Running Autotests

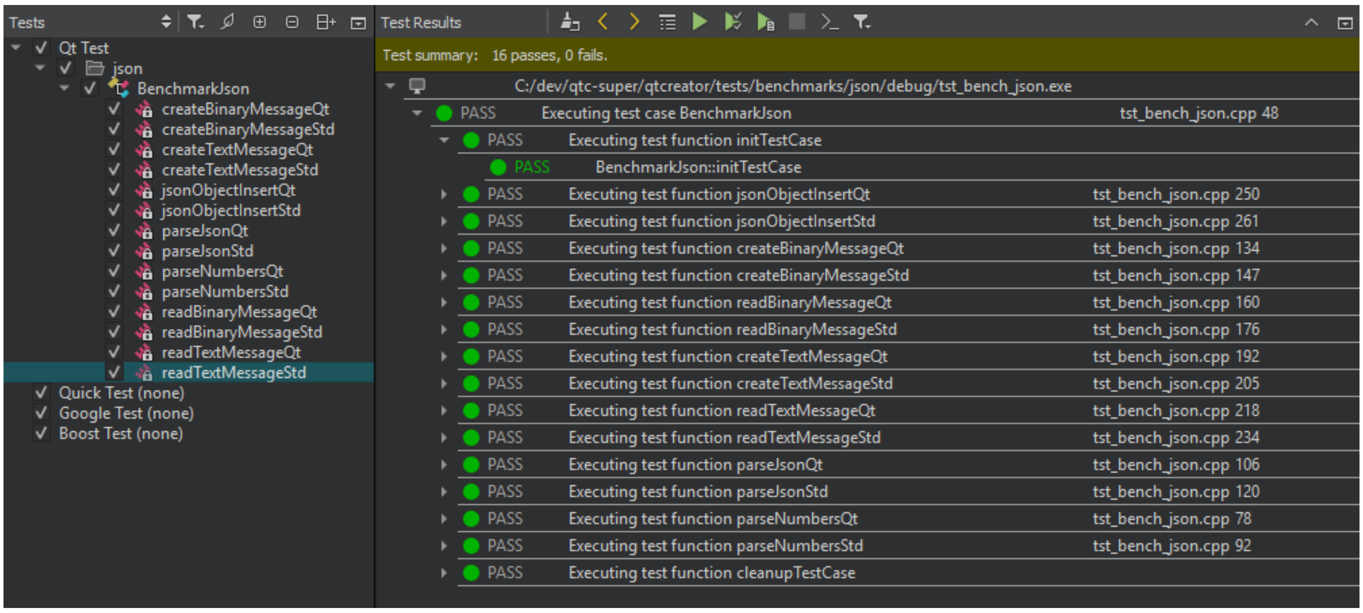
Qt Creator supports both *code based tests* and *build system based tests*. Code based testing provides special handling for particular testing frameworks that is strongly tied to the underlying code models or specialized parsers. Build system based testing is independent from any testing frameworks. It retrieves information directly from the underlying build system and uses it or even the build system as such to execute the respective tests.

Qt Creator integrates the following testing frameworks for unit testing applications and libraries:

- Boost.Test
- Catch2 test framework
- Google C++ Testing Framework
- Qt Test framework

Additional build system based support is provided for CTest.

You can use Qt Creator to create, build, and run code based tests for your projects.



```
Qt Test
  json
    BenchmarkJson
      createBinaryMessageQt
      createBinaryMessageStd
      createTextMessageQt
      createTextMessageStd
      jsonObjectInsertQt
      jsonObjectInsertStd
      parseJsonQt
      parseJsonStd
      parseNumbersQt
      parseNumbersStd
      readBinaryMessageQt
      readBinaryMessageStd
      readTextMessageQt
      readTextMessageStd
Quick Test (none)
Google Test (none)
Boost Test (none)
```

```
Test summary:  16 passes, 0 fails.
C:/dev/qtc-super/qtcreator/tests/benchmarks/json/debug/tst_bench_json.exe
PASS  Executing test case BenchmarkJson                           tst_bench_json.cpp 48
  PASS  Executing test function initTestCase
    PASS  BenchmarkJson::initTestCase
  PASS  Executing test function jsonObjectInsertQt                tst_bench_json.cpp 250
  PASS  Executing test function jsonObjectInsertStd               tst_bench_json.cpp 261
  PASS  Executing test function createBinaryMessageQt             tst_bench_json.cpp 134
  PASS  Executing test function createBinaryMessageStd            tst_bench_json.cpp 147
  PASS  Executing test function readBinaryMessageQt               tst_bench_json.cpp 160
  PASS  Executing test function readBinaryMessageStd              tst_bench_json.cpp 176
  PASS  Executing test function createTextMessageQt               tst_bench_json.cpp 192
  PASS  Executing test function createTextMessageStd              tst_bench_json.cpp 205
  PASS  Executing test function readTextMessageQt                 tst_bench_json.cpp 218
  PASS  Executing test function readTextMessageStd                tst_bench_json.cpp 234
  PASS  Executing test function parseJsonQt                       tst_bench_json.cpp 106
  PASS  Executing test function parseJsonStd                      tst_bench_json.cpp 120
  PASS  Executing test function parseNumbersQt                    tst_bench_json.cpp 78
  PASS  Executing test function parseNumbersStd                   tst_bench_json.cpp 92
  PASS  Executing test function cleanupTestCase
```

Build System Based Tests

The handling of build system based tests is disabled by default to avoid interference with code based parsers. To enable build system based tests, select the respective test tool in **Preferences** > **Testing** > **General**.

The detection of tests is usually much faster for build system based tests as this normally does not involve

If you have enabled code based and build system based tests together you may duplicate tests inside the tests tree. See also Selecting Tests to Run.

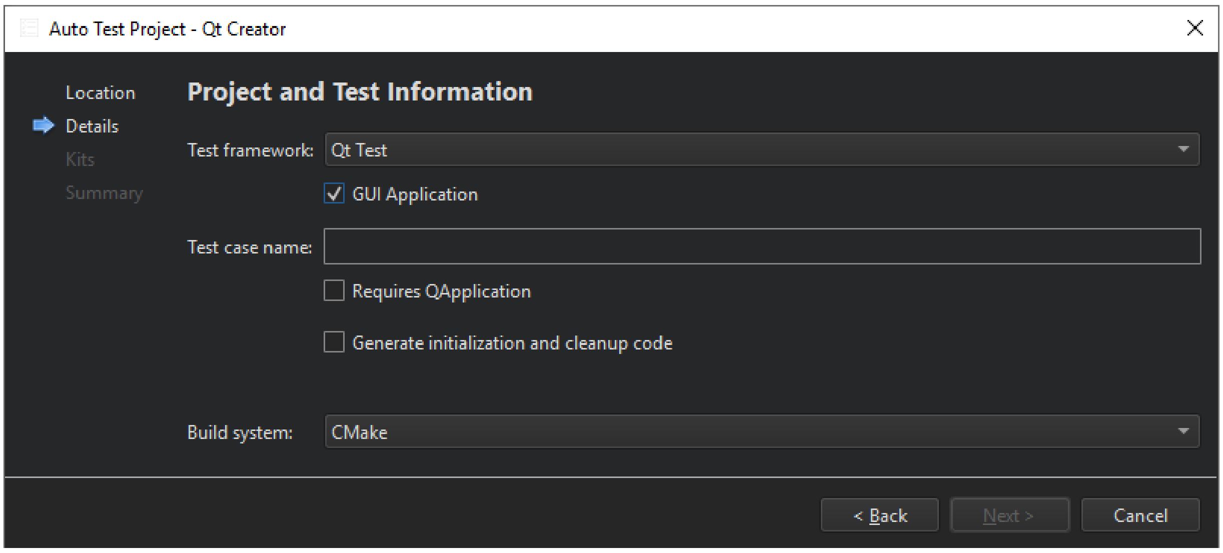
Creating Tests

You can use a wizard to create projects that contain tests.

Creating Qt and Qt Quick Tests

To create a Qt or Qt Quick test:

1. Select **File** > **New Project** > **Other Project** > **Auto Test Project** > **Choose** to create a project with boilerplate code for a Qt test or a Qt Quick test.
2. In the **Project and Test Information** dialog, specify settings for the project and test:
    1. In the **Test framework** field, select **Qt Test** or **Qt Quick Test**.
    2. For a Qt test, select the **GUI Application** check box to create a Qt application.



Project and Test Information

Test framework: Qt Test

[✓] GUI Application

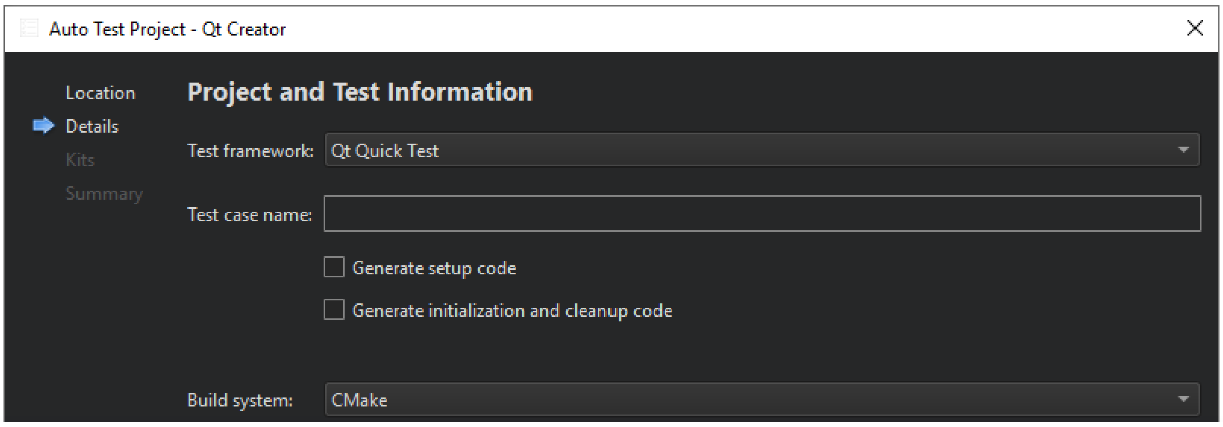
Test case name:

[ ] Requires QApplication

[ ] Generate initialization and cleanup code

Build system: CMake

    3. In the **Test case name** field, enter a name for the test case.
    4. For a Qt test, select the **Requires QApplication** check box to add the include statement for QApplication to the main.cpp file of the project.
    5. For a Qt Quick test, select the **Generate setup code** check box to execute C++ code before any of the QML tests are run. The testing framework will call slots and invokable functions, as described in Executing C++ Before QML Tests.



Project and Test Information

Test framework: Qt Quick Test

Test case name:

[ ] Generate setup code

[ ] Generate initialization and cleanup code

Build system: CMake

6. Select the **Generate initialization and cleanup code** checkbox to add functions to your test that are executed by the testing framework to initialize and clean up the test.
7. In the **Build system** field, select the build system to use for building the project: qmake, CMake, or Qbs.

Qt Creator creates the test in the specified project directory. Edit the .cpp file to add private slots for each test function in your test. For more information about creating Qt tests, see Creating a Test.

Creating Google Tests

To create a Google test:

1. Select **File** > **New Project** > **Other Project** > **Auto Test Project** > **Choose** to create a project with boilerplate code for a Google test.
2. In the **Project and Test Information** dialog, specify settings for the project and test:
    1. In the **Test framework** field, select **Google Test**.
    2. In the **Test suite name** field, enter a name for the test suite.
    3. In the **Test case name** field, enter a name for the test case.
    4. Select the **Enable C++ 11** check box to support C++ 11 features in the test.
    5. In the **Google test repository** field, select a directory that contains a clone of the googletest repository.

       To use an installed Google C++ Testing framework instead, see Setting Up the Google C++ Testing Framework.
    6. In the **Build system** field, select the build system to use for building the project: qmake, CMake, or Qbs.

Qt Creator creates the test in the specified project directory. For more information about creating Google tests, see the Google Test Primer.

Creating Boost Tests

To build and run Boost tests, you must have the Boost.Test installed on the development host. Typically, it is installed when you install Boost. You can download Boost from Boost.org.

If Boost libraries can be found by the used compiler and build system, you do not need to specify the include directory when creating the test.

To create a Boost test:

1. Select **File** > **New Project** > **Other Project** > **Auto Test Project** > **Choose** to create a project with boilerplate code for a Boost test.
2. In the **Project and Test Information** dialog, specify settings for the project and test:
    1. In the **Test framework** field, select **Boost Test**.
    2. In the **Test suite name** field, enter a name for the test suite.
    3. In the **Test case name** field, enter a name for the test case.
    4. In the **Boost include dir (optional)** field, enter the path to the directory that contains files needed by Boost.Test, such as *version.hpp* and a subfolder called *test* that contains the test header files.
    5. In the **Build system** field, select the build system to use for building the project: qmake, CMake, or Qbs.

Qt Creator creates the test in the specified project directory. For more information about creating Boost tests, see

To build and run Catch2 tests, you either must have Catch2 libraries and headers installed, or you can use the single include header file provided by the Catch2 repository.

If the Catch2 headers can be found by the used compiler and build system automatically, you do not need to specify the include directory when creating the test.

To create a basic Catch2 test:

1. Select **File** > **New Project** > **Other Project** > **Auto Test Project** > **Choose** to create a project with boilerplate code for a Catch2 test.
2. In the **Project and Test Information** dialog, specify settings for the project and test:
    1. In the **Test framework** field, select **Catch2**.
    2. In the **Test case name** field, specify a name to be used for the test case file.
    3. Select the **Use Qt libraries** check box to use a self defined main function and set up the project to use Qt features.
    4. In the **Catch2 include directory (optional)** field, you may enter a path to the directory that contains the Catch2 header files.
    5. In the **Build system** field, select the build system to use for building the project: qmake, CMake, or Qbs.

Qt Creator creates the test in the specified project directory. For more information about creating Catch2 tests, see Catch2.

Creating CTest Based Tests

CTest provides capabilities to execute tests for CMake based projects and is not limited to a special test framework. You simply configure tests inside the project files, usually CMakeLists.txt. Basically this is done by enabling testing for the project and registering the test applications or even special commands.

```
enable_testing()
add_test(NAME test_example COMMAND test_example)
```

`test_example` must of course be added as an executable before trying to register it as test or it may be any command that can be executed including arguments. For detailed information on how to use CTest see Testing with CTest.

Setting Up the Google C++ Testing Framework

To build and run Google tests, you must have the Google C++ Testing framework installed and configured on the development host. You can either clone it from Git Hub or install it from an installation package.

To configure a project to use a cloned Google testing framework, edit the `INCLUDEPATH` variable in the project file (.pro) to include the source and `include` folders of Google Test's `googletest` and `googlemock`. Usually, you need to add the following subfolders:

- `googletest`
- `googlemock`
- `googletest/include`

- `googletest/src/gtest-all.cc`
- `googlemock/src/gmock-all.cc`

To configure a project to use an installed Google testing framework package, add the following include paths to the .pro file:

- `<googletest_install_path>/include/gtest`
- `<googletest_install_path>/include/gmock`

Then add linker options to be able to find the libraries and to link against them. For example, for qmake based projects, you typically need to add the following values to the .pro file:

- `LIBS += -lgtest -L<path_to_gtest_lib>`
- `LIBS += -lgmock -L<path_to_gmock_lib>`

Building and Running Tests

To build and run tests:

1. Open a project that contains tests.
2. In the **Tests** view, select the tests to run.
3. In the **Test Results**, select:
   - (**Run All Tests**) to run all tests.
   - (**Run Selected Tests**) to run the selected tests.
   - (**Run Failed Tests**) to re-run the tests which failed in the last run. Depending on the framework this may select additional tests if it is impossible to distinguish or to fully address the test.
   - (**Run Tests for Current File**) to run the tests in the file currently open in the code editor.

By default, Qt Creator builds a project before deploying and running it. To run all tests without building and deploying them again, select **Run All Tests Without Deployment** in the context menu. To run the selected tests without deployment, select **Run Selected Tests Without Deployment**.

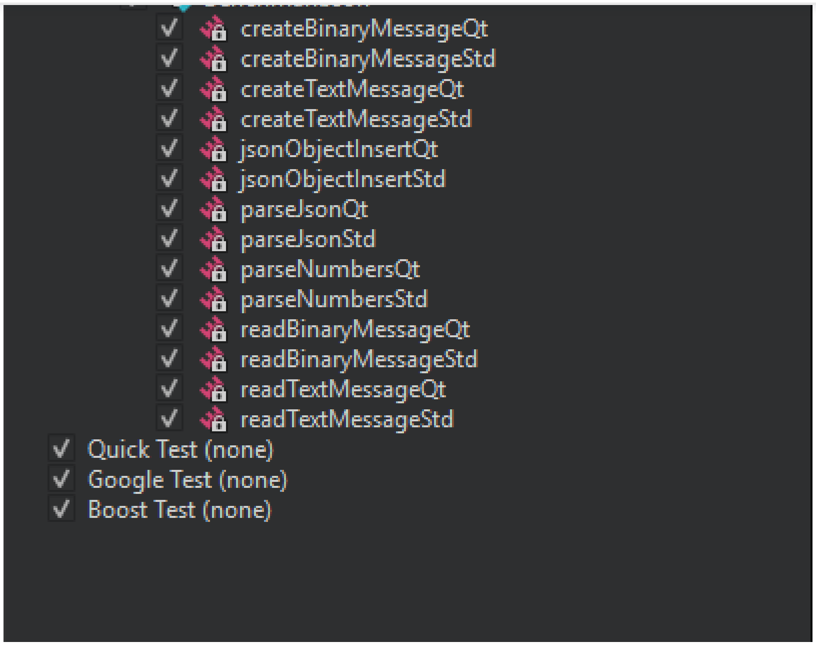
The functions to run tests are also available in the context menu in the **Tests** view and in **Tools** > **Tests**.

> **Note:** If you have enabled build system based and code based tests, you may run tests twice when using **Run All Tests** or **Run Selected Tests**. This happens if the tests can be found by the code based test frameworks and are registered as test with the build system.

If a test takes more than a minute to execute, the default timeout might stop the test execution. To increase the timeout, select **Edit** > **Preferences** > **Testing** > **General**.

Selecting Tests to Run

The **Tests** view shows all the tests found for the currently active test frameworks in the current project. Select the test cases to run.



If a Qt Quick test case does not have a name, it is marked **Unnamed** in the list. Unnamed test cases are executed when you select **Run All Tests**. You cannot select or deselect them.

Qt Creator scans the project for tests when you open the project and updates the test list for the currently active test frameworks when you edit tests. To refresh the view, select **Rescan Tests** in the context menu.

To show or hide init and cleanup or data functions in the **Tests** view, select (**Filter Test Tree**), and then select **Show Init and Cleanup Functions** or **Show Data Functions**. Double-click a function in the list to open its source code in the code editor.

The test cases are listed in alphabetic, case insensitive order. To list them in the order in which they are defined in the source code, select (**Sort Naturally**).

Running and Debugging Tests from Code Editor

You can run and debug tests in the file currently open in the code editor. To run all tests in the open file, select **Tools** > **Tests** > **Run Tests for Current File**.

Note: Available only for code based test frameworks.

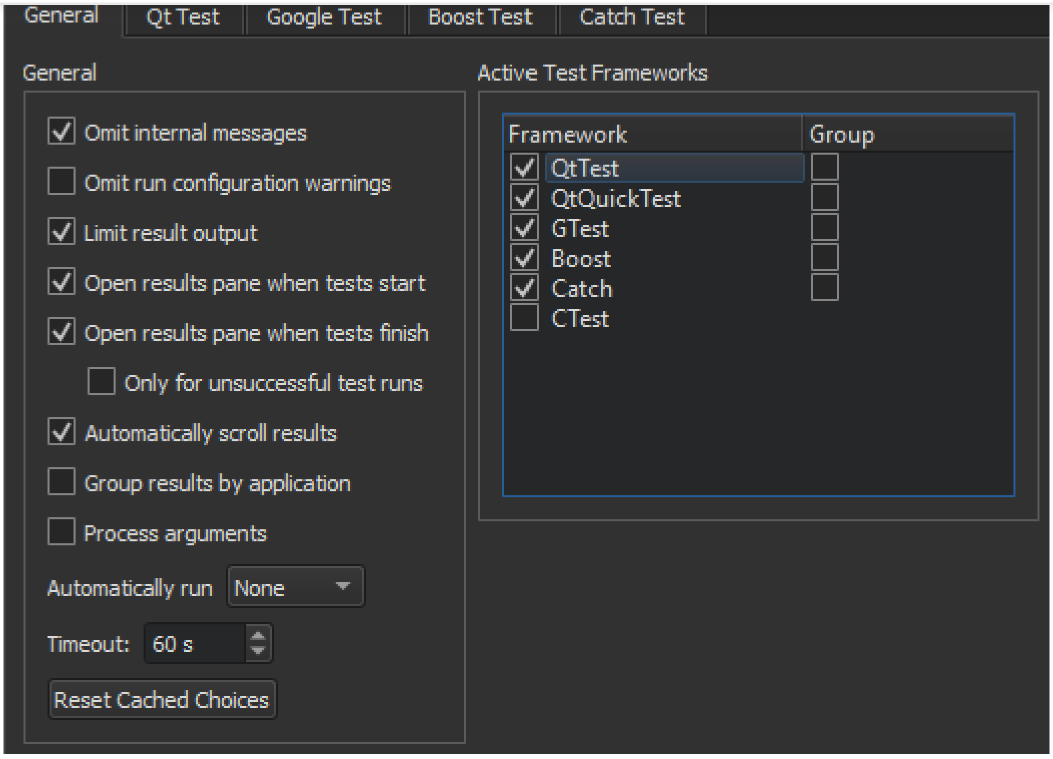
To run the test currently selected in the open file, select **Run Test Under Cursor** in the context menu.

Note: Available only for code based test frameworks.

To debug the currently selected test, select **Debug Test Under Cursor** in the context menu.

Note: Availability for build system based tests depends on the build system tool's abilities.

Specifying Test Settings


General | Qt Test | Google Test | Boost Test | Catch Test

General

- ✓ Omit internal messages
- ☐ Omit run configuration warnings
- ✓ Limit result output
- ✓ Open results pane when tests start
- ✓ Open results pane when tests finish
  - ☐ Only for unsuccessful test runs
- ✓ Automatically scroll results
- ☐ Group results by application
- ☐ Process arguments

Automatically run: None

Timeout: 60 s

Reset Cached Choices

Active Test Frameworks

Framework	Group
✓ QtTest	☐
✓ QtQuickTest	☐
✓ GTest	☐
✓ Boost	☐
✓ Catch	☐
☐ CTest	

You can customize some settings at project level. To change settings for the current project instead of globally, select **Projects** > **Project Settings** > **Testing**.

In the **Active Test Frameworks** list you can select which tests Qt Creator will handle. To improve the performance of full scans for tests, disable test frameworks you are not using.

To group related test cases for an active test framework, select the **Group** check box next to the framework name in the **Active Test Frameworks** list. By default, tests are grouped based on the directory where they are located.

Internal messages and run configuration warnings for deduced configurations are omitted by default. To view them, deselect the **Omit internal messages** and **Omit run configuration warnings** check boxes.

By default, test result output is limited to 100,000 characters. The view is automatically scrolled down when new results are added. To display full results, deselect the **Limit result output** check box. To disable automatic scrolling, deselect the **Automatically scroll results** check box.

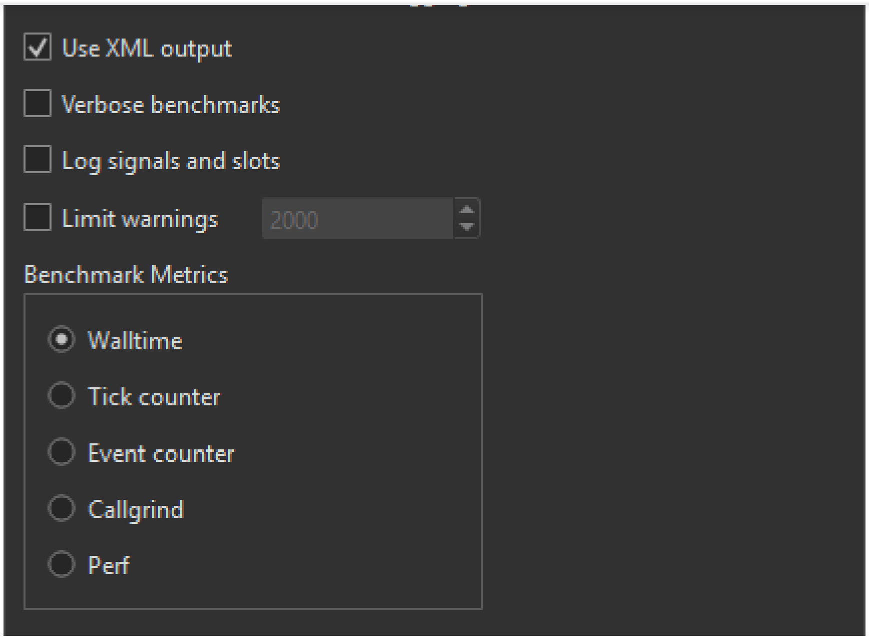
Test results can be grouped by the executable path that was used to run the tests. This is useful if you have multiple test executables and run them all at once. To enable this functionality you need to select the **Group results by application** check box.

It is possible to automatically run the currently available tests after successfully building the current project. In **Automatically run**, select which tests should be run after a successful build.

In some special setups, Qt Creator cannot deduce which executable or run configuration it should use. If Qt Creator repeatedly asks you to select the tests to run when trying to execute tests, you can enable it to cache your choices and use them were appropriate. The cached information is cleared when you switch to another project, close the current one, or select **Reset Cached Choices**.

Specifying Settings for Running Qt Tests

The code inside a benchmark test is measured, and possibly also repeated several times in order to get an accurate measurement. This depends on the measurement back-end that you can select in the **Benchmark Metrics** group in **Edit** > **Preferences** > **Testing** > **Qt Test**: walltime, CPU tick counter, event counter, Valgrind Callgrind, and Linux Perf. For more information, see Creating a Benchmark.



☑ Use XML output

☐ Verbose benchmarks

☐ Log signals and slots

☐ Limit warnings 2000

Benchmark Metrics

● Walltime

○ Tick counter

○ Event counter

○ Callgrind

○ Perf

To receive verbose output when running benchmarks, select the **Verbose benchmarks** check box.

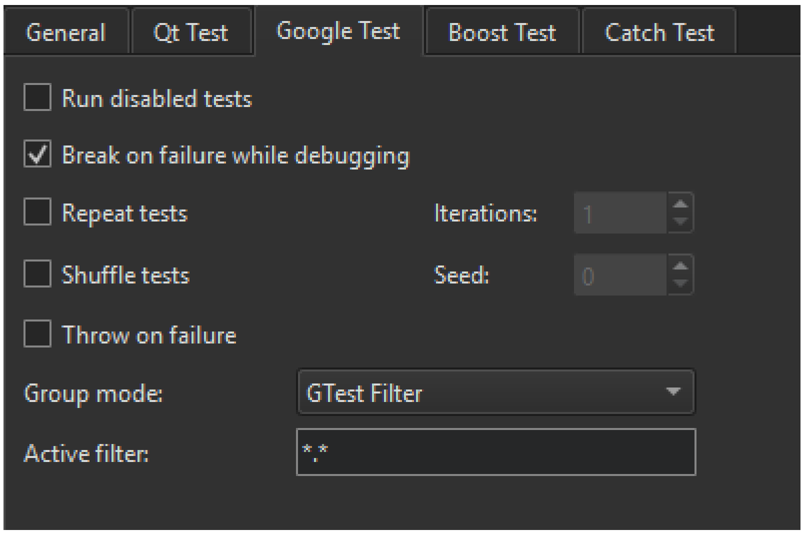
To allow the debugger to interrupt Qt tests on assertions, select the **Disable crash handler while debugging** check box.

To record information about signals and slots in the test log, select the **Log signals and slots** check box.

To explicitly limit the maximum number of warnings in the test log, select the **Limit warnings** check box and specify the intended number inside the spin box next to it. Set the number to 0 if you want no limit at all. The default number is 2000.

Specifying Settings for Running Google Tests

To specify settings for running Google tests, select **Edit** > **Preferences** > **Testing** > **Google Test**.



☐ Run disabled tests

☑ Break on failure while debugging

☐ Repeat tests Iterations: 1

☐ Shuffle tests Seed: 0

☐ Throw on failure

Group mode: GTest Filter

Active filter: *.*

To run disabled tests, select the **Run disabled tests** check box.

To run several iterations of the tests, select the **Repeat tests** check box and enter the number of times the tests should be run in the **Iterations** field. To make sure that the tests are independent and repeatable, you can run them in a different order each time by selecting the **Shuffle tests** check box.

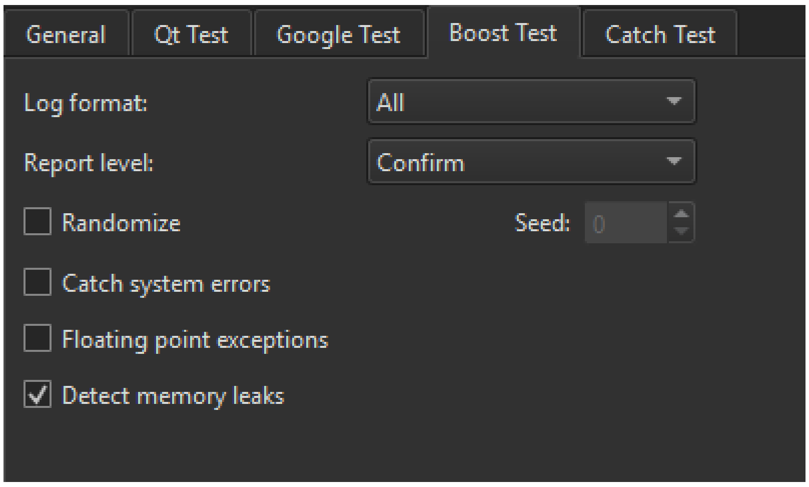
To turn failures into debugger breakpoints, select the **Break on failure while debugging** check box. To turn assertion

Qt DOCUMENTATION

in the **Active filter** field. For more information about GTest filters, see Running a Subset of the Tests.

Specifying Settings for Running Boost Tests

1. To specify settings for running Boost tests, select **Edit** > **Preferences** > **Testing** > **Boost Test**.



Log format: All

Report level: Confirm

☐ Randomize   Seed: 0

☐ Catch system errors

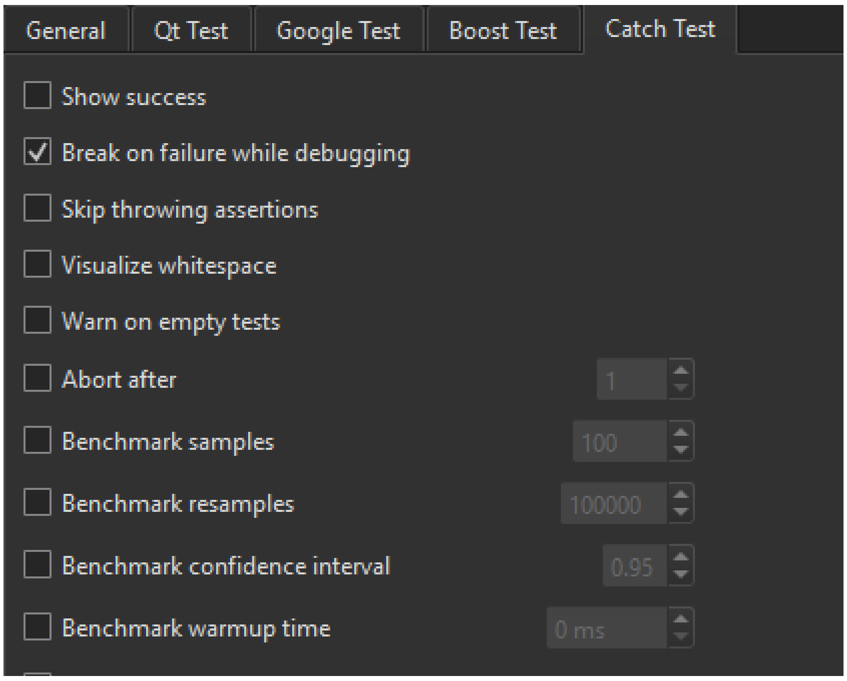
☐ Floating point exceptions

☑ Detect memory leaks

2. In the **Log format** field, select the error report format to specify the type of events you want recorded in the test report.
3. In the **Report level** field, select the verbosity level of the test result report. Select **No** if you do not want a report.
4. Select the **Randomize** check box to execute the tests in a random order, using the seed specified in the **Seed** field for initializing the randomizer.
5. Select the **Catch system errors** check box to catch system errors.
6. Select the **Floating point exceptions** check box to detect floating point exceptions.
7. Select the **Detect memory leaks** check box to detect memory leaks.

Specifying Settings for Running Catch2 Tests

1. To specify settings for running Catch2 tests, select **Edit** > **Preferences** > **Testing** > **Catch Test**.



☐ Show success

☑ Break on failure while debugging

☐ Skip throwing assertions

☐ Visualize whitespace

☐ Warn on empty tests

☐ Abort after 1

☐ Benchmark samples 100

☐ Benchmark resamples 100000

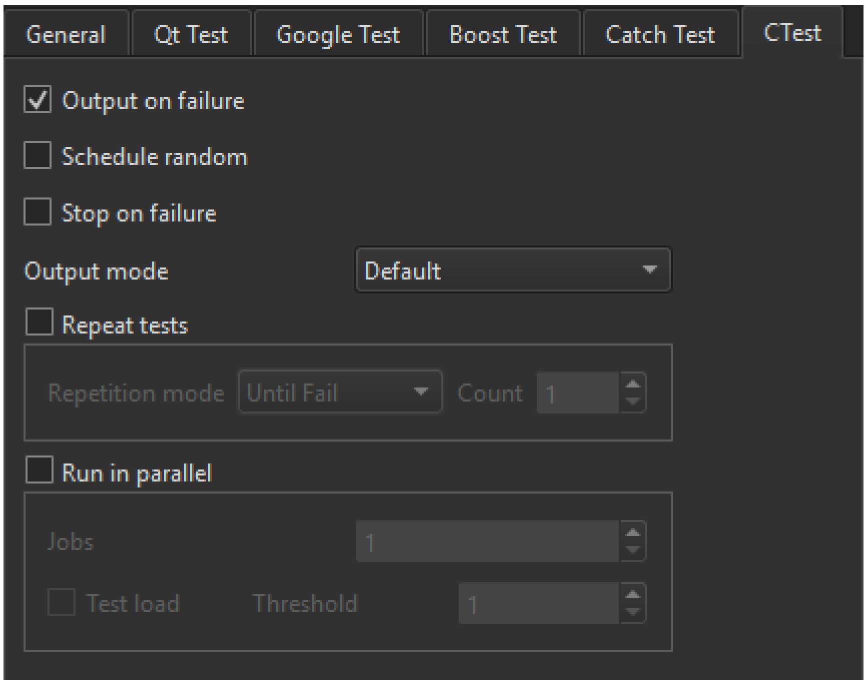
☐ Benchmark confidence interval 0.95

☐ Benchmark warmup time 0 ms

2. Select the **Show success** check box to show succeeding expressions as well. By default Catch2 will print only fails.
3. Select the **Break on failure while debugging** check box to turn failures into debugger breakpoints.
4. Select the **Skip throwing assertions** check box to skip any assertion that test for throwing an exception.
5. Select the **Visualize whitespace** check box to turn whitespace into escape sequences.
6. Select the **Warn on empty tests** check box to get a warning when a test case does not check any assertion.
7. Select the **Abort after** check box to abort the test after the number of failures specified inside the spin box.
8. Select the **Benchmark samples** check box to specify the number of samples to be collected while running benchmarks.
9. Select the **Benchmark resamples** check box to specify the number of resamples to be used for the statistical bootstrapping performed after the benchmarking.
10. Select the **Benchmark confidence interval** check box to specify the confidence interval used for the statistical bootstrapping.
11. Select the **Benchmark warmup time** check box to specify the warmup time for each test before benchmarking start.
12. Select the **Disable analysis** check box to disable the statistical analysis and bootstrapping.

Specifying Settings for Running CTest-Based Tests

1. To specify settings for running CTest-based tests, select **Edit** > **Preferences** > **Testing** > **CTest**.



✓ Output on failure
☐ Schedule random
☐ Stop on failure
Output mode: Default
☐ Repeat tests
Repetition mode: Until Fail   Count: 1
☐ Run in parallel
Jobs: 1
☐ Test load   Threshold: 1

2. Select the **Output on failure** check box to show test specific output if a test fails. Contrary to the CTest default this is enabled by default.
3. Select **Schedule random** to execute the tests in random order.
4. Select **Stop on failure** to automatically stop the test execution on the first failing test.
5. In the **Output mode** field, select the verbosity level of the CTest output.

> **Note:** This only influences the output on the text display.

can be specified in the **Count** field.

8. Select **Run in parallel** to run the tests in parallel using the specified number of **Jobs**.
9. Select **Test load** to be able to limit the parallel execution. CTest will not start a new test if it would cause the CPU load to pass the threshold given in **Threshold**.

Viewing Test Output

The test results are displayed in Test Results in XML format. XML can be parsed more easily and reliably than plain text.

However, if a Qt test crashes, it might not produce complete XML code that can be parsed, which might lead to information loss. The lost information might be retrievable when viewing the results as plain text. To view the results of Qt tests as plain text, select **Edit** > **Preferences** > **Testing** > **Qt Test**, and then deselect the **Use XML output** check box. Then select the (**Switch Between Visual and Text Display**) button in **Test Results** to switch to the text display.

The following table lists the messages that **Test Results** displays:

Result	Description
BENCH	Benchmark test.
DEBUG	Debug message.
FAIL	Test case failed. Double-click the line for more information.
FATAL	A fatal error occurred that stops the test case from being run, for example.
INFO	Informative message.
INTERNAL	Internal message.
PASS	Test case passed.
SKIP	Test case was skipped.
SYSTEM	An error message received from or influenced by the OS.
XFAIL	Test case is expected to fail, so it is marked by using the QEXPECT_FAIL macro. If the test case passes instead, an unexpected pass (XPASS) is written to the test log.
XPASS	Test case passed even though it was expected to fail.
WARN	Warning message.

Since Qt 5.4, you can provide a BLACKLIST file for tests. It is mainly used internally by the Qt CI system.

Result	Description
BFAIL	Blacklisted test case failed.
BPASS	Blacklisted test case passed.
BXFAIL	Blacklisted test case failed but was marked to be expected to fail.
BXPASS	Blacklisted test case passed even though it was expected to fail.

To view only messages of a particular type, select (**Filter Test Results**), and then select the types of messages to

Qt DOCUMENTATION

< Visualizing Chrome Trace Events   Advanced Use >

Qt The Qt Company

Contact Us

Company

About Us
Investors
Newsroom
Careers
Office Locations

Licensing

Terms & Conditions
Open Source
FAQ

Support

Support Services
Professional Services
Partners
Training

For Customers

Support Center
Downloads
Qt Login
Contact Us
Customer Success

Community

Contribute to Qt
Forum
Wiki
Downloads
Marketplace

Feedback   Sign In