window shows that the test failed. This is because the test tries to select the check box but is not aware that the check box control is disabled until the progress bar is 100% complete. You can correct this and similar issues by using the various `UITestControl.WaitForControlXXX()` methods that are available for coded UI testing. The next procedure will demonstrate using the `WaitForControlEnabled()` method to correct the issue that caused this test to fail. For more information, see [Make coded UI tests wait for specific events during playback](#).

Edit and rerun the coded UI test

1. In the **Test Explorer** window, select the failed test and in the **StackTrace** section, choose the first link to **UIMap.SimpleAppTest()**.
2. The *UIMap.Designer.cs* file opens with the point of error highlighted in the code:

```
// Select 'CheckBox' check box
uICheckBoxCheckBox.Checked = this.SimpleAppTestParams.UICheckBoxCheckBoxChecked;
```

3. To correct this problem, you can make the coded UI test wait for the CheckBox control to be enabled before continuing on to this line using the `WaitForControlEnabled()` method.

WARNING

Do not modify the *UIMap.Designer.cs* file. Any code changes you make will be overwritten every time you generate code using **UIMap - Coded UI Test Builder**. If you have to modify a recorded method, copy it to the *UIMap.cs* file and rename it. The *UIMap.cs* file can be used to override methods and properties in the *UIMapDesigner.cs* file. You must remove the reference to the original method in the *CodedUITest.cs* file and replace it with the renamed method name.

4. In **Solution Explorer**, locate *UIMap.uitest* in your coded UI test project.
5. Open the shortcut menu for *UIMap.uitest* and choose **Open**.

The coded UI test is displayed in the Coded UI Test Editor. You can now view and edit the coded UI test.

6. In the **UI Action** pane, select the test method (SimpleAppTest) that you want to move to the *UIMap.cs* or *UIMap.vb* file. Moving the method to a different file allows custom code to be added that won't be overwritten when the test code is recompiled.
7. Choose the **Move Code** button on the **Coded UI Test Editor** toolbar.
8. A Microsoft Visual Studio dialog box is displayed. It warns you that the method will be moved from the *UIMap.uitest* file to the *UIMap.cs* file, and that you'll no longer be able to edit the method using the Coded UI Test Editor. Choose **Yes**.

The test method is removed from the *UIMap.uitest* file and no longer is displayed in the UI Actions pane. To edit the moved test file, open the *UIMap.cs* file from **Solution Explorer**.

9. On the Visual Studio toolbar, choose **Save**.

The updates to the test method are saved in the *UIMap.Designer* file.

WARNING

Once you have moved the method, you can no longer edit it using the Coded UI Test Editor. You must add your custom code and maintain it using the Code Editor.

10. Rename the method from `SimpleAppTest()` to `ModifiedSimpleAppTest()`

11. Add the following using statement to the file:

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

12. Add the following `WaitForControlEnabled()` method before the offending line of code identified previously:

```
uICheckBoxCheckBox.WaitForControlEnabled();  
  
// Select 'CheckBox' check box  
uICheckBoxCheckBox.Checked = this.SimpleAppTestParams.UICheckBoxCheckBoxChecked;
```

13. In the *CodedUITest1.cs* file, locate the **CodedUITestMethod** method and either comment out or rename the reference to the original `SimpleAppTest()` method and then replace it with the new `ModifiedSimpleAppTest()`:

```
[TestMethod]  
public void CodedUITestMethod1()  
{  
    // To generate code for this test, select "Generate Code for Coded UI Test" from the  
    // shortcut menu and select one of the menu items.  
    // For more information on generated code, see http://go.microsoft.com/fwlink/?  
LinkId=179463  
    //this.UIMap.SimpleAppTest();  
    this.UIMap.ModifiedSimpleAppTest();  
}
```

14. On the **Build** menu, choose **Build Solution**.

15. Right-click the **CodedUITestMethod** method and select **Run Tests**.

16. This time the coded UI test successfully completes all the steps in the test, and **Passed** is displayed in the **Test Explorer** window.

Refactor a control in SimpleWPFApp

1. In the *MainWindow.xaml* file, in the designer, select the button control.
2. At the top of the **Properties** window, change the **Name** property value from **button1** to **buttonA**.
3. On the **Build** menu, choose **Build Solution**.
4. In **Test Explorer**, run **CodedUITestMethod1**.

The test fails because the coded UI test cannot locate the button control that was originally mapped in the *UIMap* as `button1`. Refactoring can impact coded UI tests in this manner.

5. In **Test Explorer**, in the **StackTrace** section, choose the first link next to

UIMpa.ModifiedSimpleAppTest()

The *UIMpa.cs* file opens. The point of error is highlighted in the code:

```
// Click 'Start' button  
Mouse.Click(uiStartButton, new Point(27, 10));
```

Notice that the line of code earlier in this procedure is using `uiStartButton`, which is the *UIMpa* name before it was refactored.

To correct the issue, you can add the refactored control to the *UIMpa* by using the **Coded UI Test Builder**. You can update the test's code to use the code, as demonstrated in the next procedure.

Map refactored control rerun the test

1. In the *CodedUITest1.cs* file, in the **CodedUITestMethod1()** method, right-click, select **Generate Code for Coded UI Test** and then choose **Use Coded UI Test Builder**.

The **UIMpa - Coded UI Test Builder** appears.

2. Using the desktop shortcut you created earlier, run the SimpleWPFApp application that you created earlier.
3. On the **UIMpa - Coded UI Test Builder** dialog, drag the crosshair tool to the **Start** button on SimpleWPFApp.

The **Start** button is enclosed in a blue box. **Coded UI Test Builder** takes a few seconds to process the data for the selected control and display the control's properties. Notice that the value of **AutomationUid** is **buttonA**.

4. In the properties for the control, choose the arrow at the upper-left corner to expand the UI Control Map. Notice that **UIStartButton1** is selected.
5. In the toolbar, choose the **Add control to UI Control Map**.

The status at the bottom of the window verifies the action by displaying **Selected control has been added to the UI control map**.

6. On the **UIMpa - Coded UI Test Builder** dialog, choose **Generate Code**.

The **Coded UI Test Builder - Generate Code** dialog appears with a note indicating that no new method is required, and that code will only be generated for the changes to the UI control map.

7. Choose **Generate**.
8. Close SimpleWPFApp.
9. Close **UIMpa - Coded UI Test Builder**.
10. In **Solution Explorer**, open the *UIMpa.Designer.cs* file.
11. In the *UIMpa.Designer.cs* file, locate the **UIStartButton1** property. Notice the `SearchProperties` is set to `"buttonA"`:

```

public WpfButton UIStartButton1
{
    get
    {
        if ((this.mUIStartButton1 == null))
        {
            this.mUIStartButton1 = new WpfButton(this);
            #region Search Criteria
            this.mUIStartButton1.SearchProperties[WpfButton.PropertyNames.AutomationId] =
"buttonA";
            this.mUIStartButton1.WindowTitles.Add("MainWindow");
            #endregion
        }
        return this.mUIStartButton1;
    }
}

```

Now you can modify the coded UI test to use the newly mapped control. As pointed out in the previous procedure if you want to override any methods or properties in the coded UI test, you must do so in the *UIMap.cs* file.

12. In the *UIMap.cs* file, add a constructor and specify the `SearchProperties` property of the `UIStartButton` property to use the `AutomationID` property with a value of `"buttonA"`:

```

public UIMap()
{
    this.UIMainWindowWindow.UIStartButton.SearchProperties[WpfButton.PropertyNames.AutomationId] =
"buttonA";
}

```

13. On the **Build** menu, choose **Build Solution**.

14. In **Test Explorer**, run **CodedUITestMethod1**.

This time, the coded UI test successfully completes all the steps in the test. In the **Test Results** window, you see a status of **Passed**.

Videos



[Get started with coded UI tests](#)

FAQ

[Coded UI tests FAQ](#)

See also

- [Use UI automation to test your code](#)
- [Supported configurations and platforms for coded UI tests and action recordings](#)
- [Edit coded UI tests using the coded UI test editor](#)

Create a coded UI test to test a UWP app

1/1/2020 • 5 minutes to read • [Edit Online](#)

This article explains how to create a coded UI test for a Universal Windows Platform (UWP) app.

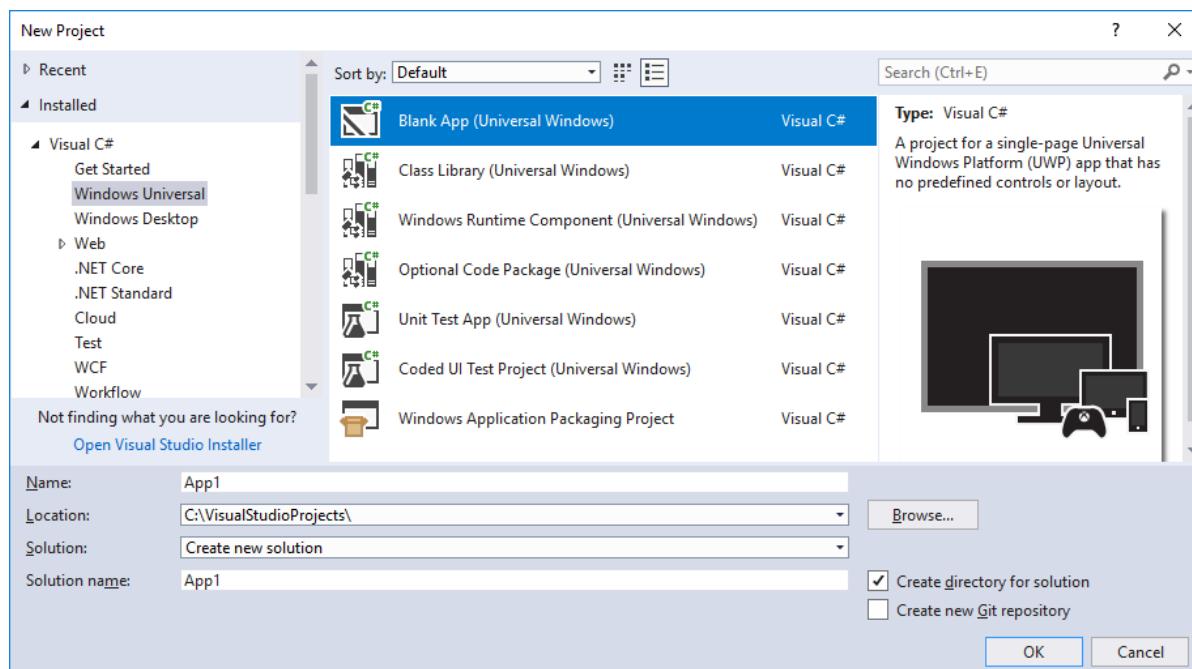
NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Create a UWP app to test

The first step is to create a simple UWP app to run the test against.

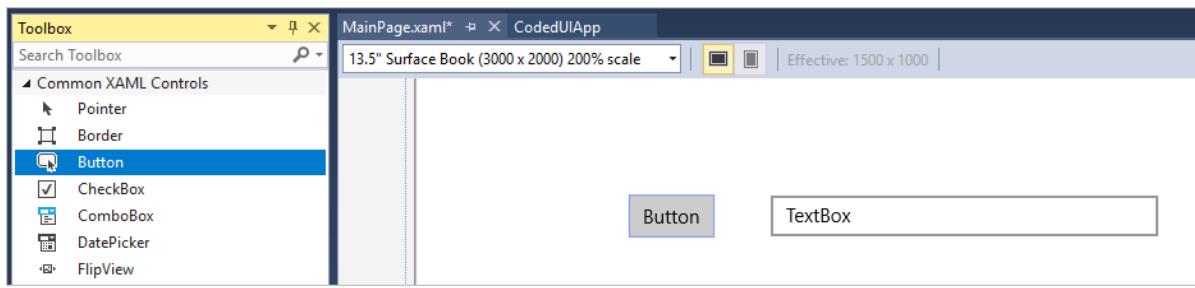
1. In Visual Studio, create a new project using the **Blank App (Universal Windows)** template for Visual C# or Visual Basic.



2. In the **New Universal Windows Platform Project** dialog, select **OK** to accept the default platform versions.
3. From **Solution Explorer**, open *MainPage.xaml*.

The file opens in the **XAML Designer**.

4. Drag a button control and a textbox control from **Toolbox** to the design surface.

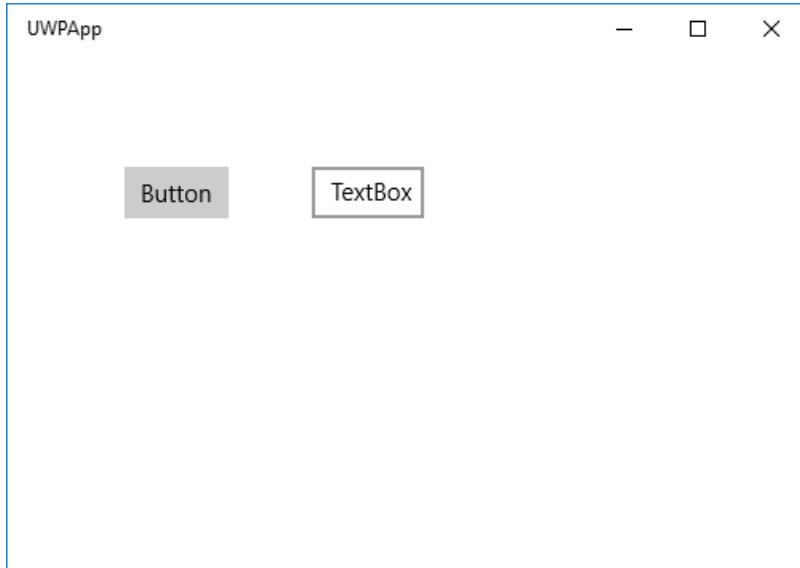


5. Give names to the controls. Select the textbox control, and then in the **Properties** window, enter **textBox** in the **Name** field. Select the button control, and then in the **Properties** window, enter **button** in the **Name** field.
6. Double-click the button control and add the following code to the body of the **Button_Click** method. This code simply sets the text in the textbox to the name of the button control, just to give us something to verify with the coded UI test we'll create later.

```
this.textBox.Text = this.button.Name;
```

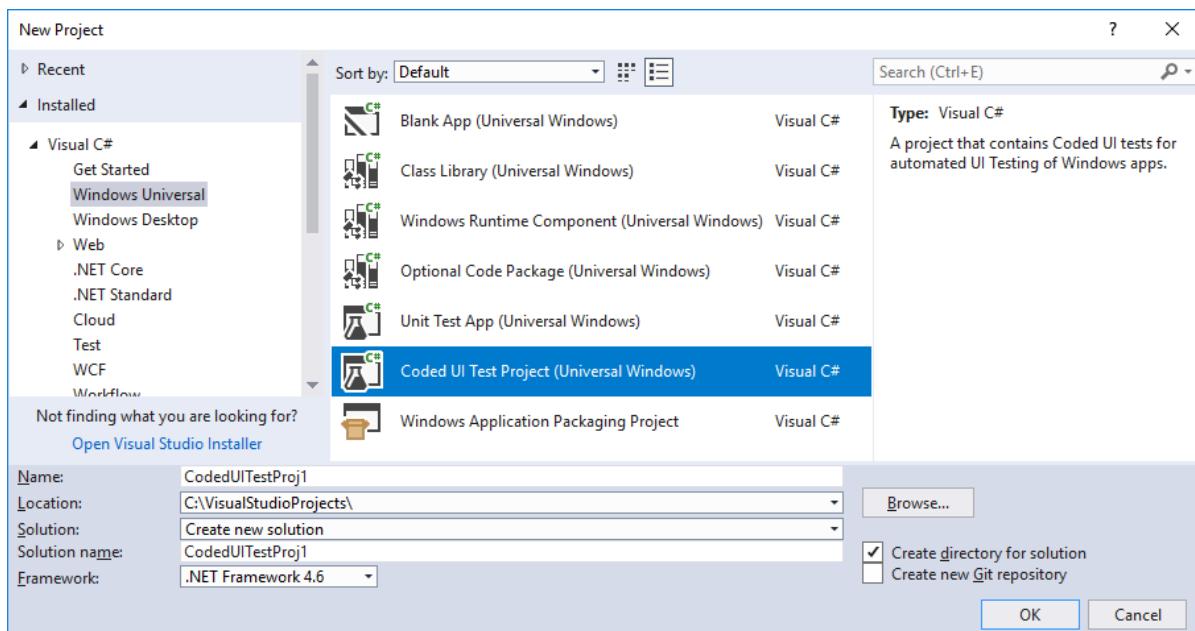
```
Me.textBox.Text = Me.button.Name
```

7. Press **Ctrl+F5** to run the app. You should see something like the following:



Create a coded UI test

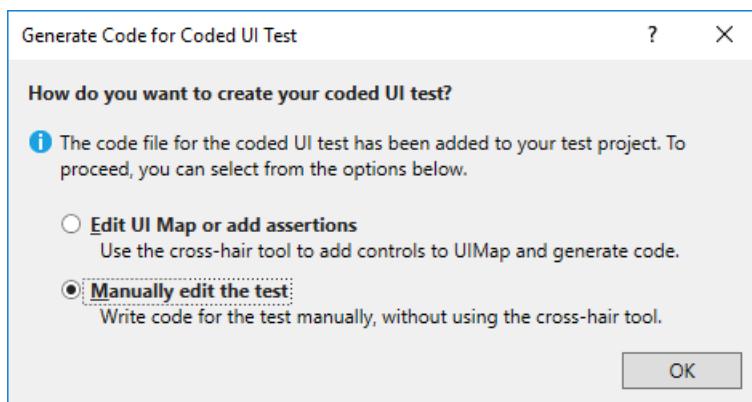
1. To add a test project to the solution, right-click on the solution in **Solution Explorer** and choose **Add > New Project**.
2. Search for and select the **Coded UI Test Project (Universal Windows)** template.



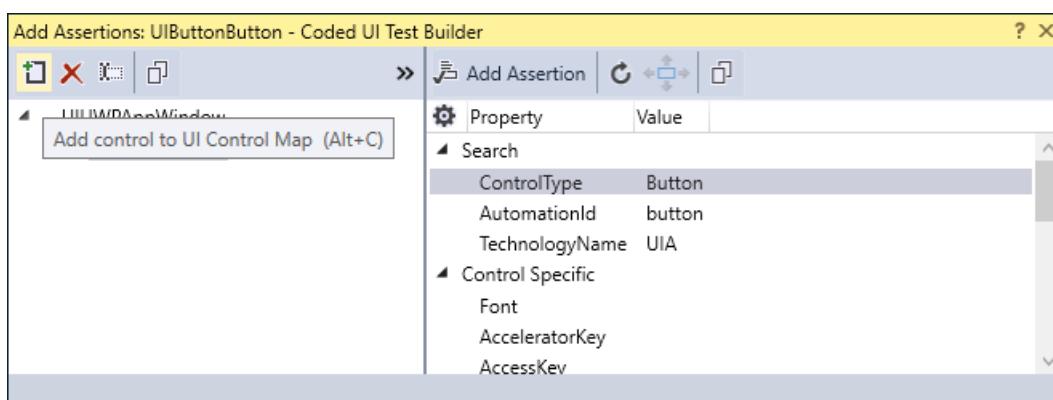
NOTE

If you don't see the **Coded UI Test Project (Universal Windows)** template, you need to [install the coded UI test component](#).

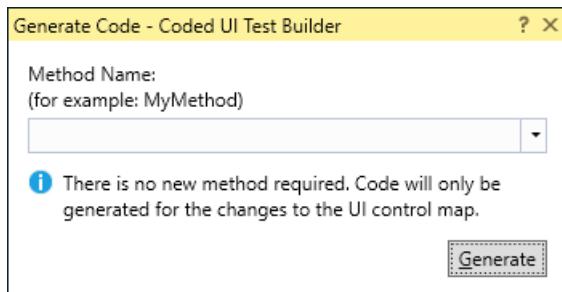
3. In the **Generate Code for Coded UI Test** dialog, select **Manually edit the test**.



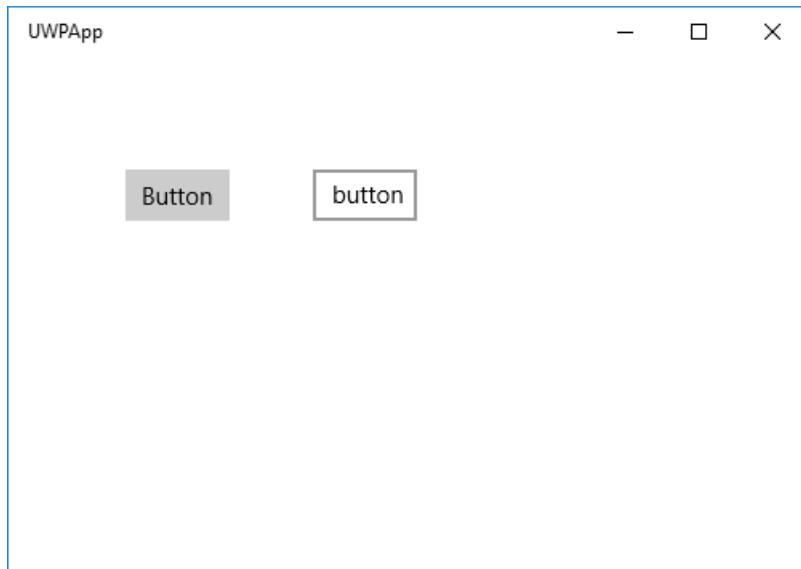
4. If your UWP app is not already running, start it by pressing **Ctrl+F5**.
5. Open the **Coded UI Test Builder** dialog by placing the cursor in the `CodedUITestMethod1` method and then choosing **Test > Generate Code for Coded UI Test > Use Coded UI Test Builder**.
6. Add the controls to the UI control map. Use the **Coded UI Test Builder** cross-hair tool to select the button control in the UWP app. In the **Add Assertions** dialog, expand the **UI Control Map** pane if necessary, and then select **Add control to UI Control Map**.



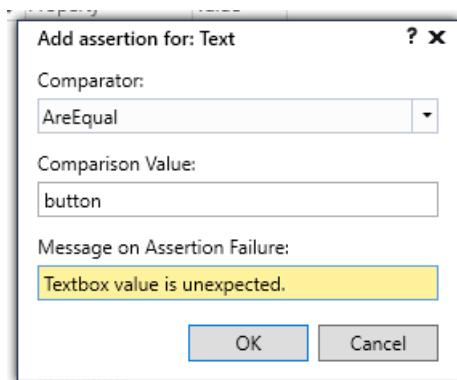
7. Repeat the previous step to add the textbox control to the UI control map.
8. In the **Coded UI Test Builder** dialog, select **Generate Code** or press **Ctrl+G**. Then select **Generate** to create code for changes to the UI control map.



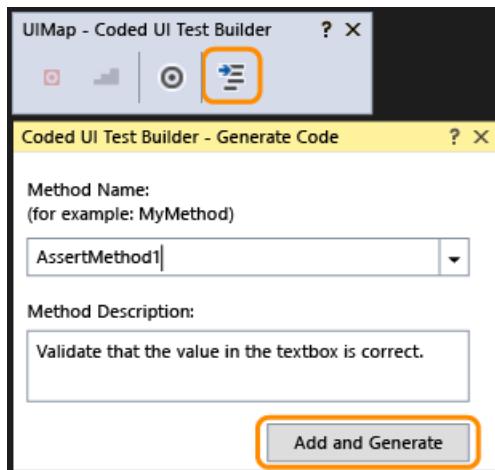
9. To verify that the text in the textbox changes to **button** when the button is clicked, click the button.



10. Add an assertion to verify the text in the textbox control. Use the cross-hair tool to select the textbox control, and then select the **Text** property in the **Add Assertions** dialog. Then, select **Add Assertion** or press **Alt+A**. In the **Message on Assertion Failure** box, enter **Textbox value is unexpected.** and then select **OK**.



11. Generate test code for the assertion. In the **Coded UI Test Builder** dialog, select **Generate Code**. In the **Generate Code** dialog, select **Add and Generate**.



In **Solution Explorer**, open *UIMap.Designer.cs* to view the added code for the assert method and the controls.

TIP

If you're using Visual Basic, open *CodedUITest1.vb*. Then, in the `CodedUITestMethod1()` test method code, right-click on the call to the assert method `Me.UIMap.AssertMethod1()` and choose **Go To Definition**. *UIMap.Designer.vb* opens in the code editor, and you can view the added code for the assert method and the controls.

WARNING

Do not modify the *UIMap.designer.cs* or *UIMap.Designer.vb* files directly. If you do, your changes will be overwritten when the test is generated.

The assert method looks like this:

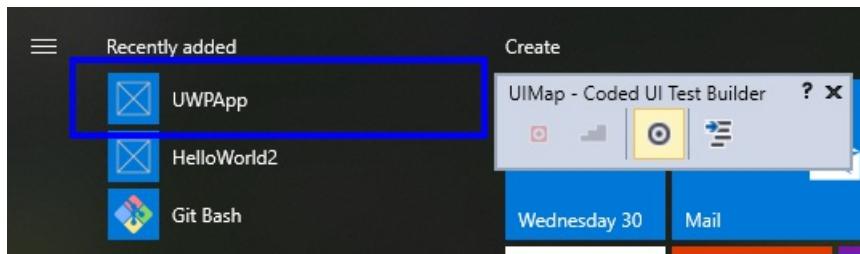
```
public void AssertMethod1()
{
    #region Variable Declarations
    XamlEdit uITextBoxEdit = this.UIUWPAppWindow.UITextBoxEdit;
    #endregion

    // Verify that the 'Text' property of 'textBox' text box equals 'button'
    Assert.AreEqual(this.AssertMethod1ExpectedValues.UITextBoxEditText, uITextBoxEdit.Text, "Textbox
value is unexpected.");
}
```

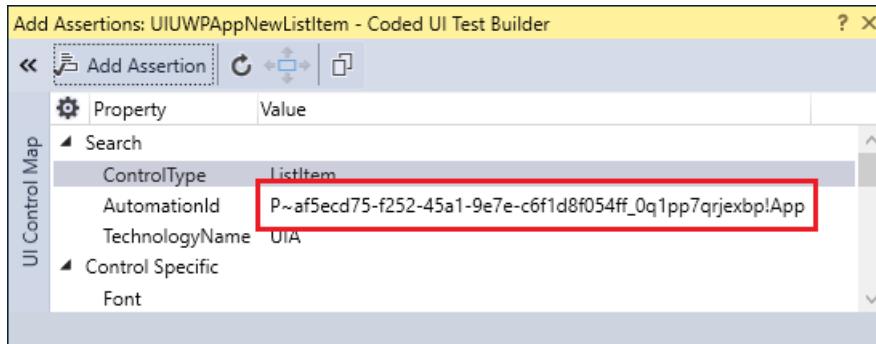
```
Public Sub AssertMethod1()
    Dim uITextBoxEdit As XamlEdit = Me.UIApp2Window.UITextBoxEdit

    'Verify that the 'Text' property of 'textBox' text box equals 'button'
    Assert.AreEqual(Me.AssertMethod1ExpectedValues.UITextBoxEditText, uITextBoxEdit.Text, "Textbox value
is unexpected.")
End Sub
```

12. Next, we need to obtain the **AutomationId** of the UWP app that we want to test. Open the Windows **Start** menu to see the tile for the app. Then, drag the cross-hair tool from the **Coded UI Test Builder** dialog to the tile for your app. When a blue box surrounds the tile, release your mouse.



The **Add Assertions** dialog box opens and displays the **AutomationId** for your app. Right-click **AutomationId** and choose **Copy Value to Clipboard**.



13. Add code to the test method to launch the UWP app. In **Solution Explorer**, open *CodedUITest1.cs* or *CodedUITest1.vb*. Above the call to `AssertMethod1`, add code to launch the UWP app:

```
XamlWindow.Launch("af5ecd75-f252-45a1-9e7e-c6f1d8f054ff_0q1pp7qrjexbp!App")
```

```
XamlWindow myAppWindow = XamlWindow.Launch("af5ecd75-f252-45a1-9e7e-c6f1d8f054ff_0q1pp7qrjexbp!App");
```

Replace the automation ID in the example code with the value you copied to the clipboard in the previous step.

IMPORTANT

Trim the beginning of the automation ID to remove characters such as `P~`. If you don't trim these characters, the test throws a `Microsoft.VisualStudio.TestTools.UITest.Extension.PlaybackFailureException` when it tries to launch the app.

14. Next, add code to the test method to click the button. On the line after `XamlWindow.Launch`, add a gesture to tap the button control:

```
Gesture.Tap(this.UIMap.UIUWPAppWindow.UIButtonButton);
```

```
Gesture.Tap(Me.UIMap.UIUWPAppWindow.UIButtonButton)
```

After adding the code, the complete `CodedUITestMethod1` test method should appear as follows:

```

[TestMethod]
public void CodedUITestMethod1()
{
    XamlWindow.Launch("af5ecd75-f252-45a1-9e7e-c6f1d8f054ff_0q1pp7qrjexbp!App");

    Gesture.Tap(this.UIMap.UIUWPAppWindow.UIButtonButton);

    // To generate code for this test, select "Generate Code for Coded UI Test" from the shortcut menu
    // and select one of the menu items.
    this.UIMap.AssertMethod1();
}

```

```

<CodedUITest(CodedUITestType.WindowsStore)>
Public Class CodedUITest1

    <TestMethod()>
    Public Sub CodedUITestMethod1()

        ' Launch the app.
        XamlWindow.Launch("af5ecd75-f252-45a1-9e7e-c6f1d8f054ff_0q1pp7qrjexbp!App")

        ' // Tap the button.
        Gesture.Tap(Me.UIMap.UIUWPAppWindow.UIButtonButton)

        Me.UIMap.AssertMethod1()
    End Sub

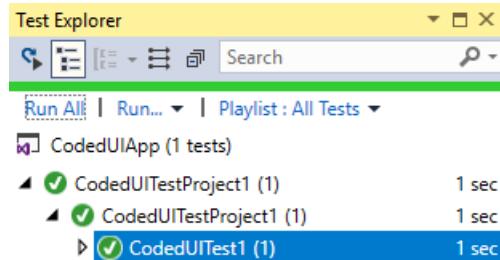
```

15. Build the test project, and then open **Test Explorer** by selecting **Test > Windows > Test Explorer**.

16. Select **Run All** to run the test.

The app opens, the button is tapped, and the textbox's **Text** property is populated. The assert method validates the textbox's **Text** property.

After the test completes, **Test Explorer** displays that the test passed.



CodedUITestMethod1 [Copy All](#)
Source: [CodedUITest1.cs line 26](#)

Test Passed - CodedUITestMethod1
Elapsed time: 0:00:01.1849946

Q & A

Q: Why don't I see the option to record my coded UI test in the Generate Code for a Coded UI Test dialog?

A: The option to record is not supported for UWP apps.

Q: Can I create a coded UI test for my UWP apps based on WinJS?

A: No, only XAML-based apps are supported.

Q: Why can't I modify the code in the *UIMap.Designer* file?

A: Any code changes you make in the *UIMapDesigner.cs* file are overwritten every time you generate code using the **Coded UI Test Builder**. If you have to modify a recorded method, copy it to the *UIMap.cs* file and rename it. The *UIMap.cs* file can be used to override methods and properties in the *UIMapDesigner.cs* file. Remove the reference to the original method in the *CodedUITest.cs* file and replace it with the renamed method name.

See also

- [Use UI automation to test your code](#)
- [Set unique automation properties for UWP controls](#)

Set a unique automation property for UWP controls for testing

1/1/2020 • 4 minutes to read • [Edit Online](#)

If you want to run coded UI tests for your XAML-based UWP application, each control must be identified by a unique automation property. You can assign a unique automation property based on the type of XAML control in your application.

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Static XAML definition

To specify a unique automation property for a control that is defined in your XAML file, you can set the **AutomationProperties.AutomationId** or **AutomationProperties.Name** implicitly or explicitly, as shown in the examples that follow. Setting either of these values gives the control a unique automation property that can be used to identify the control when you create a coded UI test or action recording.

Set the property implicitly

Set **AutomationProperties.AutomationId** to **ButtonX** using the **Name** property in the XAML for the control.

```
<Button Name="ButtonX" Height="31" HorizontalAlignment="Left" Margin="23,26,0,0" VerticalAlignment="Top" Width="140" Click="ButtonX_Click" />
```

Set **AutomationProperties.Name** to **ButtonY** using the **Content** property in the XAML for the control.

```
<Button Content="ButtonY" Height="31" HorizontalAlignment="Left" Margin="23,76,0,0" VerticalAlignment="Top" Width="140" Click="ButtonY_Click" />
```

Set the property explicitly

Set **AutomationProperties.AutomationId** to **ButtonX** explicitly in the XAML for the control.

```
<Button AutomationProperties.AutomationId="ButtonX" Height="31" HorizontalAlignment="Left" Margin="23,26,0,0" VerticalAlignment="Top" Width="140" Click="ButtonX_Click" />
```

Set the **AutomationProperties.Name** to **ButtonY** explicitly in the XAML for the control.

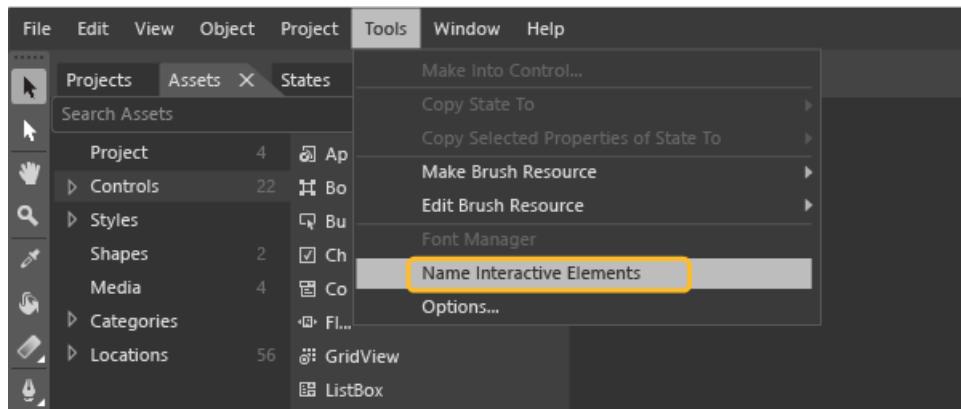
```
<Button AutomationProperties.Name="ButtonY" Height="31" HorizontalAlignment="Left" Margin="23,76,0,0" VerticalAlignment="Top" Width="140" Click="ButtonY_Click" />
```

Assign unique names

In Blend for Visual Studio, you can select an option to assign unique names to interactive elements such as

buttons, list boxes, combo boxes, and text boxes, which gives the controls unique values for **AutomationProperties.Name**.

To assign unique names to existing controls, select **Tools > Name Interactive Elements**.



To automatically give unique names to new controls that you add, select **Tools > Options** to open the **Options** dialog. Select **XAML Designer** and then select **Automatically name interactive elements on creation**. Select **OK** to close the dialog box.

Use a data template

You can define a simple template using **ItemTemplate** to bind the values in a list box to variables:

```
<ListBox Name="listBox1" ItemsSource="{Binding Source={StaticResource employees}}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding EmployeeName}" />
                <TextBlock Text="{Binding EmployeeID}" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

You can also use a template with **ItemContainerStyle** to bind the values to variables:

```
<ListBox Name="listBox1" ItemsSource="{Binding Source={StaticResource employees}}">
    <ListBox.ItemContainerStyle>
        <Style TargetType="ListBoxItem">
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="ListBoxItem">
                        <Grid>
                            <Button Content="{Binding EmployeeName}" AutomationProperties.AutomationId="{Binding EmployeeID}"/>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </ListBox.ItemContainerStyle>
</ListBox>
```

For both of these examples, you must then override the **ToString()** method of **ItemSource**, as shown using the code example that follows. This code makes sure that the **AutomationProperties.Name** value is set and is unique, because you cannot set a unique automation property for each data-bound list item using binding. Setting a unique value for the **Automation Properties.Name** is sufficient in this case.

NOTE

Using this approach, the inner contents of the list item can also be set to a string in the Employee class through binding. As shown in the example, the button control inside each list item is assigned a unique automation id, which is the Employee ID.

```
Employee[] employees = new Employee[]
{
    new Employee("john", "4384"),
    new Employee("margaret", "7556"),
    new Employee("richard", "8688"),
    new Employee("george", "1293")
};

listBox1.ItemsSource = employees;

public override string ToString()
{
    return EmployeeName + EmployeeID; // Unique Identification to be set as the AutomationProperties.Name
}
```

Use a control template

You can use a control template so that each instance of a specific type obtains a unique automation property when it is defined in the code. Create the template so that the **AutomationProperty** binds to a unique ID in the control instance. The following XAML demonstrates one approach to create this binding with a control template:

```
<Style x:Key="MyButton" TargetType="Button">
<Setter Property="Template">
    <Setter.Value>
        <ControlTemplate TargetType="Button">
            <Grid>
                <CheckBox HorizontalAlignment="Left" AutomationProperties.AutomationId="{TemplateBinding Content}">
                </CheckBox>
                <Button Width="90" HorizontalAlignment="Right" Content="{TemplateBinding Content}"
                    AutomationProperties.AutomationId="{TemplateBinding Content}"></Button>
            </Grid>
        </ControlTemplate>
    </Setter.Value>
</Setter>
</Style>
```

When you define two instances of a button using this control template, the automation ID is set to the unique content string for the controls in the template, as shown in the following XAML:

```
<Button Content="Button1" Style="{StaticResource MyButton}" Width="140"/>
<Button Content="Button2" Style="{StaticResource MyButton}" Width="140"/>
```

Dynamic controls

If you have controls that are created dynamically from your code and not created statically or through templates in XAML files, you must set the **Content** or **Name** properties for the control. This action makes sure that each dynamic control has a unique automation property. For example, if you have a check box that must be displayed when you select a list item, you can set these properties, as shown here:

```
private void CreateCheckBox(string txt, StackPanel panel)
{
    CheckBox cb = new CheckBox();
    cb.Content = txt; // Sets the AutomationProperties.Name
    cb.Height = 50;
    cb.Width = 100;
    cb.Name = "DynamicCheckBoxAid"+ txt; // Sets the AutomationProperties.AutomationId
    panel.Children.Add(cb);
}
```

See also

- [Test UWP apps with coded UI tests](#)

Using HTML5 controls in coded UI tests

1/1/2020 • 3 minutes to read • [Edit Online](#)

Coded UI tests include support for some of the HTML5 controls that are included in Internet Explorer 9 and Internet Explorer 10.

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Requirements

- Visual Studio Enterprise

WARNING

In versions prior to Internet Explorer 10, it was possible to run coded UI tests in a higher privilege level compared to that of the Internet Explorer process. When running coded UI tests on Internet Explorer 10, both the coded UI test and the Internet Explorer process must be at the same privilege level. This is because of more secure AppContainer features in Internet Explorer 10.

WARNING

If you create a coded UI test in Internet Explorer 10, it might not run using Internet Explorer 9 or Internet Explorer 8. This is because Internet Explorer 10 includes HTML5 controls such as Audio, Video, ProgressBar and Slider. These HTML5 controls are not recognized by Internet Explorer 9, or Internet Explorer 8. Likewise, your coded UI test using Internet Explorer 9 might include some HTML5 controls that also will not be recognized by Internet Explorer 8.

Audio Control

Audio control: Actions on the HTML5 Audio control are correctly recorded and played back.



ACTION	RECORDING	GENERATED CODE
Play audio Directly from control, or from control's right-click menu.	Play <name> Audio from 00:00:00	HtmlAudio.Play(TimeSpan)
Seek to a specific time in the audio	Seek <name> Audio to 00:01:48	HtmlAudio.Seek(TimeSpan)
Pause audio Directly from control, or from control's right-click menu.	Pause <name> Audio at 00:01:53	HtmlAudio.Pause(TimeSpan)

ACTION	RECORDING	GENERATED CODE
Mute audio Directly from control, or from control's right-click menu.	Mute <name> Audio	HtmlAudio.Mute()
Unmute audio Directly from control, or from control's right-click menu.	Unmute <name> Audio	HtmlAudio.Unmute()
Change volume of audio	Set volume of <name> Audio to 79%	HtmlAudio.SetVolume(float)

See [HTMLAudioElement](#) for a list of properties on which you can add an assertion.

Search properties: The search properties for `HtmlAudio` are `Id`, `Name` and `Title`.

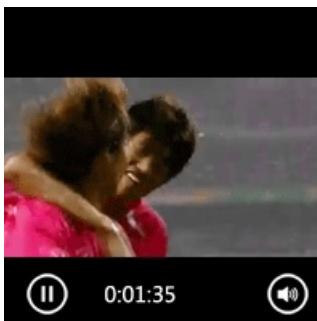
Filter properties: The filter properties for `HtmlAudio` are `Src`, `Class`, `ControlDefinition` and `TagInstance`.

NOTE

The amount of time for Seek and Pause can be significant. During playback, the coded UI test will wait until the specified time in `(TimeSpan)` before Pausing the audio. If by some special circumstance, the specified time has passed before hitting the Pause command, an exception will be thrown.

Video Control

Video control: Actions on the HTML5 Video control are correctly recorded and played back.



ACTION	RECORDING	GENERATED CODE
Play video Directly from control, or from control's right-click menu.	Play <name> Video from 00:00:00	HtmlVideo.Play(TimeSpan)
Seek to a specific time in the video	Seek <name> Video to 00:01:48	HtmlVideo.Seek(TimeSpan)
Pause video Directly from control, or from control's right-click menu.	Pause <name> Video at 00:01:53	HtmlVideo.Pause(TimeSpan)

ACTION	RECORDING	GENERATED CODE
Mute video Directly from control, or from control's right-click menu.	Mute <name> Video	HtmlVideo.Mute()
Unmute video Directly from control, or from control's right-click menu.	Unmute <name> Video	HtmlVideo.Unmute()
Change volume of video	Set volume of <name> Video to 79%	

See [HTMLVideoElement](#) for a list of properties on which you can add an assertion.

Search properties: The search properties for `HtmlVideo` are `Id`, `Name` and `Title`.

Filter properties: The filter properties for `HtmlVideo` are `Src`, `Poster`, `Class`, `ControlDefinition` and `TagInstance`.

NOTE

If you rewind or fast forward the video using -30s or +30s labels, this will be aggregated to seek to the appropriate time.

ProgressBar

ProgressBar control: The ProgressBar is a non-interactable control. You can add assertions on the `Value` and `Max` properties of this control. For more information, see [HTMLProgressElement](#).



See also

- [HTML elements](#)
- [Use UI automation to test your code](#)
- [Create coded UI tests](#)
- [Supported configurations and platforms for coded UI tests and action recordings](#)

Create a data-driven coded UI test

1/1/2020 • 6 minutes to read • [Edit Online](#)

To test different conditions, you can run your tests multiple times with different parameter values. Data-driven coded UI tests are a convenient way to do this. You define parameter values in a data source, and each row in the data source is an iteration of the coded UI test. The overall result of the test will be based on the outcome for all the iterations. For example, if one test iteration fails, the overall test result is failure.

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Requirements

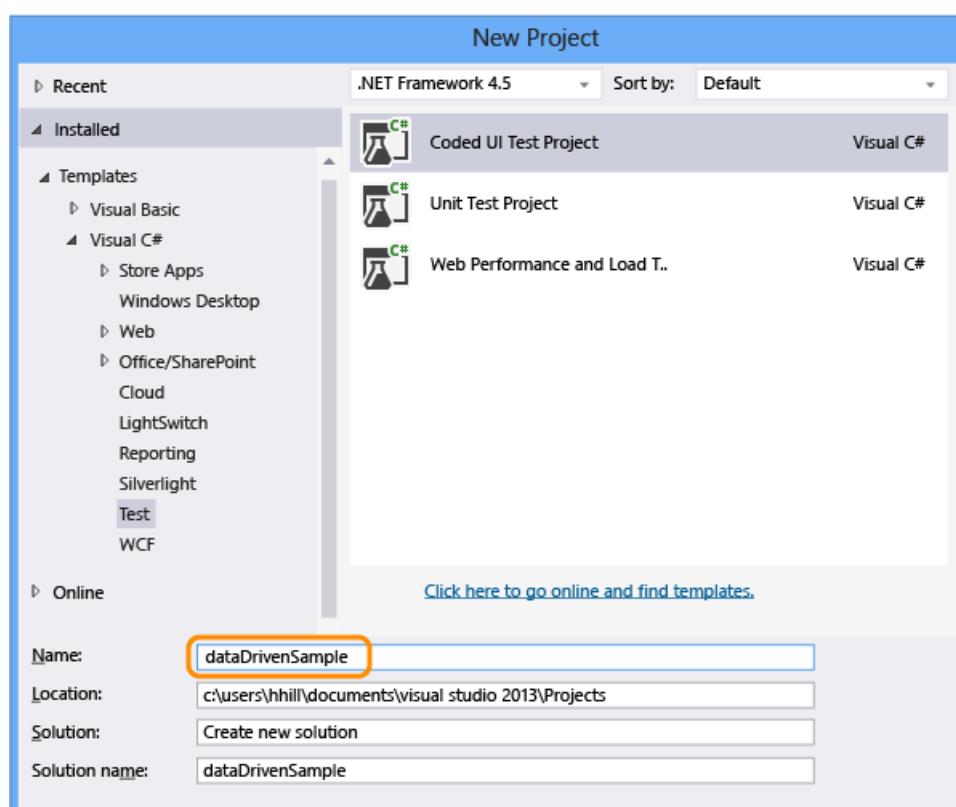
- Visual Studio Enterprise
- Coded UI test component

Create a test project

This sample creates a coded UI test that runs on the Windows Calculator application. It adds two numbers together and uses an assertion to validate that the sum is correct. Next, the assertion and the parameter values for the two numbers are coded to become data-driven and stored in a comma-separated value (.csv) file.

Step 1 - Create a coded UI test

1. Create a project.



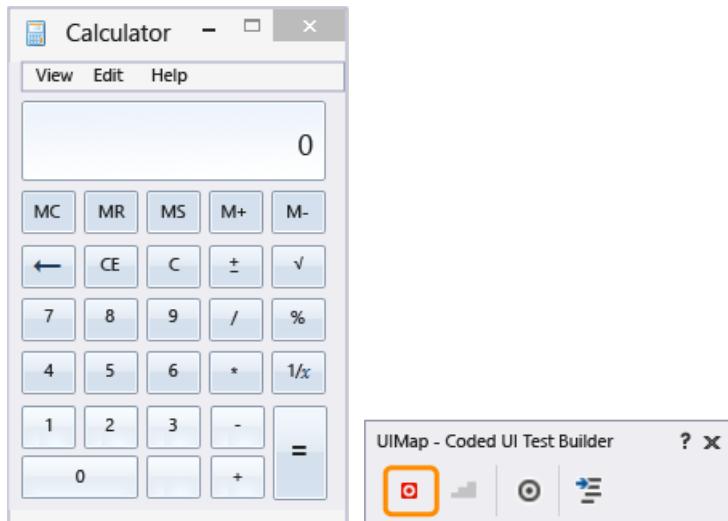
NOTE

If you don't see the **Coded UI Test Project** template, you need to [install the coded UI test component](#).

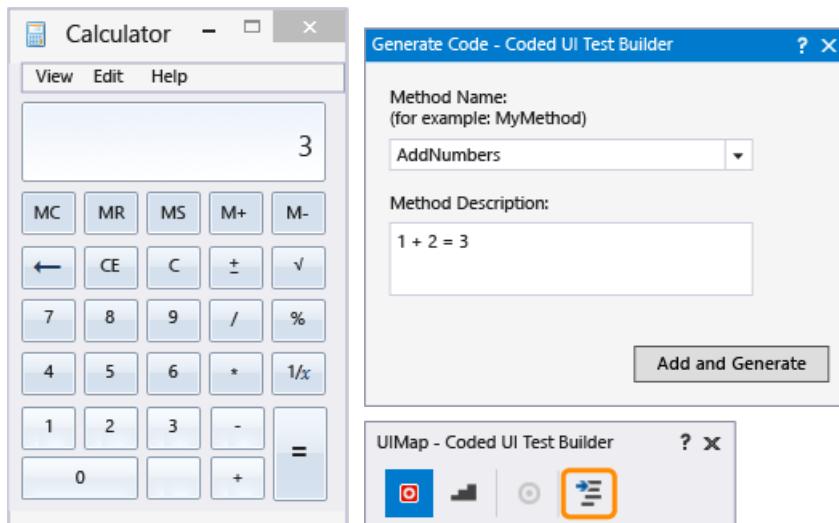
2. Choose to **record the actions**.



3. Open the calculator app and start recording the test.



4. Add 1 plus 2, pause the recorder, and generate the test method. Later we'll replace the values of this user input with values from a data file.



Close the test builder. The method is added to the test:

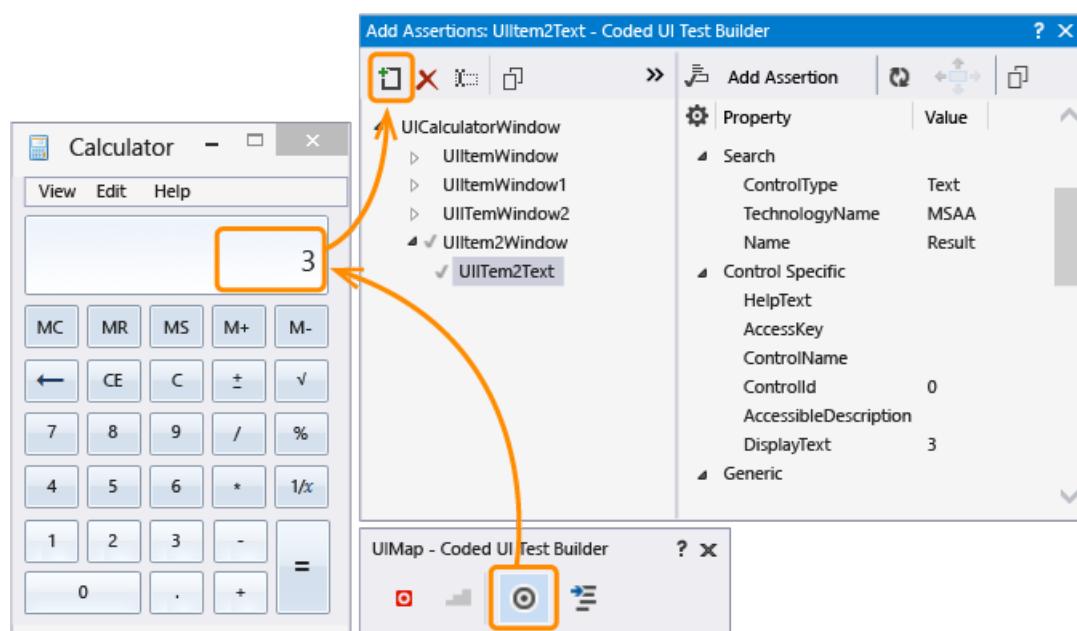
```
[TestMethod]
public void CodedUITestMethod1()
{
    // To generate code for this test, select "Generate Code for Coded UI Test"
    // from the shortcut menu and select one of the menu items.
    this.UIMap.AddNumbers();
}
```

5. Use the `AddNumbers()` method to verify that the test runs. Place the cursor in the test method shown above, open the right-click menu, and choose **Run Tests**. (Keyboard shortcut: **Ctrl+R,T**).

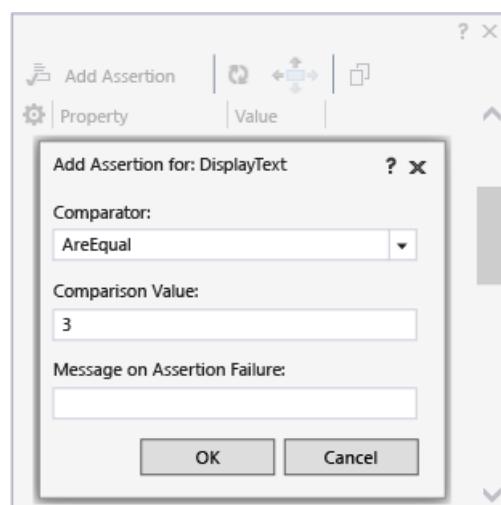
The test result that shows if the test passed or failed is displayed in the **Test Explorer** window. To open the Test Explorer window, from the **Test** menu, choose **Windows** and then choose **Test Explorer**.

6. Because a data source can also be used for assertion parameter values—which are used by the test to verify expected values—let's add an assertion to validate that the sum of the two numbers is correct. Place the cursor in the test method shown above, open the right-click menu and choose **Generate Code for Coded UI Test**, and then **Use Coded UI Test Builder**.

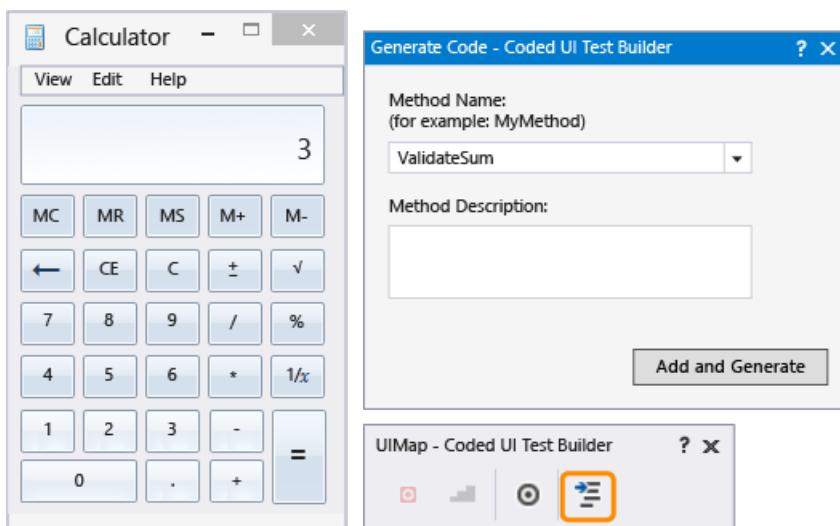
Map the text control in the calculator that displays the sum.



7. Add an assertion that validates that the value of the sum is correct. Choose the **DisplayText** property that has the value of **3** and then choose **Add Assertion**. Use the **AreEqual** comparator and verify that the comparison value is **3**.



- After configuring the assertion, generate code from the builder again. This creates a new method for the validation.



Because the `ValidateSum` method validates the results of the `AddNumbers` method, move it to the bottom of the code block.

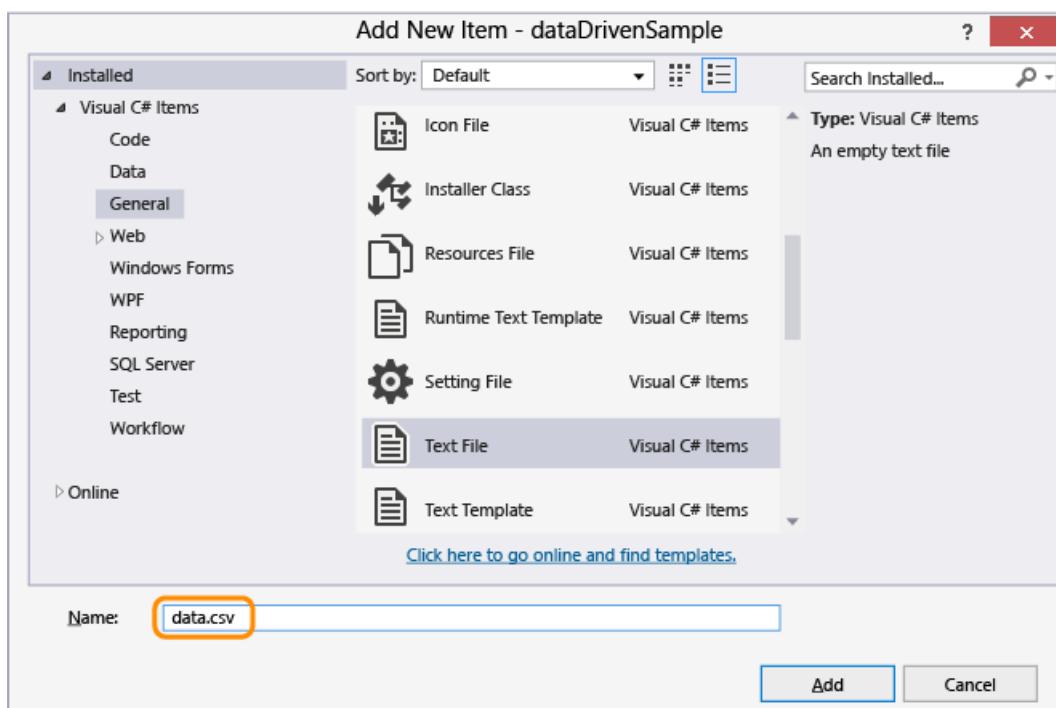
```
public void CodedUITestMethod1()
{
    this.UIMap.AddNumbers();
    this.UIMap.ValidateSum();
}
```

- Verify that the test runs by using the `ValidateSum()` method. Place the cursor in the test method shown above, open the right-click menu, and choose **Run Tests**. (Keyboard shortcut: **Ctrl+R,T**).

At this point, all the parameter values are defined in their methods as constants. Next, let's create a data set to make our test data-driven.

Step 2 - Create a data set

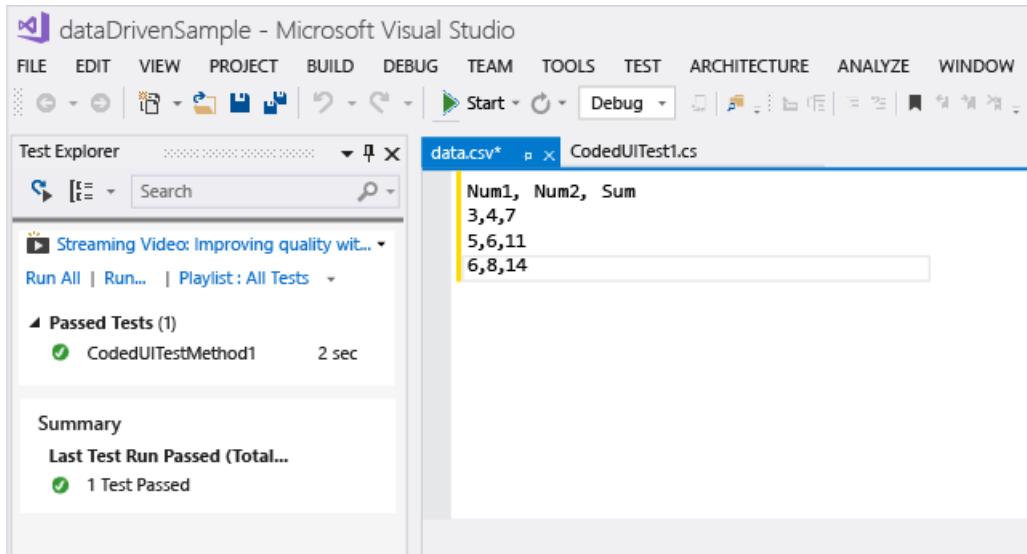
- Add a text file to the dataDrivenSample project named `data.csv`.



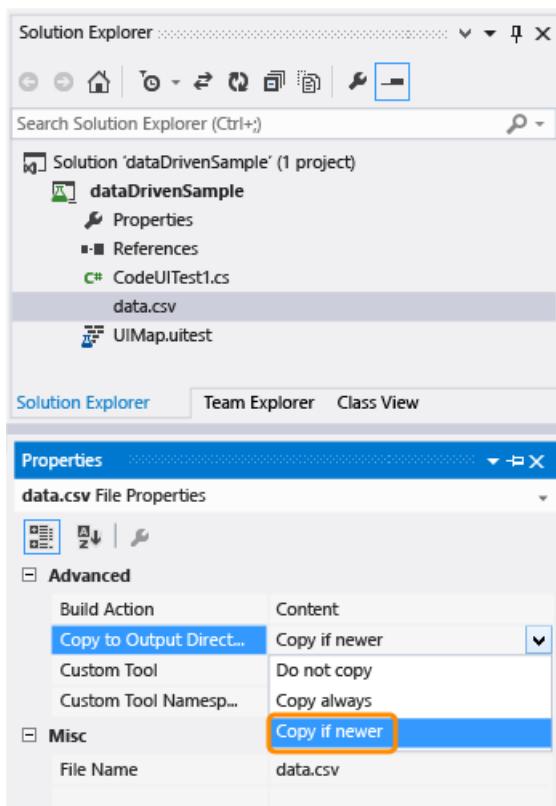
- Populate the `.csv` file with the following data:

NUM1	NUM2	SUM
3	4	7
5	6	11
6	8	14

After adding the data, the file should appear as the following:



3. It is important to save the .csv file using the correct encoding. On the **File** menu, choose **Advanced Save Options** and choose **Unicode (UTF-8 without signature) - Codepage 65001** as the encoding.
4. The .csv file must be copied to the output directory, or the test can't run. Use the **Properties** window to copy it.



Now that we have the data set created, let's bind the data to the test.

Step 3 - Add data source binding

1. To bind the data source, add a `DataSource` attribute within the existing `[TestMethod]` attribute that is immediately above the test method.

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV", "|DataDirectory|\\data.csv", "data#csv",
DataAccessMethod.Sequential), DeploymentItem("data.csv"), TestMethod]
public void CodedUITestMethod1()
{
    this.UIMap.AddNumbers();
    this.UIMap.ValidateSum();
}
```

The data source is now available for you to use in this test method.

TIP

See [data source attribute samples](#) in the Q & A section for samples of using other data source types such as XML, SQL Express and Excel.

2. Run the test.

Notice that the test runs through three iterations. This is because the data source that was bound contains three rows of data. However, you will also notice that the test is still using the constant parameter values and is adding 1 + 2 with a sum of 3 each time.

Next, we'll configure the test to use the values in the data source file.

Step 4 - Use the data in the coded UI test

1. Add `using Microsoft.VisualStudio.TestTools.UITesting.WinForms;` to the top of the `CodedUITest.cs` file:

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;
using System.Windows.Input;
using System.Windows.Forms;
using System.Drawing;
using Microsoft.VisualStudio.TestTools.UITesting;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.VisualStudio.TestTools.UITest.Extension;
using Keyboard = Microsoft.VisualStudio.TestTools.UITesting.Keyboard;
using Microsoft.VisualStudio.TestTools.UnitTesting.WinForms;
```

2. Add `TestContext.DataRow[]` in the `CodedUITestMethod1()` method which will apply values from the data source. The data source values override the constants assigned to UIMap controls by using the controls `SearchProperties`:

```

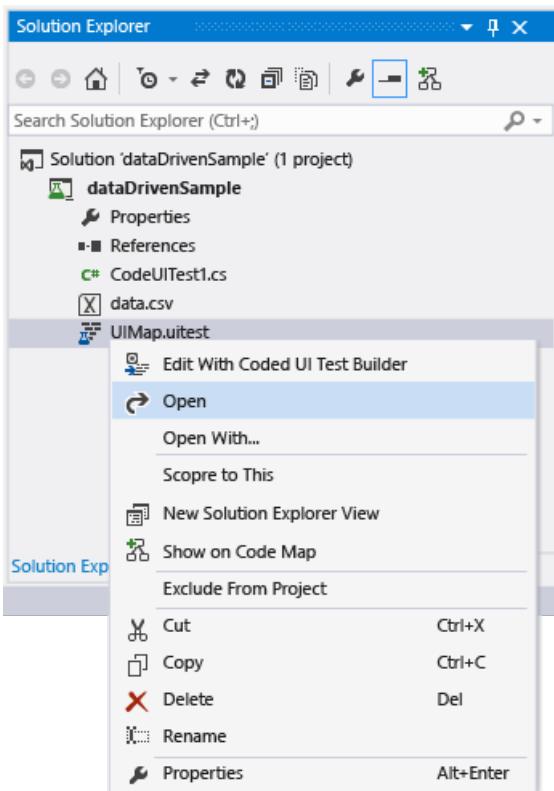
public void CodedUITestMethod1()
{
    this.UIMap.UICalculatorWindow.UIItemWindow.UIItem1Button.SearchProperties[WinButton.PropertyNames.Name]
        = TestContext.DataRow["Num1"].ToString();

    this.UIMap.UICalculatorWindow.UIItemWindow2.UIItem2Button.SearchProperties[WinButton.PropertyNames.Name]
        = TestContext.DataRow["Num2"].ToString();
    this.UIMap.AddNumbers();
    this.UIMap.ValidateSumExpectedValues.UIItem3TextDisplayText = TestContext.DataRow["Sum"].ToString();
    this.UIMap.ValidateSum();
}

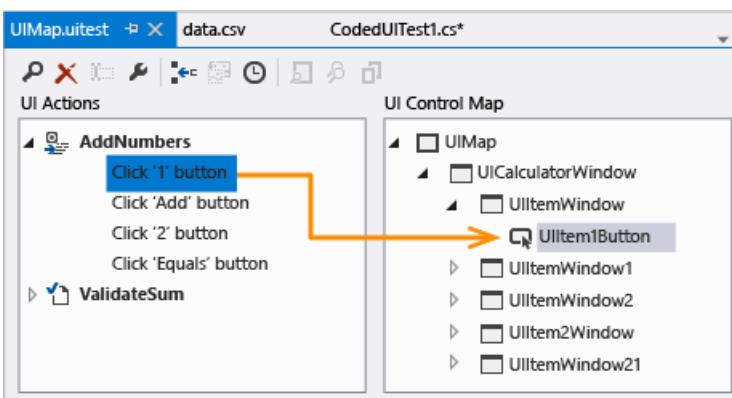
```

To figure out which search properties to code the data to, use the Coded UI Test Editor.

- Open the *UIMap.uitest* file.



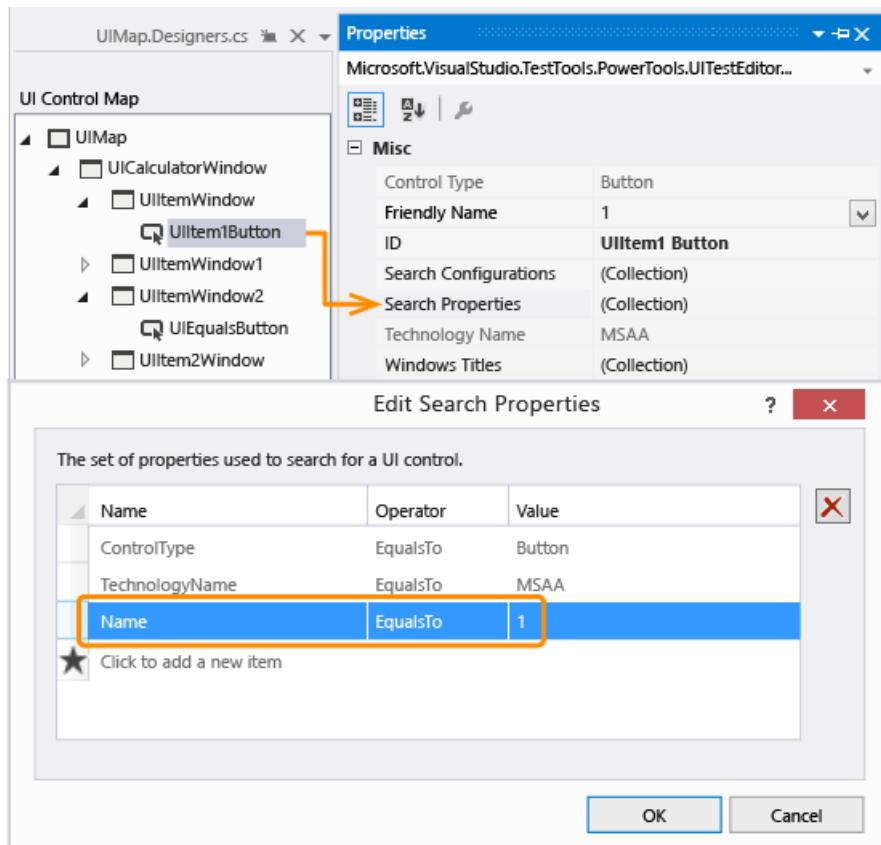
- Choose the UI action and observe the corresponding UI control mapping. Notice how the mapping corresponds to the code, for example, `this.UIMap.UICalculatorWindow.UIItemWindow.UIItem1Button`.



- In the **Properties** Window, open **Search Properties**. The search properties **Name** value is what is being manipulated in the code using the data source. For example, the `SearchProperties` is being assigned the values in the first column of each data row:

```
UIItem1Button.SearchProperties[WinButton.PropertyNames.Name] =
    TestContext.DataRow["Num1"].ToString();
```

- . For the three iterations, this test will change the **Name** value for the search property to 3, then 5, and finally 6.



3. Save the solution.

Step 5 - Run the data-driven test

Verify that the test is now data-driven by running the test again.

You should see the test run through the three iterations using the values in the .csv file. The validation should work as well and the test should display as passed in the Test Explorer.

Q & A

What are the data source attributes for other data source types, such as SQL Express or XML?

A: You can use the sample data source strings in the table below by copying them to your code and making the necessary customizations.

Data Source Types and Attributes

- CSV

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV", "|DataDirectory|\\data.csv", "data#csv",
    DataAccessMethod.Sequential), DeploymentItem("data.csv"), TestMethod]
```

- Excel

```
DataSource("System.Data.Odbc", "Dsn=ExcelFiles;Driver={Microsoft Excel Driver (*.xls)};dbq=|DataDirectory|\\Data.xls;defaultdir=.;driverid=790;maxbuffersize=2048;pagetimeout=5;readonly=true",
    "Sheet1$", DataAccessMethod.Sequential), DeploymentItem("Sheet1.xls"), TestMethod]
```

- Test case in Team Foundation Server

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.TestCase",
    "http://vml13261329:8080/tfs/DefaultCollection;Agile", "30", DataAccessMethod.Sequential), TestMethod]
```

- XML

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML", "|DataDirectory|\\data.xml", "Iterations",
DataAccessMethod.Sequential), DeploymentItem("data.xml"), TestMethod]
```

- SQL Express

```
[DataSource("System.Data.SqlClient", "Data Source=.\sqlExpress;Initial Catalog=tempdb;Integrated
Security=True", "Data", DataAccessMethod.Sequential), TestMethod]
```

Q: Why can't I modify the code in the *UIMap.Designer* file?

A: Any code changes you make in the *UIMapDesigner.cs* file will be overwritten every time you generate code using the UIMap - Coded UI Test Builder. In this sample, and in most cases, the code changes needed to enable a test to use a data source can be made to the test's source code file (that is, *CodedUITest1.cs*).

If you have to modify a recorded method, you must copy it to *UIMap.cs* file and rename it. The *UIMap.cs* file can be used to override methods and properties in the *UIMapDesigner.cs* file. You must remove the reference to the original method in the Coded *UITest.cs* file and replace it with the renamed method name.

See also

- [UIMap](#)
- [Assert](#)
- [Use UI automation to test your code](#)
- [Create coded UI tests](#)
- [Best practices for coded UI tests](#)
- [Supported configurations and platforms for coded UI tests and action recordings](#)

Make coded UI tests wait for specific events during playback

1/1/2020 • 4 minutes to read • [Edit Online](#)

In a coded UI test playback, you can instruct the test to wait for certain events to occur, such as a window to appear, the progress bar to disappear, and so on. To do this, use the appropriate `UITestControl.WaitForControlXXX()` method, as described in the following table. For an example of a coded UI test that waits for a control to be enabled using the `WaitForControlEnabled` method, see [Walkthrough: Creating, editing and maintaining a coded UI test](#).

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Requirements

Visual Studio Enterprise

TIP

You can also add delays before actions using the Coded UI Test Editor. For more information, see [How to: Insert a delay before a UI action using the Coded UI Test Editor](#).

`UITestControl.WaitForControlXXX()` Methods

[WaitForControlReady](#)

Waits for the control to be ready to accept mouse and keyboard input. The engine implicitly calls this API for all actions to wait for the control to be ready before doing any operation. However, in certain esoteric scenario, you may have to do explicit call.

[WaitForControlEnabled](#)

Waits for the control to be enabled when the wizard is doing some asynchronous validation of the input by making calls to the server. For example, you can method to wait for the **Next** button of the wizard to be enabled (). For an example of this method, see [Walkthrough: Creating, editing and maintaining a coded UI test](#).

[WaitForControlExist](#)

Waits for the control to appear on the UI. For example, you are expecting an error dialog after the application has done the validation of the parameters. The time taken for validation is variable. You can use this method to wait for the error dialog box.

[WaitForControlNotExist](#)

Waits for the control to disappear from the UI. For example, you can wait for the progress bar to disappear.

[WaitForControlPropertyEqual](#)

Waits for the specified property of the control to have the given value. For example, you wait for the status text to

change to **Done**.

WaitForControlPropertyNotEqual

Waits for the specified property of the control to have the opposite of a specified value. For example, you wait for the edit box to be not read-only, that is, editable.

WaitForControlCondition

Waits for the specified predicate returns to be `true`. This can be used for complex wait operation (like OR conditions) on a given control. For example, you can wait until the status text is **Succeeded** or **Failed** as shown in the following code:

```
// Define the method to evaluate the condition
private static bool IsStatusDone(UITestControl control)
{
    WinText statusText = control as WinText;
    return statusText.DisplayText == "Succeeded" || statusText.DisplayText == "Failed";
}

// In test method, wait till the method evaluates to true
statusText.WaitForControlCondition(IsStatusDone);
```

WaitForCondition

All the previous methods are instance methods of `UITestControl`. This method is a static method. This method also waits for the specified predicate to be `true` but it can be used for complex wait operation (like OR conditions) on multiple controls. For example, you can wait until the status text is **Succeeded** or until an error message appears, as shown in the following code:

```
// Define the method to evaluate the condition
private static bool IsStatusDoneOrError(UITestControl[] controls)
{
    WinText statusText = controls[0] as WinText;
    WinWindow errorDialog = controls[1] as WinWindow;
    return statusText.DisplayText == "Succeeded" || errorDialog.Exists;
}

// In test method, wait till the method evaluates to true
UITestControl.WaitForCondition<UITestControl[]>(new UITestControl[] { statusText, errorDialog },
IsStatusDoneOrError);
```

All these methods have the following behavior:

The methods return true if the wait is successful and false if the wait failed.

The implicit timeout for the wait operation is specified by `WaitForReadyTimeout` property. The default value of this property is 60000 milliseconds (one minute).

The methods have an overload to take explicit timeout in milliseconds. However, when the wait operation results in an implicit search for the control or, when the application is busy, the actual wait time could be more than the timeout specified.

The previous functions are powerful and flexible and should satisfy almost all conditions. However, in case these methods do not satisfy your needs and you need to code either a `Wait`, or a `Sleep` in your code, it is recommended that you use the `Playback.Wait()` instead of `Thread.Sleep()` API. The reasons for this are:

You can use the `ThinkTimeMultiplier` property to modify the duration of sleep. By default, this variable is 1 but you

can increase or decrease it to change the wait time all over the code. For example, if you are specifically testing over slow network, or some other slow performance case, you can change this variable at one place (or even in the configuration file) to 1.5 to add 50% extra wait at all places.

Playback.Wait() internally calls Thread.Sleep() (after above computation) in smaller chunks in a for-loop while checking for user cancel\break operation. In other words, Playback.Wait() lets you cancel playback before the end of the wait whereas sleep might not or throw exception.

TIP

The Coded UI Test Editor lets you easily modify your coded UI tests. Using the Coded UI Test Editor, you can locate, view, and edit your test methods. You can also edit UI actions and their associated controls in the UI control map. For more information, see [Edit coded UI tests using the Coded UI Test Editor](#).

See also

- [Use UI automation to test your code](#)
- [Create coded UI tests](#)
- [Walkthrough: Creating, editing and maintaining a coded UI test](#)
- [Anatomy of a coded UI test](#)
- [Supported configurations and platforms for coded UI tests and action recordings](#)
- [How to: Insert a delay before a UI action using the coded UI test editor](#)

Use different web browsers with coded UI tests

1/1/2020 • 4 minutes to read • [Edit Online](#)

Coded UI tests can automate testing for web applications by recording your tests using Internet Explorer. You can then customize your test and play it back using either Internet Explorer or other browser types for these web applications.

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

First, install the [Selenium components for coded UI cross browser testing](#).

What's supported across all web browsers?

- [Add custom code for controlling features](#) such as properties, search, and playback waiters.
- Pop-ups and dialog boxes
- [Execute basic JavaScript with no return type](#)
- Search resilience (using smart match) and [performance improvements](#)

Why should I use coded UI tests across multiple web browser types?

By testing your web application using a variety of web browser types, you better emulate the UI experience of your users who may run different browsers. For example, your application might include a control or code in Internet Explorer that is not compatible with other web browsers. By running your coded UI tests across other browsers, you can discover and correct any issue before it impacts your customers.

How do I record and play back coded UI tests on web applications using the supported web browsers?

Recording: You must use the Coded UI Test Builder to record your web application test using Internet Explorer. You can optionally add validation and custom code for the tested controls using a predefined set of properties as you would normally do for coded UI tests. For more information, see [Use UI automation to test your code](#).

NOTE

You cannot record coded UI tests using Google Chrome or Mozilla Firefox browsers.

Play back with Internet Explorer: When no browser is explicitly specified, tests will run on Internet Explorer by default. You can explicitly state the browser to be used by setting the **BrowserWindow.CurrentBrowser** property in your test code. For Internet Explorer, this property should be set to **IE** or **Internet Explorer**.

Play back with non-Internet Explorer web browsers: To play back on non-Internet Explorer web browsers, change `BrowserWindow.CurrentBrowser` property in your test code to either **Firefox** or **Chrome**.

To play back tests on non-IE web browsers, you must install the [Selenium components for Coded UI Cross](#)

Browser Testing.

Install Selenium components

1. On the **Tools** menu, choose **Extensions and Updates**.
2. In the **Extensions and Updates** dialog box, search for `Selenium components for Cross Browser Testing`.
1. On the **Extensions** menu, choose **Manage Extensions**.
2. In the **Manage Extensions** dialog box, search for `Selenium components for Cross Browser Testing`.
3. Highlight the extension and choose **Download**.

TIP

You can also download the Selenium components for Coded UI Cross Browser Testing from [here](#).

For more information about creating and using coded UI tests, see [Create coded UI tests](#).

Enable debugging

To enable debugging your web application, you must complete the following configuration options:

1. Enable Just My Code:
 - a. On the **Tools** menu, choose **Options** and then choose **Debugging**.
 - b. Select **Enable Just My Code**.
2. Disable CLR exceptions:
 - a. On the **Debug** menu, choose **Exceptions**.
 - b. For **Common Language Runtime Exceptions**, uncheck **User-unhandled**.

If don't see the option to change `BrowserWindow.CurrentBrowser` in the coded UI test, you might be using a version of Visual Studio that does not support coded UI tests using various web browsers. To use such coded UI tests, you must use Visual Studio Enterprise edition.

Here are some other things you should know:

- Apple Safari web browser is not supported.
- The action of starting the web browser must be part of the coded UI test.

If you have a web browser already open and you want to run steps on it, the playback will fail unless you are using Internet Explorer. Therefore, it is a best practice to include the startup of your web browser as part of your coded UI tests.
- Automating browser specific based UI actions such as maximize, minimize and restore is not supported.

Tips

You can configure the output to include screenshots in the coded UI logs. To do so, you need to set some configuration settings in the `QTAgent32.exe.config` file. By default, this file is installed in the following location:

`%ProgramFiles(x86)%\Microsoft Visual Studio\2017\Enterprise\Common7\IDE`

Set the following values:

- `EqtTraceLevel` in the `system.diagnostics` section.

- `<add name="EqtTraceLevel" value="4" />`

By setting the value to 3 or higher, screenshots are taken for each action. When the value is set to either 1 or 2, screenshots are taken for error actions only.

For more information, see [Analyze coded UI tests using coded UI test logs](#).

Video resources

[Record on IE and playback everywhere](#)

[Author cross browser tests with coded UI test builder](#)

[Author cross browser tests using plain hand coding without UI Map](#)

[Run cross browser tests sequentially on multiple browsers](#)

[Troubleshoot cross browser test failures](#)

See also

- [Use UI automation to test your code](#)
- [Supported configurations and platforms for coded UI tests and action recordings](#)
- [Analyze coded UI tests using coded UI test logs](#)

Edit coded UI tests using the Coded UI Test Editor

1/1/2020 • 11 minutes to read • [Edit Online](#)

The Coded UI Test Editor lets you easily modify your coded UI tests. Using the Coded UI Test Editor, you can locate, view, and edit the properties of your test methods and UI actions. In addition, you can use the UI control map to view and edit their corresponding controls.

NOTE

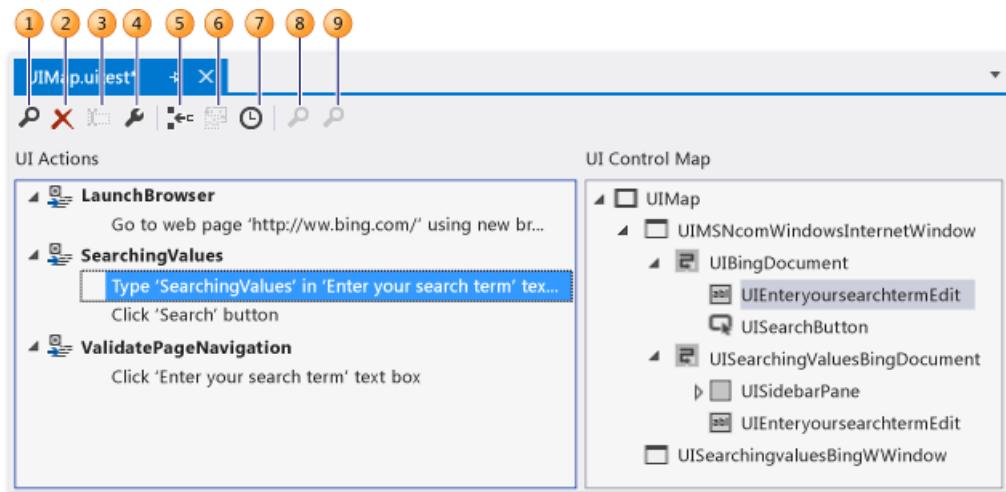
Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Requirements

- Visual Studio Enterprise
- Coded UI test component

Features of the Coded UI Test Editor

Using the Coded UI Test Editor is quicker and more efficient than editing the code in your coded UI test methods using the Code Editor. With the Coded UI Test Editor, you can use the toolbar and shortcut menus to quickly locate and modify property values associated with UI actions and controls. For example, you can use the Coded UI Test Editor's toolbar to perform the following commands:



1. **Find** helps you locate UI actions and controls.
2. **Delete** removes unwanted UI actions.
3. **Rename** changes the names for test methods and controls.
4. **Properties** opens the **Properties** window for the selected item.
5. **Split into a new method** lets you modularize the UI actions.
6. **Move Code** adds custom code to your test methods.
7. **Insert Delay Before** adds a pause prior to a UI action, specified in milliseconds.

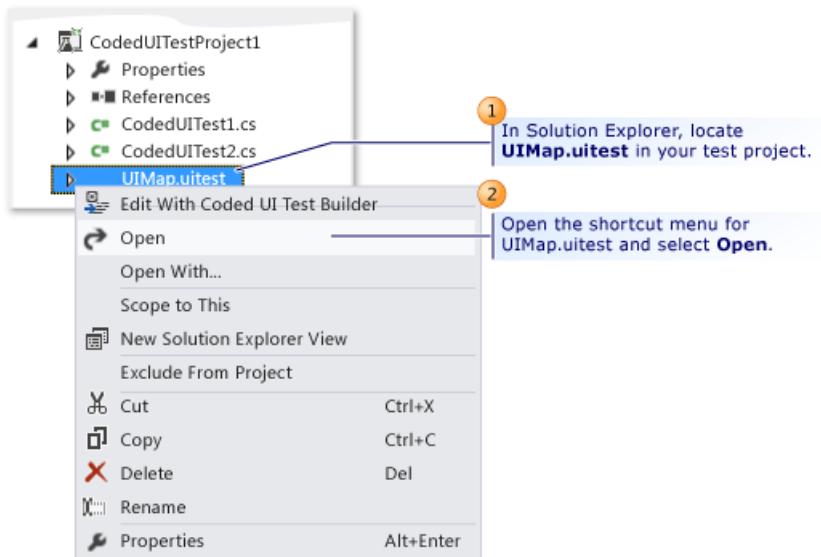
8. **Locate the UI Control** identifies the location of the control in the UI of application under test.

9. **Locate All** helps you verify control property and significant changes to the application's controls.

When you open the *UIMap.uitest* file affiliated with your coded UI test, the coded UI test opens in the **Coded UI Test Editor**. The following procedures describe how you can then locate and edit your test methods, and properties for the UI actions, and controls using the editor's toolbar and shortcut menus.

Open a coded UI test

You can view and edit your Visual C# and Visual Basic-based coded UI test using the **Coded UI Test Editor**.



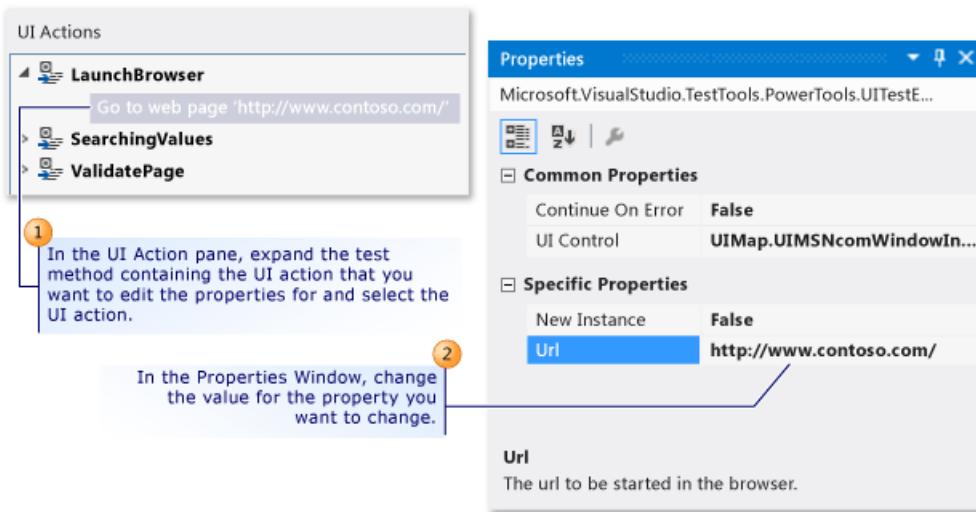
In **Solution Explorer**, open the shortcut menu for *UIMap.uitest* and choose **Open**. The coded UI test is displayed in the **Coded UI Test Editor**. You can now view and edit the recorded methods, actions, and corresponding controls in the coded UI test.

TIP

When you select a UI action that is located in a method in the **UI Actions** pane, the corresponding control is highlighted. You can also modify the UI action or the controls properties.

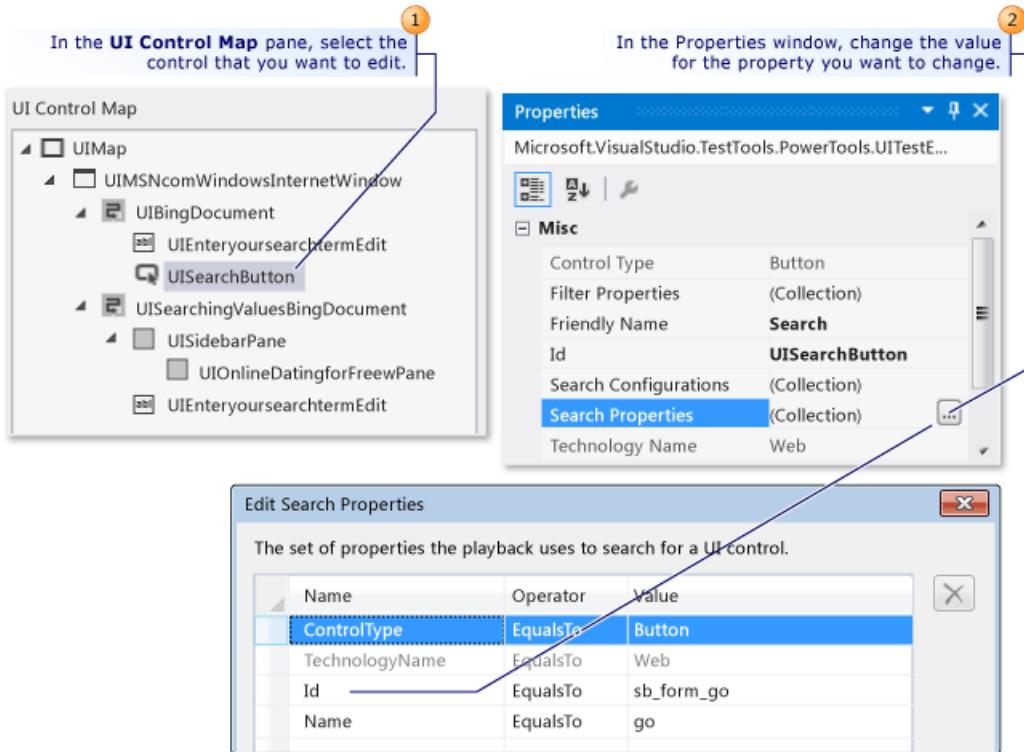
Modify UI action and control properties

Using the Coded UI Test Editor, you can quickly locate and view all the UI actions in your test methods. When you select the UI action in the editor, the corresponding control is automatically highlighted. Likewise, if you select a control, the associated UI actions are highlighted. When you select either a UI action or a control, it is then easy to use the **Properties** window to modify the properties that correspond with it.



To modify the properties for a UI action, in the **UI Action** pane, expand the test method that contains a UI action that you want to edit the properties for, select the UI action, and then modify the properties using the Properties window.

For example, if a server is unavailable, and you have a UI action associated with your web browser that states **Go to Web page 'http://Contoso1/default.aspx'**, you could change the URL to **'http://Contoso2/default.aspx'**.



Modifying the properties for a control is done in the same way as the UI actions. In the **UI Control Map** pane, select the control that you want to edit and modify its properties using the **Properties** window.

For example, a developer might have changed the **(ID)** property on a button control in the source code for the application being tested from "idSubmit" to "idLogin." With the **(ID)** property changed in the application, the coded UI test will not be able to locate the button control and will fail. In this case, the tester can open the **Search Properties** collection and change the **Id** property to match the new value that the developer used in the application. The tester could also change the **Friendly Name** property value from "Submit" to "Login." By making this change, the associated UI action in the Coded UI Test Editor is updated from "Choose 'Submit' button" to "Choose 'Login' button."

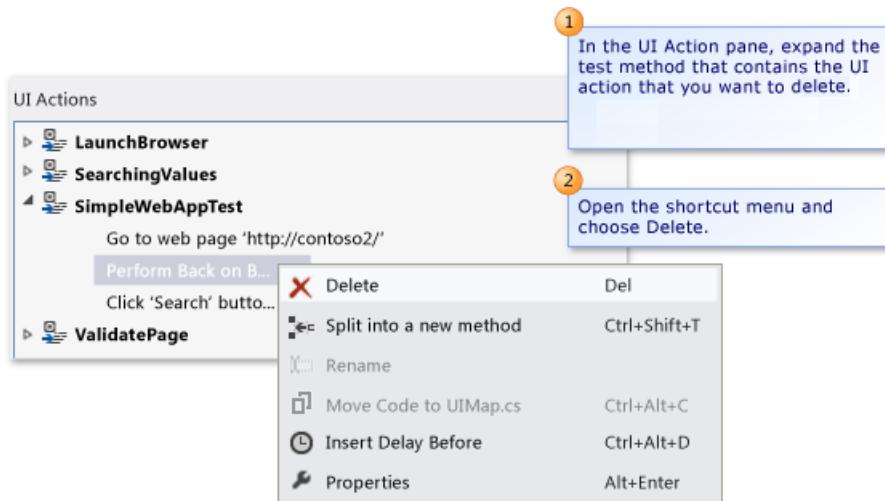
After completing your modifications, save the changes to the *UIMap.Designer* file by choosing **Save** on the Visual Studio toolbar.

Tips

- If the **Properties** window is not displayed, press and hold **Alt** while you press **Enter**, or press **F4**.
- To undo the property changes you made, select **Undo** from the **Edit** menu, or press **Ctrl+Z**.
- You can use the **Find** button in the Coded UI Test editor toolbar to open the **Find and Replace** tool in Visual Studio. You can then use the **Find** control to locate a UI action in the Coded UI Test editor. For example, you can try to find "Click 'Login' button." This can be useful in large tests. You cannot use the replace functionality in the **Find and Replace** tool in the Coded UI Test Editor. For more information, see [Find control in Find and replace text](#).
- Sometimes, it can be difficult to visualize where controls are located in the UI of the application under test. One of the capabilities of the coded UI Test Editor is that you can select a control listed in the UI control map and view its location in the application under test. For more information, see [Locate a UI control in the application under test](#) located further below in this article.
- It might be necessary to expand the container control that contains the control that you want to edit. For more information, see [Locate a control and its descendants](#) located further below in this article.

Delete unwanted UI actions

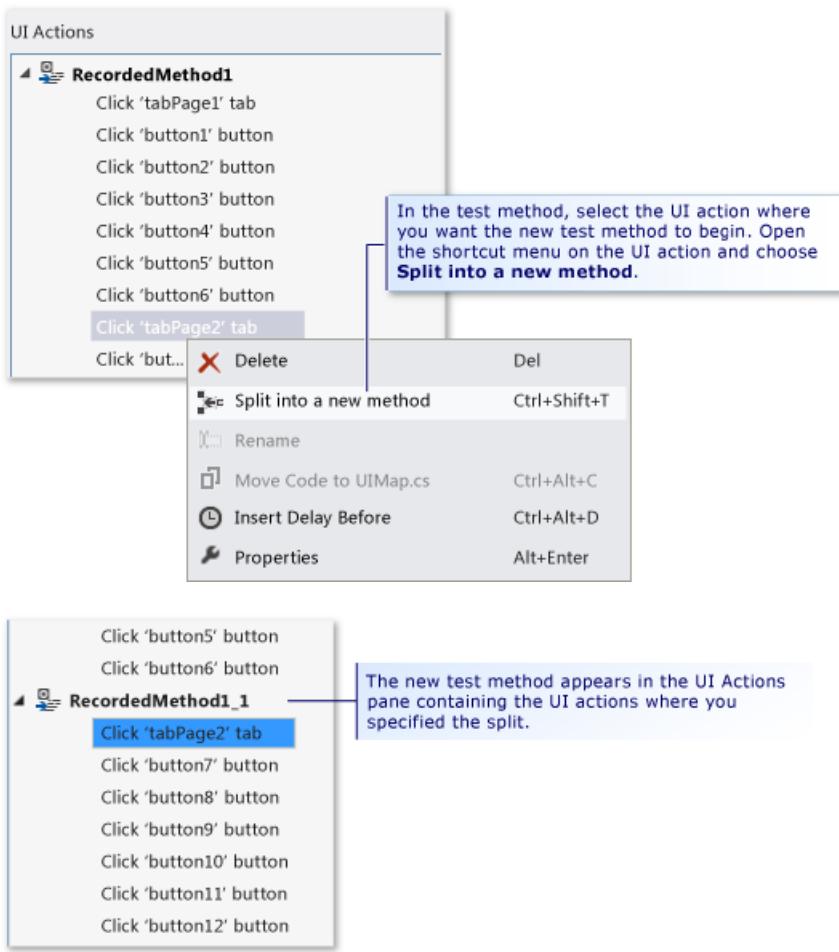
You can easily remove unwanted UI actions in your coded UI test.



In the **UI Action** pane, expand the test method that contains the UI action that you want to delete. Open the shortcut menu for the UI action and choose **Delete**.

Split a test method into two separate methods

You can split a test method to refine or to modularize the UI actions. For example, your test might have a single test method with UI actions in two container controls. The UI actions might be better modularized in two methods that correspond with one container.



In the **UI Action** pane, expand the test method that you want to split into two separate methods and select the UI action where you want the new test method to begin. Either open the shortcut menu for the UI action and then choose **Split into a new method**, or choose the **Split into a new method** button on the Coded UI Test Editor toolbar. The new test method appears in the **UI Actions** pane. It contains the UI actions starting from the action where you specified the split.

After you are done splitting the method, save the changes to the *UIMap.Designer* file by choosing **Save** on the Visual Studio toolbar.

WARNING

If you split a method, you must modify any code that calls the existing method to also call the new method you are about to create if you still want those UI actions included. When you split a method, a Microsoft Visual Studio dialog box is displayed. It warns you that you must modify any code that calls the existing method to also call the new method you are about to create. Choose **Yes**.

Tips

- To undo the split, choose **Undo** from the **Edit** menu, or press **Ctrl+Z**.
- You can rename the new method. Select it in the **UI Actions** pane and choose the **Rename** button in the Coded UI Test Editor toolbar.

-or-

Open the shortcut menu for the new test method and choose **Rename**.

A Microsoft Visual Studio dialog box is displayed. It warns you that you must modify any code that references the method. Choose **Yes**.

Move a test method to the *UIMap* file to facilitate customization

If you determine that one of your test methods in your coded UI test requires custom code, you must move it into either the *UIMap.cs* or *UIMap.vb* file. Otherwise, your code will be overwritten whenever the coded UI test is recompiled. If you do not move the method, your custom code will be overwritten each time the test is recompiled.

In the **UI Action** pane, select the test method that you want to move to the *UIMap.cs* or *UIMap.vb* file to facilitate custom code functionality that won't be overwritten when the test code is recompiled. Next, choose the **Move Code** button on the Coded UI Test Editor toolbar, or open the shortcut menu for the test method and choose **Move Code**. The test method is removed from the *UIMap.uitest* file and no longer is displayed in the **UI Actions** pane. To edit the test file that you moved, open the *UIMap.cs* or the *UIMap.vb* file from **Solution Explorer**.

After you are done moving the method, save the changes to the *UIMap.Designer* file by choosing **Save** on the Visual Studio toolbar.

WARNING

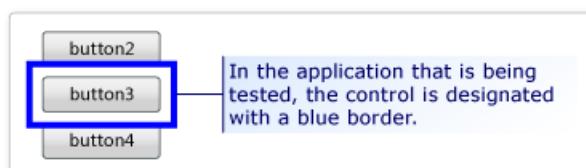
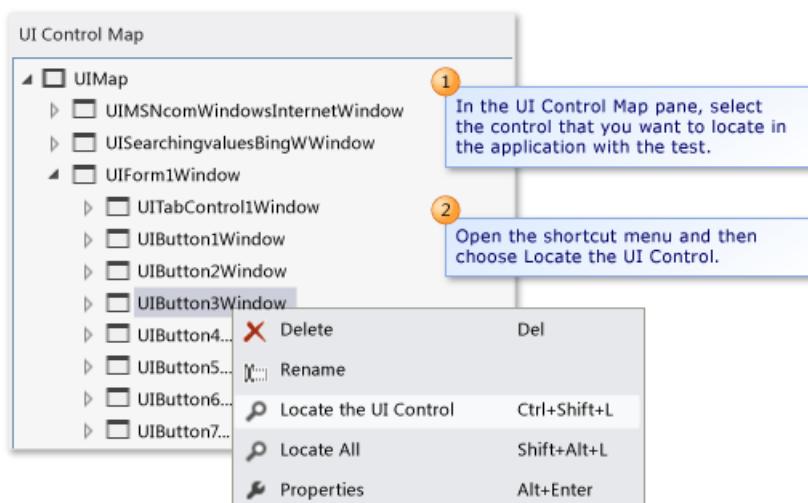
Once you have moved a method, you can no longer edit it using the Coded UI Test Editor. You must add your custom code and maintain it using the Code Editor. When you move a method, a Microsoft Visual Studio dialog box is displayed. It warns you that the method will be moved from the *UIMap.uitest* file to the *UIMap.cs* or *UIMap.vb* file and that you will no longer be able to edit the method using the Coded UI Test Editor. Choose **Yes**.

Tips

To undo the move, select **Undo** from the **Edit** menu, or press **Ctrl+Z**. However, you must then manually remove the code from the *UIMap.cs* or *UIMap.vb* file.

Locate a UI control in the application under test

Sometimes, it can be difficult to visualize where controls are located in the UI of the application under test. One of the capabilities of the coded UI Test Editor is that you can select a control listed in the UI control map and view its location in the application under test. Using the **Locate the UI Control** feature on the application under test can also be used to verify search property modifications you have made to a control.



In the **UI Control Map** pane, select the control that you want to locate in the application associated with the test.

Next, open the shortcut menu for the control and then choose **Locate the UI Control**. In the application that is being tested, the control is designated with a blue border.

NOTE

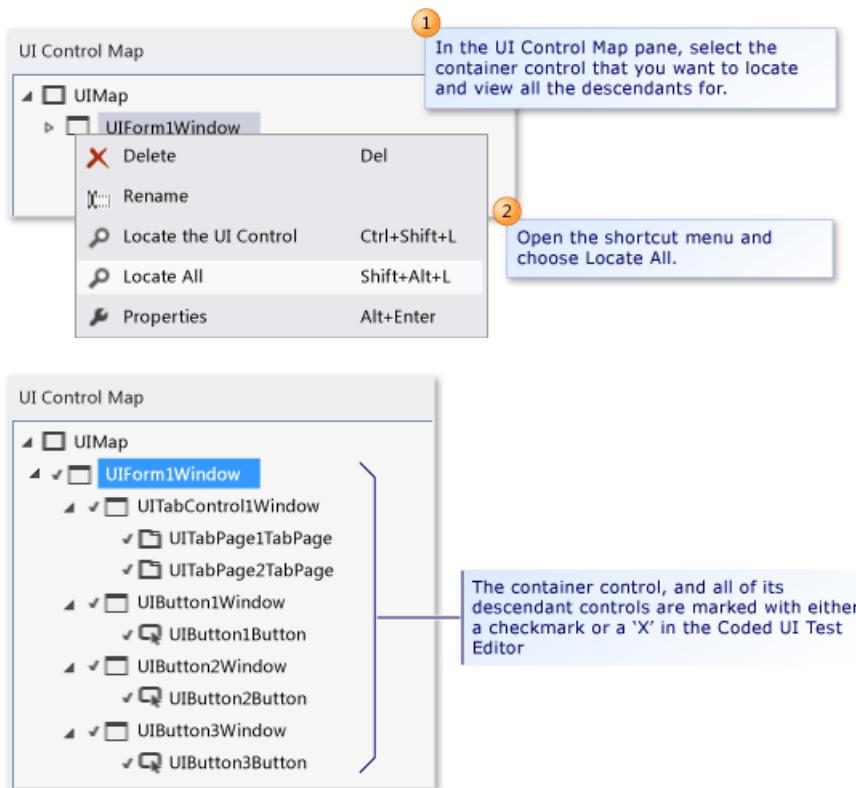
Before you locate a UI control, verify that the application associated with the test is running.

Tips

You can use the **Locate All** option to verify that all the controls under a container can be correctly located. This option is described in the next section.

Locate a control and its descendants

You can verify that all the controls under a container can be correctly located in the UI of the application under test. This can be helpful in verifying search property changes you may have made on the container. Additionally, if there have been significant changes in the UI of the application under test, you can validate that the existing control search properties are still correct.



In the **UI Control Map** pane, select the container control that you want to locate and view all the descendants for. Next, open the shortcut menu for the control and choose **Locate All**. The container control, and all its descendant controls, are marked in the Coded UI Test Editor with either a green check mark or a red 'X'. These marks let you know if the controls were successfully located in the application under test.

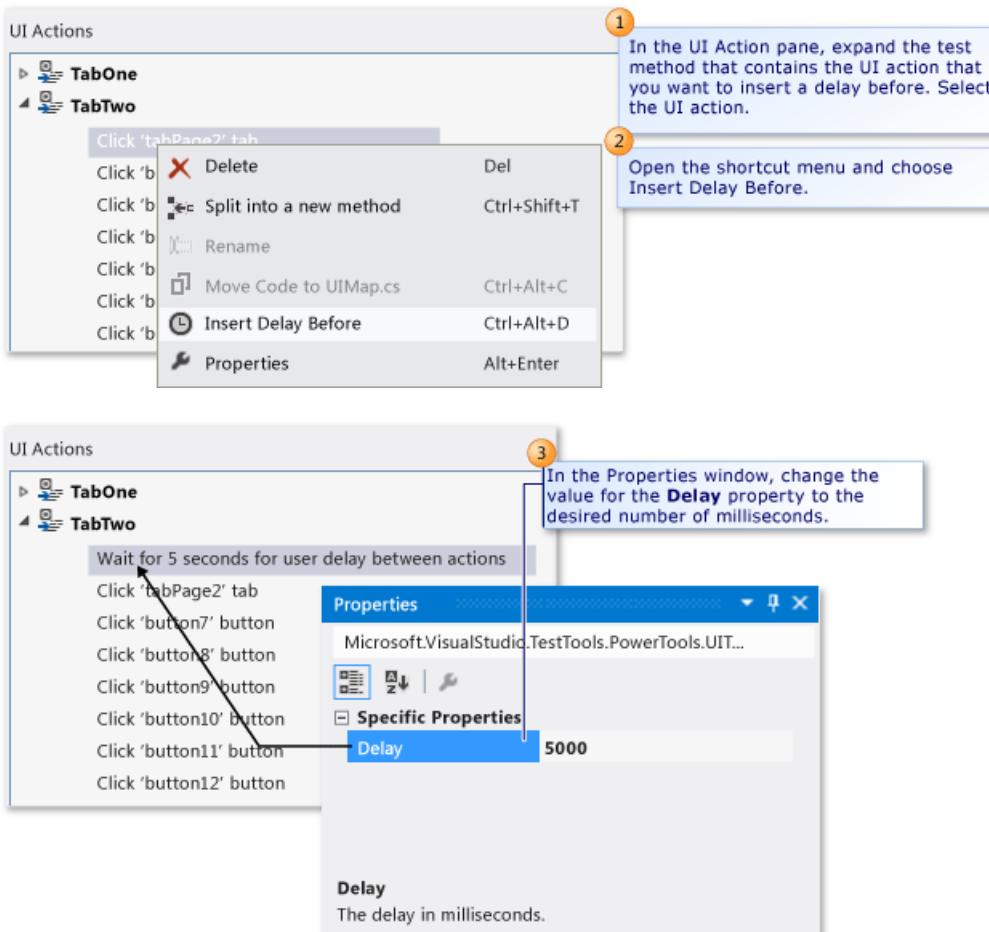
NOTE

Prior to locating the UI controls, verify that the application associated with the test is running.

Insert a delay before a UI action

Sometimes, you might want to make the test wait for certain events to occur, such as a window to appear, the progress bar to disappear, and so on. Using the Coded UI Test Editor, you can accomplish this by inserting a delay

before a UI action. You can specify how many seconds you want the delay to be.



In the **UI Action** pane, expand the test method that contains the UI action that you want to insert a delay before. Select the UI action. Next, open the shortcut menu for the UI action and choose **Insert Delay Before**. A delay is inserted and highlighted before the selected UI action with the following text: **Wait for 1 seconds for user delay between actions**. In the **Properties** window, change the value for the **Delay** property to the desired number of milliseconds.

After you are done inserting the delay, save the changes to the *UIMap.Designer* file by choosing **Save** on the Visual Studio toolbar.

If you need to ensure that a specific control is available before a UI action, you should consider adding custom code to your test method using the appropriate `UITestControl.WaitForControlXXX()` method. For more information, see [Making coded UI tests wait for specific events during playback](#).

See also

- [Use UI automation to test your code](#)
- [Create coded UI tests](#)
- [Create a data-driven coded UI test](#)
- [Walkthrough: Creating, editing and maintaining a coded UI test](#)

Analyzing coded UI tests using coded UI test logs

1/1/2020 • 2 minutes to read • [Edit Online](#)

Coded UI test logs filter and record important information about your coded UI test runs. The logs are presented in a format that allows for debugging issues quickly.

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Step 1: Enable logging

Depending on your scenario, use one of the following methods to enable the log:

- If there's no *App.config* file present in your test project:
 1. Determine which *QTAgent*.exe* process is launched when you run your test. One way to do this is to watch the **Details** tab in Windows **Task Manager**.
 2. Open the corresponding *.config* file from the *%ProgramFiles(x86)%\Microsoft Visual Studio\<version>\<edition>\Common7\IDE* folder. For example, if the process that runs is *QTAgent_40.exe*, open *QTAgent_40.exe.config*.
 3. Modify the value of **EqtTraceLevel** to the log level you want.

```
<!-- You must use integral values for "value".  
     Use 0 for off, 1 for error, 2 for warn, 3 for info, and 4 for verbose. -->  
<add name="EqtTraceLevel" value="4" />
```

4. Save the file.
- If there's an *App.config* file present in your test project:
 - Open the *App.config* file in the project, and add the following code under the configuration node:

```
<system.diagnostics>  
  <switches>  
    <add name="EqtTraceLevel" value="4" />  
  </switches>  
</system.diagnostics>
```

- Enable logging from the test code itself:

```
Microsoft.VisualStudio.TestTools.UnitTestingPlaybackSettings.LoggerOverrideState =  
  HtmlLoggerState.AllActionSnapshot;
```

Step 2: Run your coded UI test and view the log

When you run a coded UI test with the modifications to the *QTAgent*.exe.config* file in place, you see an output

link in the **Test Explorer** results. Log files are produced not only when your test fails but also for successful tests when the trace level is set to **verbose**.

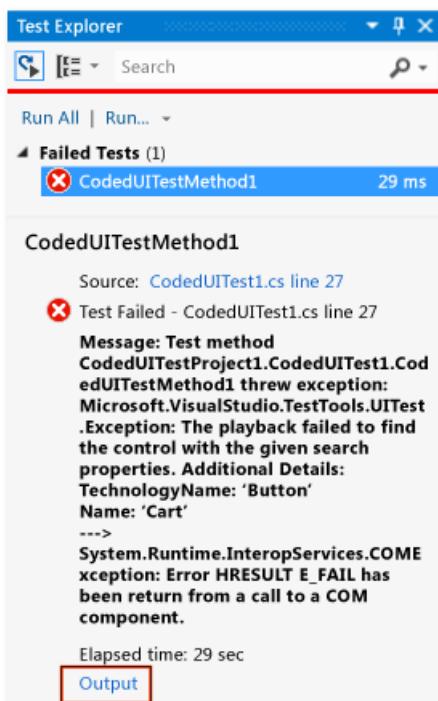
1. On the **Test** menu, choose **Windows** and then select **Test Explorer**.
2. On the **Build** menu, choose **Build Solution**.
3. In **Test Explorer**, select the coded UI test you want to run, open its shortcut menu, and then choose **Run Select Tests**.

The automated tests run and indicate if they passed or failed.

TIP

To view **Test Explorer**, choose **Test > Windows**, and then choose **Test Explorer**.

4. Choose the **Output** link in the **Test Explorer** results.



This displays the output for the test, which includes a link to the action log.

A screenshot of a web-based test result details page. It shows the following information:

- Test Name:** CodedUITestMethod1
- Test Result:** Failed (indicated by a red X icon)
- Message:** Test method CodedUITestProject1.CodedUITest1.CodedUITestMethod1 threw exception: Microsoft.VisualStudio.TestTools.UITest.Extension.UITestControlNotFoundException...
TechnologyName: 'MSAA'
ControlType: 'Button'
Name: 'Cart'
--->
System.Runtime.InteropServices.COMException: Error HRESULT E_FAIL has been returned from a call to a COM component.
- Attachments:** A link labeled 'UITestActionLog.html' is highlighted with a red box.

5. Choose the *UITestActionLog.html* link.

The log is displayed in your web browser.

Coded UI Test Log

TOTAL TIME: 0:20.146

▼ ✖ CodedUITestMethod1	0:20.146
▶ ✓ Click 'Update' button	0:09.969
▶ ✓ Click 'Submit' button	0:01.664
▼ ✖ Click 'Cart' 'Button'	0:07.933

✖ The playback failed to find the control with the given search properties. Additional Details:
TechnologyName: 'MSAA';
ControlType: 'Button' Name: 'Cart'



Microsoft.VisualStudio.TestTools.UITest.Extension.UITestCon...
at
Microsoft.VisualStudio.TestTools.UITesting.UITestLogGenerator.G...
(UITestControl control)

See also

- [Use UI automation to test your code](#)
- [How to: Run tests from Microsoft Visual Studio](#)

Anatomy of a coded UI test

1/1/2020 • 8 minutes to read • [Edit Online](#)

When you create a Coded UI Test in a coded UI test project, several files are added to the solution. This article provides information about the files.

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Contents of a coded UI test

When you create a Coded UI Test, the **Coded UI Test Builder** creates a map of the user interface under test, and also the test methods, parameters, and assertions for all tests. It also creates a class file for each test.

FILE	CONTENTS	EDITABLE?
UIMap.Designer.cs	Declarations section UIMap class (partial, auto-generated) Methods Properties	No
UIMap.cs	UIMap class (partial)	Yes
CodedUITest1.cs	CodedUITest1 class Methods Properties	Yes
UIMap.uitest	The XML map of the UI for the test.	No

UIMap.Designer.cs

This file contains code that is automatically created by the **Coded UI Test Builder** when a test is created. This file is re-created every time that a test changes, so that it is not a file in which you can add or modify code.

Declarations section

This section includes the following declarations for a Windows UI.

```
using System;
using System.CodeDom.Compiler;
using System.Collections.Generic;
using System.Drawing;
using System.Text.RegularExpressions;
using System.Windows.Input;
using Microsoft.VisualStudio.TestTools.UITest.Extension;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.VisualStudio.TestTools.UnitTesting.WinControls;
using Microsoft.VisualStudio.TestTools.UnitTesting.UnitTesting;
using Keyboard = Microsoft.VisualStudio.TestTools.UnitTesting.UnitTesting.Keyboard;
using Mouse = Microsoft.VisualStudio.TestTools.UnitTesting.UnitTesting.Mouse;
using MouseButtons = System.Windows.Forms.MouseButtons;
```

The [Microsoft.VisualStudio.TestTools.UnitTesting.WinControls](#) namespace is included for a Windows user interface (UI). For a web page UI, the namespace would be [Microsoft.VisualStudio.TestTools.UnitTesting.HtmlControls](#); for a Windows Presentation Foundation UI, the namespace would be [Microsoft.VisualStudio.TestTools.UnitTesting.WpfControls](#).

UIMap class

The next section of the file is the [UIMap](#) class.

```
[GeneratedCode("Coded UITest Builder", "10.0.21221.0")]
public partial class UIMap
```

The class code starts with a [GeneratedCodeAttribute](#) attribute that is applied to the class, which is declared as a partial class. Notice that the attribute is also applied to every class in this file. The other file that can contain more code for this class is *UIMap.cs*, which is discussed later.

The generated [UIMap](#) class includes code for each method that was specified when the test was recorded.

```
public void LaunchCalculator()
public void AddItems()
public void VerifyTotal()
public void CleanUp()
```

This part of the [UIMap](#) class also includes the generated code for each property that is required by the methods.

```
public virtual LaunchCalculatorParams LaunchCalculatorParams
public virtual AddItemsParams AddItemsParams
public virtual VerifyTotalExpectedValues VerifyTotalExpectedValues
public virtual CalculateItemsParams CalculateItemsParams
public virtual VerifyMathAppTotalExpectedValues
    VerifyMathAppTotalExpectedValues
public UIStartMenuWindow UIStartMenuWindow
public UIRunWindow UIRunWindow
public UICalculatorWindow UICalculatorWindow
public UIStartWindow UIStartWindow
public UIMathApplicationWindow UIMathApplicationWindow
```

UIMap methods

Each method has a structure that resembles the [AddItems\(\)](#) method. This is explained in more detail under the code, which is presented together with line breaks to add clarity.

```

/// <summary>
/// AddItems - Use 'AddItemsParams' to pass parameters into this method.
/// </summary>
public void AddItems()
{
    #region Variable Declarations
    WinControl uICalculatorDialog =
        this.UICalculatorWindow.UICalculatorDialog;
    WinEdit UIItemEdit =
        this.UICalculatorWindow.UIItemWindow.UIItemEdit;
    #endregion

    // Type '{NumPad7}' in 'Calculator' Dialog
    Keyboard.SendKeys(uICalculatorDialog,
        this.AddItemsParams.UICalculatorDialogSendKeys,
        ModifierKeys.None);

    // Type '{Add}{NumPad2}{Enter}' in 'Unknown Name' text box
    Keyboard.SendKeys(UIItemEdit,
        this.AddItemsParams.UIItemEditSendKeys,
        ModifierKeys.None);
}

```

The summary comment for each method definition tells which class to use for parameter values for that method. In this case, it is the `AddItemsParams` class, which is defined later in the `UIMap.cs` file, and which is also the value type that is returned by the `AddItemsParams` property.

At the top of the method code is a `Variable Declarations` region that defines local variables for the UI objects that are used by the method.

In this method, both `UIItemWindow` and `UIItemEdit` are properties that are accessed by using the `UICalculatorWindow` class, which is defined later in the `UIMap.cs` file.

Next are lines that send text from the keyboard to the Calculator application by using properties of the `AddItemsParams` object.

The `VerifyTotal()` method has a similar structure, and includes the following assertion code:

```

// Verify that 'Unknown Name' text box's property 'Text' equals '9. '
Assert.AreEqual(
    this.VerifyTotalExpectedValues.UIItemEditText,
    UIItemEdit.Text);

```

The text box name is listed as unknown because the developer of the Windows Calculator application did not provide a publicly available name for the control. The `Assert.AreEqual` method fails when the actual value is not equal to the expected value, which would cause the test to fail. Also notice that the expected value includes a decimal point that is followed by a space. If you ever have to modify the functionality of this particular test, you must allow for that decimal point and the space.

`UIMap` properties

The code for each property is also standard throughout the class. The following code for the `AddItemsParams` property is used in the `AddItems()` method.

```

public virtual AddItemsParams AddItemsParams
{
    get
    {
        if ((this.mAddItemsParams == null))
        {
            this.mAddItemsParams = new AddItemsParams();
        }
        return this.mAddItemsParams;
    }
}

```

Notice that the property uses a private local variable that is named `mAddItemsParams` to hold the value before it returns it. The property name and the class name for the object it returns are the same. The class is defined later in the `UIMap.cs` file.

Each class that is returned by a property is structured similarly. The following is the `AddItemsParams` class:

```

/// <summary>
/// Parameters to be passed into 'AddItems'
/// </summary>
[GeneratedCode("Coded UITest Builder", "10.0.21221.0")]
public class AddItemsParams
{
    #region Fields
    /// <summary>
    /// Type '{NumPad7}' in 'Calculator' Dialog
    /// </summary>
    public string UICalculatorDialogSendKeys = "{NumPad7}";

    /// <summary>
    /// Type '{Add}{NumPad2}{Enter}' in 'Unknown Name' text box
    /// </summary>
    public string UIItemEditSendKeys = "{Add}{NumPad2}{Enter}";
    #endregion
}

```

As with all classes in the `UIMap.cs` file, this class starts with the `GeneratedCodeAttribute`. In this small class is a `Fields` region that defines the strings to use as parameters for the `Keyboard.SendKeys` method that is used in the `UIMap.AddItems()` method that was discussed earlier. You can write code to replace the values in these string fields before the method in which these parameters are used is called.

UIMap.cs

By default, this file contains a partial `UIMap` class that has no methods or properties.

UIMap class

This is where you can create custom code to extend the functionality of the `UIMap` class. The code that you create in this file is not overwritten by the **Coded UI Test Builder** every time that a test is modified.

All parts of the `UIMap` can use the methods and properties from any other part of the `UIMap` class.

CodedUITest1.cs

This file is generated by the **Coded UI Test Builder**, but is not re-created every time that the test is modified, so that you can modify the code in this file. The name of the file is generated from the name that you specified for the test when you created it.

CodedUITest1 class

By default, this file contains the definition for only one class.

```
[CodedUITest]
public class CodedUITest1
```

The [CodedUITestAttribute](#) is automatically applied to the class, which allows the testing framework to recognize it as a testing extension. Also notice that this is not a partial class. All class code is contained in this file.

CodedUITest1 properties

The class contains two default properties that are located at the bottom of the file. Do not modify them.

```
/// <summary>
/// Gets or sets the test context which provides
/// information about and functionality for the current test run.
/// </summary>
public TestContext TestContext
public UIMap UIMap
```

CodedUITest1 methods

By default, the class contains only one method.

```
public void CodedUITestMethod1()
```

This method calls each [UIMap](#) method that you specified when you recorded your test, which is described in the section on the [UIMap Class](#).

A region that is titled [Additional test attributes](#), if uncommented, contains two optional methods.

```
// Use TestInitialize to run code before running each test
[TestInitialize()]
public void MyTestInitialize()
{
    // To generate code for this test, select "Generate Code for Coded
    // UI Test" from the shortcut menu and select one of the menu items.
    // For more information on generated code, see
    // http://go.microsoft.com/fwlink/?LinkId=179463

    // You could move this line from the CodedUITestMethod1() method
    this.UIMap.LaunchCalculator();
}

// Use TestCleanup to run code after each test has run
[TestCleanup()]
public void MyTestCleanup()
{
    // To generate code for this test, select "Generate Code for Coded
    // UI Test" from the shortcut menu and select one of the menu items.
    // For more information on generated code, see
    // http://go.microsoft.com/fwlink/?LinkId=179463

    // You could move this line from the CodedUITestMethod1() method
    this.UIMap.CloseCalculator();
}
```

The [MyTestInitialize\(\)](#) method has the [TestInitializeAttribute](#) applied to it, which tells the testing framework to call this method before any other test methods. Similarly, the [MyTestCleanup\(\)](#) method has the [TestCleanupAttribute](#) applied to it, which tells the testing framework to call this method after all other test methods have been called. Use of these methods is optional. For this test, the [UIMap.LaunchCalculator\(\)](#) method could be called from [MyTestInitialize\(\)](#) and the [UIMap.CloseCalculator\(\)](#) method could be called from [MyTestCleanup\(\)](#) instead of from [CodedUITest1Method1\(\)](#).

If you add more methods to this class by using the [CodedUITestAttribute](#), the testing framework calls each method as part of the test.

UIMap.uitest

This is an XML file that represents the structure of the coded UI test recording and all its parts. These include the actions and the classes in addition to the methods and properties of those classes. The [UIMap.Designer.cs](#) file contains the code that is generated by the Coded UI Builder to reproduce the structure of the test and provides the connection to the testing framework.

The *UIMap.uitest* file is not directly editable. However, you can use the Coded UI Builder to modify the test, which automatically modifies the *UIMap.uitest* file and the [UIMap.Designer.cs](#) file.

See also

- [UIMap](#)
- [Microsoft.VisualStudio.TestTools.UnitTesting.WinControls](#)
- [Microsoft.VisualStudio.TestTools.UnitTesting.HtmlControls](#)
- [Microsoft.VisualStudio.TestTools.UnitTesting.WpfControls](#)
- [GeneratedCodeAttribute](#)
- [Assert.AreEqual](#)
- [Keyboard.SendKeys](#)
- [CodedUITestAttribute](#)
- [TestInitializeAttribute](#)
- [TestCleanupAttribute](#)
- [Use UI automation to test your code](#)
- [Creating coded UI tests](#)
- [Best practices for coded UI tests](#)
- [Testing a large application with multiple UI maps](#)
- [Supported configurations and platforms for coded UI tests and action recordings](#)

Best practices for coded UI tests

1/1/2020 • 4 minutes to read • [Edit Online](#)

This topic describes some recommendations for developing coded UI tests.

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Best practices

Use the following guidelines to create a flexible coded UI test.

- Use the **Coded UI Test Builder** whenever possible.
- Do not modify the *UIMap.designer.cs* file directly. If you modify the file, the changes to the file will be overwritten.
- Create your test as a sequence of recorded methods. For more information about how to record a method, see [Creating coded UI tests](#).
- Each recorded method should act on a single page, form, or dialog box. Create a new test method for each new page, form, or dialog box.
- When you create a method, use a meaningful method name instead of the default name. A meaningful name helps identify the purpose of the method.
- When possible, limit each recorded method to fewer than 10 actions. This modular approach makes it easier to replace a method if the UI changes.
- Create each assertion using the **Coded UI Test Builder**, which automatically adds an assertion method to the *UIMap.Designer.cs* file.
- If the user interface (UI) changes, re-record the test methods, or the assertion methods, or re-record the affected sections of an existing test method.
- Create a separate [UIMap](#) file for each module in your application under test. For more information, see [Testing a large application with multiple UI maps](#).
- In the application under test, use meaningful names when you create the UI controls. Using meaningful names gives greater clarity and usability to the automatically generated control names.
- If you are creating assertions by coding with the API, create a method for each assertion in the part of the [UIMap](#) class that is in the *UIMap.cs* file. To execute the assertion, call this method from your test method.
- If you are directly coding with the API, use the properties and methods in the classes generated in the *UIMap.Designer.cs* file in your code as much as you can. These classes will make your work easier, more reliable, and will help you be more productive.

Coded UI tests automatically adapt to many changes in the user interface. If, for example, a UI element has changed position or color, most of the time the coded UI test will still find the correct element.

During a test run, the UI controls are located by the testing framework by using a set of search properties. The search properties are applied to each control class in the definitions created by the **Coded UI Test Builder** in the `UIMap.Designer.cs` file. The search properties contain name-value pairs of property names and property values that can be used to identify the control, such as the `FriendlyName`, `Name`, and `ControlType` properties of the control. If the search properties are unchanged, the coded UI test will successfully find the control in the UI. If the search properties are changed, coded UI tests have a smart match algorithm that applies heuristics to find controls and windows in the UI. When the UI has changed, you might be able to modify the search properties of previously identified elements to make sure that they are found.

If your user interface changes

User interfaces frequently change during development. Here are some ways to reduce the effect of these changes:

- Find the recorded method that references this control, and use the **Coded UI Test Builder** to re-record the actions for this method. You can use the same name for the method to overwrite the existing actions.
- If a control has an assertion that is no longer valid:
 - Delete the method that contains the assertion.
 - Remove the call to this method from the test method.
 - Add a new assertion by dragging the cross-hair button onto the UI control, open the UI map, and add the new assertion.

For more information about how to record coded UI tests, see [Use UI automation to test your code](#).

If a background process needs to complete before the test can continue

You might have to wait until a process finishes before you can continue with the next UI action. To do this you can use `WaitForReadyLevel` to wait before the test continues, as in the following example:

```
// Set the playback to wait for all threads to finish
Playback.PlaybackSettings.WaitForReadyLevel = WaitForReadyLevel.AllThreads;

// Press the submit button
this.UIMap.ClickSubmit();

// Reset the playback to wait only for the UI thread to finish
Playback.PlaybackSettings.WaitForReadyLevel = WaitForReadyLevel.UIThreadOnly;
```

See also

- [UIMap](#)
- [Microsoft.VisualStudio.TestTools.UnitTesting](#)
- [Use UI automation to test your code](#)
- [Creating coded UI tests](#)
- [Testing a large application with multiple UI maps](#)
- [Supported configurations and platforms for coded UI tests and action recordings](#)

Test a large application with multiple UI Maps

1/1/2020 • 4 minutes to read • [Edit Online](#)

This topic discusses how to use coded UI tests when you are testing a large application by using multiple UI Maps.

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Requirements

- Visual Studio Enterprise

When you create a new coded UI test, the Visual Studio testing framework generates code for the test by default in a [UI Map](#) class. For more information about how to record coded UI tests, see [Create coded UI tests](#) and [Anatomy of a coded UI test](#).

The generated code for the UI Map contains a class for each object that the test interacts with. For each generated method, a companion class for method parameters is generated specifically for that method. If there are a large number of objects, pages, and forms and controls in your application, the UI Map can grow very large. Also, if several people are working on tests, the application becomes unwieldy with a single large UI Map file.

Using multiple UI Map files can provide the following benefits:

- Each map can be associated with a logical subset of the application. This makes changes easier to manage.
- Each tester can work on a section of the application and check in their code without interfering with other testers working on other sections of the application.
- Additions to the application UI can be scaled incrementally with minimal effect on tests for other parts of the UI.

Do you need multiple UI Maps?

Create multiple UI Maps in each of these types of situations:

- Several complex sets of composite UI controls that together perform a logical operation, such as a registration page in a website, or the purchase page of a shopping cart.
- An independent set of controls that is accessed from various points of the application, such as a wizard with several pages of operations. If each page of a wizard is especially complex, you could create separate UI Maps for each page.

Add multiple UI Maps

To add a UI Map to your coded UI test project

1. In **Solution Explorer**, to create a folder in your coded UI test project to store all the UI Maps, right-click the coded UI test project file, point to **Add**, and then choose **New Folder**. For example, you could name it **UIMaps**.

The new folder is displayed under the coded UI test project.

2. Right-click the `UIMaps` folder, point to **Add**, and then choose **New Item**.

The **Add New Item** dialog box is displayed.

NOTE

You must be in a coded UI test project to add a new coded UI test map.

3. Select **Coded UI Test Map** from the list.

In the **Name** box, enter a name for the new UI Map. Use the name of the component or page that the map will represent, for example, `HomePageMap`.

4. Choose **Add**.

The Visual Studio window minimizes and the **Coded UI Test Builder** dialog box is displayed.

5. Record the actions for the first method and choose **Generate Code**.

6. After you have recorded all actions and assertions for the first component or page and grouped them into methods, close the **Coded UI Test Builder** dialog box.

7. Continue to create UI Maps. Record the actions and assertions, group them into methods for each component, and then generate the code.

In many cases, the top-level window of your application remains constant for all wizards, forms, and pages. Although each UI Map has a class for the top-level window, all maps are probably referring to the same top-level window within which all components of your application run. Coded UI tests search for controls hierarchically from the top down, starting from the top-level window, so in a complex application, the real top-level window could be duplicated in every UI Map. If the real top-level window is duplicated, multiple modifications will result if that window changes. This could cause performance problems when you switch between UI Maps.

To minimize this effect, you can use the `CopyFrom()` method to make sure that the new top-level window in that UI Map is the same as the main top-level window.

Example

The following example is part of a utility class that provides access to each component and their child controls, which are represented by the classes generated in the various UI Maps.

For this example, a web application named `Contoso` has a Home Page, a Product Page, and a Shopping Cart Page. Each of these pages shares a common top-level window, which is the browser window. There is a UI Map for each page and the utility class has code similar to the following:

```

using ContosoProject.UIMaps;
using ContosoProject.UIMaps.HomePageClasses;
using ContosoProject.UIMaps.ProductPageClasses;
using ContosoProject.UIMaps.ShoppingCartClasses;

namespace ContosoProject
{
    public class TestRunUtility
    {
        // Private fields for the properties
        private HomePage homepage = null;
        private ProductPage productPage = null;
        private ShoppingCart shoppingCart = null;

        public TestRunUtility()
        {
            homepage = new HomePage();
        }

        // Properties that get each UI Map
        public HomePage HomePage
        {
            get { return homepage; }
            set { homepage = value; }
        }

        // Gets the ProductPage from the ProductPageMap.
        public ProductPage ProductPageObject
        {
            get
            {
                if (productPage == null)
                {
                    // Instantiate a new page from the UI Map classes
                    productPage = new ProductPage();

                    // Since the Product Page and Home Page both use
                    // the same browser page as the top level window,
                    // get the top level window properties from the
                    // Home Page.
                    productPage.UIContosoFinalizeWindow.CopyFrom(
                        HomePage.UIContosoWindowsIWindow);
                }
                return productPage;
            }
        }

        // Continue to create properties for each page, getting the
        // page object from the corresponding UI Map and copying the
        // top level window properties from the Home Page.
    }
}

```

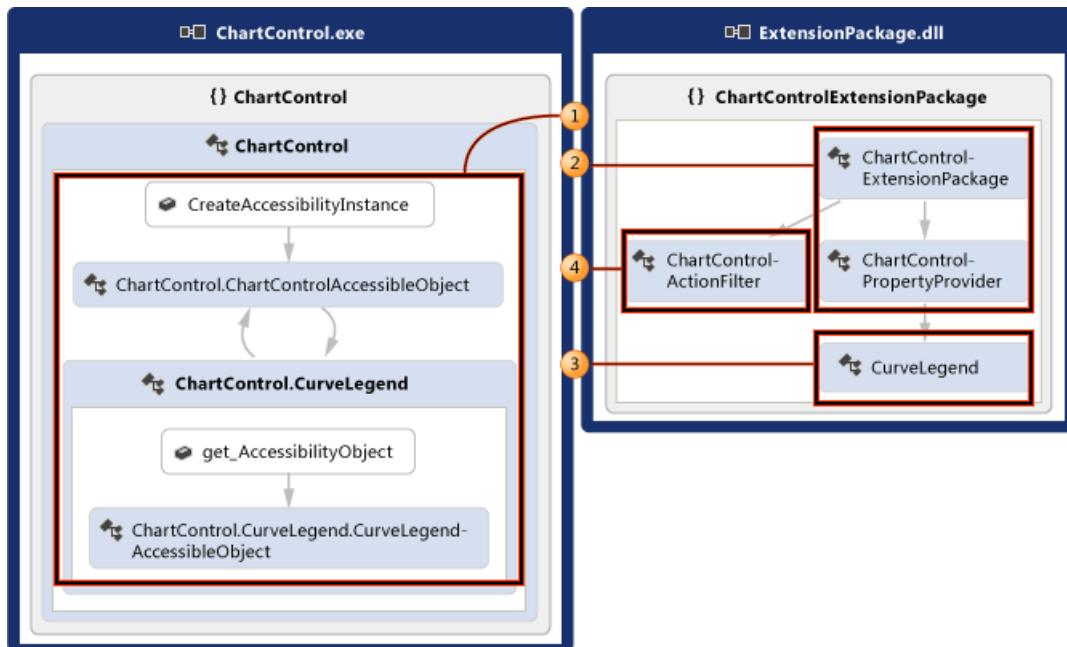
See also

- [UIMap](#)
- [CopyFrom](#)
- [Use UI automation to test your code](#)
- [Create coded UI tests](#)
- [Anatomy of a coded UI test](#)

Enable coded UI testing of your controls

1/1/2020 • 6 minutes to read • [Edit Online](#)

Implement support for the coded UI testing framework to make your control more testable. You can add increasing levels of support incrementally. Start by supporting record and playback and property validation. Then, build on that to enable the coded UI test builder to recognize your control's custom properties. Provide custom classes to access those properties from generated code. You can also help the coded UI test builder capture actions in a way that is closer to the intent of the action being recorded.

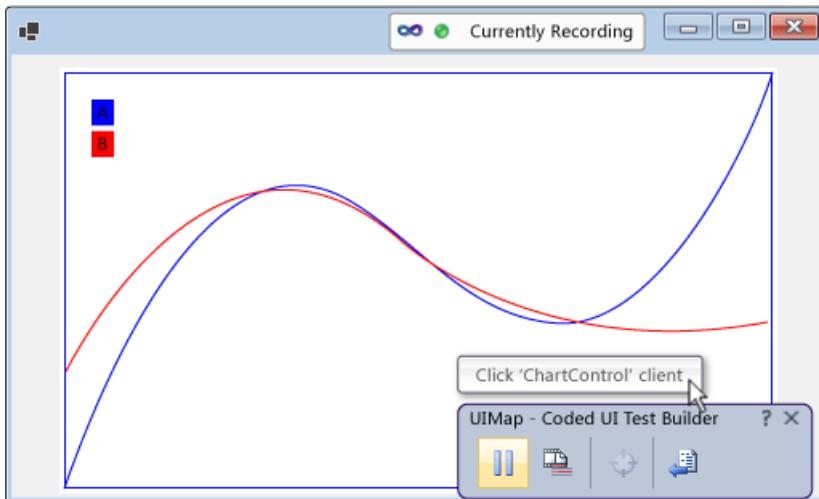


NOTE

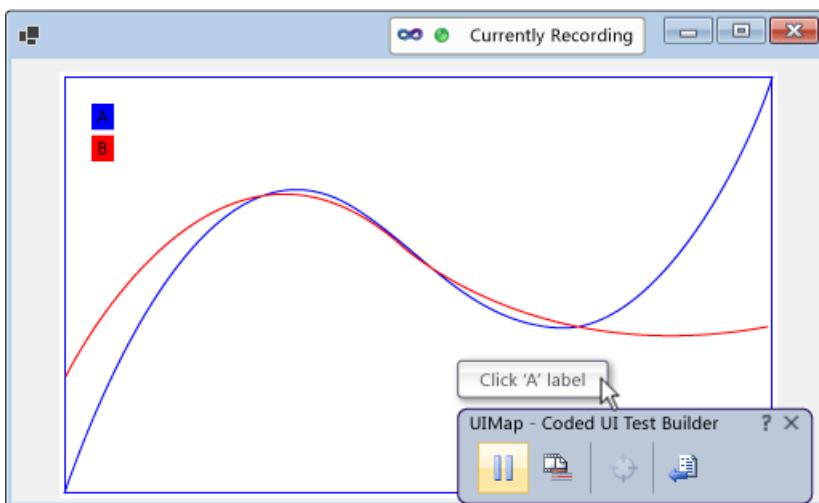
Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Support record and playback and property validation by implementing accessibility

The coded UI test builder captures information about the controls that it encounters during a recording and then generates code to replay that session. If your control doesn't support accessibility, then the coded UI test builder captures actions (like mouse clicks) using screen coordinates. When the test is played back, the generated code issues the actions in the same screen coordinates. If your control appears in a different place on the screen when the test is played back, the generated code will fail to perform the action. By not implementing accessibility for your control, you might see test failures if the test is played back on different screen configurations, in different environments, or when the UI layout changes.

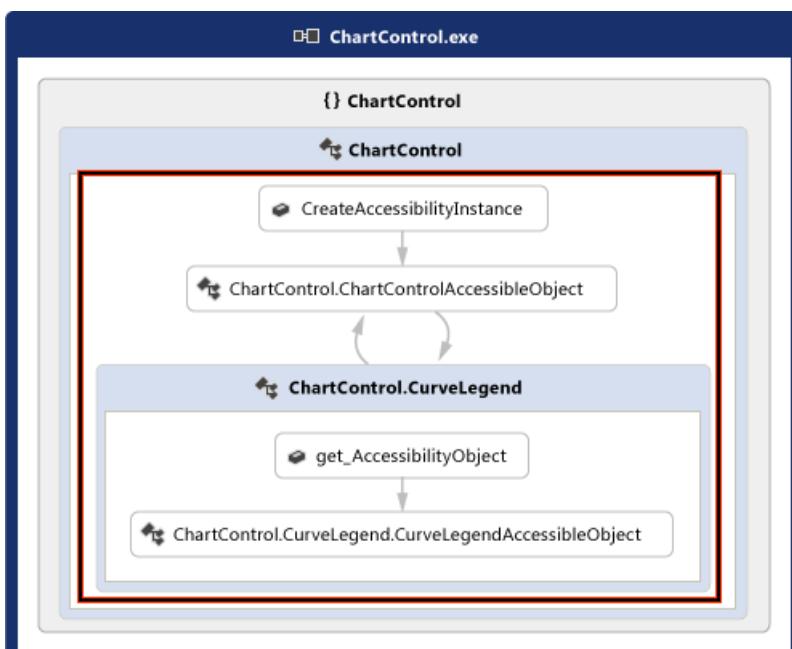


If you implement accessibility, the coded UI test builder uses that to capture information about your control when it records a test. Then, when you run the test, the generated code will replay those events against your control, even if it's somewhere else in the user interface. Test authors can also create asserts using the basic properties of your control.



To support record and playback, property validation, and navigation for a Windows Forms control

Implement accessibility for your control as outlined in the following procedure, and explained in detail in [AccessibleObject](#).



1. Implement a class that derives from [Control.ControlAccessibleObject](#), and override the [AccessibilityObject](#) property to return an object of your class.

```
public partial class ChartControl : UserControl
{
    // Overridden to return the custom AccessibleObject for the control.
    protected override AccessibleObject CreateAccessibilityInstance()
    {
        return new ChartControlAccessibleObject(this);
    }

    // Inner class ChartControlAccessibleObject represents accessible information
    // associated with the ChartControl and is used when recording tests.
    public class ChartControlAccessibleObject : ControlAccessibleObject
    {
        ChartControl myControl;
        public ChartControlAccessibleObject(ChartControl ctrl)
            : base(ctrl)
        {
            myControl = ctrl;
        }
    }
}
```

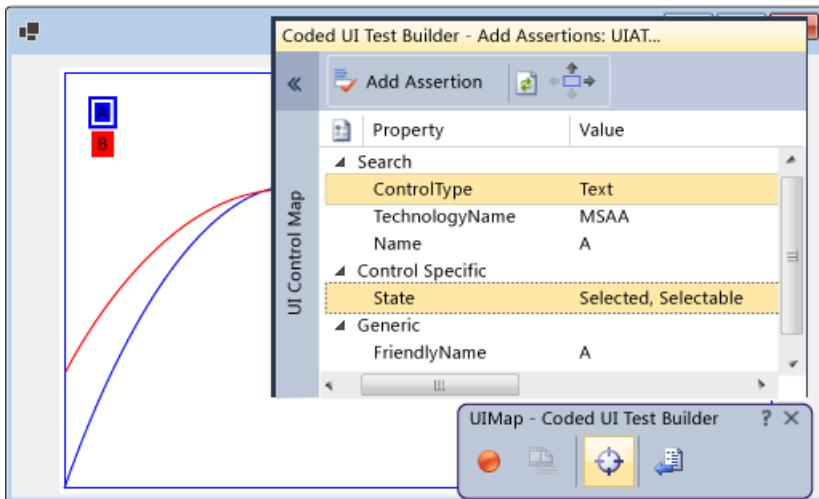
2. Override the accessible object's [Role](#), [State](#), [GetChild](#) and [GetChildCount](#) properties and methods.
3. Implement another accessibility object for the child control and override the child control's [AccessibilityObject](#) property to return the accessibility object.
4. Override the [Bounds](#), [Name](#), [Parent](#), [Role](#), [State](#), [Navigate](#), and [Select](#) properties and methods for the child control's accessibility object.

NOTE

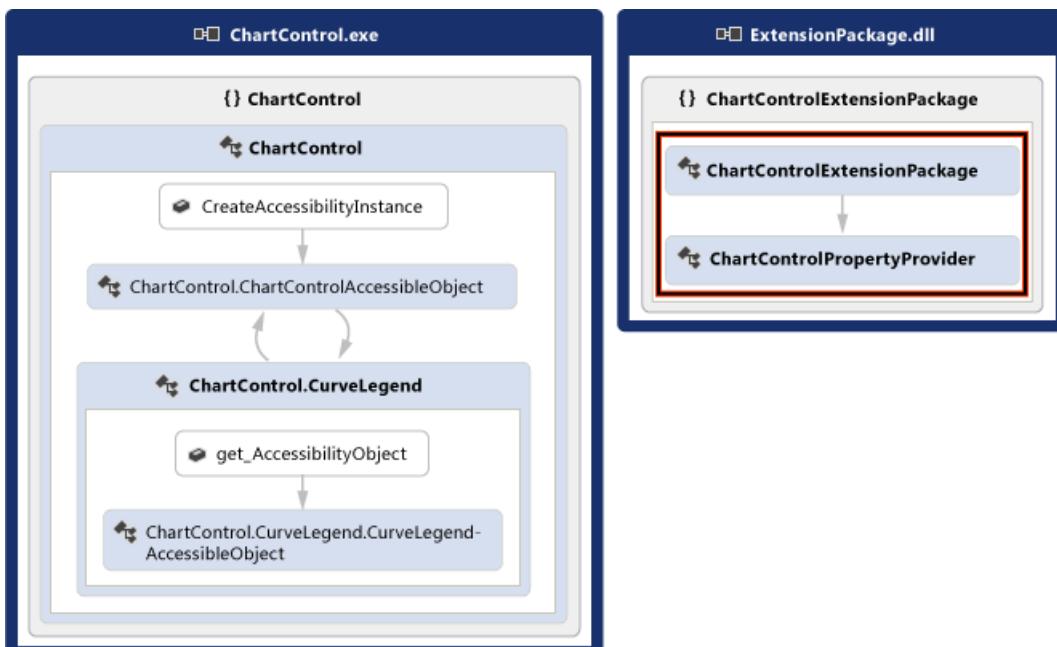
This topic starts with the accessibility sample in [AccessibleObject](#), and then builds on that sample in the remaining procedures. If you want to create a working version of the accessibility sample, create a console application and then replace the code in *Program.cs* with the sample code. Add references to Accessibility, System.Drawing, and System.Windows.Forms. Change the **Embed Interop Types** for Accessibility to **False** to eliminate a build warning. You can change the project's output type from **Console Application** to **Windows Application** so that a console window doesn't appear when you run the application.

Support custom property validation by implementing a property provider

After you implement basic support for record and playback and property validation, you can make your control's custom properties available to coded UI tests by implementing a [UITestPropertyProvider](#) plug-in. For example, the following procedure creates a property provider that allows coded UI tests to access the [State](#) property of the chart control's [CurveLegend](#) child controls:



To support custom property validation



1. Override the curve legend accessible object's **Description** property to pass rich property values in the description string. Separate multiple values with semicolons (;).

```

public class CurveLegendAccessibleObject : AccessibleObject
{
    // add the state property value to the description
    public override string Description
    {
        get
        {
            // Add ";" and the state value to the end
            // of the curve legend's description
            return "CurveLegend; " + State.ToString();
        }
    }
}

```

2. Create a UI test extension package for your control by creating a class library project. Add references to Accessibility, Microsoft.VisualStudio.TestTools.UITesting, Microsoft.VisualStudio.TestTools.UnitTesting, and Microsoft.VisualStudio.TestTools.Extension. Change the **Embed Interop Types** for Accessibility to **False**.
3. Add a property provider class that's derived from [UITestPropertyProvider](#):

```

using System;
using System.Collections.Generic;
using Accessibility;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.VisualStudio.TestTools.UnitTesting.Extension;
using Microsoft.VisualStudio.TestTools.UnitTesting.WinControls;
using Microsoft.VisualStudio.TestTools.UnitTesting.Common;

namespace ChartControlExtensionPackage
{
    public class ChartControlPropertyProvider : UITestPropertyProvider
    {
    }
}

```

4. Implement the property provider by placing property names and property descriptors in a [Dictionary< TKey, TValue >](#).
5. Override [UITestPropertyProvider.GetControlSupportLevel](#) to indicate that your assembly provides control-specific support for your control and its children.
6. Override the remaining abstract methods of [Microsoft.VisualStudio.TestTools.UnitTesting.UITestPropertyProvider](#)
7. Add an extension package class that's derived from [UITestExtensionPackage](#).
8. Define the `UITestExtensionPackage` attribute for the assembly.
9. In the extension package class, override [UITestExtensionPackage.GetService](#) to return the property provider class when a property provider is requested.
10. Override the remaining abstract methods and properties of [UITestExtensionPackage](#).
11. Build your binaries and copy them to `%ProgramFiles%\Common\Microsoft Shared\VSTT\10.0\UITestExtensionPackages`.

NOTE

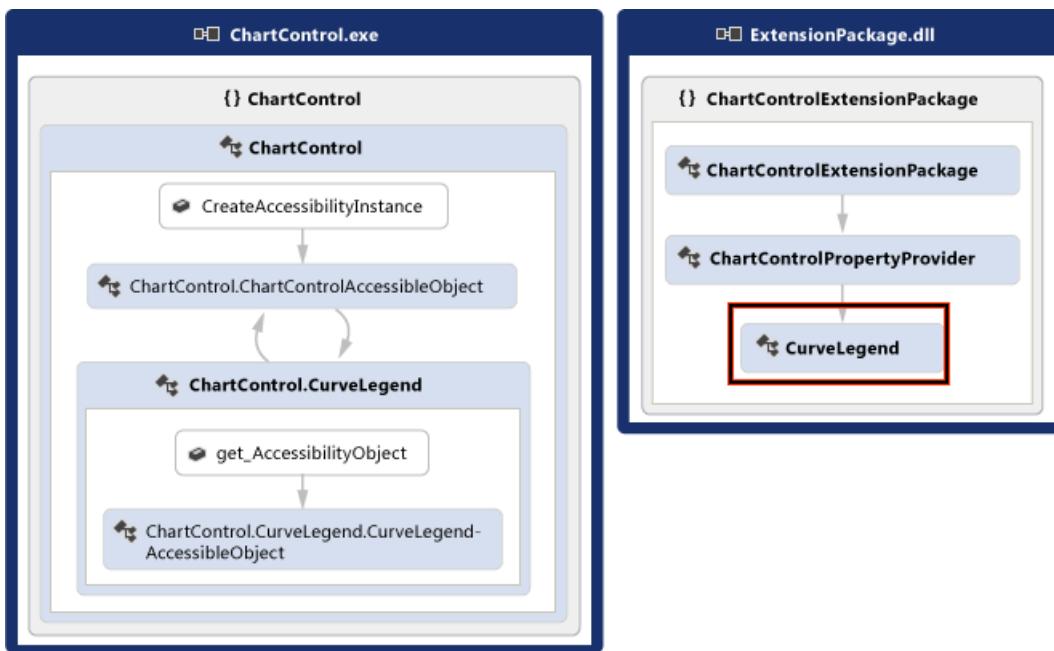
This extension package is applied to any control that is of type "Text". If you're testing multiple controls of the same type, test them separately so you can manage which extension packages are deployed when you record the tests.

Support code generation by implementing a class to access custom properties

When the coded UI test builder generates code from a session recording, it uses the [UITestControl](#) class to access your controls.

If you've implemented a property provider to provide access to your control's custom properties, you can add a specialized class that is used to access those properties. Adding a specialized class simplifies the generated code.

To add a specialized class to access your control

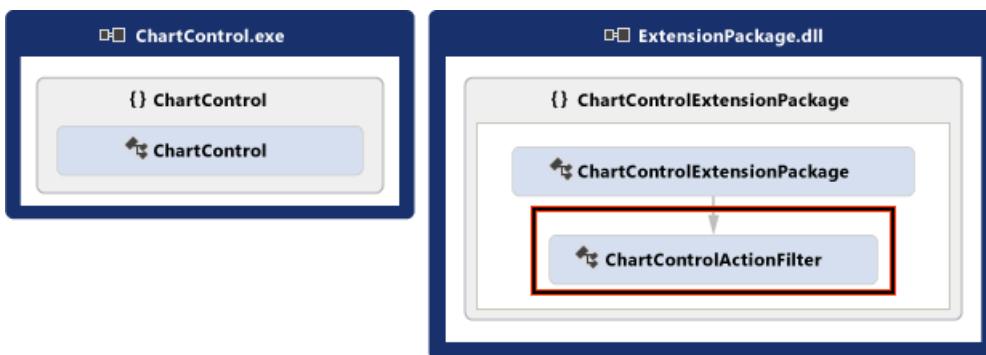


1. Implement a class that's derived from [WinControl](#) and add the control's type to the search properties collection in the constructor.
2. Implement your control's custom properties as properties of the class.
3. Override your property provider's [UITestPropertyProvider.GetSpecializedClass](#) method to return the type of the new class for the curve legend child controls.
4. Override your property provider's [GetPropertyNamesClassType](#) method to return the type of the new class' [PropertyNames](#) method.

Support intent-aware actions by implementing an action filter

When Visual Studio records a test, it captures each mouse and keyboard event. However, in some cases, the intent of the action can be lost in the series of mouse and keyboard events. For example, if your control supports autocomplete, the same set of mouse and keyboard events may result in a different value when the test is played back in a different environment. You can add an action filter plug-in that replaces the series of keyboard and mouse events with a single action. This way, you can replace the series of mouse and keyboard events that select a value with a single action that sets the value. Doing that protects coded UI tests from the differences in autocomplete from one environment to another.

To support intent-aware actions



1. Implement an action filter class that's derived from [UITestActionFilter](#), overriding the properties [ApplyTimeout](#), [Category](#), [Enabled](#), [FilterType](#), [Group](#) and [Name](#).
2. Override [ProcessRule](#). The example here replaces a double-click action with a single-click action.
3. Add the action filter to the [GetService](#) method of your extension package.

4. Build your binaries and copy them to %ProgramFiles%\Common Files\Microsoft Shared\VSTT\10.0\UITestExtensionPackages.

NOTE

The action filter does not depend on the accessibility implementation or on the property provider.

Debug your property provider or action filter

Your property provider and action filter are implemented in an extension package. The test builder runs the extension package in a separate process from your application.

To debug your property provider or action filter

1. Build the debug version of your extension package copy the .dll and .pdb files to %ProgramFiles%\Common Files\Microsoft Shared\VSTT\10.0\UITestExtensionPackages.
2. Run your application (not in the debugger).
3. Run the coded UI test builder.

```
codedUITestBuilder.exe /standalone
```
4. Attach the debugger to the codedUITestBuilder process.
5. Set breakpoints in your code.
6. In the coded UI test builder, create asserts to exercise your property provider, and record actions to exercise your action filters.

See also

- [AccessibleObject](#)
- [Use UI automation to test your code](#)

Supported configurations and platforms for coded UI tests and action recordings

1/10/2020 • 5 minutes to read • [Edit Online](#)

The supported configurations and platforms for coded UI tests for Visual Studio Enterprise are listed in the following table. These configurations also apply to action recordings created by using Test Runner.

NOTE

The coded UI test process must have the same privileges as the app under test.

NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Requirements

- Visual Studio Enterprise

Supported configurations

CONFIGURATION	SUPPORTED
Operating Systems	Windows 7 Windows Server 2008 R2 Windows 8 Windows 10
32-bit / 64-bit Support	32-bit Windows that is running 32-bit Microsoft Test Manager can test 32-bit applications. 64-bit Windows that is running 32-bit Microsoft Test Manager can test 32-bit WOW Applications that have UI Synchronization.n. 64-bit Windows that is running 32-bit Microsoft Test Manager can test 64-bit Windows Forms and WPF Applications that do not have UI Synchronization.
Architecture	x86 and x64 Note: Internet Explorer is not supported in 64-bit mode except when running under Windows 8 or later versions.

CONFIGURATION	SUPPORTED
.NET	.NET 2.0, 3.0, 3.5, 4 and 4.5. Note: Microsoft Test Manager and Visual Studio will both require .NET 4 to operate. However, applications developed by using the listed .NET versions are supported.

NOTE

UI Synchronization is a feature where the playback is verified in the message queue of each control. If a control did not respond to the event that was sent to it, then the event is sent again.

Platform support

PLATFORM	LEVEL OF SUPPORT
Windows Phone Apps	Only WinRT-XAML based Phone apps are supported.
UWP apps	Only XAML-based UWP apps are supported.
Universal Windows Apps	Only XAML-based Universal Windows Apps on Phone and Desktop are supported.
Edge	Recording of action steps or using the builder to view object properties is not supported. Tests can be played back on the Edge browser, using Visual Studio 2015 Update 2 and later versions by using the Coded UI cross browser testing extension .

PLATFORM	LEVEL OF SUPPORT
Internet Explorer 8	Fully supported.
Internet Explorer 9	<ul style="list-style-type: none"> - Support for HTML5 in Internet Explorer 9 and Internet Explorer 10: Coded UI tests support record, playback, and validation of the HTML5 controls: Audio, Video, ProgressBar and Slider. For more information, see Using HTML5 controls in coded UI tests. Warning: If you create a coded UI tests in Internet Explorer 10, it might not run using Internet Explorer 9 or Internet Explorer 8. This is because Internet Explorer 10 includes HTML5 controls such as Audio, Video, ProgressBar, and Slider. These HTML5 controls are not recognized by Internet Explorer 9, or Internet Explorer 8. Likewise, your coded UI test using Internet Explorer 9 might include some HTML5 controls that also will not be recognized by Internet Explorer 8.
Internet Explorer 10 Important: Internet Explorer 10 is only supported on the desktop.	
Internet Explorer 11 Important: Internet Explorer 11 is only supported on the desktop.	<ul style="list-style-type: none"> - Support for Internet Explorer 10 Spell Checking: Internet Explorer 10 includes spell checking capabilities for all text boxes. This allows you to choose from a list of suggested corrections. Coded UI Test will ignore user actions like selecting an alternative spelling suggestion. Only the final text typed into the text box will be recorded.
	<p>The following actions are recorded for coded UI test that use the spell checking control: Add to Dictionary, Copy, Select All, Add To Dictionary, and Ignore.</p>
	<ul style="list-style-type: none"> - Support for 64-bit Internet Explorer running under Windows 8: Previously, 64-bit versions of Internet Explorer were not supported for recording and playback. With Windows 8 and Visual Studio 2012, coded UI tests have been enabled for 64-bit versions of Internet Explorer.
	<p>Warning: 64-bit support for Internet Explorer applies only when you are running Windows 8 or later.</p>
	<ul style="list-style-type: none"> - Support for Pinned Sites in Internet Explorer 9: In Internet Explorer 9, pinned sites were introduced. With Pinned Sites, you can get to your favorite sites directly from the Windows taskbar—without having to open Internet Explorer first. Coded UI tests can now generate intent-aware actions on pinned sites. For more information about pinned sites, see Pinned sites.
	<ul style="list-style-type: none"> - Support for Internet Explorer 9 Semantic Tags: Internet Explorer 9 introduced the following semantic tags: section, nav, article, aside, hgroup, header, footer, figure, figcaption and mark. Coded UI tests ignore all of these semantic tags while recording. You can add assertions on these tags using the Coded UI Test Builder. You can use the navigation dial in the Coded UI Test Builder to navigate to any of these elements and view their properties.
	<ul style="list-style-type: none"> - Seamless Handling of White Space Characters between Versions of Internet Explorer: There are differences in the handling of white space characters between Internet Explorer 8, Internet Explorer 9, and Internet Explorer 10. Coded UI Test handles these differences seamlessly.
	<p>Therefore, a coded UI test created in Internet Explorer 8 for example, will play back successfully in Internet Explorer 9 and Internet Explorer 10.</p>
	<ul style="list-style-type: none"> - The Notification Area of Internet Explorer Are Now Recorded With the "Continue on Error" Attribute Set: All actions on the Notification Area of Internet Explorer are now recorded with the "Continue on Error" attribute set. If the notification bar does not appear during playback, the actions on it will be ignored and coded UI test will continue with the next action.

PLATFORM	LEVEL OF SUPPORT
Windows Forms and WPF third party controls	Fully supported. To enable third party controls in Windows Forms and WPF applications, you must add references and code. For more information, see Enable coded UI testing of your controls .
Internet Explorer 6	Not supported.
Internet Explorer 7	
Chrome	Recording of action steps is not supported. Coded UI Tests can be played back on Chrome and Firefox browsers with Visual Studio 2012 Update 4 or later. Go here for more details.
Firefox	
Opera	Not supported.
Safari	
Silverlight	Not supported. For Visual Studio 2013 however, you can download the Microsoft Visual Studio 2013 coded UI test plugin for Silverlight from the Visual Studio Gallery.
Flash/Java	Not supported.
Windows Forms 2.0 and later	Fully supported. Note: NetFx controls are fully supported, but not all third-party controls are supported.
WPF 3.5 and later	Fully supported. Note NetFx controls are fully supported, but not all third-party controls are supported.
Windows Win32	May work with some known issues, but not officially supported.
MFC	Partially supported. See the UITest framework for details of what features are supported.
SharePoint	Fully supported.
Office Client Applications	Not supported.
Dynamics CRM web client	Fully supported.
Dynamics (Ax) 2012 client	Action recording and playback are partially supported. See Visual Studio 10 coded UI / action recordings support for Microsoft Dynamics for details.
SAP	Not supported.

PLATFORM	LEVEL OF SUPPORT
Citrix/Terminal Services	We don't recommend recording actions on a terminal server. The recorder doesn't support running multiple instances at the same time.
PowerBuilder	Partially supported. The support is to the extent accessibility is enabled for PowerBuilder controls.

For information about how to create extensions to support other platforms, see [Enable coded UI testing of your controls](#) and [Extend coded UI tests and action recordings](#).

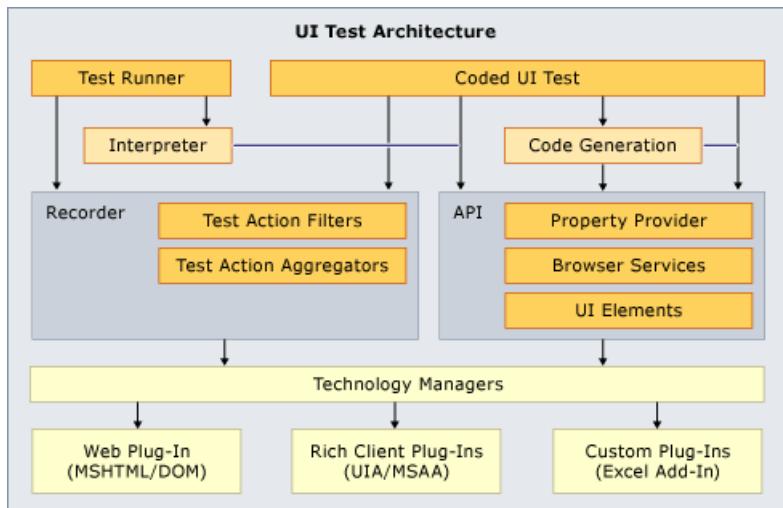
See also

- [Use UI automation to test your code](#)

Extend coded UI tests and action recordings

1/1/2020 • 2 minutes to read • [Edit Online](#)

The testing framework for coded UI tests and action recordings does not support every possible user interface. It might not support the specific UI that you want to test. For example, you cannot immediately create a coded UI test or an action recording for a Microsoft Excel spreadsheet. However, you can create your own extension to the coded UI test framework that supports your specific UI by taking advantage of the extensibility of the coded UI test framework.



NOTE

Coded UI Test for automated UI-driven functional testing is deprecated. Visual Studio 2019 is the last version where Coded UI Test will be available. We recommend using [Selenium](#) for testing web apps and [Appium with WinAppDriver](#) for testing desktop and UWP apps. Consider [Xamarin.UITest](#) for testing iOS and Android apps using the NUnit test framework.

Sample extension to test Microsoft Excel

This [blog post](#) contains a link to a [sample extension](#) for the coded UI test framework. You can also view the entire [blog post series for coded UI test extensibility](#).

NOTE

The sample is intended for use with Microsoft Excel 2010. It may or may not work with other versions of Excel.

See also

- [UITestPropertyProvider](#)
- [UITechnologyElement](#)
- [UITestActionFilter](#)
- [UITestExtensionPackage](#)
- [Use UI automation to test your code](#)
- [Best practices for coded UI tests](#)
- [Supported configurations and platforms for coded UI tests and action recordings](#)

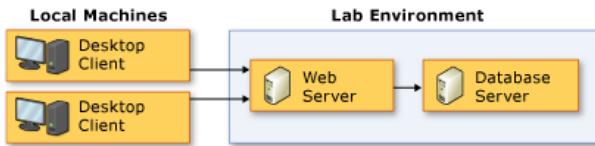
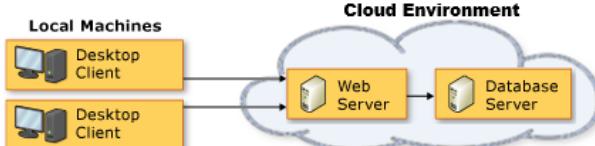
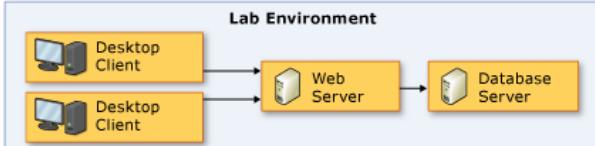
Use a lab environment for your devops

1/10/2020 • 7 minutes to read • [Edit Online](#)

A lab environment is a collection of virtual and physical machines that you can use to develop and test applications. A lab environment can contain multiple roles needed to test multi-tiered applications, such as workstations, web servers, and database servers. In addition, you can use a build-deploy-test workflow with your lab environment to automate the process of building, deploying, and running automated tests on your application.

- **Use a test plan to run automated tests** - You can run a collection of automated tests, called a *test plan*, and view the progress.
- **Use a build-deploy-test workflow** - You can use a build-deploy-test workflow to test multi-tiered applications automatically. A typical example is a workflow that starts a build, deploys the build files onto the appropriate machines in a lab environment, and then performs automated tests. In addition, you can schedule your workflow to run at specific intervals.
- **Collect diagnostic data from all machines, even during manual testing** - You can collect diagnostic data from multiple machines simultaneously. For example, during a single test run, you can collect IntelliTrace, test impact, and other forms of data from a web server, a database server, and a client.

Here are examples of common lab environment topologies:

TOPOLOGY	DESCRIPTION
	This lab environment has a <i>server topology</i> , which is often used to run manual tests on server applications, and which allows testers to use their own client machines to verify bugs in the environment. In a backend topology, your lab environment contains only servers. When you use this type of topology, you typically connect to the servers in the lab environment using a client machine that is not part the environment.
	This lab environment provides similar capabilities and features as the <i>server topology</i> , but removes the requirement for physical or virtual machines running in a local environment; which can reduce setup time, simplify maintenance, and minimize cost. Setting up multiple websites and virtual machines, together with custom networking, is quick and easy in a cloud environment such as Microsoft Azure.
	This lab environment has a <i>client-server topology</i> , which is often used to test an application that has server and client components. In a client/server topology, all of the client and server machines used to test your application are in your lab environment. When you use this topology, you can collect test data from every machine that impacts your tests.
	Watch a video on managing lab environments for testing.

Use the cloud with Azure Pipelines or Team Foundation Server Build

and Release

You can perform automated testing and build-deploy-test automation using the [build and release](#) features in Team Foundation Server (TFS) and Azure Test Plans. Some of the benefits are:

- You do not need a Build controller or Test controller.
- The Test agent is installed through a task as part of the build or release.
- It is easy to customize the deployment steps. You are no longer restricted to use a single script. You can also take advantage of the many tasks that are available in the product as well as in Visual Studio Marketplace.
- You don't have to maintain test suites. You can directly run tests from binaries.
- You get a richer inline reporting experience for the tests that run within each build or release.
- You can track which assets (release, build, work items, commits) are currently deployed and tested on each environment.
- You can customize and extend the automation to easily deploy to multiple test environments, and even to production.
- You can schedule the automation to happen whenever there is a check-in or commit, or at a specific time every day.

For more information, see [Use Build or Release management](#).

Use the Visual Studio Lab Management features of Microsoft Test Manager

You can create and manage lab environments with the Visual Studio Lab Management features of Microsoft Test Manager when you use Visual Studio Enterprise edition.

Lab Management automatically installs test agents on every machine in your environment.

If you use Lab Management in conjunction with System Center Virtual Machine Manager (SCVMM), you also get these benefits when you use lab environments:

- **Quickly reproduce machine configurations** - You can store collections of virtual machines that are configured to recreate typical production environments. You can then perform each test run on a new copy of a stored environment.
- **Reproduce the exact conditions of a bug** - When a test run fails, you can store a copy of the state of your lab environment and access it from your build results or a work item.
- **Run multiple copies of a lab environment at the same time** - You can run multiple copies of your lab environment at the same time without naming conflicts.

Standard Environments and SCVMM Environments

There are two types of lab environments that you can create with Visual Studio Lab Management: **standard environments** and **SCVMM environments**. However, the capabilities of each type of environment are different.

Standard environments: can contain a mix of virtual and physical machines. You can also add virtual machines to a standard environment that are managed by third-party virtualization frameworks. In addition, standard environments do not require additional server resources such as an SCVMM server.

SCVMM environments: can only contain virtual machines that are managed by SCVMM (System Center Virtual Machine Manager), so the virtual machines in SCVMM environments can only run on the Hyper-V virtualization framework. However, SCVMM environments provide the following automation and management features that are not available in standard environments:

- **Environment snapshots:** Environment snapshots contain the state of a lab environment, so you can quickly restore a clean environment, or save the state of an environment that has been modified. You can

also use a build-deploy-test workflow to automate the process of saving and restoring environment snapshots.

- **Stored environments:** You can store a copy of an SCVMM environment, and then deploy multiple copies of that environment.
- **Network isolation:** Network isolation allows you to simultaneously run multiple identical copies of an SCVMM environment without computer name conflicts.
- **Virtual machine templates:** A virtual machine template is a virtual machine that has had its name and other identifiers removed. When a VM template is deployed in an SCVMM environment, Microsoft Test Manager generates new identifiers. This allows you deploy multiple copies of a virtual machine in the same environment, or multiple environments, and then run the virtual machines simultaneously.
- **Stored Virtual Machines:** A virtual machine that is stored in your project library and includes unique identifiers.

NOTE

Lab Management does not support SCVMM 2016.

For information about SCVMM, see [Virtual Machine Manager](#).

Standard environments and SCVMM environments support many of the same features. However, there are some important differences to consider. The following table compares the features that are available for standard environments and SCVMM environments.

CAPABILITY	SCVMM ENVIRONMENTS	STANDARD ENVIRONMENTS
Testing		
Run manual tests	Supported	Supported
Run coded UI and other automated tests	Supported	Supported
File rich bugs using diagnostic adapters	Supported	Supported
Build deployment		
Automatic build-deploy-test workflows	Supported	Supported
Environment creation and management		
Use physical machines in addition to virtual machines	Not supported	Supported
Use third-party virtual machines	Not supported	Supported
Automatically install test agents onto machines in the lab environment	Supported	Supported

CAPABILITY	SCVMM ENVIRONMENTS	STANDARD ENVIRONMENTS
Save and deploy the state of a lab environment using environment snapshots	Supported	Not supported
Create lab environments from VM templates	Supported	Not supported
Start/stop/snapshot environment	Supported	Not supported
Connect to the environment using Environment Viewer	Supported	Supported
Run multiple copies of an environment at the same time using network isolation	Supported	Not supported

Lab management concepts

Here are some additional concepts that you should be familiar with before you continue:

TERM	DESCRIPTION
Lab Center	The area of Microsoft Test Manager where you create and manage lab environments.
Azure DevOps Project Lab	The collection of lab environments that have been set up so you can connect to them and run their virtual machines.
Azure DevOps Project Library	An archive of stored virtual machines, templates, and stored lab environments that have been imported into the host group of your project. You can use the items in your library with SCVMM environments; however, you can't add them directly to a standard environment. You can't run the items in your library; instead you use them to deploy a new environment.
Deployed Environment	A lab environment that has been deployed to your project lab so that you can connect to it and run its machines.

For more information about lab management, see:

- [Plan your lab](#)
- [Administer your lab](#)
- [Set up for SCVMM environments](#)
- [Manage permissions](#)
- [Change setup](#)
- [Troubleshooting](#)

For information about setting up environments, see:

- [Build and release cloud environments](#)
- [Standard lab environments](#)
- [SCVMM \(virtual\) environments](#)
- [Creating and using a network isolated environment](#)

See also

- [Install and configure test agents](#)
- [Visual Studio Lab Management Guide](#)
- [Microsoft DevOps Blog](#)

Use Azure Test Plans instead of Lab Management for automated testing

1/1/2020 • 5 minutes to read • [Edit Online](#)

If you use Microsoft Test Manager and Lab Management for automated testing or for build-deploy-test automation, this topic explains how you can achieve the same goals using the [build and release](#) features in Azure Pipelines and Team Foundation Server (TFS).

Build-deploy-test automation

Microsoft Test Manager and Lab Management rely on a XAML build definition to automate build, deployment, and testing of your applications. The XAML build relies on various constructs created in Microsoft Test Manager such as a lab environment, test suites, and testing settings, and on various infrastructure components such as a Build controller, Build agents, Test controller, and Test agents to achieve this goal. You can accomplish the same with fewer steps using Azure Pipelines or TFS.

STEPS	WITH XAML BUILD	IN A BUILD OR RELEASE
Identify the machines to deploy the build to and run tests.	Create a standard lab environment in Microsoft Test Manager with those machines.	n/a
Identify the tests to be run.	Create a test suite in Microsoft Test Manager, create test cases, and associate automation with each test case. Create test settings in Microsoft Test Manager identifying the role of machines in the lab environment in which tests should be run.	Create automated test suite in Microsoft Test Manager in the same manner if you plan to manage your testing through test plans. Alternatively, you can skip this if you want to run tests directly from test binaries produced by your builds. There is no need to create test settings in either case.
Automate deployment and testing.	Create a XAML build definition using LabDefaultTemplate.*.xaml. Specify the build, test suites, and lab environment in the build definition.	Create a build or release pipeline with a single environment. Run the same deployment script (from the XAML build definition) using the Command line task, and run automated tests using Test Agent Deployment and Run Functional Tests tasks. Specify the list of machines and their credentials as inputs to these tasks.

Some of the benefits of using Azure Pipelines or TFS for this scenario are:

- You do not need a Build controller or Test controller.
- The Test agent is installed through a task as part of the build or release.
- It is easy to customize the deployment steps. You are no longer restricted to use a single script. You can also take advantage of the many tasks that are available in the product as well as in Visual Studio Marketplace.
- You do not have to maintain test suites. You can directly run tests from binaries.
- You get a richer inline reporting experience for the tests that ran within each build or release.
- You can track which assets (release, build, work items, commits) are currently deployed and tested on each environment.

- You can customize and extend the automation to easily deploy to multiple test environments, and even to production.
- You can schedule the automation to happen whenever there is a check-in or commit, or at a specific time every day.

Self-service management of SCVMM environments

The [Test Center in Microsoft Test Manager](#) supports the ability to manage a library of environment templates as well as provision environments on demand using an [SCVMM server](#).

The self-service provisioning features of Lab Center have two distinct goals:

- Provide a simpler way to manage the infrastructure. Managing VM and environment templates and automatically creating private networks to isolate clones of environments from each other were examples of infrastructure management.
- Provide a simpler way for teams to consume the virtual machines in their test and deployment activities. Making lab environments accessible through the same project security model, and integrated use of those virtual machines in test scenarios were examples of easy consumption.

However, given the evolution of richer public and private cloud management systems such as [Microsoft Azure](#) and [Microsoft Azure Stack](#), there is no evolution of infrastructure management features in TFS 2017 and beyond. Instead, the focus on easy consumption of resources managed through such cloud infrastructures continues.

The following table summarizes the typical activities you perform in Lab Center, and how you can accomplish them through SCVMM or Azure (if they are infrastructure management activities) or through TFS and Azure DevOps Services (if they are test or deployment activities):

STEPS	WITH LAB CENTER	IN A BUILD OR RELEASE
Manage a library of environment templates.	Create a lab environment. Install necessary software on the virtual machines. Sysprep and store the environment as a template in library.	Use SCVMM administration console directly to create and manage either virtual machine templates or service templates. When using Azure, select one of the Azure quickstart templates .
Create a lab environment.	Select an environment template in the library and deploy it. Provide the necessary parameters to customize the virtual machine configurations.	Use SCVMM administration console directly to create VMs or service instances from templates. Use Azure portal directly to create resources. Or, create a release definition with an environment. Use the Azure tasks or tasks from the SCVMM Integration extension to create new virtual machines. Creating a new release of this definition is equivalent to creating a new environment in Lab Center.
Connect to machines.	Open the lab environment in Environment viewer.	Use SCVMM administration console directly to connect to the virtual machines. Alternatively, use the IP address or DNS names of the virtual machines to open remote desktop sessions.

Steps	With Lab Center	In a Build or Release
Take a checkpoint of an environment, or restore an environment to a clean checkpoint.	Open the lab environment in Environment viewer. Select the option to take a checkpoint or to restore to a previous checkpoint.	Use SCVMM administration console directly to perform these operations on virtual machines. Or, to perform these steps as part of a larger automation, include the checkpoint tasks from the SCVMM Integration extension as part of the environment in a release definition.

Create network-isolated environments

A network isolated lab environment is a group of SCVMM virtual machines that can be cloned safely without causing network conflicts. This was done in MTM using a series of instructions that used a set of network interface cards to configure the virtual machines in a private network, and another set of network interface cards to configure the virtual machines in a public network.

However, Azure Pipelines and TFS, in conjunction with the SCVMM build and deploy task, can be used to manage SCVMM environments, provision isolated virtual networks, and implement build-deploy-test scenarios. For example, you can use the task to:

- Create, restore, and delete checkpoints
- Create new virtual machines using a template
- Start and stop virtual machines
- Run custom PowerShell scripts for SCVMM

For more information, see [Create a virtual network isolated environment for build-deploy-test scenarios](#).

Install test agents and test controllers

1/1/2020 • 2 minutes to read • [Edit Online](#)

For test scenarios that use Visual Studio and Azure Test Plans or Team Foundation Server (TFS), you don't need a test controller. Agents for Visual Studio handle orchestration by communicating with Azure Test Plans or TFS. A scenario could be that you run continuous tests for build and release workflows in Azure Test Plans or TFS.

You might also consider if it's better to use [build or release management](#) instead of lab management.

System requirements

The following table shows the system requirements for installing the test agent or test controller for Visual Studio:

ITEM	REQUIREMENTS
Agent	Windows 10 Windows 8, Windows 8.1 Windows 7 Service Pack 1 Windows Server 2016 Standard and Datacenter Windows Server 2012 R2
Controller	Windows 10 Windows 8, Windows 8.1 Windows 7 Service Pack 1 Windows Server 2016 Standard and Datacenter Windows Server 2012 R2
.NET Framework	.NET Framework 4.5

Install the test controller and test agents

You can download agents for Visual Studio from [visualstudio.microsoft.com](#). Look for *Agents for Visual Studio 2019*, select either *Agent* or *Controller*, and then choose *Download*. Run the downloaded executable to install the test agent or controller.

You can download agents for Visual Studio 2017, Visual Studio 2015, and Visual Studio 2013 from the [older downloads](#) page.

These installers are available as ISO files for easy installation on virtual machines.

Compatible versions of TFS, Microsoft Test Manager, the test controller, and test agent

You can mix different versions of TFS, Microsoft Test Manager, the test controller, and the test agent, according to the following table:

TFS	MICROSOFT TEST MANAGER WITH LAB CENTER	CONTROLLER	AGENT
2017: upgrade from 2015 or new install	2017	2017	2017
2017: upgrade from 2015 or new install	2017	2013 Update 5	2013 Update 5
2017: upgrade from 2015 or new install	2015	2013 Update 5	2013 Update 5
2015: upgrade from 2013	2013	2013	2013
2015: new install	2013	2013	2013
2015: upgrade from 2013 or new install	2015	2013	2013
2013	2015	2013	2013

NOTE

Lab management scenarios in TFS 2018 and Azure DevOps Services are deprecated. For more information see [TFS 2018 Release Notes](#).

Upgrade from Visual Studio 2013 test agents

We recommend that you use agents for Visual Studio in all new automated testing scenarios. You can use the *Deploy Test Agents* task in a build pipeline to download and install the test agents on your machine.

The following table shows the scenarios supported by Agents for Visual Studio 2013, and the alternatives for Team Foundation Server (TFS) 2015 and Azure Test Plans:

SCENARIOS SUPPORTED BY AGENTS FOR VISUAL STUDIO 2013	ALTERNATIVE IN TFS AND AZURE TEST PLANS
Build-Deploy-Test workflow in Visual Studio	Users can use a build pipeline (not a XAML build) for build, deploy, and test scenarios in TFS.
Load testing (performance testing) using on-premises remote machines	Use Test Controller and Test Agents 2013 Update 5 to run load tests on-premises.
Remote execution of automated tests from Microsoft Test Manager using a lab environment	Currently there is no alternative for this scenario. We recommend you use the Run Functional Tests task in build and release definitions (not in a XAML build) to execute tests remotely.
Developers executing remote tests in Visual Studio	No longer supported.