Unity Test Tools Documentation

Contents

Unity Test Tools

Integration Test Framework

Integration Tests allow you to automate the verification process of your assets directly in a scene. They are designed to be used on existing content, directly within the Editor, to build tests which verify the behaviour of single assets or the interaction between them.

How to use Integration Test Framework

Assertion component

The Assertion Component is used to setup invariants on GameObjects. Setting up the component doesn't require writing any code - it's all done in the Editor UI. It is easily extensible, customizable, and can be configured for your own needs.

How to use the Assertion Component

Using Assertion Component with Integration Tests

Unit Test Runner

The integration of the NUnit Framework in the Editor allows you to execute unit tests from inside Unity. This means you can instantiate GameObjects and operate on them which would not be possible outside of Unity. We provide an integrated test runner that runs the tests and reports results.

How to use Unit Test Runner

NSubstitute library

NSubstitute is shipped with the Unity Test Framework. Please use its documentation for help: **http://nsubstitute.github.io/help.html**

Examples

Examples provided with the framework

General Q&A

- Which version of Unity is the framework compatible with?
  The framework is compatible with Unity 4.X.
- Can I write unit test only in C#?
  Although we focus on C# there should be no problems with writing tests in UnityScript or Boo.
- Can I move the tools to any subfolder?
  Yes.
- Why the framework doesn't work when I move it to *Standard Assets* folder?
  The current folder structure design will not work with the way *Standard Assets* folder is handled. However, with a little bit of work, you can move it to *Standard Assets.* Simply move all editor-dependant code to "*Standard Assets/Editor"* and place the rest under "*Standard Assets"*.
- What are the currently known issues and limitations?
  The limitations are as follows:
  - NSubstitute that comes with the framework is not threadsafe.
  - Windows Apps store (Metro) does not support Assertion Component or Integration Tests framework due to limitations in functionality
  - Result won't be send when running integration tests for WebPlayer

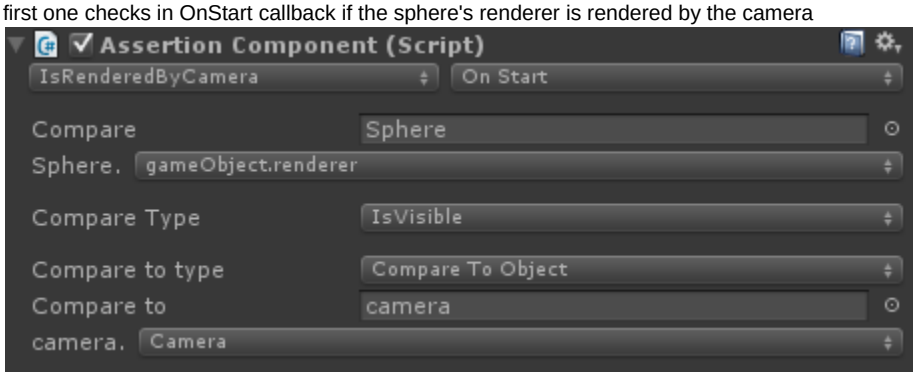
Examples provided with the framework

Scene examples

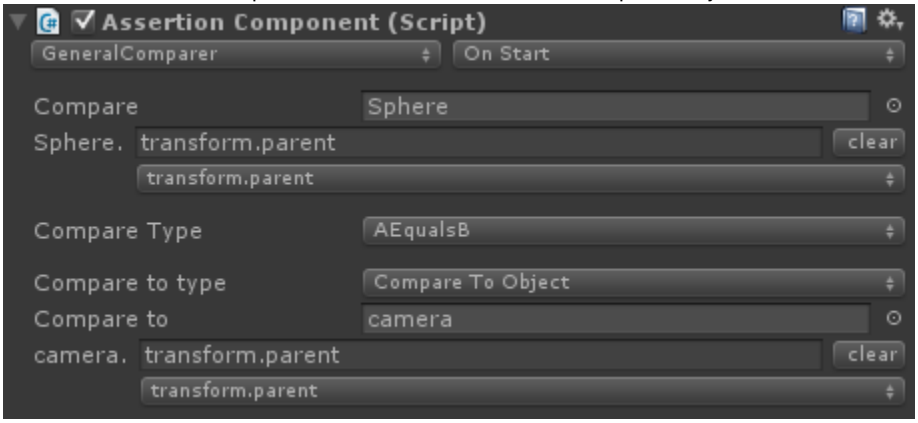
The examples are located int the *Examples* folder.

- ***IntegrationTestsExample.unity* scene**

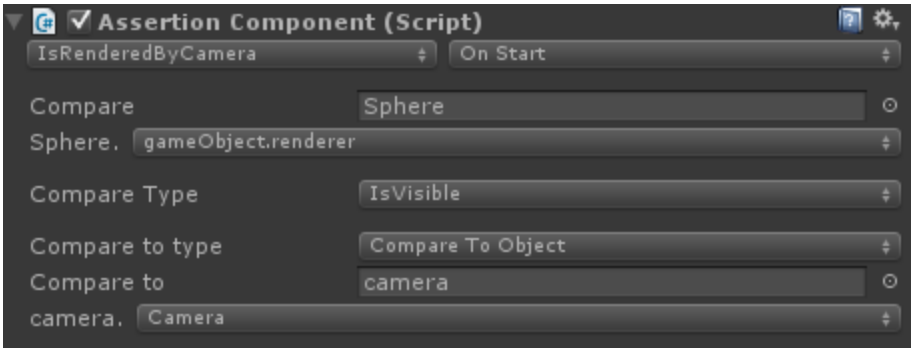
  This scene shows the functionality of the Game Tests runner. To work with it you need to open the runner (menu *Tests/Game Test Runner,* or *ctrl+alt+shift+t*). You will find 6 examples. On the scene there are also two common objects (prefabs) shared by all tests: *Cube TriggerFailure* and *CubeTriggerSuccess*. The prefabs are simple objects with colliders attached and scripts that call *Testing.Fail()* and *Te sting.Succeed()*, respectively. The tests show the technical side of the framework, therefore the examples may seem trivial. The tests on the scene have the following purpose:
  - *Test1 - Success*
    A sphere falls onto a cube that triggers success. Results in a successful test.
  - *Test2 - Timeout*
    A test has a very low (0.1 seconds) timeout value and the sphere won't have enough time to fall on the trigger cube.
  - *Test3 - FailurePlayerReceivesDamageWhenSpiderExplodes*
    A sphere falls onto a cube that triggers failure. Results in a failed test.
  - *Test4 - Ignored*
    Test with *ignore* check set. Will be ignored when running all tests.
  - *Test with Assertions*
    Test with *Succeed after all assertions are executed* checked. The sphere is this test has two assertions set:
    - first one checks in OnStart callback if the sphere's renderer is rendered by the camera

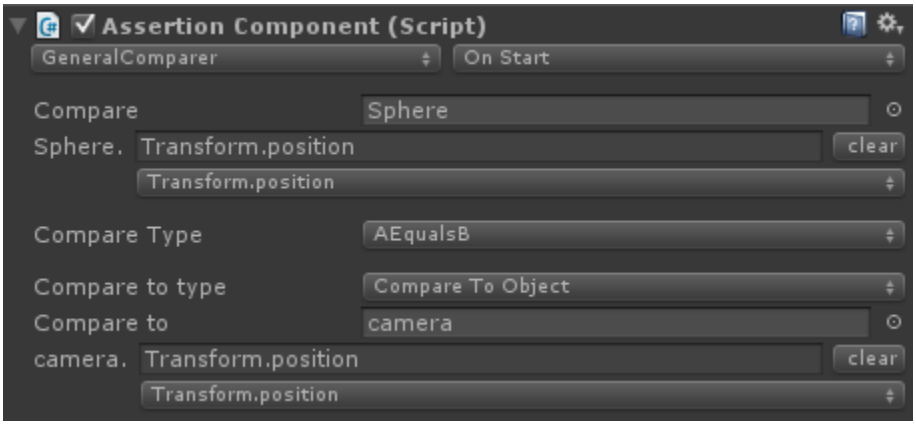
    - second one checks if the sphere and the camera share the same parent object

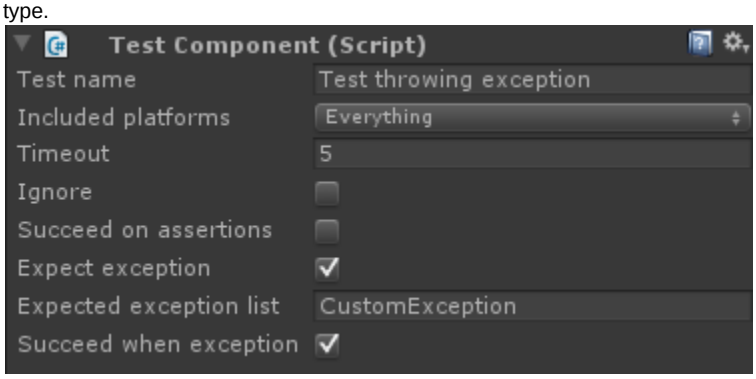
  - *Test with Assertion Fails*
    - first one checks in OnStart callback if the renderer is rendered by the camera



- second one checks if the sphere's transform.position is equal to transform.position of the camera. This condition is not true, therefore the test will fail.



- *Test throwing exception*
  A test that will succeeds when an exception is thrown. In this case the exception has to be CustomException or derive from this type.



- ***AssertionExampleScene.unity* scene**

  A simple example of how to use the Assertion Component to debug undesired state. The scene contains a sphere falling onto a plane and rolling out of it. We set two assertions on the sphere. First assertion verifies that the sphere needs to be rendered in the camera on every OnUpdate call. Second assertion makes sure the y value of the position vector of the sphere is always higher than y value of plane's position vector. In other words the sphere can not fall below the plane.

- ***AngryBotsTests/TestScene.unity* scene**

  This example uses assets from Unity's example project, Angry Bots. Two prefabs from Angry Bots are used: *PlayerPrefab* and *EnemySpider*. The first one is a standard player controller. The other prefab is an enemy spider that wakes up when the the player approaches it. On this test scene we want to check three things:
  - The spider wakes up and walks towards the player when the player is close enough
  - The spider doesn't wake up when the player is not close enough
  - The spider does damage to the player when it explodes

  The scene uses Game Tests to automate this procedure. You will find following tests on the scene:
  - *Test_PlayerReceivesDamageWhenSpiderExplodes*
    The spider attacks the player and explodes. To verify the player has taken damage an assertion has been set to verify the health is lower than a certain value.
  - *Test_SpiderSleepsWhenPlayerNotInRange*
    The player is set to be outside of spider's visibility range to make sure it doesn't wake up and attack the player. The verification is done by setting the *CubeCollisionFailure* between the spider and the player that would fail the test if the spider has stepped on it. Additionally, there is a GameObject with assertion that is checked after some period of time that spider's attack move controller is

disabled. This makes sure that the spider is still in sleep mode. The test will succeed when the assertion is checked.

- *Test_SpiderWakesWhenPlayerInRange*
  The player is situated within the range of spider's visibility. The spider wakes up and starts to move towards the player. Between the spider and the player there is a *CubeCollisionSuccess* that call Testing.Succeed() on collision. When the spider walks toward the player, it steps on the trigger and the test passes.

NUnit Examples

SampleTests.cs contains examples showing the basic NUnit usage.

- public void *ExceptionTest()* fails due to the exception thrown.
- public void *IgnoredTest()* shows the usage of ignore attribute.
- public void *SlowTest(),* a test which takes 1 second to run. You can try to the *Notify when test is slow* option on it.
- public void *FailingTest()* - demonstrates the usage of assertions and fails due to the call of Assert.Fail().
- public void *PassingTest()* - demonstrates the usage of assertions and succeeds due to the call of Assert.Pass().
- *ParameterizedTest, RandomTest, RangeTest* show capabilities of nUnit.

NSubstituteDemo.cs contains classes needed to demonstrate the usage of **NSubstitute** in a simple scenario.

IGameEvent - an interface that represents an abstract game event.

IGameEventListener - an interface that represents an abstract event consumer.

GameEventSink - is an object that gets events from the system and passes them to registered listeners.

A test RegisteredEventListenersGetEvents checks that ReceiveEvent method was called on a registered listener that represents IGameEventListener.

A substitute for IGameEventListener is used instead of concrete implementation.

```
public void RegisteredEventListenersGetEvents()
{
GameEventSink sink = new GameEventSink();
//a proxy for IGameEventListener is created.
IGameEventListener listener = Substitute.For<IGameEventListener>();
sink.RegisterListener(listener);
sink.ReceiveEvent(Substitute.For<IGameEvent>());
//In this line a check that the method was called (with any arguments).
listener.Received().ReceiveEvent(Arg.Any<IGameEvent>());
}
```

Please refer to NSubstitute documentation to learn more about NSubstitute functionality.

Unit tests in UnityScript and Boo

Example of unit tests written in UnityScript

```
class UnityScriptUnitTests
{
@NUnit.Framework.Test()
function UnityScriptTest () {
NUnit.Framework.Assert.Pass();
}
}
```

Example of unit tests written in Boo

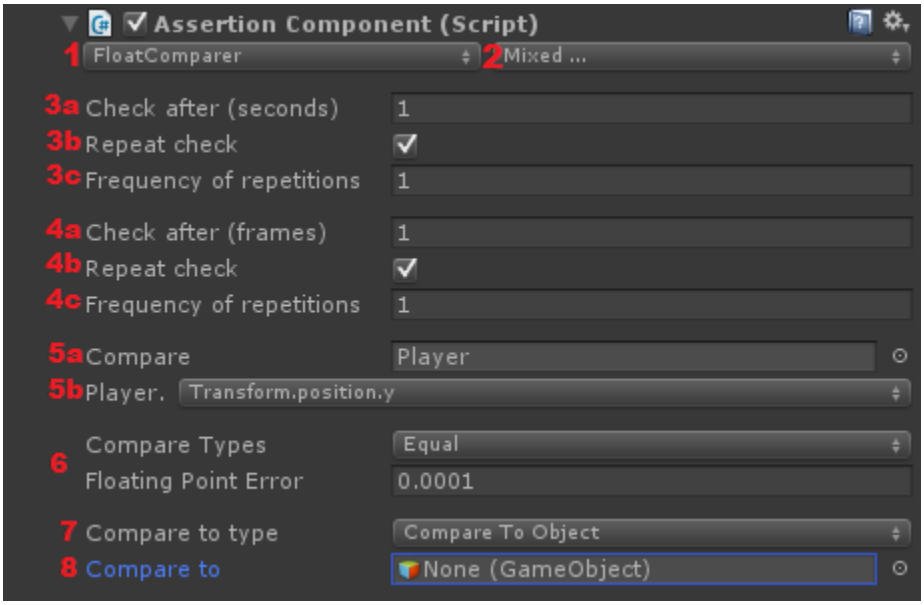
```
namespace UnityTest

import NUnit.Framework

class BooUnitTests:

[Test]

def BooTest ():

Assert.Pass();
```

How to use the Assertion Component

Assertion Component overview

The Assertion Component brings you the possibility to assert desired states of your game objects. It's a visual tool that doesn't require writing any code. It was designed to be extensible and adaptable to the content of your project and your needs.

The way the component works is simple. You need to set up an invariant - a condition you expect to be always true. You specify when the condition should be checked (for example every Update method). Now, when your project is running and an assertion you set fails, an exception is throw. This way you get notified that your application got into an undesired state and you can investigate the issue. In most cases you would want to have the "*Error pause*" option enabled in the *Console* window to pause the run when an error occurs.



1. Comparer selector - A comparer defines how two values should be compared with each other. It determines the result of the assertion.
2. Frequency of checks - Multiselectable control where you can define when the assertion should be checked.
3. Custom menu for *After period of time* frequency option (see 2.) It won't be visible unless the option is selected
    a. After how many seconds should the first check be done
    b. Should the checks be repeated
    c. How often to repeat the checks
4. Custom menu for *Update* frequency option (see 2.) It won't be visible unless the option is selected.
    a. After how many frames first check should be done
    b. Should the checks be repeated
    c. How often to repeat the checks
5. First GameObject that is used in the compare method. By default it's the GameObject to which the component is attached
    a. GameObject reference field
    b. Path to the variable to be checked
6. Custom field from the selected comparer (*Float Comparer*). In this case they define operation type and precision.
7. What to compare the object (selected in 5.) with. It's possible to compare it with another GameObject, static value or null value.
8. The other object to compare with.

Setting up the Assertion Component

The Assertion Component is really easy to set up. A simple assertion can be set up in just a few steps:

1. Choose the Comparer (1) that will be used when checking the assertions. A Comparer usually defines acceptable types which will be a helpful filter when selecting property to compare
2. Select when you would like the assertion to be checked (2). Most of the callback methods of MonoBehaviour are available (like. OnStart, OnUpdate). You can also set the time after you would like the check to be done (*After period of time*). OnUpdate and AfterPeriodOfTime allow you to select an extra parameter defining the frequency of checks (3, 4).
3. Choose path to the property (5b) which value you would like to compare. The values will be filtered out based on types accepted by selected Comparer. For example the Float Comparer accepts only float values, so only properties and fields of float type will be presented.
4. A Comparer can expose fields which can be used to customize behaviour. For example the Float Comparer allow you to select the type of compare operation (Equal, Greater, Less) and the precision of floating point operations (6).
5. Next, you can select what you would like to compare the value with. By default, you can compare it with another GameObject's property. You can also compare it with a static value (if it's supported by the Comparer) or to null.
6. Depending on your previous choice, select the other value to compare with.

Create assertion from code

It's possible to create the assertion from script. To do so, use one of the following APIs from the **UnityTest.AssertionComponent** class:

- public static **T Create**<**T**> (CheckMethod **checkOnMethods**, GameObject **gameObject**, string **propertyPath**)
- public static **T Create**<**T**> (CheckMethod **checkOnMethods**, GameObject **gameObject**, string **propertyPath**, GameObject **gameObject 2**, string **propertyPath2**)
- public static **T Create**<**T**> (CheckMethod **checkOnMethods**, GameObject **gameObject**, string **propertyPath**, object **constValue**)

The **Create** static methods take as a generic argument the class of the comparer you want to use. The **checkOnMethods** parameter flag allows you to set the methods where you want the assertion to be checked. The **gameObject** is the instance of GameObject from with a property passed in **propertyPath** parameter will be used is the evaluation.

The **gameObject2** and **propertyPath2** represent to the path to a property from the second GameObject you want to use in the assertion. Pass the **constValue** to use is in comparison.

In addition, each of the method has an overload with an **out** parameter that return reference to the **AssertionComponent** that is created. Use it to configure the validation frequency if necessary.

Examples of usage:

```
public class InitAssertions : MonoBehaviour
{
 public float FloatField = 3;

 public GameObject goReference;

 public void Awake ()
 {
  //An assertion that will compare a foat value from a custom component attached to a GameObject to a con
stant variable equal to 3.
  //The comparison will happen Start method and every 5 frames in the Update method
  //Additionally, the comparer is configured to have accuracy of 0.1 for floating equality check.
  AssertionComponent ac;
  var c = AssertionComponent.Create<FloatComparer> (out ac, CheckMethod.Update | CheckMethod.Start, gameO
bject, "InitAssertions.FloatField", 3f);
  ac.repeatEveryFrame = 5;
  c.floatingPointError = 0.1;
  c.compareTypes = FloatComparer.CompareTypes.Equal;

  //Create an assertion that will fail is the FloatField from InitAssertions component of gameObject will
 change it's value
  AssertionComponent.Create<ValueDoesNotChange> (CheckMethod.Update | CheckMethod.Start, gameObject, "Ini
tAssertions.FloatField");

  //Validate the gameObject.transform.y is always equal to 3 (defined in this component)
  transform.position = new Vector3(0, 3, 0);
  AssertionComponent.Create<FloatComparer> (out ac, CheckMethod.Update, gameObject, "InitAssertions.Float
Field", gameObject, "transform.position.y");

  //Check with the goReference field from this component is set to null
  var gc = AssertionComponent.Create<GeneralComparer> (CheckMethod.Update, gameObject, "InitAssertions.go
Reference", null);
  gc.compareType = GeneralComparer.CompareType.AEqualsB;
 }
}
```

Assertion Component features:

- **Comparers**

  Comparers define the assertion action. A Comparer must derive from *ObjectComparerBase* class and implement the *Compare* method.

  Example of a Comparer implementation:

```
public class FloatComparer : ObjectComparerBase<float> 1
{
    public enum CompareTypes
    {
        Equal,
        NotEqual,
        Greater,
        Less
    }

    public CompareTypes compareTypes;
    public double floatingPointError = 0.0001f;                    2

    protected override bool Compare (float a, float b)
    {
        switch (compareTypes)
        {
            case CompareTypes.Equal:
                return Math.Abs (a - b) < floatingPointError;
            case CompareTypes.NotEqual:
                return Math.Abs (a - b) > floatingPointError;
            case CompareTypes.Greater:
                return a > b;
            case CompareTypes.Less:
                return a < b;
        }
        throw new Exception();                                     3
    }
    public override int GetDepthOfSearch() 4
    {
        return 3;
    }
}
```

1. A Comparer needs to inherit from *ObjectComparerBase* class.

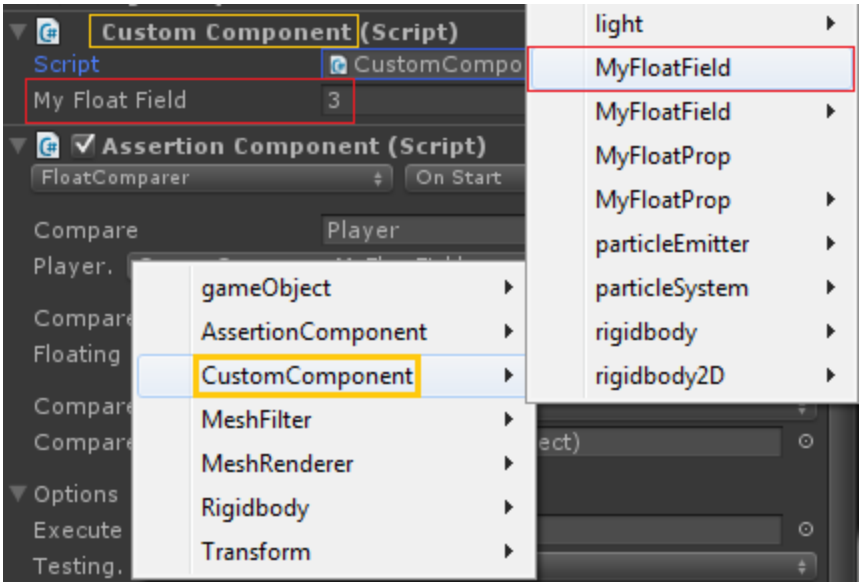
2. Any public serializable fields will be exposed in the Comparer. They can be used for customizing the Comparer.

3. The Compare method will be called when the assertion is performed. It's an abstract method that takes as arguments two values of types as defined by the Comparer. If types are not defined (the Comparer derives from non-generic *ObjectComparerBase*), the argument will be of *System.Object* type.

4. Additional customization can be done by overriding methods. The *GetDepthOfSearch* method overrides default depth of property search algorithm.

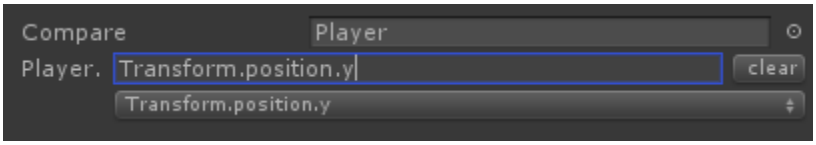
- **Selecting path to the property of a GameObject**

When you select path to the desired property, the control will show you a list of fields which types are the same as Comparer's accepted type. The list will contain properties of GameObject itself and properties of all Components attached to it, including custom scripts.



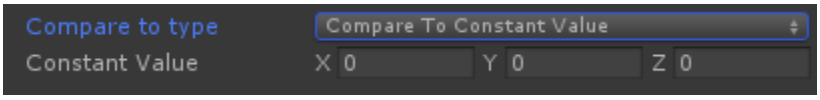
- **Manual path selection**

  If the Comparer doesn't have accepted type specified, it is impossible to present possible value to pick from. You will need to type in the path to the property by hand. The editor will give you a tip if such path might not be correct and will allow you to pick some values form a hint list. Hint: if you press down arrow while typing the path, the path hints popup will be displayed.



- **Comparing to constant value**

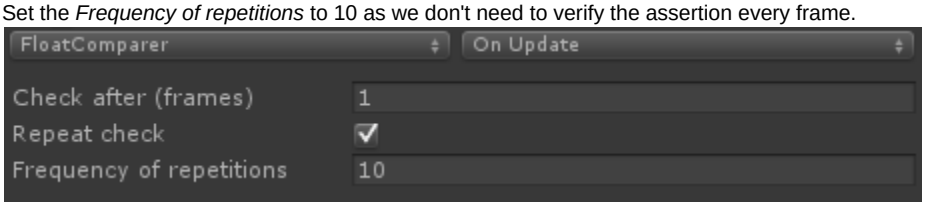
  You can compare the first value with a static value you will provide in the component. To do that, in the *Compare to type* field select *Compare To Constant Value*. If the type accepted by the Comparer is serializable by default in Unity, an appropriate control will appear to put in the values.



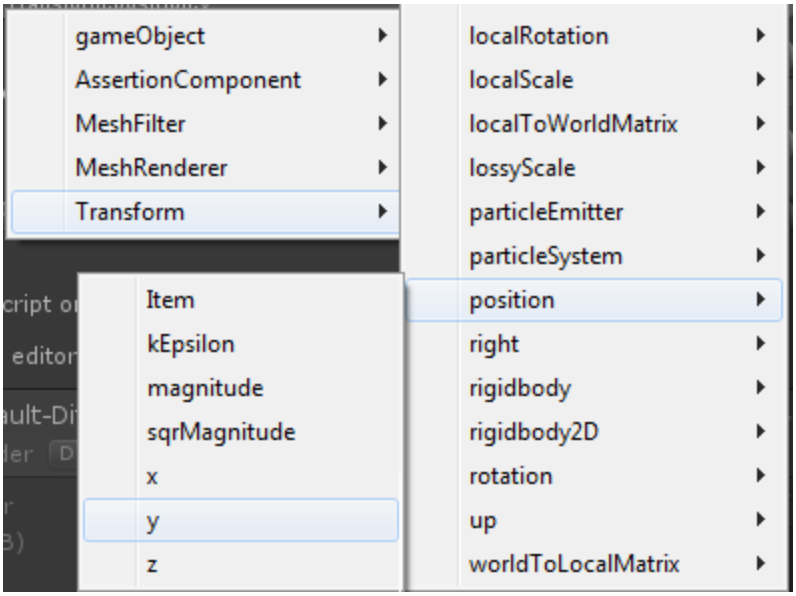
How to start

This steps will guide you through the process of setting up a simple Assertion Component

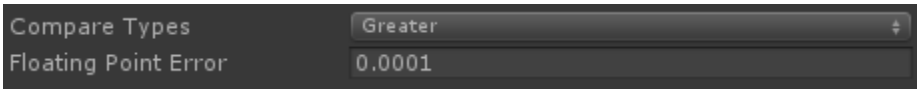
1. Create a new object on the scene, for example a Sphere, and add a Rigidbody component to it. Select the object and go the Inspector
2. Add the Assertion Component
3. Select the Float Comparer
4. Select the OnUpdate as the moment to verify the assertion. Remember to uncheck the default *OnStart* selection since it's a multi select control.
5. Set the *Frequency of repetitions* to 10 as we don't need to verify the assertion every frame.



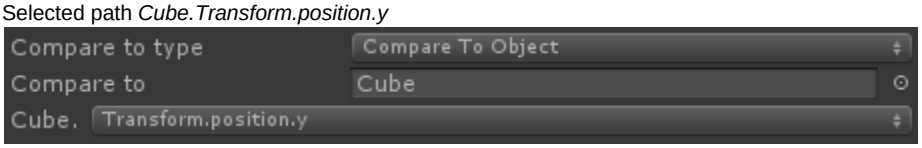
6. Select *Sphere.Transform.position.y* value.



7. Select Compare type to Greater as we want the *Sphere.Transform.position.y* value to be always greater than the value we will compare it with.



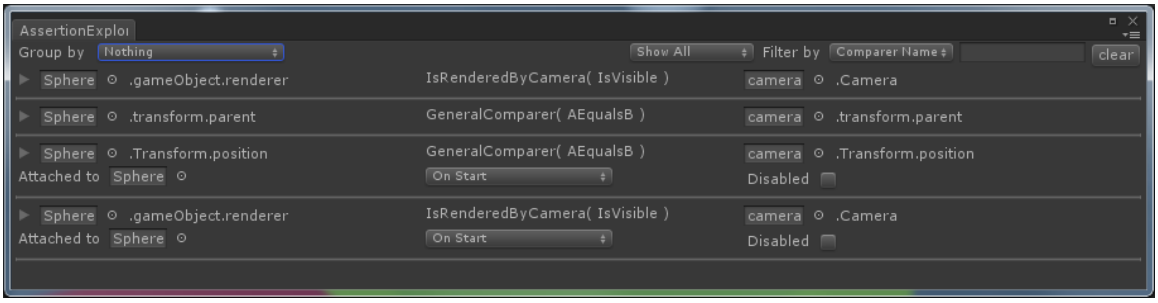
8. Add another Game Object, for example a Cube, and place it somewhere below the first object.
9. Drag the cube's Game Object to the *Compare to* field in the Assertion Component.
10. Selected path *Cube.Transform.position.y*



11. Run the scene
12. You will see the sphere falling down and once it falls below the cube, the editor should pause. It happened because the assertion check has failed (the condition *Sphere.Transform.position.y > Cube.Transform.position.y* was no longer valid). If the editor doesn't pause, make sure you selected *Error Pause* option in the *Console*.

Assertion Explorer

The Assertion Explorer is available from the *Unit Test Tools* menu. It shows all assertion placed on objects of the current scene. Most fields are read-only. You can only disable and enable single components from the explorer. It allows grouping the list and basic filtering.



Code stripping

The assertions will be stripped out from GameObjects in non developers builds. To completely remove any code dependencies a custom build pipeline needs to be incorporated. An example of code stripping build pipeline:

```
using UnityEditor;
using UnityEngine;

public class StripCodeAndBuild
{
 private static string assetsPath = "Assets/UnityTestTools";
 private static string buildTempPath = "Assets/Editor/UnityTestTools";

 [MenuItem ("Unity Test Tools/StripCodeAndBuild")]
 public static void StripCodeAndBuildStandalone ()
 {
  AssetDatabase.CreateFolder ("Assets", "Editor");
  var result = AssetDatabase.MoveAsset (assetsPath, buildTempPath);

  if (string.IsNullOrEmpty (result))
  {
   BuildPipeline.BuildPlayer (new[] {EditorApplication.currentScene}, @"C:\temp\release.exe", BuildTarget
.StandaloneWindows, BuildOptions.None);
   result = AssetDatabase.MoveAsset (buildTempPath, assetsPath);
   if (string.IsNullOrEmpty (result))
    AssetDatabase.Refresh ();
   else
    Debug.LogWarning (result);
  }
  else
  {
   Debug.LogWarning (result);
  }
 }
}
```

Remarks

- Known issue: The component will not work with a MonoBehaviour attached in which the class name starts with a lower case.

How to use the Integration Test Framework

How to open the runner

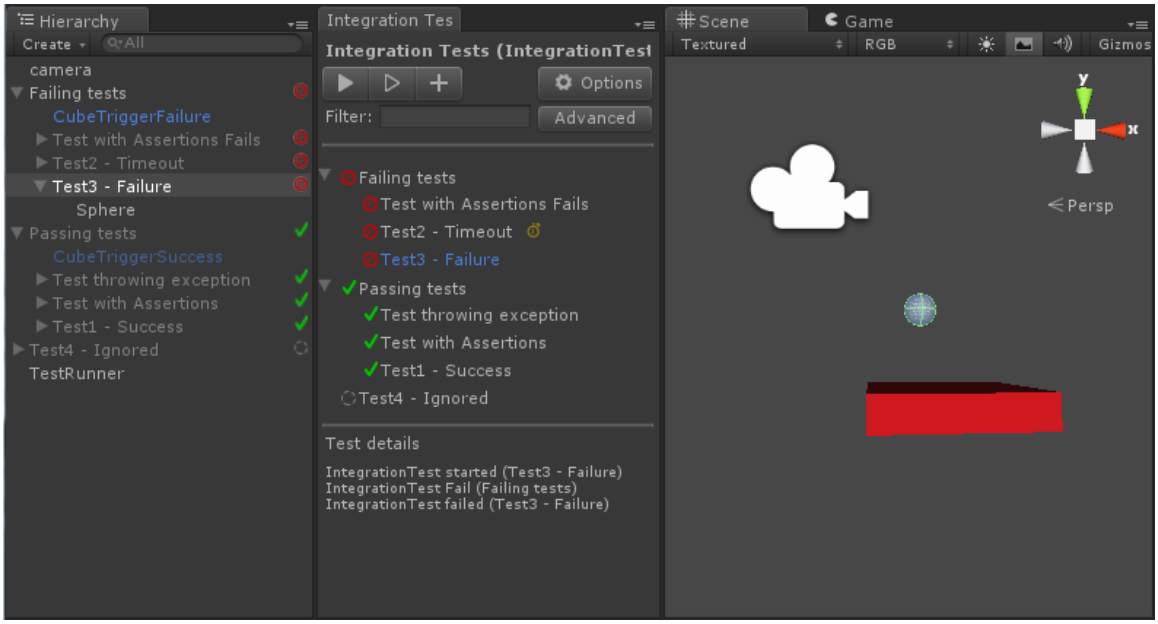
You open the Integration Tests Runner from the menu bar *Unity Test Tools/Integration Tests Runner* or by *Shift+Ctrl+Alt+T* combination.

How does the Integration Test Runner work?

The Integration Tests are designed to run on a separate scene. You can consider the scene where you place your tests as a test suite. One test scene can contain multiple tests. You should not put your tests on your production scenes. Instead, create a separate scene for them.

A Test Object is a GameObject on the scene that has TestComponent attached to it. Everything under the Test Object in the hierarchy is considered to belong to this test. Any object that is not under a Test Object will be common for every test on the scene (it's usually environment like floor, walls etc.). You shouldn't care about creating the Test Object manually. Everything is done through the Test Runner. Tests can have hierarchical structure. One Test Object can be a direct child of another Test Object. A Test Object with any Test Object under it becomes a group and it's not executed as a test.

Only one test can be active at any time. When you select a test, all other tests will become disabled, so you can work only with one test at a time.



This scene is an exemplary scene shipped with the framework. If you look at the hierarchy view you will notice 7 Test Objects and 2 groups. First group has all the tests that are passing in this example. The second group consists of tests that are failing. The remaining, non grouped test, is set to be ignored and not executed. Each of them has an icon showing the result from last time the test was run. In this example the *Test3 - Failure* is the active selection, so all other tests are disabled. This test contains only one GameObject, the Sphere. If you take a look at the scene now you will notice two additional cubes: red and green. Those cubes are not places under any Test Object, so they will be shared by any test in the scene. In the hierarchy window you will find them as *CubeTriggerSuccess* and *CubeTriggerFailure*. Additionally, you can also see a *TestRunner* object. This object is responsible for driving the test run once execution starts. It will be added automatically when you add the first test on a scene.

When you run the tests the following steps are performed by the runner:

1. Play mode is enabled
2. The first test gets enabled (becomes active)
3. Wait until the test has finished (or a timeout has occurred)
4. The current active test gets disabled
5. If there are more tests in the queue, enable next test and go to step 3
6. Report results and finish test run

Creating Integration Tests from code (Dynamic Integration Tests)

It's possible to mark a MonoBehaviour with an attribute that make it loaded by the test runner. On your class that derives from MonoBehaviour, put the **[IntegrationTest.DynamicTest (string)]** attribute and pass the name of the test scene as the argument. The runner will show the test on the test list and you will be able to run it. The test will contain the script that was marked with the attribute. Additionally, you can mark the test with attribute change test's parameters:

- **[IntegrationTest.Ignore]**
- **[IntegrationTest.ExpectExceptions]**
- **[IntegrationTest.SucceedWithAssertions]**
- **[IntegrationTest.Timeout]**
- **[IntegrationTest.ExcludePlatform]**

```
[IntegrationTest.DynamicTest ("IntegrationTestsExample")]
[IntegrationTest.Ignore]
[IntegrationTest.ExpectExceptions (false, typeof (ArgumentException))]
[IntegrationTest.SucceedWithAssertions]
[IntegrationTest.Timeout(1)]
[IntegrationTest.ExcludePlatform(RuntimePlatform.Android, RuntimePlatform.LinuxPlayer)]
public class DynamicIntegrationTest : MonoBehaviour
{
    void Start()
    {
        IntegrationTest.Pass(gameObject);
    }
}
```

If the currently opened scene matches the string provided in the attribute constructor, the test runner will try to create the test on scene. It will be,

however, marked with HideFlags.DontSave so you dynamic tests won't be saved on the scene. If you change any of the attributes related to the tests, remember to refresh the object on the scene (in the inspector) or simply delete it from the scene to get it rebuilt.

How to control a test flow (How to start and finish a test)

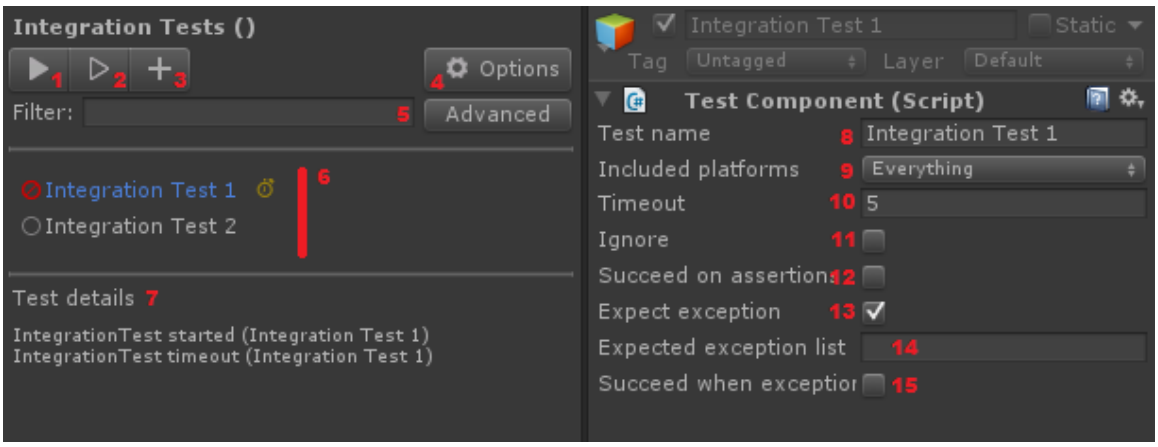
A test starts once the Test Object gets enabled. The Test can finish its run in multiple ways:

- Function *Testing.Pass()* is called. This will successfully finish the test.
- Function *Testing.Fail()* is called. This will fail the test.
- Execution times out. This can happen when none of the above functions is called within a specified period of time (you can set the timeout value per test).
- An unhandled exception is thrown.
- An expected exception is thrown (*Expect exception* must be checked)
- Every Assertion Component on objects under tests is checked at least once (the "Succeed after all assertions are executed" option needs to be selected)

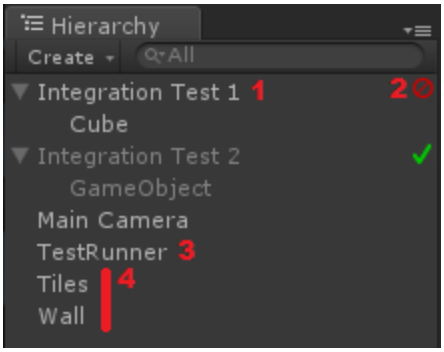
Note that you can use a set of pre-made assets for controlling test flow. They are placed in the *IntegrationTestsFramework / TestingAssets* folder.

Integration Test Runner

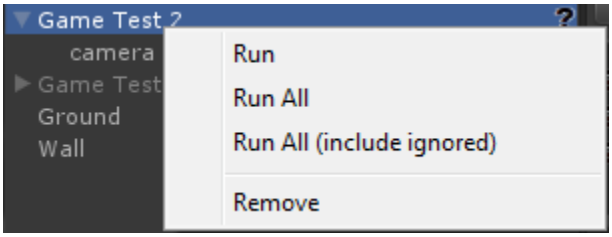
The Integration Test Runner window functionality:



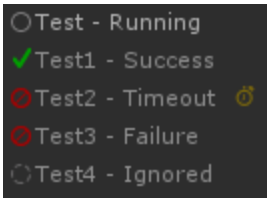
1. Run all tests in the scene (excluding ignored tests)
2. Run selected test(s).
3. Create a new test - creates new test object on the scene
4. Options - options for working with Integration Tests
    a. Add GameObjects under selected test - when selected, when you add a new object to the scene it will be automatically placed under the test GameObject instead of the hierarchy root
    b. Block UI when running - when selected, a dialog will appear during test execution
5. Test Filter - will filter out tests where name does not not contain the string
    a. Show succeeded - show tests that succeeded
    b. Show failed - show tests that failed
    c. Show ignored - show tests that are ignored
    d. Show not runed - show tests that hasn't been run
6. Test list - list of all tests available in the scene
7. Test log and exception messages
8. Test name - name of the test
9. Included platform - on what platform the test should included
10. Timeout - number of second after the test will timeout
11. Ignored - ignore the test when running all tests
12. Succeed after all assertions are executed - select if the test should finish after all assertions from Game Object in the test got checked at least once.
13. Expect exception - the test will not fail if an exception if thrown.
14. Expected exception list - a list of exception that will not fail the test when thrown. Separate the exceptions with comma (","). Derived types from types on the list will also be considered as expected. If the list is empty, any exception type will be accepted.
15. Succeed when exception is thrown - the test will succeed when one of the excepted exceptions is thrown.



1. Selected test
2. Result icon
3. Test Runner object
4. Common objects - objects that are not under a test node will be active in every test



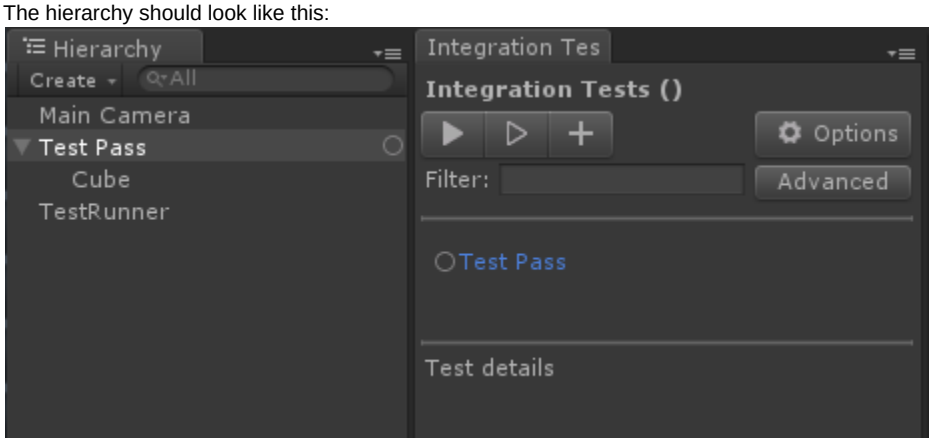
Test context menu (right click on a test)



Icons

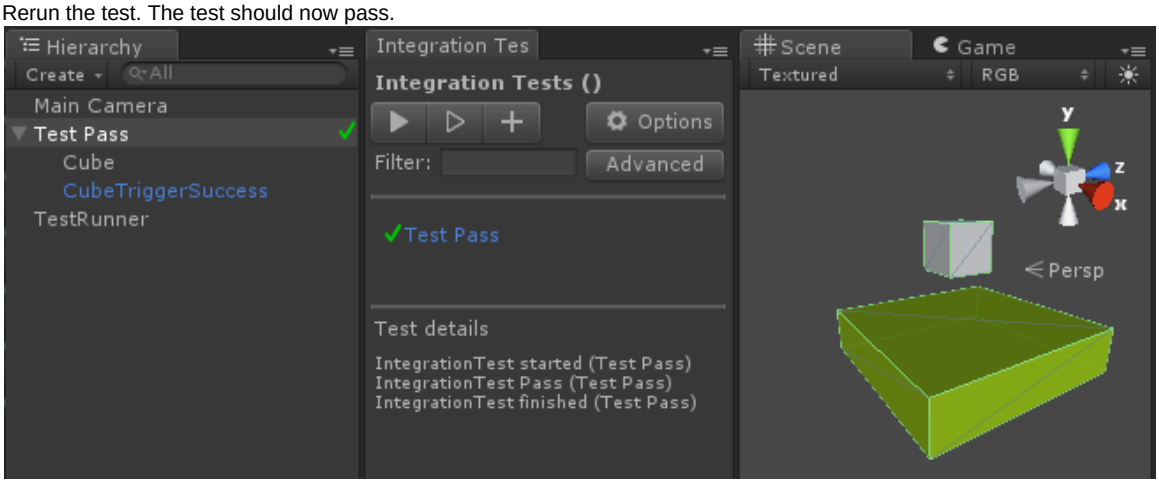
Creating simple tests

These steps will walk you through writing two simple tests:

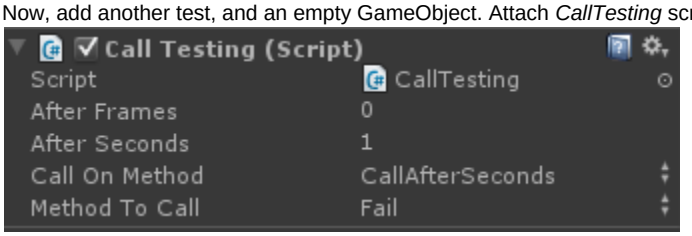
1. Create a new scene that will contain the tests
2. Open the Integration Tests Runner Window. (in menu bar, Unity Test Tools/Integration Tests Runner, or ctrl+alt+shift+t)
3. Click the "plus" button to add new test and rename it to *Test Pass*. Notice that a TestRunner object should be automatically added.
4. Select the test
5. Add a Cube to the scene. If the *Add new GameObjects under selected test* option is checked the Cube will automatically be placed under the test node. Otherwise move it there manually.
   The hierarchy should look like this:
   

6. Add a Rigidbody component to the cube. You can run the test and verify the cube is falling down. To run the test right click on it and select *Run* (you can also use ctrl+t combination if the test is selected). The test will timeout after 5 seconds (by default) because *Testing. Pass()* was never called.

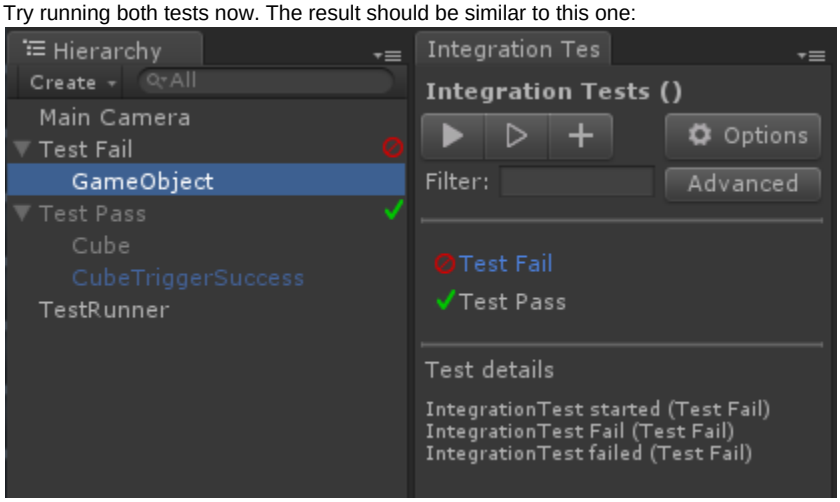
7. Add another Cube below the first one that will call *Testing.Pass()* on collision. You can use the test assets provided with the framework. Find *CubeTriggerSuccess* prefab under *Assets\UnityTestFramework\IntegrationTestsFramework* and put it on the scene below the first Cube so it will fall on it. If you look at the prefab you will see that it has a script attached. The script is responsible for calling *Testing.Pass ()* in *OnCollisionEnter* function.
8. Rerun the test. The test should now pass.



9. Now, add another test, and an empty GameObject. Attach *CallTesting* script from the Testing Assets and make it fail after 1 second.



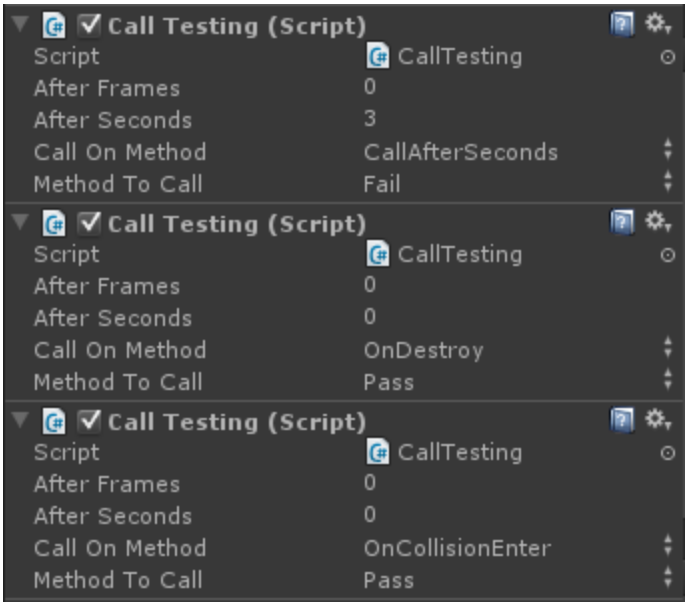
10. Try running both tests now. The result should be similar to this one:



Testing Assets

A few testing assets are provided with the framework:

- *CallTesting.cs* script - allows you to call Testing.Pass() or Testing.Fail() automatically from selected methods or after a desired amount of time or number of frames.

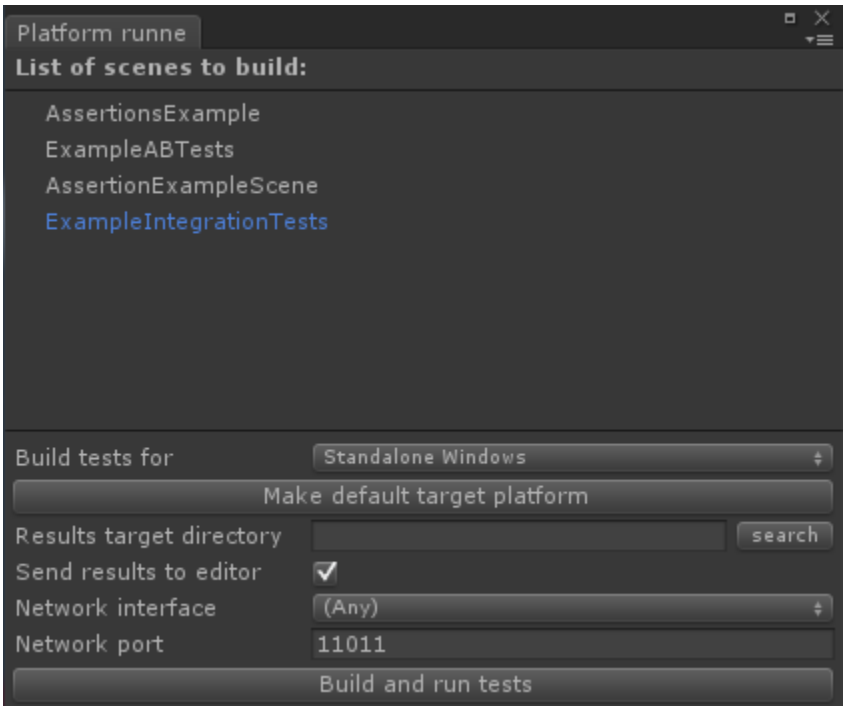


This configuration will make the test succeed if the object enters a collision (OnCollisionEnter) or gets destroyed (OnDestroy). Otherwise, the test will fail after 3 seconds.

- *CubeTriggerSuccess*, *CubeTriggerFailure* - cube prefabs that will succeed/fail on OnTriggerEvent.

Running tests on platforms

To automatically run build and run a test scene on a target platform use the *Platform Runner* (*Unity Test Tools/Integration Tests/Platform Runner/Run on platform, Ctrl+Shift+R*).



1. List of scenes to build - Multiselectable list of scene to build. Selected scenes need to be test scenes in order to run properly
2. Build tests for - A list of target platform player will be built for. The list correspond to BuildTarget.
3. Make default target platform - will make currently selected platform the default one
4. Result target directory - a target directory where the results should be saved. The path should be valid and exist. It's possible to specify network location, i.e. *\\network-drive\results*
5. Send results to editor - the runner will communicate with the editor via TCP protocol in order to send the resuts. The target device needs to be connected to the same network.
6. Network interface - the interface the runner should use in order to connect with the editor. Useful when the target device is connected to a different subnetwork.
7. Network port - which network port should be used

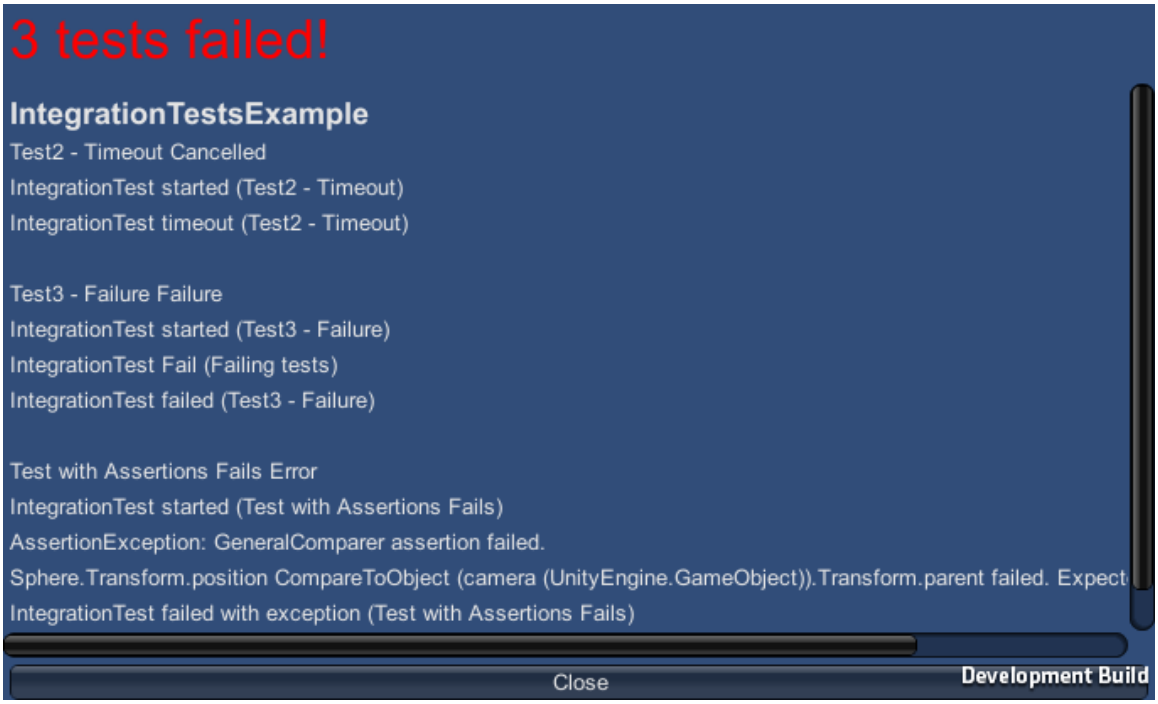
8. Build and run tests - will do what it says

Additionally, *Unity Test Tools/Integration Tests/Platform Runner/Run current scene (Ctrl+Alt+Shift+R)* will run currently open scene in the default platform.

Reporting results

When running tests from the editor, the results are available in the Test Runner.

You build and run test using the Player Runner, after the tests are executed you will see a result report on the screen of the targeted device.



The player runner will now send the result back to the editor via TCP if possible. The results can be saved under specified path. This will only work on platforms that support TCP communication.

If you run test in batch mode, a file with results will be generated and the application will close itself. The result file is an XML file and it's located in project's root folder, unless a parameter is specified. The results are in nUnit result format. The schema for the file can be found here: http://www.nunit.org/docs/2.6.2/files/Results.xsd.

For platform that don't support file system, you can parse platform's log output for results.

Headless running (batch mode)

It is possible to execute integration tests from command line. In order to do that, run Unity in batch mode and execute *UnityTest.Batch.RunIntegrationTests* method on start.

Parameter names	Description
`testscenes`	Comma-separated list of scene names to be run. If the parameter is not provided, the runner will automatically pick all scene matching following pattern: "*Test[s].unity"
`targetPlatform`	Platform you want to run the tests on. Value has to correspond BuildTarget enum. If not present, the tests will be run in the editor.
`resultsFileDirectory`	A path to a folder where the results should be placed. If not specified, the results will be placed in project's root folder.

Example:

```
>Unity.exe -batchmode -projectPath PATH_TO_YOUR_PROJECT -executeMethod UnityTest.Batch.RunIntegrationTests
-testscenes=TestScene1,TestScene2 -targetPlatform=StandaloneWindows -resultsFileDirectory=C:\temp\
```

This will run tests from scenes *TestScene1* and *TestScene2* in Standalone player for Windows and generate results in C:\temp\ folder.

The editor will exit with a return code according to the result of the run:

Return code	description
0	Run succeeded, no failures occurred
2	Run succeeded, some tests failed
3	Run failure (other failure)

TIP: On Windows, for batch mode, run Unity with followind command in order to receive the return code properly:

```
start /WAIT Unity.exe ARGUMENT_LIST
```

It will allow you to get the return code from ERRORLEVEL variable.

How to use Unit Test Runner

How to open the Unit Test Runner

You open the Unit Test Runner from the menu bar *Unity Test Tools/Unit Test Runner* or by *shift+ctrl+alt+u* combination.

Getting Started with NUnit

After importing Unity Test Framework package, NUnit library(version 2.6.2) is included into your project.

If you are new to NUnit please visit NUnit's Quick Start guide to get started. This article demonstrates the development process with NUnit in the context of a C# banking application.

To start your unit testing experience open the test runner window by clicking "Test -> Unit Test Runner" and it will show you the windows with test examples that are supplied with Unity Test Framework.

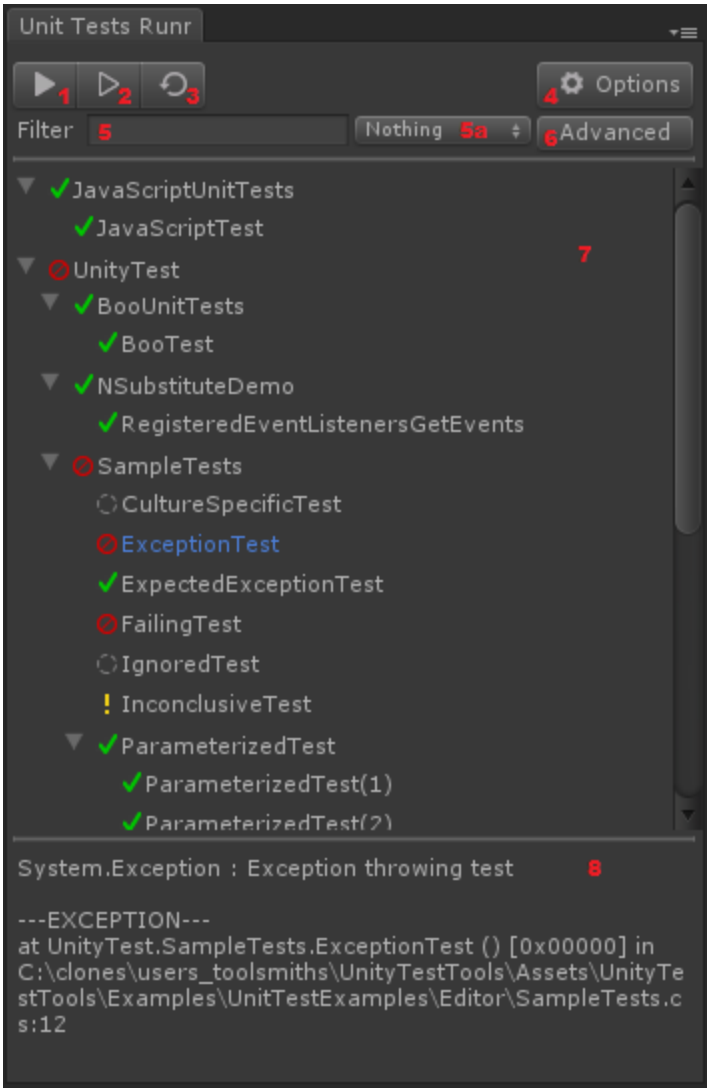
How Unit Test Runner works

The Unit Test Runner uses NUnit library that's included into the project (nunit.core.dll, nunit.core.interfaces.dll, nunit.framework.dll). The runner looks for tests in Assembly-CSharp.dll and Assembly-Editor-CSharp.dll.

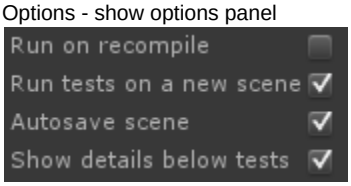
Before executing the tests the runner will open a new scene (unless you disable it in the options), therefore you may get a prompt to save your scene. After the run is finished the previous scene will be loaded automatically in between the run no cleanup is done and it must be done within the test suite if necessary. For managing *GameObjects* on the scene you can use the *UnityUnitTest* class which provides you with a method for creating *GameObjects* and does the cleanup automatically.

It's recommended to keep the unit test files under Editor folder so they won't be included in the build.

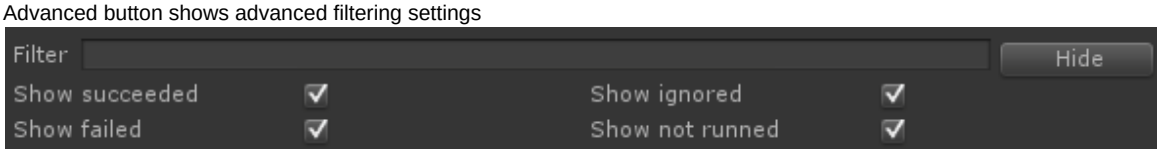
Unit Tests Runner window



1. Run all tests
2. Run selected test
3. Run failed tests
4. Options - show options panel



   a. Run on recompilation - the tests will automatically run after every compilation (unless the compilation failed)
   b. Run tests on a new scene - the runner will open new scene to run the tests and load back the current one after the run has finished. The user will be prompt to save the scene first.
   c. Autosave scene - will automatically save the scene before the run start (available when "Run tests on a new scene" is checked)
   d. Show details below tests - positions the details tab below the test list.
5. Filter that allows to show only methods and classes that match the string in this field
   a. Category filter - if tests have Category attribute set on them, this field will allow you to filter the list by selecting one or many categories to show
6. Advanced button shows advanced filtering settings



7. Tests hierarchy window that shows the tests and the execution results
8. Displays the exception and the stacktrace for failed tests

Reporting results

After each run the results are reported in the Test Runner window. When running in a batch mode, an XML file is generated with nUnit style results. It's located in project's root folder or under specified path. The schema for the file can be found here: http://www.nunit.org/docs/2.6.2/files/Results.xsd.

Include unity tests from external libraries

The runner will scan all dll libraries in the project and include those that depend on 'nunit.framework'.

Headless running (batch mode)

It is possible to run test from command line. In order to do that, run unity in batch mode and execute *UnityTest.Batch.RunUnitTests* method on start.

Parameter names	Description
`resultFilePath`	A path where the result file should be placed. If the path is a folder, a default file name will be used. If not specified, the results will be places in project's root folder.
`filter`	Filter tests by names.
`categories`	Filter tests by categories

Example:

```
>Unity.exe -batchmode -projectPath PATH_TO_YOUR_PROJECT -executeMethod UnityTest.Batch.RunUnitTests
-resultFilePath=C:\temp\results.xml
```

This will run all available tests and generate result file under *C:\temp\results.xml*.

The editor will exit with a return code according to the result of the run:

Return code	description
0	Run succeeded, no failures occurred
2	Run succeeded, some tests failed
3	Run failure (other failure)

NOTE: In order to get the result code, **do not** use the -quit parameter. Read more about command line argument here.

TIP: On Windows, for batch mode, run Unity with followind command in order to receive the return code properly:

```
start /WAIT Unity.exe ARGUMENT_LIST
```

It will allow you to get the return code from ERRORLEVEL variable.

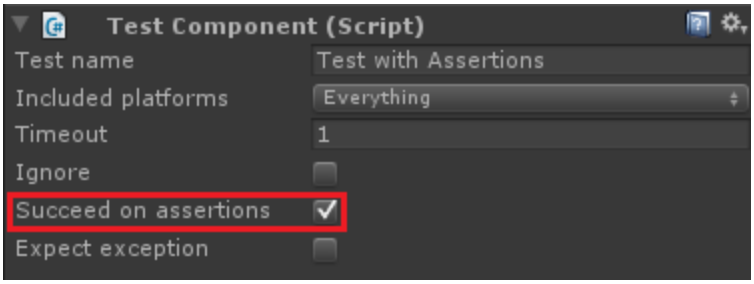
Using Assertion Component with Integration Tests

The Assertion Component can be used in Integration Tests to verify expected behaviour. When an Integration Test test is selected, in the inspector you can select *Succeed on assertions* option.

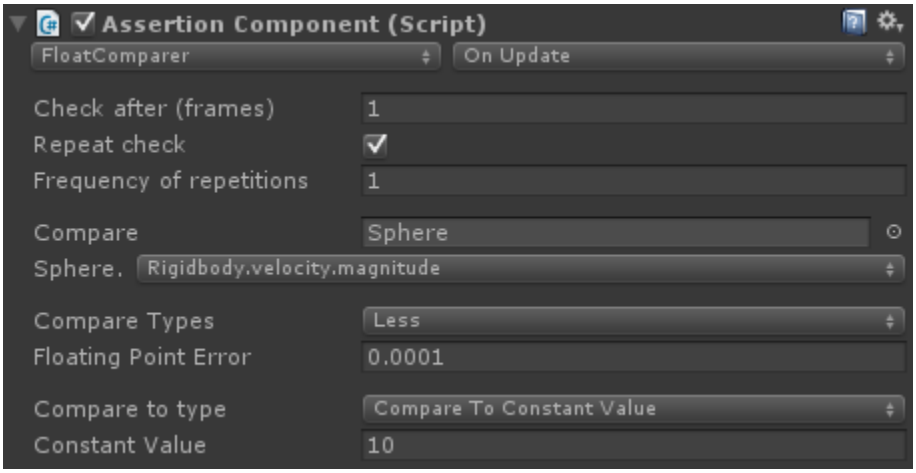
When the option is selected, there is no need to call the *Testing.Succeed()* or *Testing.Fail()* methods explicitly for that test. The test runner will check for all Assertion Component attached to every object under the test and pass if each Assertion was checked at least one time.

Getting started

1. Create new scene
2. Open the Integration Tests runner window. (in menu bar, *Unity Test Tools/Game Tests Runner*, or *ctrl+alt+shift+t*)
3. Create new test
4. In the inspector, check the *Succeed on assertions* option.



5. Add a Sphere to the scene and attach a Rigidbody component to it
6. Attach an Assertion Component
7. Configure the Assertion Component as follows:
   Comparer: **FloatComparer**
   Check: **On Update**
   Value: **Sphere.Rigidbody.velocity.magnitude**
   Compare Type: **Less**
   Compare to type: **Compare to constant value**
   Constant value: **10**



   This can be read as: Make sure in every Update call that velocity of the Sphere is always lower than 10

8. Start the test from the *Integration Tests Runner* window (or use *ctrl+t* combination if the test is selected). The sphere will start to fall down and gain velocity. Once it's velocity reached 10, the assertion will fail, therefore the test will fail.