Write Low-Level Tests

Learn how to write Low-Level Tests in Unreal Engine, including naming conventions and best practices.

PREREQUISITE TOPICS

In order to understand and use the content on this page, make sure you are familiar with the following topics:

- Low-Level Tests
- Types of Low-Level Tests

This document discusses the following:

- Write Low-Level Tests, including:

Write Low-Level Tests

This page primarily discusses structure, guidelines, and best practices for writing **Low-Level Tests (LLTs)** with Catch2 in the context of **Unreal Engine (UE)**. See the Catch2 GitHub Repository for information specific to Catch2. For a complete guide on writing tests, see the Catch2 Reference.

> Be sure to use Unreal C++ coding conventions for tests. See the Unreal Coding Standard for more information.

Before You Begin

Review the Types of Low-Level Tests documentation for the following items:

- Ensure that your directories match the ones outlined in that document's steps.

Once you are ready to start writing test `.cpp` files, follow these steps:

1. Give your `.cpp` test file a descriptive name, such as `<NAME_OF_FILE>Test.cpp`
    - See the Naming Conventions section of this document for more information.
2. Include all necessary header files.
    - At minimum, you need to have optional `#include "CoreMinimal.h"` and `#include "TestHarness.h"`.
    - After you include the minimum headers, include only the headers that are necessary to complete your tests.

3. Now you can write your test using either the **TDD (Test-Driven Development)** or **BDD (Behavior-Driven Development)** paradigms.
    - BDD Test Example
    - TDD Test Example

Behavior Driven Development Test Example

BDD-style tests focus testing through a `SCENARIO`. A file can include multiple scenarios. The core structure of a scenario is:

- `GIVEN`: setup conditions
- `WHEN`: actions are performed
- `THEN`: expected result holds.

The `GIVEN` and `WHEN` sections can contain additional initialization and changes to internal state. The `THEN` section should perform checks to determine whether the desired result holds true. `CHECK` failures continue execution while `REQUIRE` stops execution of a single test.

The code example below provides a general outline of a BDD-style test in the Low-Level Tests framework in UE:

```
1
2  #include "CoreMinimal.h"
3  #include "TestHarness.h"
4
5  // Other includes must be placed after CoreMinimal.h and TestHarness.h, grouped by scope (std libraries, UE modules, third party etc)
6
7  /* A BDD-style test */
8
9  SCENARIO("Summary of test scenario", "[functional][feature][slow]") // Tags are placed in brackets []
10 {
11 GIVEN("Setup phase")
12 {
```

```
// Initialize variables, setup test preconditions etc

[...]

WHEN("I perform an action")
{
// Change internal state

[...]

THEN("Check for expectations")
{
REQUIRE(Condition_1);
REQUIRE(Condition_2);
// Not reached if previous require fails
CHECK(Condition_3);
}
}
}
}

```

Copy full snippet

Test Driven Development Test Example

TDD-style tests focus testing through a `TEST_CASE`. Each `TEST_CASE` can include code to set up the case being tested. The actual test case can then be broken down into multiple `SECTION` blocks. Each of the `SECTION` blocks performs checks to determine whether the desired result holds true. After all the checks are performed in `SECTION` blocks, the end of the `TEST_CASE` can include any necessary teardown code.

The code example below provides a general outline of a TDD-style test in the Low Level Tests framework in UE:

```

#include "CoreMinimal.h"
#include "TestHarness.h"

// Other includes must be placed after CoreMinimal.h and TestHarness.h, grouped by scope (std libraries, UE modules, third party etc)

/* Classic TDD-style test */

TEST_CASE("Summary of test case", "[unit][feature][fast]")
{
// Setup code for this test case

[...]

// Test can be divided into sections
SECTION("Test #1")
{
REQUIRE(Condition_1);
}

...

SECTION("Test #n")
{
REQUIRE(Condition_n);
}

// Teardown code for this test case

```

```
30 [...]
31
32 }
33
```

Copy full snippet

Test cases can also use double colon `::` notation to create a hierarchy in tests:

```
TEST_CASE("Organic::Tree::Apple::Throw an apple away from the tree") { ... }
```

Copy full snippet

The examples contained in this section are not exhaustive of all the features of Low-Level Tests or Catch2 in Unreal Engine. Generators, benchmarks, floating point approximation helpers, matchers, variable capturing, logging and more are all detailed in the external Catch2 Documentation.

More Examples

There are several UE-specific Low-Level Test examples in the engine directory `Engine/Source/Runtime/Core/Tests`. To continue the example from the Types of Low-Level Tests, you can see an example of TDD-style tests in the file `ArchiveReplaceObjectRefTests.cpp` located in `Engine/Source/Programs/LowLevelTests/Tests`.

Additional Low-Level Tests Features

Test Groups and Lifecycle Events

Grouping tests is a feature of UE's extended Catch2 library. By default, all test cases are grouped under a group with an empty name. To add a test to a group, specify its name as the first parameter and use `GROUP_*` versions of test cases:

```
GROUP_TEST_CASE("Apples", "Recipes::Baked::Pie::Cut slice", "[baking][recipe]")
GROUP_TEST_CASE_METHOD("Oranges", OJFixture, "Recipes::Raw::Juice Oranges", "[raw][recipe]")
GROUP_METHOD_AS_TEST_CASE("Pears", PoachInWine, "Recipes::Boiled::Poached Pears", "[desert][recipe]")
GROUP_REGISTER_TEST_CASE("Runtime", UnregisteredStaticMethod, "Dynamic", "[dynamic]")
```

Copy full snippet

For each group there are six lifecycle events that are self descriptive. The following code section illustrates these events:

```
GROUP_BEFORE_ALL("Apples") {
std::cout << "Called once before all tests in group Apples, use for one-time setup.\n";
}

GROUP_AFTER_ALL("Oranges") {
std::cout << "Called once after all tests in group Oranges, use for one-time cleanup.\n";
}

GROUP_BEFORE_EACH("Apples") {
std::cout << "Called once before each test in group Apples, use for repeatable setup.\n";
}

GROUP_AFTER_EACH("Oranges") {
std::cout << "Called once after each tests in group Oranges, use for repeatable cleanup.\n";
}

GROUP_BEFORE_GLOBAL() {
std::cout << "Called once before all groups, use for global setup.\n";
```

```
}

GROUP_AFTER_GLOBAL() {
std::cout << "Called once after all groups, use for global cleanup.\n";
}

GROUP_TEST_CASE("Apples", "Test #1") {
std::cout << "Apple #1\n";
}

GROUP_TEST_CASE("Apples", "Test #2") {
std::cout << "Apple #2\n";
}

GROUP_TEST_CASE("Oranges", "Test #1") {
std::cout << "Orange #1\n";
}

GROUP_TEST_CASE("Oranges", "Test #2") {
std::cout << "Orange #2\n";
}
```

Copy full snippet

This produces the output:

```
Called once before all groups, use for global setup.
Called once before all tests in group Apples, use for one-time setup.
Called once before each test in group Apples, use for repeatable setup.
Apple #1.
Called once before each test in group Apples, use for repeatable setup.
```

```
Apple #2.
Orange #1.
Called once after each tests in group Oranges, use for repeatable cleanup.
Orange #2.
Called once after each tests in group Oranges, use for repeatable cleanup.
Called once after all tests in group Oranges, use for one-time cleanup.
Called once after all groups, use for global cleanup.
```

Copy full snippet

Guidelines for Writing and Organizing Tests

Naming Conventions for Files and Folders

- Give your test files descriptive names.
  - If `SourceFile.cpp` is the source file you want to test, name your test file `SourceFileTest.cpp` or `SourceFileTests.cpp`.
- Mirror the tested module's folder structure.
  - `Alpha/Omega/SourceFile.cpp` maps to `Alpha/Omega/SourceFileTests.cpp` for explicit tests.
- Avoid using terms derived from unit test in test file names if they aren't unit tests — this definition is restrictive and misnaming can cause confusion.

A unit test should target the smallest testable unit — a class or a method — and run in seconds or less. The same principle applies for any other type of specialized test — integration, functional, smoke, end to end, performance, stress or load test. You can also place all unit tests in a **Unit** subfolder.

Explicit Tests Resources Folder

Test files, such as arbitrary binary files, assets files, or any other file-system based resource, must be placed into a *resource folder* for explicit tests. Set this folder in the `.Build.cs` module:

```
SetResourcesFolder("TestFilesResources");
```

Copy full snippet

When **Unreal Build Tool (UBT)** runs the platform deploy step, UBT copies this folder and its entire contents into the binary path so tests can relatively locate and load resources from it.

Best Practices

- Provide tags to test cases and scenarios.
  - Use consistent names and keep them short.
  - Use tags to your advantage. For example, you can choose to parallelize the run of tests tagged `[unit]` or tag all slow running tests to be run on a nightly build.
- Ensure each `SECTION` or `THEN` block includes at least one `REQUIRE` or a `CHECK`.
  - Tests that don't have expectations are useless.
- Use `REQUIRE` when test preconditions must be satisfied.
  - `REQUIRE` immediately stops on failure but `CHECK` doesn't.
- Design tests that are deterministic and fit a certain type.
  - Create and group tests by type whether they are unit, integration, functional, stress test etc.
- Tag slow tests with `[slow]` or `[performance]` if they are intended as performance tests.
  - This can be used to filter them out into a nightly build in the Continuous Integration/Continuous Delivery (CI/CD) pipeline.
- Ensure test code supports all platforms that the tested module requires.
  - For example, when working with the platform file system, use the `FPlatformFileManager` class, don't assume the test will run exclusively on a desktop platform.
- Use test groups and lifecycle events to initialize certain tests independently from others.
  - Refer to the test groups and lifecycle events section.

- Follow best practices for each type of test.
    - For example, unit tests should use mocking and not rely on external dependencies (other modules, a local database etc) and should not have order dependencies.
    - See the Low-Level Tests Overview for more information on different types of tests and their characteristics.

Next Step

Once you are finished writing your tests, see the Build and Run Low-Level Tests documentation to execute them.

Build and Run Low-Level Tests

Learn about the different ways to build and run Low-Level Tests.