

Quick CMake Tutorial

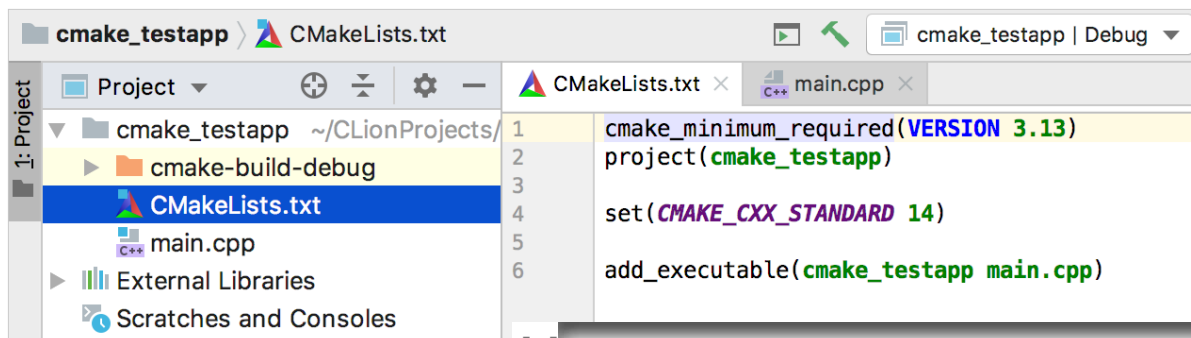
This tutorial will guide you through the process of creating and developing a simple CMake project. Step by step, we will learn the basics of [CMake](#) as a build system, along with the CLion settings and actions for CMake projects.

1. Basic CMake project

[CMake](#) is a meta build system that uses scripts called **CMakeLists** to generate build files for a specific environment (for example, makefiles on Unix machines). When you create a new CMake project in CLion, a `CMakeLists.txt` file is automatically generated under the project root.

Let's start with creating a new CMake project. For this, go to **File | New Project** and choose **C++ Executable**. In our example, the project name is `cmake_testapp` and the selected language standard in `C++14`.

By default, we get the project with a single source file `main.cpp` and the automatically created root [CMakeLists.txt](#) containing the following commands:



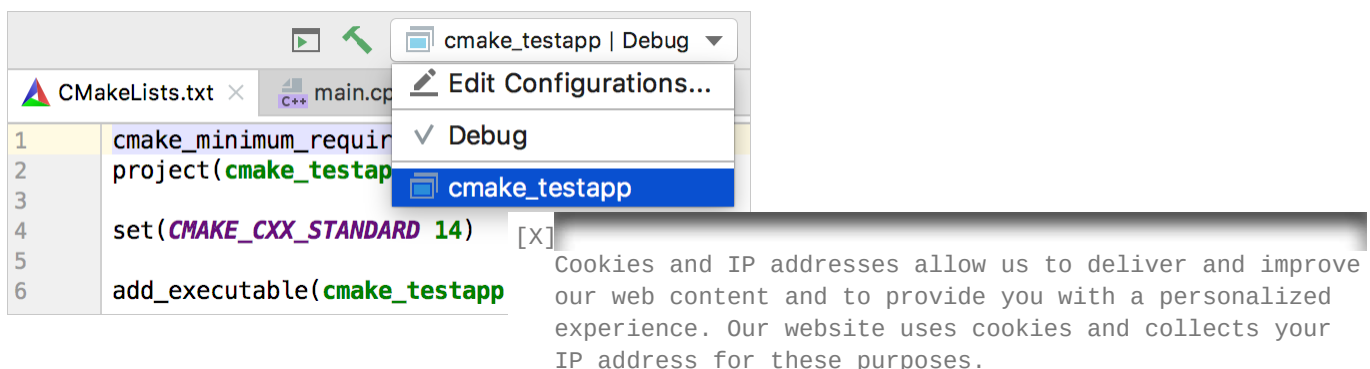
[X] Cookies and IP addresses allow us to deliver and improve our web content and to provide you with a personalized experience. Our website uses cookies and collects your IP address for these purposes.

Command	Description
<code>cmake_minimum_required(VERSION 3.13)</code>	Specifies the minimum required version of CMake. It is set to the version of CMake bundled in CLion (always one of the newest versions available).
<code>project(cmake_testapp)</code>	Defines the project name according to what we provided during project creation.
<code>set(CMAKE_CXX_STANDARD 14)</code>	Sets the <code>CMAKE_CXX_STANDARD</code> variable to the value of <code>14</code> , as we selected when creating the project.
<code>add_executable(cmake_testapp main.cpp)</code>	Adds the <code>cmake_testapp</code> executable target which will be built from <code>main.cpp</code> .

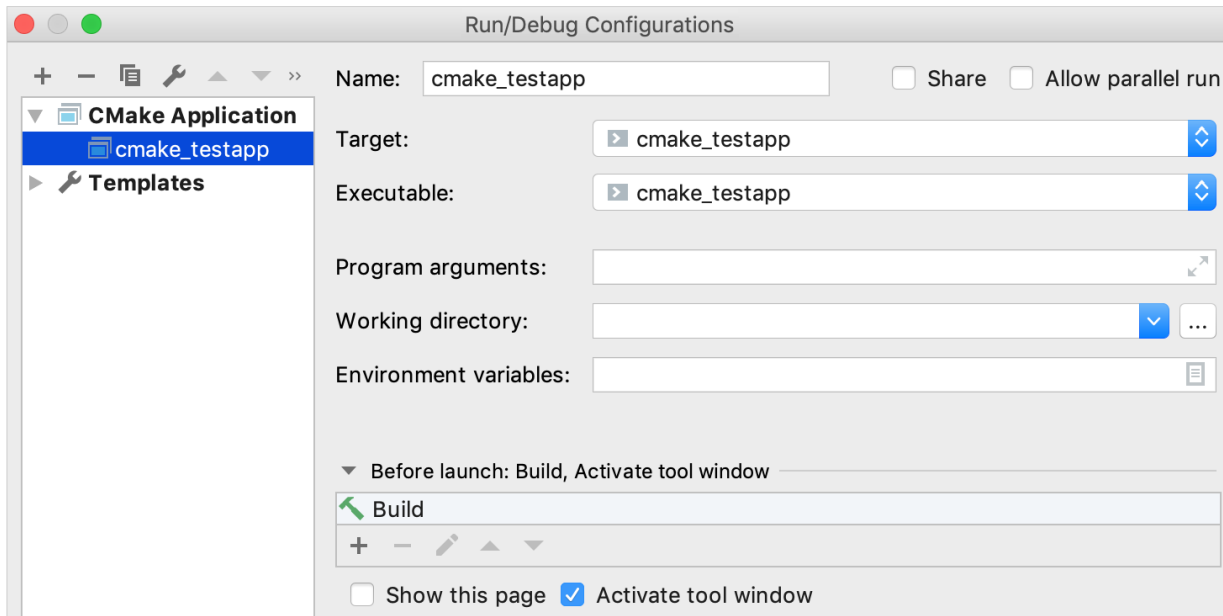
2. Build targets and Run/Debug configurations

[Target](#) is an executable or a library to be built using a CMake script. You can define multiple build targets in a single script.

For now, our test project has only one build target, `cmake_testapp`. Upon the first project loading, CLion automatically adds a Run/Debug configuration associated with this target:



Click **Edit Configurations** in the switcher or select **Run | Edit Configurations** from the main menu to view the details. The target name and the executable name were taken directly from the `CMakeLists.txt` :

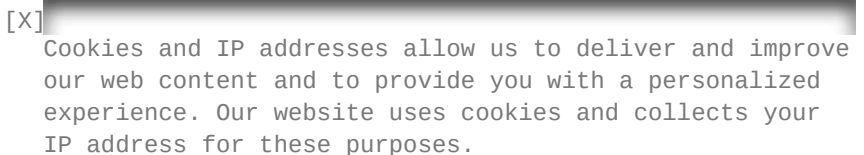


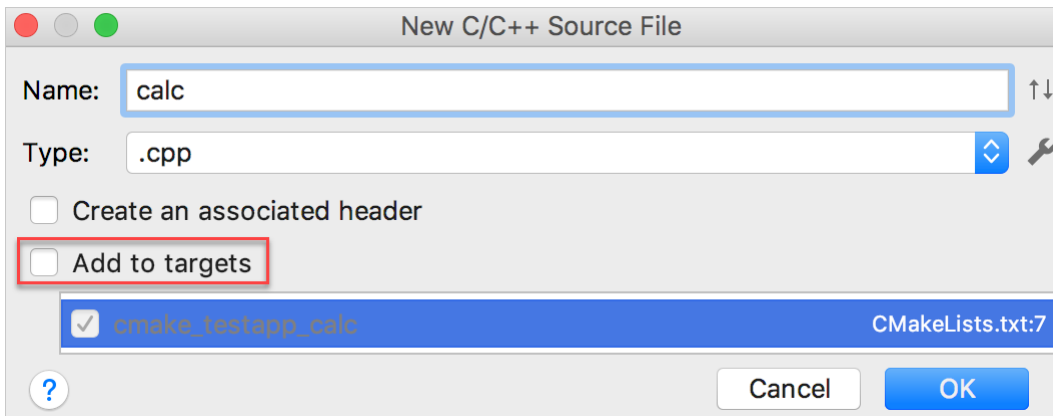
Notice the **Before launch** area of this dialog: **Build** is set as a before launch step by default. So we can use this configuration not only to debug or run our target but also to perform the build. To learn more about various build actions available in CLion, see [Build and Rebuild Projects](#).

3. Adding targets and reloading the project

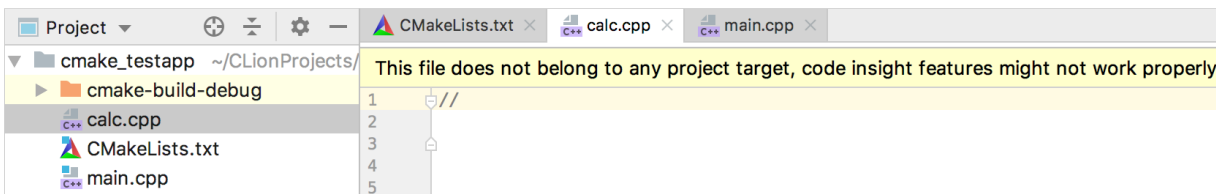
Now let's add another source file `calc.cpp` and create a new executable target from it.

Right-click the root folder in the Project tree and select **New | C/C++ Source File**. CLion prompts to add the file to an existing target:

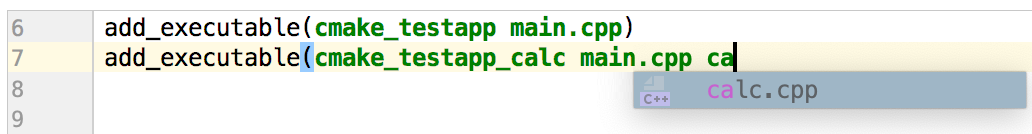




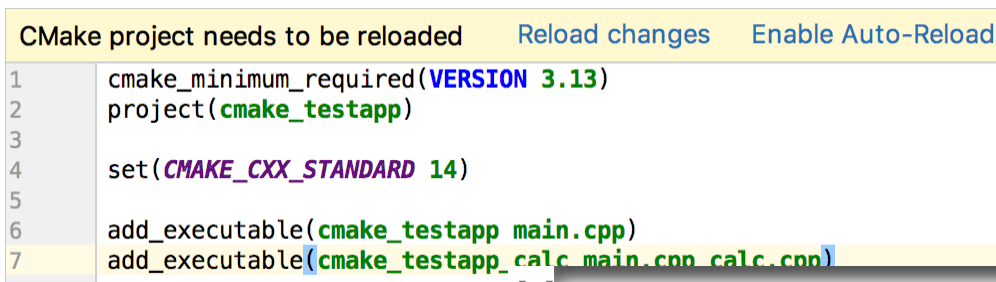
Since our goal is to create a new target, we clear the **Add to targets** checkbox. Accordingly, CLion notifies us that the new file currently does not belong to any target:



Now let's declare a new target manually in the `CMakeLists.txt`. Note that CLion treats **CMake scripts as regular code files**, so we can use code assistance features like syntax highlighting, auto-completion, and navigation:



When we make changes in `CMakeLists.txt`, CLion needs to reload it in order to update the project structure:

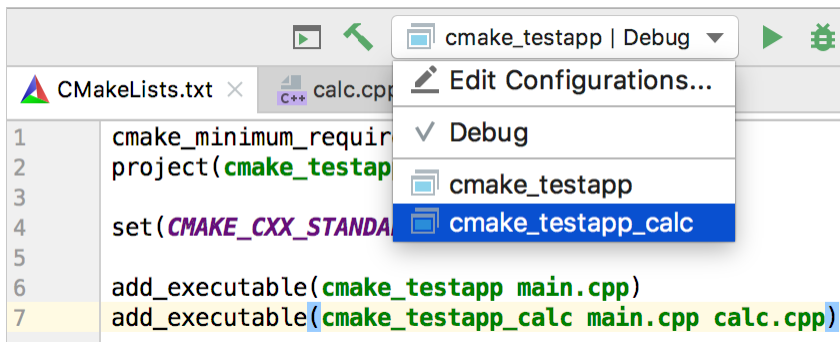


We can either reload the project or reload to let CLion silently apply al

Cookies and IP addresses allow us to deliver and improve our web content and to provide you with a personalized experience. Our website uses cookies and collects your IP address for these purposes.

for enabling/disabling auto-reload is also available in **Settings / Preferences | Build, Execution, Deployment | CMake**.

After reloading the project, CLion adds a Run/Debug configuration for the new target:

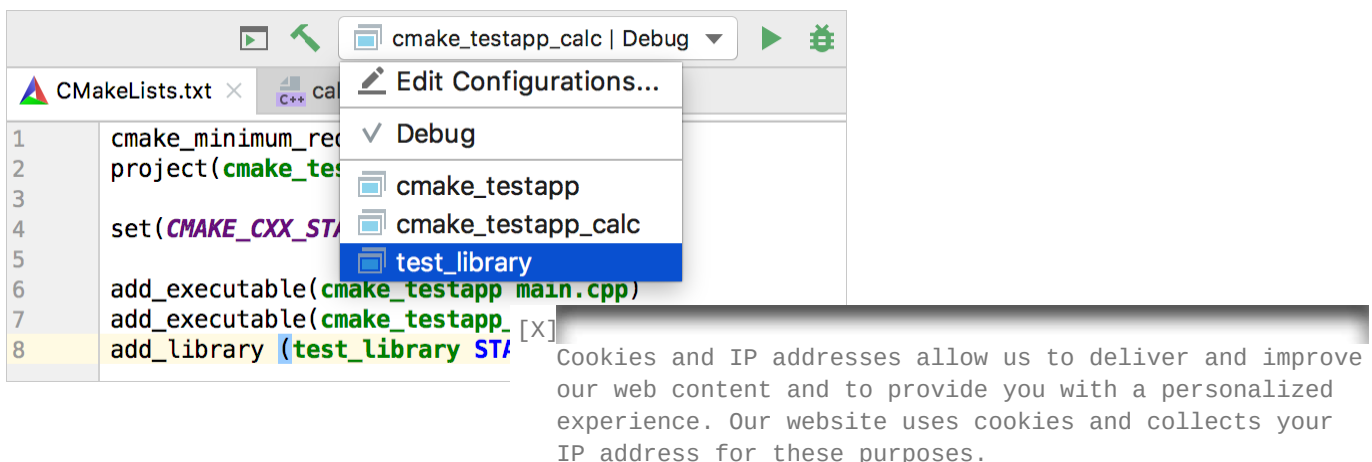



Library targets

Up to this point, the targets we added were executables, and we used **add_executable** to declare them. For library targets, we need another command - [add_library](#). As an example, let's create a static library from the `calc.cpp` source file:

```
add_library(test_library STATIC calc.cpp)
```

As well as for executables, CLion adds a Run/Debug configuration for the library target after reloading the project:

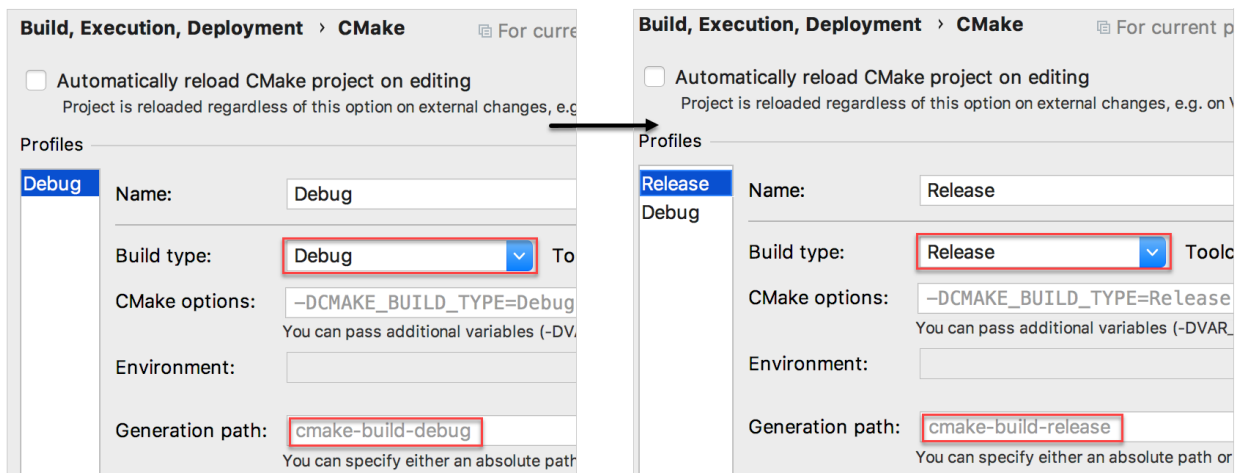



However, this is a non-executable configuration, so if we attempt to run or debug it, we will get the  *Executable not specified* error message.

4. Build types and CMake profiles

All the Run/Debug configurations created by far were *Debug* configurations, which is the default build type of the [CMake profile](#) that was automatically configured for our project. **CMake profile** is a set of options for the project build. It specifies the toolchain, build type, CMake flags, path for storing build artifacts, *make* build options, and environment variables.

For example, to separate the *Debug* and *Release* builds, we need to add (+) a new CMake profile in **Settings / Preferences | Build, Execution, Deployment | CMake** and set its build type to **Release**:

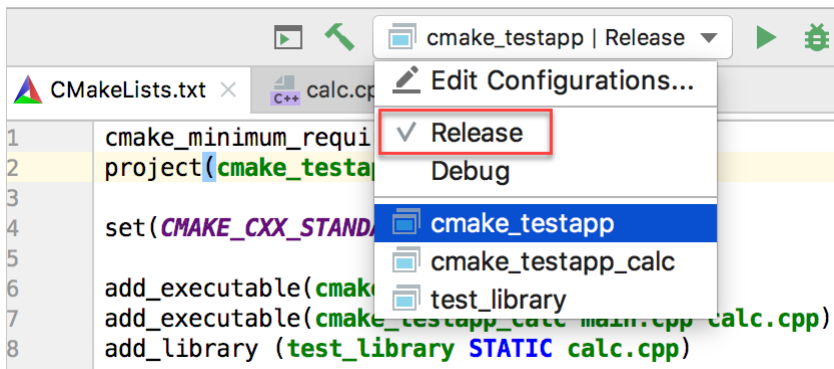


Notice the **Generation path** field that specifies the location of build results. The default folders are `cmake-build-debug` for *Debug* profiles and `cmake-build-release` for *Release* profiles. You can always set  other locations of your choice.

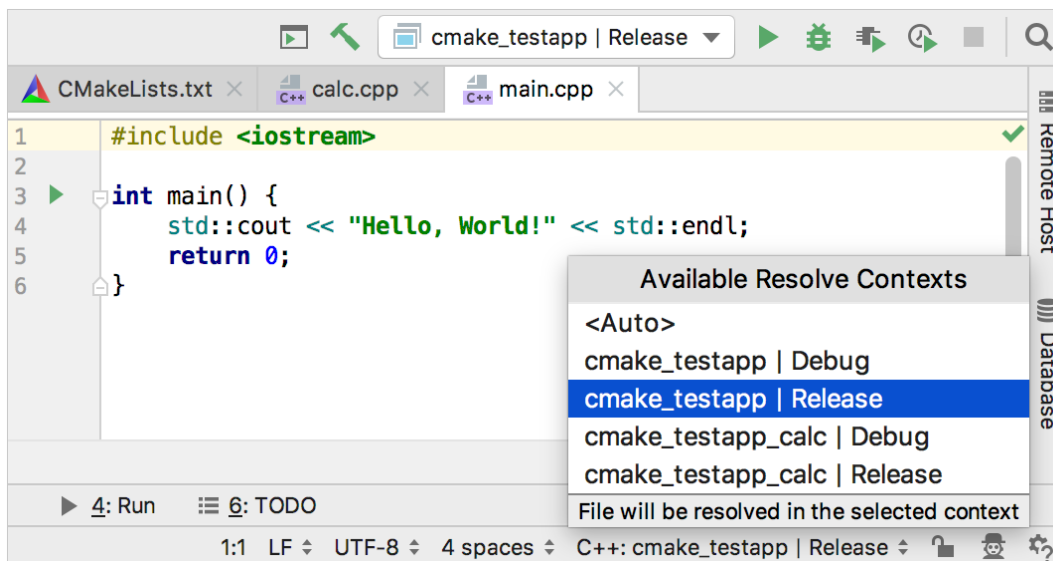
Now the Run/Debug configuration  shows two available profiles:

[X]

Cookies and IP addresses allow us to deliver and improve our web content and to provide you with a personalized experience. Our website uses cookies and collects your IP address for these purposes.



Switching configurations or CMake profiles may affect preprocessor definitions used while resolving the code: for example, when there are separate flags for Debug and Release builds or some variables that take different values depending on the build type. This is called [resolve context](#); it defines how CLion performs syntax highlighting and other code insights like Find Usages, refactorings, and code completion. When you switch between configurations, the resolve context for the current file is changed automatically. Also, you can select it manually in the context switcher (<Auto> restores the automatic selection):



5. Adding include directories

In order to use additional headers them either to all the targets or to

[X] Cookies and IP addresses allow us to deliver and improve our web content and to provide you with a personalized experience. Our website uses cookies and collects your IP address for these purposes.

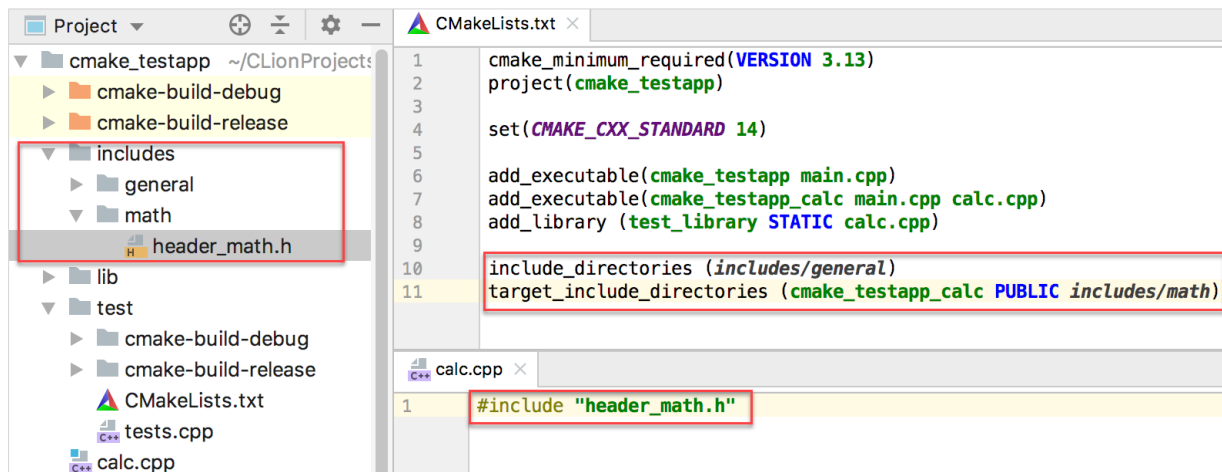
three directories under the project root, `includes`, `includes/general`, `includes/math`, and write the following in `CMakeLists.txt`:

1. `include_directories(includes/general)`
- to include `general` for all targets;
2. `target_include_directories (cmake_testapp_calc PUBLIC includes/math)`

- to include `math` only for the `cmake_testapp_calc` target.

Note that [target_include_directories](#) should be placed after `add_executable` (or `add_library`) for the target name to be already available.

Now the headers from `includes/general` or `includes/math` can be included in the sources directly, for example:



Headers and sources that you add to the project will be resolved correctly only if you include them explicitly in `CMakeLists.txt` or if you include them in other files already belonging to the project (see [Managing CMake project files](#)).

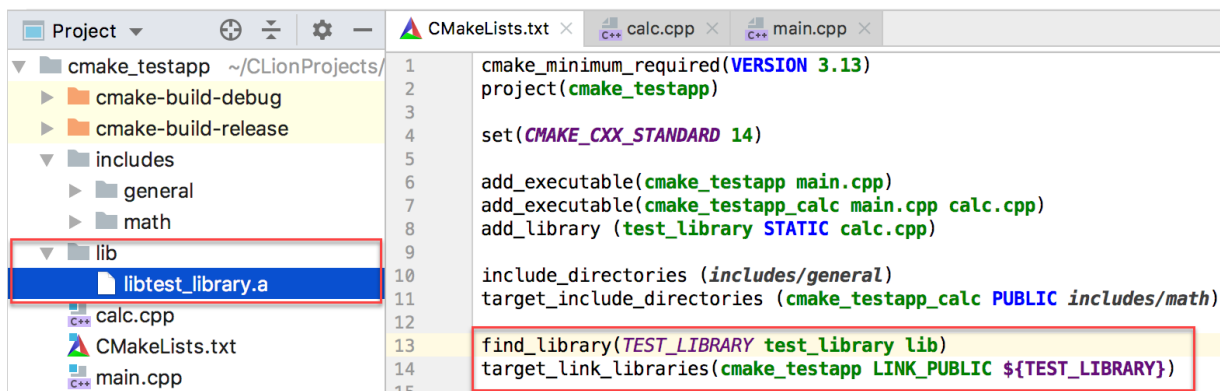
6. Linking libra

[X] Cookies and IP addresses allow us to deliver and improve our web content and to provide you with a personalized experience. Our website uses cookies and collects your IP address for these purposes.

Static libraries

On [step 3](#), we created a static library called *test_library*. Let's take it from the default location, which is `cmake-build-debug`, place it in the `lib` directory under the project root, and link it to the *cmake_testapp* target.

We will use two commands to link a static library: [find_library](#) provides the full path, which we then pass directly into the [target_link_libraries](#) command via the `${TEST_LIBRARY}` variable.



Note: make sure to place `target_link_libraries` after the `add_executable` command, so that CMake actually builds the target before linking the library.

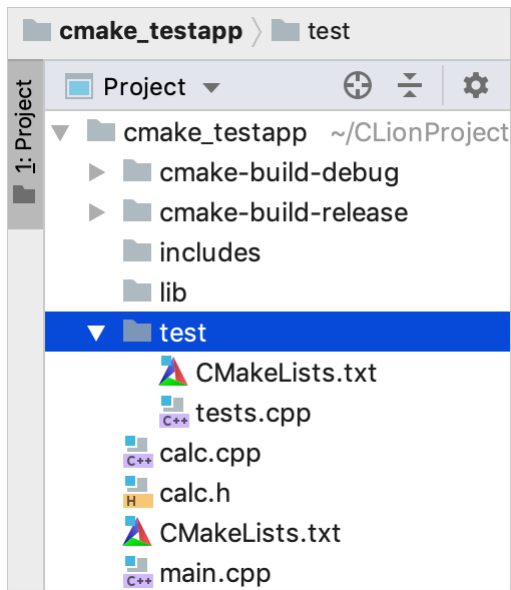
Dynamic libraries (Boost example)

To illustrate linking dynamic libraries, we will take an example of using [Boost.Test framework](#).

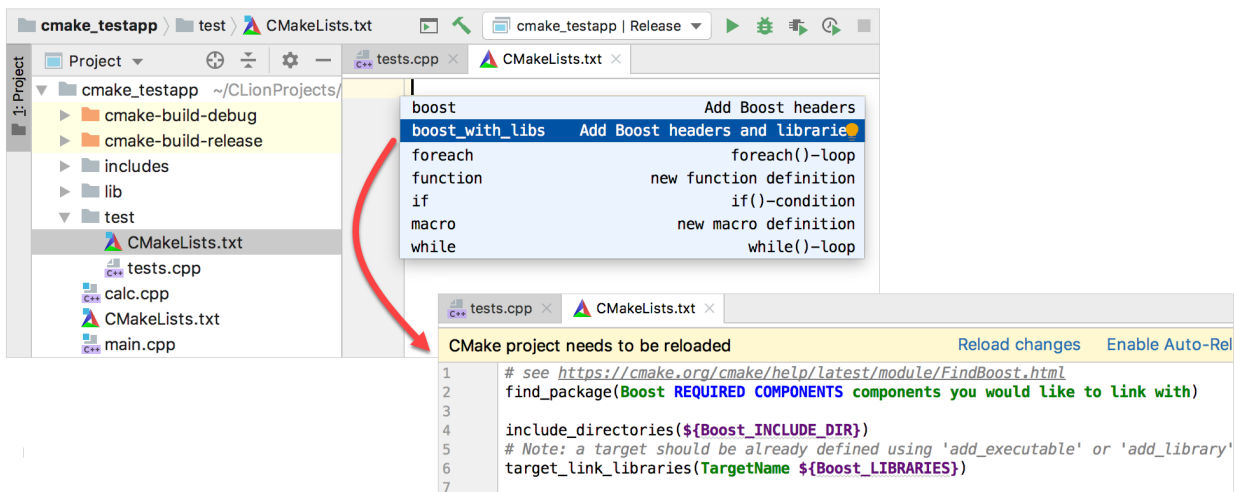
Let's write a simple function `int add_values (int a, int b) { return a+b;}` in `calc.cpp` and create an associated header `calc.h` with the function declaration. We will test this function with the help of Boost.Test framework.

As our project gets more complicated, the root `CMakeLists.txt` file can become difficult to maintain. To avoid this and build a transparent project structure, we will extract the tests into a subproject. For this, we will create a separate directory called `test` and provide it with its own `CMakeLists.txt` file. To create a new subproject, go to `Project tree` and select `New | CMake`

[X] Cookies and IP addresses allow us to deliver and improve our web content and to provide you with a personalized experience. Our website uses cookies and collects your IP address for these purposes.



The subdirectory `test/CMakeLists.txt` script is initially empty. Let's start filling it up by inserting a [live template](#) for *Boost with libs*. Press `Ctrl+J` or click **Code | Insert Live Template**, and choose **boost_with_libs**:



Let's adjust the inserted code to look as follows:

```
set (Boost_USE_STATIC_LIBS OFF) #enable dynamic linking
```

```
#search for unit_test_framework
```

```
find_package (Boost REQUIRED [x]
```

```
#create a cmake_testapp_bo
add_executable (cmake_testa
```

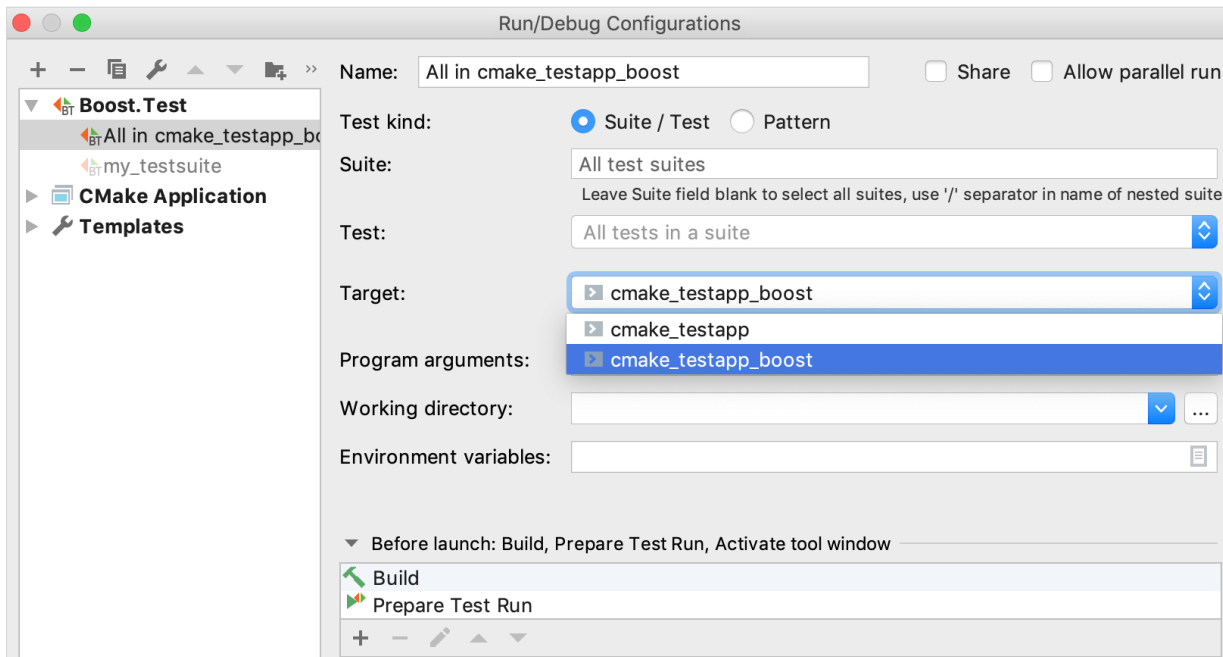
Cookies and IP addresses allow us to deliver and improve our web content and to provide you with a personalized experience. Our website uses cookies and collects your IP address for these purposes.

```
#link Boost libraries to the new target
target_link_libraries (cmake_testapp_boost ${Boost_LIBRARIES})
```

Also, we need to place the `add_subdirectory(test)` command in the `root`

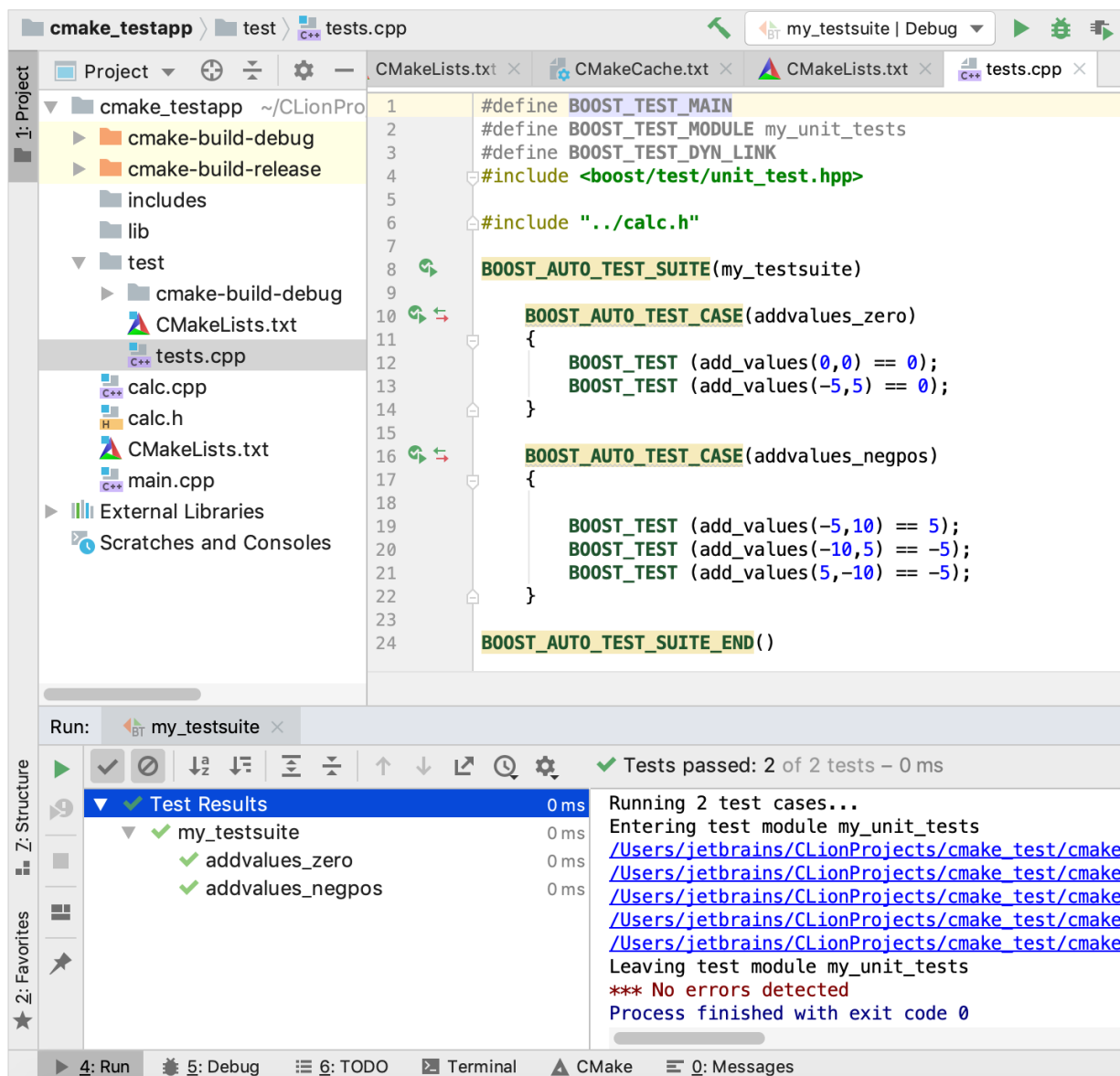
`CMakeLists.txt` to make our test target `cmake_testapp_boost` available for the main build. This command, when placed in the root CMake script, declares a subproject `test` that has its own `CMakeLists.txt`.

After reloading the changes in both `CMakeLists.txt` files, CLion creates a Run/Debug configuration for the `cmake_testapp_boost` target. This is a regular CMake Application configuration that we could run/debug right away. However, to be able to use the built-in test runner, let's create another configuration out of the [Boost.Test](#) template:



Now let's run this configuration and get the test results. Test runner shows the tree of tests in the suite, their output, status, and duration:

[X] Cookies and IP addresses allow us to deliver and improve our web content and to provide you with a personalized experience. Our website uses cookies and collects your IP address for these purposes.



7. Learn more

To dig deeper into CMake in CLion, learn how to:

- [Change project root](#)
- [Reset CMake Cache](#)
- [Switch compilers](#)

[X]

Cookies and IP addresses allow us to deliver and improve our web content and to provide you with a personalized experience. Our website uses cookies and collects your IP address for these purposes.

- Run [Build actions](#) and [CMake install](#)
- Use [environment variables](#) and the [CLION_IDE](#) macro.

Last modified: 12 June 2019

[X]

Cookies and IP addresses allow us to deliver and improve our web content and to provide you with a personalized experience. Our website uses cookies and collects your IP address for these purposes.