Other languages: 繁體中文

Testing Framework

DrogonTest is a minimal testing framework built into drogon to enable easy asynchronous testing as well as synchronous ones. It is used for Drogon's own unittests and integration tests. But could also be used for testing applications built with Drogon. The syntax of DrogonTest is inspired by both GTest and Catch2.

You don't have to use DrogonTest for your application. Use whatever you are comfortable with. But it is an option.

Basic testing

Let's start with a simple example. You have a synchronous function that computes the sum of natural numbers up to a given value. And you want to test it for correctness.

```
// Tell DrogonTest to generate `test::run()`. Only defined this in the main file
#define DROGON_TEST_MAIN
#include <drogon/drogon_test.h>

int sum_all(int n)
{
    int result = 1;
    for(int i=2;i<n;i++) result += i;
    return result;
}

DROGON_TEST(Sum)
{
    CHECK(sum_all(1) == 1);
    CHECK(sum_all(2) == 3);
    CHECK(sum_all(3) == 6);
}

int main(int argc, char** argv)
{
    return drogon::test::run(argc, argv);
}
```

Compile and run... Well, it passed but there's an obvious bug isn't it. `sum_all(0)` should have been 0. We can add that to our test

```
DROGON_TEST(Sum)
{
    CHECK(sum_all(0) == 0);
    CHECK(sum_all(1) == 1);
    CHECK(sum_all(2) == 3);
    CHECK(sum_all(3) == 6);
}
```

Now the test fails with:

```
In test case Sum
↳ /path/to/your/test/main.cc:47  FAILED:
  CHECK(sum_all(0) == 0)
With expansion
  1 == 0
```

Notice the framework printed the test that failed and the actual value at both ends of the expression. Allows us to see what's going on immediately. And the solution is simple:

```
int sum_all(int n)
{
    int result = 0;
    for(int i=1;i<n;i++) result += i;
    return result;
}
```

Types of assertions

DrogonTest comes with a variety of assertions and actions. The basic `CHECK()` simply checks if the expression evaluates to true. If not, it prints to console. `CHECK_THROWS()` checks if the expression throws an exception. If it didn't, print to console. etc.. On the other hand `REQUIRE()` checks if a expression is true. Then return if not, preventing expressions after the test being executed.

action if fail/expression	is true	throws	does not throw	throws certain type
nothing	CHECK	CHECK_THROWS	CHECK_NOTHROW	CHECK_THROWS_AS
return	REQUIRE	REQUIRE_THROWS	REQUIRE_NOTHROW	REQUIRE_THROWS_AS
co_return	CO_REQUIRE	CO_REQUIRE_THROWS	CO_REQUIRE_NOTHROW	CO_REQUIRE_THROWS_AS
kill process	MANDATE	MANDATE_THROWS	MANDATE_NOTHROW	MANDATE_THROWS_AS

Let's try a slightly practical example. Let's say you're testing if the content of a file is what you're expecting. There's no point to further test if the program failed to open the file. So, we can use `REQUIRE` to shorten and reduce duplicated code.

```
DROGON_TEST(TestContent)
{
    std::ifstream in("data.txt");
    REQUIRE(in.is_open());
    // Instead of
    // CHECK(in.is_open() == true);
    // if(in.is_open() == false)
    //    return;

    ...
}
```

Likewise, `CO_REQUIRE` is like REQUIRE. But for coroutines. And `MANDATE` can be used when an operation failed and it modifies an unrecoverable global state. Which the only logical thing to do is to stop testing completely.

Asynchronous testing

Drogon is a asynchronous web framework. It only follows DrogonTest supports testing asynchronous functions. DrogonTest tracks the testing context through the `TEST_CTX` variable. Simply capture the variable **by value**. For example, testing if a remote API is successful and returns a JSON.

```
DROGON_TEST(RemoteAPITest)
{
    auto client = HttpClient::newHttpClient("http://localhost:8848");
    auto req = HttpRequest::newHttpRequest();
    req->setPath("/");
    client->sendRequest(req, [TEST_CTX](ReqResult res, const HttpResponsePtr&
resp) {
        // There's nothing we can do if the request didn't reach the server
        // or the server generated garbage.
        REQUIRE(res == ReqResult::Ok);
        REQUIRE(resp != nullptr);

        CHECK(resp->getStatusCode() == k200OK);
        CHECK(resp->contentType() == CT_APPLICATION_JSON);
    });
}
```

Coroutines have to be wrapped inside `AsyncTask` or called through `sync_wait` due to no native support of coroutines and C++14/17 compatibility in the testing framework.

```
DROGON_TEST(RemoteAPITestCoro)
{
    auto api_test = [TEST_CTX]() {
        auto client = HttpClient::newHttpClient("http://localhost:8848");
        auto req = HttpRequest::newHttpRequest();
        req->setPath("/");

        auto resp = co_await client->sendRequestCoro(req);
        CO_REQUIRE(resp != nullptr);
        CHECK(resp->getStatusCode() == k200OK);
        CHECK(resp->contentType() == CT_APPLICATION_JSON);
    };

    sync_wait(api_test());
}
```

Starting Drogon's event loop

Some tests need Drogon's event loop running. For example, unless specified, HTTP clients runs on Drogon's global event loop. The following boilerplate handles many edge cases and guarantees the event loop is running before any test starts.

```
int main()
{
    std::promise<void> p1;
    std::future<void> f1 = p1.get_future();

    // Start the main loop on another thread
    std::thread thr([&]() {
        // Queues the promise to be fulfilled after starting the loop
        app().getLoop()->queueInLoop([&p1]() { p1.set_value(); });
        app().run();
    });

    // The future is only satisfied after the event loop started
    f1.get();
    int status = test::run(argc, argv);

    // Ask the event loop to shutdown and wait
    app().getLoop()->queueInLoop([]() { app().quit(); });
    thr.join();
    return status;
}
```

CMake integration

Like most testing frameworks, DrogonTest can integrate itself into CMake. The `ParseAndAddDrogonTests` function adds tests it sees in the source file to CMake's CTest framework.

```
find_package(Drogon REQUIRED) # also loads ParseAndAddDrogonTests
add_executable(mytest main.cpp)
target_link_libraries(mytest PRIVATE Drogon::Drogon)
ParseAndAddDrogonTests(mytest)
```

Now the test could be ran through build system (Makefile in this case).

```
❯ make test
Running tests...
Test project path/to/your/test/build/
      Start  1: Sum
 1/1  Test  #1: Sum ................................   Passed    0.00 sec
```