Qt Test Overview

Qt Test is a framework for unit testing Qt based applications and libraries. Qt Test provides all the functionality commonly found in unit testing frameworks as well as extensions for testing graphical user interfaces.

Qt Test is designed to ease the writing of unit tests for Qt based applications and libraries:

Feature	Details
Lightweight	Qt Test consists of about 6000 lines of code and 60 exported symbols.
Self-contained	Qt Test requires only a few symbols from the Qt Core module for non-gui testing.
Rapid testing	Qt Test needs no special test-runners; no special registration for tests.
Data-driven testing	A test can be executed multiple times with different test data.
Basic GUI testing	Qt Test offers functionality for mouse and keyboard simulation.
Benchmarking	Qt Test supports benchmarking and provides several measurement back-ends.
IDE friendly	Qt Test outputs messages that can be interpreted by Qt Creator, Visual Studio, and KDevelop.
Thread-safety	The error reporting is thread safe and atomic.
Type-safety	Extensive use of templates prevent errors introduced by implicit type casting.
Easily extendable	Custom types can easily be added to the test data and test output.

You can use a Qt Creator wizard to create a project that contains Qt tests and build and run them directly from Qt Creator. For more information, see Running Autotests.

Creating a Test

To create a test, subclass QObject and add one or more private slots to it. Each private slot is a test function in your test. QTest::qExec() can be used to execute all test functions in the test object.

In addition, you can define the following private slots that are *not* treated as test functions. When present, they will be executed by the testing framework and can be used to initialize and clean up either the entire test or the

current test function.

- `initTestCase()` will be called before the first test function is executed.
- `initTestCase_data()` will be called to create a global test data table.
- `cleanupTestCase()` will be called after the last test function was executed.
- `init()` will be called before each test function is executed.
- `cleanup()` will be called after every test function.

Use `initTestCase()` for preparing the test. Every test should leave the system in a usable state, so it can be run repeatedly. Cleanup operations should be handled in `cleanupTestCase()`, so they get run even if the test fails.

Use `init()` for preparing a test function. Every test function should leave the system in a usable state, so it can be run repeatedly. Cleanup operations should be handled in `cleanup()`, so they get run even if the test function fails and exits early.

Alternatively, you can use RAII (resource acquisition is initialization), with cleanup operations called in destructors, to ensure they happen when the test function returns and the object moves out of scope.

If `initTestCase()` fails, no test function will be executed. If `init()` fails, the following test function will not be executed, the test will proceed to the next test function.

Example:

```
class MyFirstTest: public QObject
{
    Q_OBJECT

private:
    bool myCondition()
    {
        return true;
    }

private slots:
    void initTestCase()
    {
        qDebug("Called before everything else.");
    }

    void myFirstTest()
    {
        QVERIFY(true); // check that a condition is satisfied
        QCOMPARE(1, 1); // compare two values
    }

    void mySecondTest()
    {
        QVERIFY(myCondition());
        QVERIFY(1 != 2);
    }

    void cleanupTestCase()
    {
        qDebug("Called after myFirstTest and mySecondTest.");
    }
};
```

Finally, if the test class has a static public `void initMain()` method, it is called by the QTEST_MAIN macros before the QApplication object is instantiated. For example, this allows for setting application attributes like Qt::AA_DisableHighDpiScaling. This was added in 5.14.

For more examples, refer to the Qt Test Tutorial.

Building a Test

You can build an executable that contains one test class that typically tests one class of production code. However, usually you would want to test several classes in a project by running one command.

See Writing a Unit Test for a step by step explanation.

Building with CMake and CTest

You can use Building with CMake and CTest to create a test. CTest enables you to include or exclude tests based on a regular expression that is matched against the test name. You can further apply the LABELS property to a test and CTest can then include or exclude tests based on those labels. All labeled targets will be run when `test` target is called on the command line.

There are several other advantages with CMake. For example, the result of a test run can be published on a web server using CDash with virtually no effort.

CTest scales to very different unit test frameworks, and works out of the box with QTest.

The following is an example of a CMakeLists.txt file that specifies the project name and the language used (here, *mytest* and C++), the Qt modules required for building the test (Qt5Test), and the files that are included in the test (*tst_mytest.cpp*).

```
project(mytest LANGUAGES CXX)

find_package(Qt5Test REQUIRED)

set(CMAKE_INCLUDE_CURRENT_DIR ON)

set(CMAKE_AUTOMOC ON)

enable_testing(true)

add_executable(mytest tst_mytest.cpp)
add_test(NAME mytest COMMAND mytest)

target_link_libraries(mytest PRIVATE Qt5::Test)
```

For more information about the options you have, see Build with CMake.

Building with qmake

If you are using qmake as your build tool, just add the following to your project file:

```
QT += testlib
```

If you would like to run the test via `make check`, add the additional line:

```
CONFIG += testcase
```

See the qmake manual for more information about `make check`.

Building with Other Tools

If you are using other build tools, make sure that you add the location of the Qt Test header files to your include path (usually `include/QtTest` under your Qt installation directory). If you are using a release build of Qt, link your test to the `QtTest` library. For debug builds, use `QtTest_debug`.

Qt Test Command Line Arguments

Syntax

The syntax to execute an autotest takes the following simple form:

```
testname [options] [testfunctions[:testdata]]...
```

Substitute `testname` with the name of your executable. `testfunctions` can contain names of test functions to be executed. If no `testfunctions` are passed, all tests are run. If you append the name of an entry in `testdata`, the test function will be run only with that test data.

For example:

```
/myTestDirectory$ testQString toUpper
```

Runs the test function called `toUpper` with all available test data.

```
/myTestDirectory$ testQString toUpper toInt:zero
```

Runs the `toUpper` test function with all available test data, and the `toInt` test function with the test data called `zero` (if the specified test data doesn't exist, the associated test will fail).

```
/myTestDirectory$ testMyWidget -vs -eventdelay 500
```

Runs the `testMyWidget` function test, outputs every signal emission and waits 500 milliseconds after each

Runs the `testMyWidget` function test, outputs every signal emission and waits 500 milliseconds after each simulated mouse/keyboard event.

Options

Logging Options

The following command line options determine how test results are reported:

- `-o` *filename,format*
  Writes output to the specified file, in the specified format (one of `txt`, `xml`, `lightxml`, `junitxml` or `tap`). The special filename `-` may be used to log to standard output.
- `-o` *filename*
  Writes output to the specified file.
- `-txt`
  Outputs results in plain text.
- `-xml`
  Outputs results as an XML document.
- `-lightxml`
  Outputs results as a stream of XML tags.
- `-junitxml`
  Outputs results as an JUnit XML document.
- `-csv`
  Outputs results as comma-separated values (CSV). This mode is only suitable for benchmarks, since it suppresses normal pass/fail messages.
- `-teamcity`
  Outputs results in TeamCity format.
- `-tap`
  Outputs results in Test Anything Protocol (TAP) format.

The first version of the `-o` option may be repeated in order to log test results in multiple formats, but no more than one instance of this option can log test results to standard output.

If the first version of the `-o` option is used, neither the second version of the `-o` option nor the `-txt`, `-xml`, `-lightxml`, `-teamcity`, `-junitxml` or `-tap` options should be used.

If neither version of the `-o` option is used, test results will be logged to standard output. If no format option is used, test results will be logged in plain text.

Test Log Detail Options

The following command line options control how much detail is reported in test logs:

- `-silent`
  Silent output; only shows fatal errors, test failures and minimal status messages.
- `-v1`
  Verbose output; shows when each test function is entered. (This option only affects plain text output.)
- `-v2`
  Extended verbose output; shows each QCOMPARE() and QVERIFY(). (This option affects all output formats and implies `-v1` for plain text output.)
- `-vs`
  Shows all signals that get emitted and the slot invocations resulting from those signals. (This option affects all output formats.)

Testing Options

The following command line options influence how tests are run:

The following command-line options influence how tests are run:

- `-functions`
  Outputs all test functions available in the test, then quits.
- `-datatags`
  Outputs all data tags available in the test. A global data tag is preceded by ' __global__ '.
- `-eventdelay` *ms*
  If no delay is specified for keyboard or mouse simulation (QTest::keyClick(), QTest::mouseClick() etc.), the value from this parameter (in milliseconds) is substituted.
- `-keydelay` *ms*
  Like -eventdelay, but only influences keyboard simulation and not mouse simulation.
- `-mousedelay` *ms*
  Like -eventdelay, but only influences mouse simulation and not keyboard simulation.
- `-maxwarnings` *number*
  Sets the maximum number of warnings to output. 0 for unlimited, defaults to 2000.
- `-nocrashhandler`
  Disables the crash handler on Unix platforms. On Windows, it re-enables the Windows Error Reporting dialog, which is turned off by default. This is useful for debugging crashes.
- `-platform` *name*
  This command line argument applies to all Qt applications, but might be especially useful in the context of auto-testing. By using the "offscreen" platform plugin (-platform offscreen) it's possible to have tests that use QWidget or QWindow run without showing anything on the screen. Currently the offscreen platform plugin is only fully supported on X11.

Benchmarking Options

The following command line options control benchmark testing:

- `-callgrind`
  Uses Callgrind to time benchmarks (Linux only).
- `-tickcounter`
  Uses CPU tick counters to time benchmarks.
- `-eventcounter`
  Counts events received during benchmarks.
- `-minimumvalue` *n*
  Sets the minimum acceptable measurement value.
- `-minimumtotal` *n*
  Sets the minimum acceptable total for repeated executions of a test function.
- `-iterations` *n*
  Sets the number of accumulation iterations.
- `-median` *n*
  Sets the number of median iterations.
- `-vb`
  Outputs verbose benchmarking information.

Miscellaneous Options

- `-help`
  Outputs the possible command line arguments and gives some useful help.

Creating a Benchmark

To create a benchmark, follow the instructions for creating a test and then add a QBENCHMARK macro or QTest::setBenchmarkResult() to the test function that you want to benchmark. In the following code snippet, the macro is used:

```
class MyFirstBenchmark: public QObject
{
    Q_OBJECT
private slots:
    void myFirstBenchmark()
    {
        QString string1;
        QString string2;
        QBENCHMARK {
            string1.localeAwareCompare(string2);
        }
    }
};
```

A test function that measures performance should contain either a single QBENCHMARK macro or a single call to `setBenchmarkResult()`. Multiple occurrences make no sense, because only one performance result can be reported per test function, or per data tag in a data-driven setup.

Avoid changing the test code that forms (or influences) the body of a QBENCHMARK macro, or the test code that computes the value passed to `setBenchmarkResult()`. Differences in successive performance results should ideally be caused only by changes to the product you are testing. Changes to the test code can potentially result in misleading report of a change in performance. If you do need to change the test code, make that clear in the commit message.

In a performance test function, the QBENCHMARK or `setBenchmarkResult()` should be followed by a verification step using QCOMPARE(), QVERIFY(), and so on. You can then flag a performance result as *invalid* if another code path than the intended one was measured. A performance analysis tool can use this information to filter out invalid results. For example, an unexpected error condition will typically cause the program to bail out prematurely from the normal program execution, and thus falsely show a dramatic performance increase.

Selecting the Measurement Back-end

The code inside the QBENCHMARK macro will be measured, and possibly also repeated several times in order to get an accurate measurement. This depends on the selected measurement back-end. Several back-ends are available. They can be selected on the command line:

Name	Command-line Argument	Availability
Walltime	(default)	All platforms
CPU tick counter	-tickcounter	Windows, macOS, Linux, many UNIX-like systems.
Event Counter	-eventcounter	All platforms
Valgrind Callgrind	-callgrind	Linux (if installed)
Linux Perf	-perf	Linux

In short, walltime is always available but requires many repetitions to get a useful result. Tick counters are usually available and can provide results with fewer repetitions, but can be susceptible to CPU frequency scaling issues. Valgrind provides exact results, but does not take I/O waits into account, and is only available on a limited number of platforms. Event counting is available on all platforms and it provides the number of events that were received by the event loop before they are sent to their corresponding targets (this might include non-Qt events).

The Linux Performance Monitoring solution is available only on Linux and provides many different counters, which can be selected by passing an additional option `-perfcounter countername`, such as `-`

`perfcounter cache-misses`, `-perfcounter branch-misses`, or `-perfcounter l1d-load-misses`. The default counter is `cpu-cycles`. The full list of counters can be obtained by running any benchmark executable with the option `-perfcounterlist`.

- Using the performance counter may require enabling access to non-privileged applications.
- Devices that do not support high-resolution timers default to one-millisecond granularity.

See Writing a Benchmark in the Qt Test Tutorial for more benchmarking examples.

Using Global Test Data

You can define `initTestCase_data()` to set up a global test data table. Each test is run once for each row in the global test data table. When the test function itself is data-driven, it is run for each local data row, for each global data row. So, if there are g rows in the global data table and d rows in the test's own data-table, the number of runs of this test is g times d.

Global data is fetched from the table using the QFETCH_GLOBAL() macro.

The following are typical use cases for global test data:

- Selecting among the available database backends in QSql tests to run every test against every database.
- Doing all networking tests with and without SSL (HTTP versus HTTPS) and proxying.
- Testing a timer with a high precision clock and with a coarse one.
- Selecting whether a parser shall read from a QByteArray or from a QIODevice.

For example, to test each number provided by `roundTripInt_data()` with each locale provided by `initTestCase_data()`:

```
void TestQLocale::roundTripInt()
{
    QFETCH_GLOBAL(QLocale, locale);
    QFETCH(int, number);
    bool ok;
    QCOMPARE(locale.toInt(locale.toString(number), &ok), number);
    QVERIFY(ok);
}
```