C++ Development Environment

The Gringofts Team

2020-03-14

1 Background

This document describes what the Gringofts team's C++ development environment may look like. Aspects from OS, compiler, to unit test framework, documentation generation tool will be discussed in detail in the following sections.

2 OS - Ubuntu 16.04.5 LTS

Ubuntu 16.04.5 LTS is chosen because:

1. It is the recommended OS which is installed across staging, pre-production and production pools.
2. The core component in real-time FAS will be based on high performance consensus solutions such as logcabin, which is developed in C++ and uses several Linux-only features.

3 IDE - CLion

Jetbrains' CLion is chosen as the IDE for the following reasons:

1. It is a smart editor, providing useful features such as navigation, code completion and code analysis.
2. Full integration with other tools needed in development, such as Git (version control system), CMake (build tool), Google Test (unit test framework), debuggers (e.g., GDB), dynamic analysis tools (e.g., Valgrind).
3. CLion has a very consistent user experience (e.g., look-and-feel, key mappings) to its Java counterpart IntelliJ, which has been widely accepted firm wide.

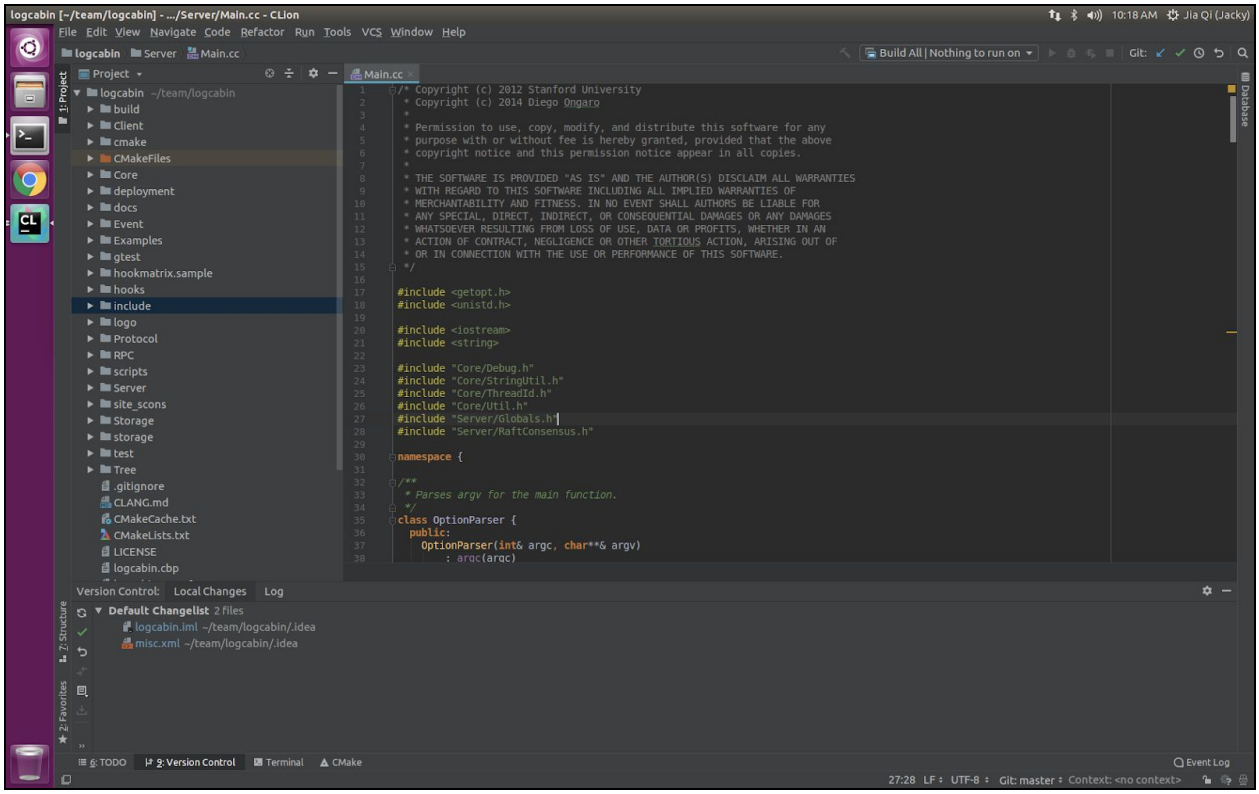


Figure 1: Screenshot of CLion on Ubuntu 16.04.5

4 Build Tool - CMake + GNU Make

We use CMake to generate build scripts such as Makefile, which are later consumed by make to generate executables.

There are many discussions on the Internet comparing various build tools, such as this, this and this. Besides all other pros, the main reason CMake is chosen is that it is so far the only build tool supported by CLion.

5 C++ Language Version - C++17

C++17 is chosen because C++17 introduces many improvements and additions (see summaries here and here) that can help make code simpler and more consistent (see posts here and here). Its standard library now has cross-platform threading support.

We have this confidence to try out C++17 is also because:

1. Open source consensus solutions, such as logcabin and cornerstone, can be compiled without any issues against C++17 by both Clang and GCC compilers.

2. We have various tools (will be introduced later) that can detect most issues - if there were - as early as possible in various environments (e.g., dev, private-ci, staging).

6 Compiler - Clang 6.0.1 or GCC 7.3.0[1]

Both Clang 6.0.1 and GCC 7.3.0 can be used to compile the C/C++ source code, and they are close in various aspects:

1. Both compilers support most features in C++11/C++14/C++17. See the full list here.
2. In many of the benchmark results the outcome is tight between these two compilers. See the results here.
3. Both provide clear diagnostics now. See the comparison here. We can leverage both compilers to get a more complete diagnostics.
4. Both can work with CMake and GNU Make.

One note is Clang has a Clang Static Analyzer which GCC lacks. And this can be the deciding factor that we prefer Clang over GCC.

1. GCC 8.x is not chosen for now because it doesn't play well with the code coverage tools. See the issue here.

7 Version Control System - Git

Due to great features provided by Git.

8 Coding Standards - Google C++ Style Guide

We have two choices, one is Google C++ Style Guide, which has been widely applied across open-source projects originated at Google, the other is C++ Core Guidelines, which is a continuous effort led by Bjarne Stroustrup, the inventor of C++.

Both are best practices distilled from real projects running in production, with the former focuses on "*keeping the code base manageable while still allowing coders to use C++ language features productively",* while the latter is focused on *"relatively higher-level issues, such as interfaces, resource management, memory management, and concurrency"*.

One good thing with the former is it also provides cpplint, a tool to assist with style guide compliance while for the latter there is a checker only available in Microsoft Visual Studio[2].

The other good thing with the former is it's supported by CLion. See below:

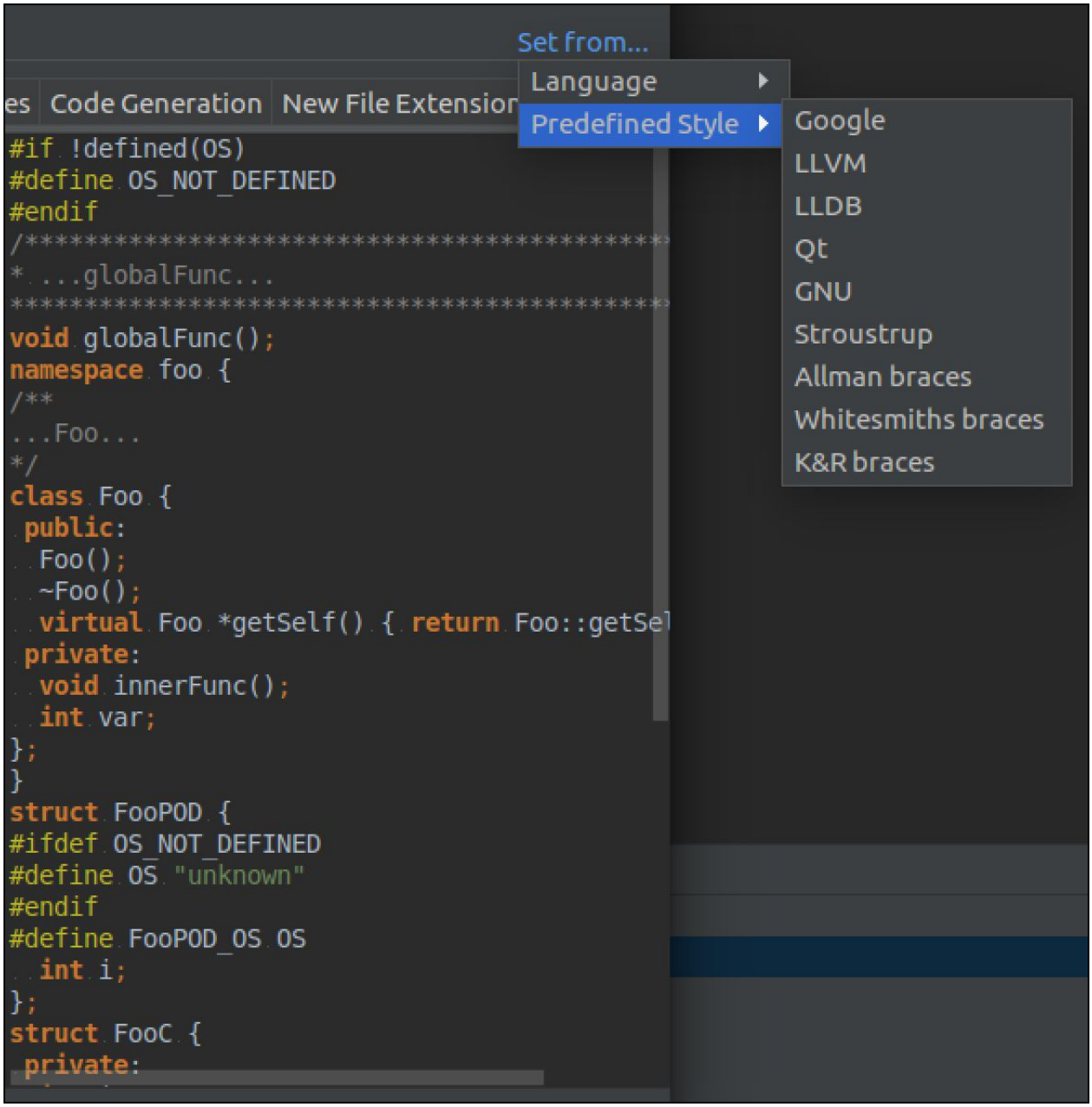


Figure 2: Apply Google C++ Style in CLion

We will critically apply a subset of the Google C++ Style Guide but not all of them (e.g., we need to take another look at the statement on exceptions) and keep the C++ Core Guidelines in mind when designing our system.

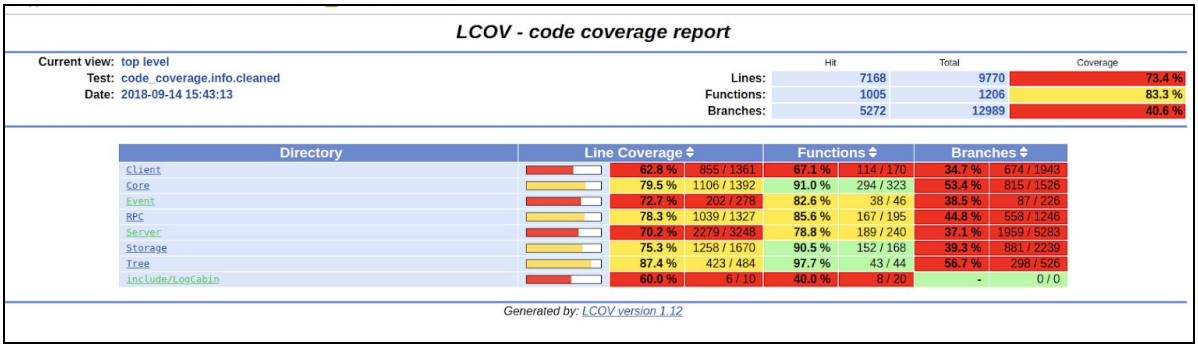
2. Clang-Tidy can check the C++ Core Guidelines for Clang under Linux. However it'll only be available in Extra Clang Tools 8, which is not released yet.

9 Code Coverage Tools - gcov + lcov

We use gcov to generate the code coverage report which will be consumed by lcov and then genhtml to generate the html reports.

Some benefits using these tools are:

1. Work well with our unit test framework.
2. Test coverage of line, function and branch are all supported.
3. Integration with Jenkins with the help of gcovr and Jenkins' Cobertura Plugin.



LCOV - code coverage report

Current view: top level
Test: code_coverage.info.cleaned
Date: 2018-09-14 15:43:13

	Hit	Total	Coverage
Lines:	7168	9770	73.4 %
Functions:	1005	1206	83.3 %
Branches:	5272	12989	40.6 %

Directory	Line Coverage		Functions		Branches	
Client	62.8 %	855 / 1361	67.1 %	114 / 170	34.7 %	674 / 1943
Core	79.5 %	1106 / 1392	91.0 %	294 / 323	53.4 %	815 / 1526
Event	72.7 %	202 / 278	82.6 %	38 / 46	38.5 %	87 / 226
RPC	78.3 %	1039 / 1327	85.6 %	167 / 195	44.8 %	558 / 1246
Server	70.2 %	2279 / 3248	78.8 %	189 / 240	37.1 %	1959 / 5283
Storage	75.3 %	1258 / 1670	90.5 %	152 / 168	39.3 %	881 / 2239
Tree	87.4 %	423 / 484	97.7 %	43 / 44	56.7 %	298 / 526
include/LogCabin	60.0 %	6 / 10	40.0 %	8 / 20	-	0 / 0

Generated by: LCOV version 1.12

Figure 3: Code Coverage Report from logcabin

10 Unit Test Framework - Google Test + Google Mock

Among all the popular frameworks (see one of the comparisons here), our choice is Google Test for following reasons:

1. logcabin uses Google Test, so unless other frameworks such as Catch2 or Boost.Test provide features that are not supported in Google Test, I'd rather stay where we are.
2. In fact, together with Google Mock, Google Test provides a more complete solution to writing/running unit tests compared with other frameworks.

11 Debuggers - LLDB or GDB

We use LLDB for Clang and GDB for GCC. Both are supported by CLion.

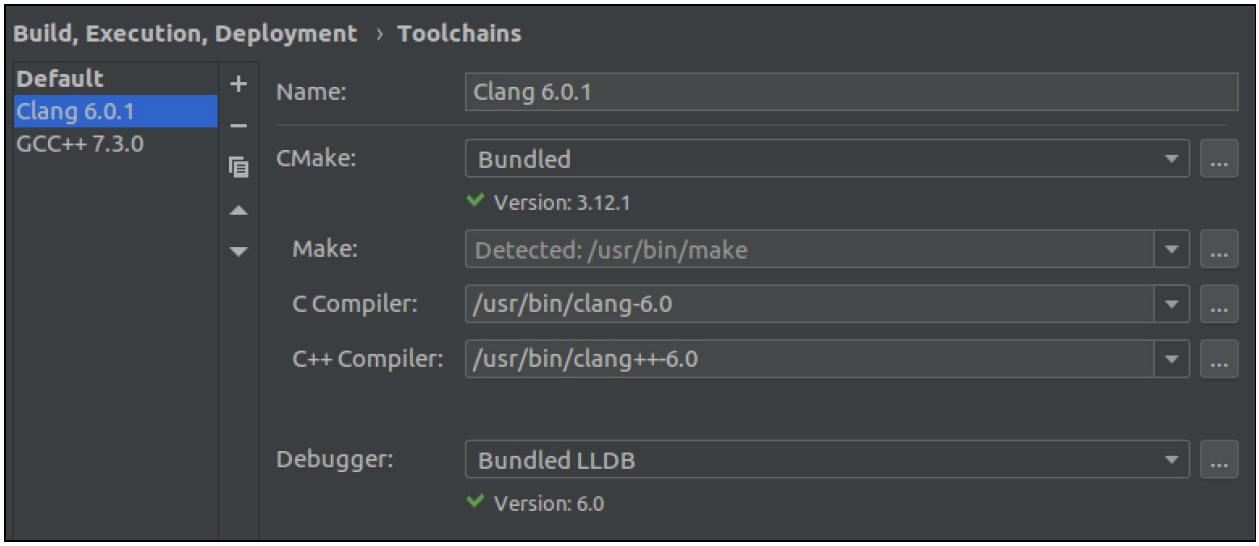


Figure 4: Clang 6.0.1 Toolchains Settings in CLion

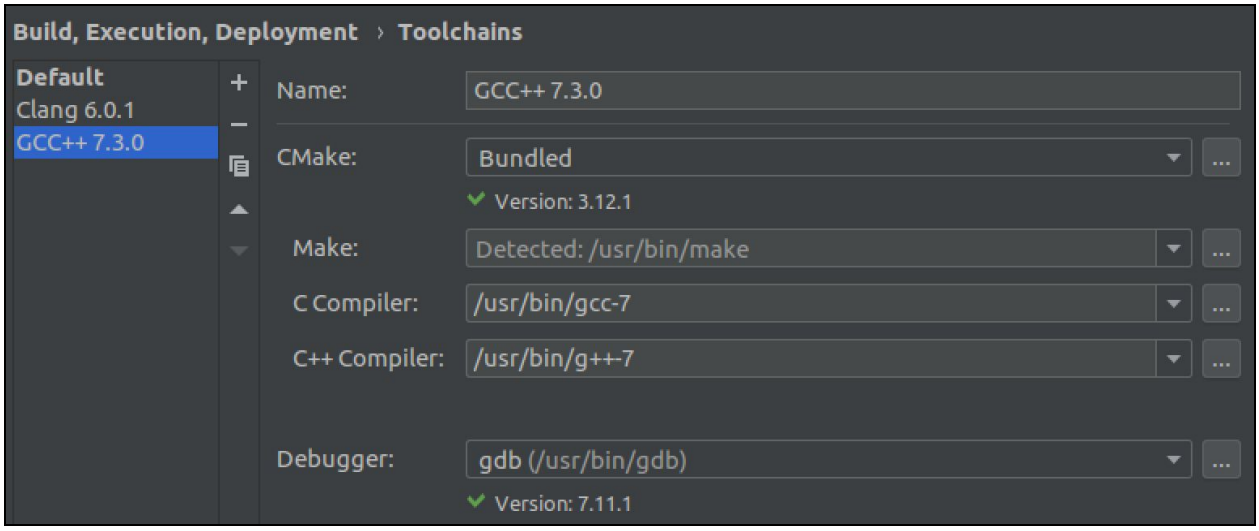


Figure 5: GCC 7.3.0 Toolchains Settings in CLion

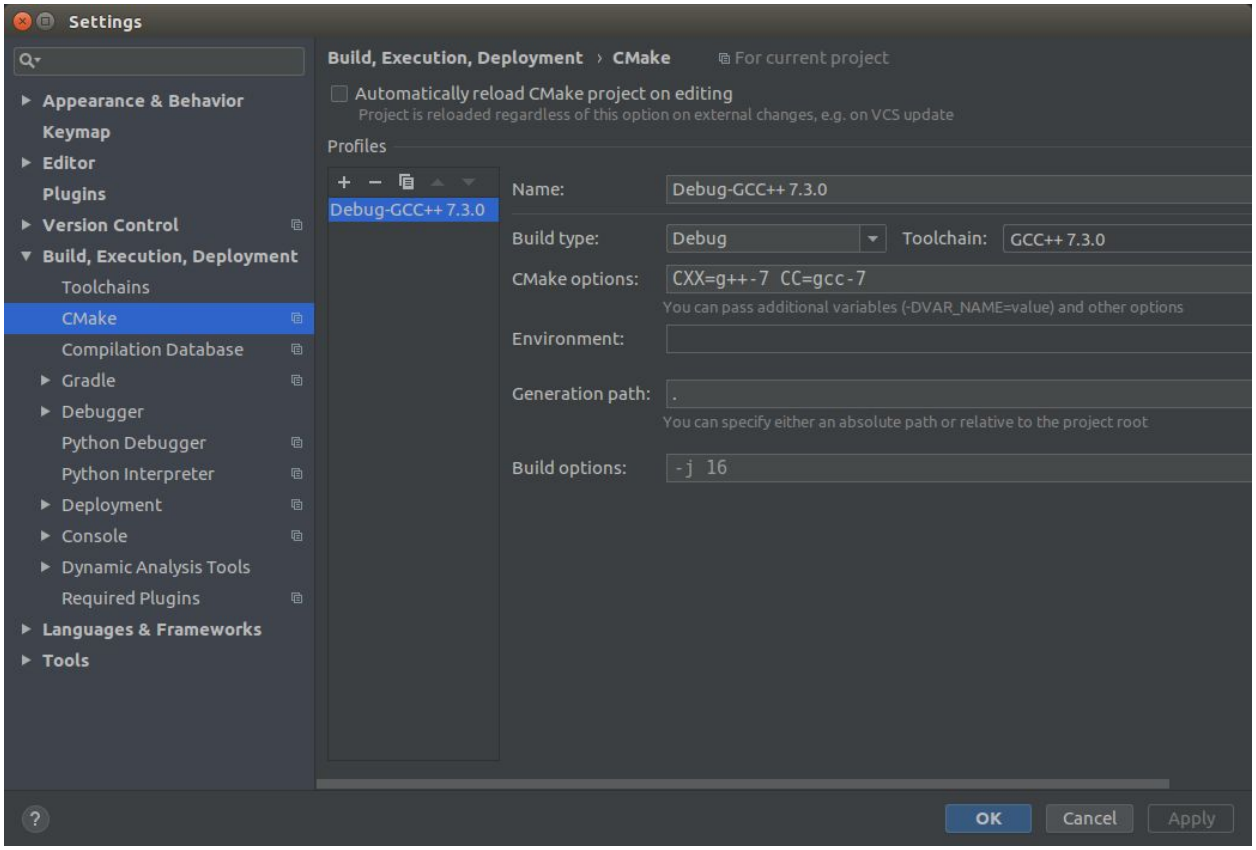


Figure 6: CMake Configurations for GCC 7.3.0 in CLion

12 Dynamic Analysis Tools - Valgrind and Sanitizers

Valgrind is the widely-used toolset which can be used to "*automatically detect many memory management and threading bugs, and profile your programs in detail.*" See one of the testimonies here.

Google Sanitizers are the only known C++ sanitizers so far.

Both are integrated into CLion now. Details are here (Valgrind) and here (Google Sanitizers).

13 Documentation Generation Tool - Doxygen

We use Doxygen to generate documentation for C++ projects. If you are from Java world, you can write the comment using JavaDoc style and Doxygen will recognize that.

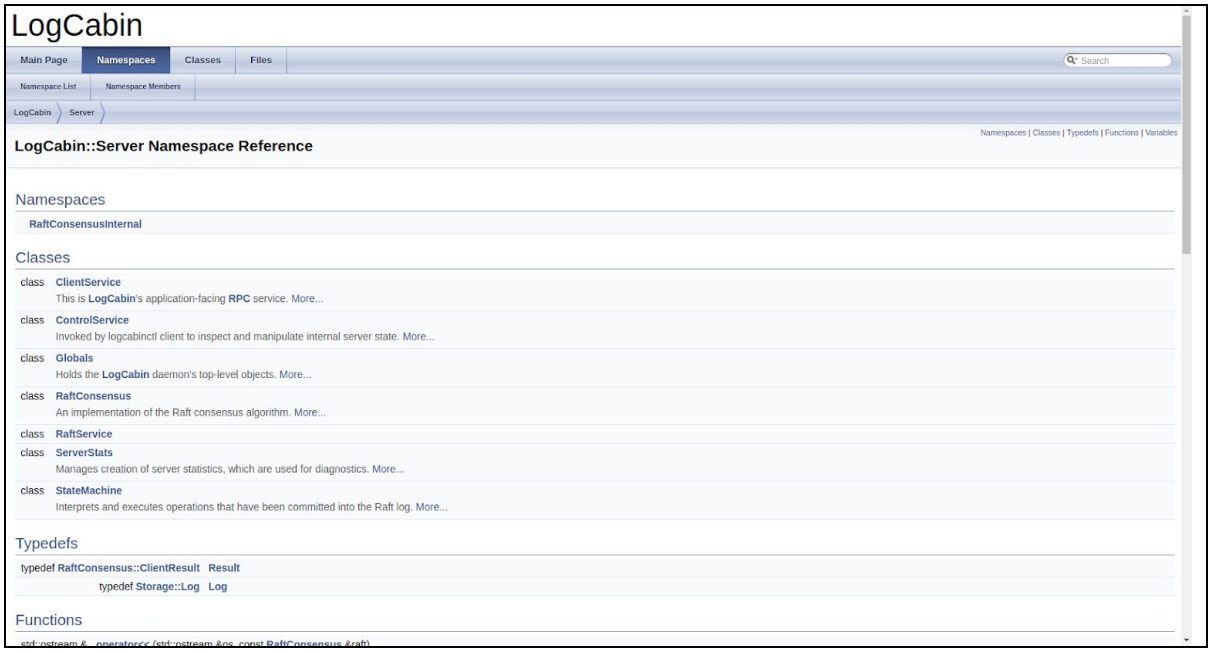


Figure 7: Documentation for logcabin Generated by Doxygen

14 Logging - spdlog

The selection criterias for logging are:

1. Support various file sinkers, e.g., rotating file sink.
2. Support formatted message, e.g., can include severity, timestamp, filename, etc
3. Thread-safe
4. Easy to use, e.g., no external dependency
5. Well documented
6. Fast

After evaluating all the famous logging frameworks (glog, boost log, spdlog, g3log, and log4cpp), spdlog is chosen as it satisfies all above criterias, especially that it's very lightweight, header only.

15 Copyright and File Template

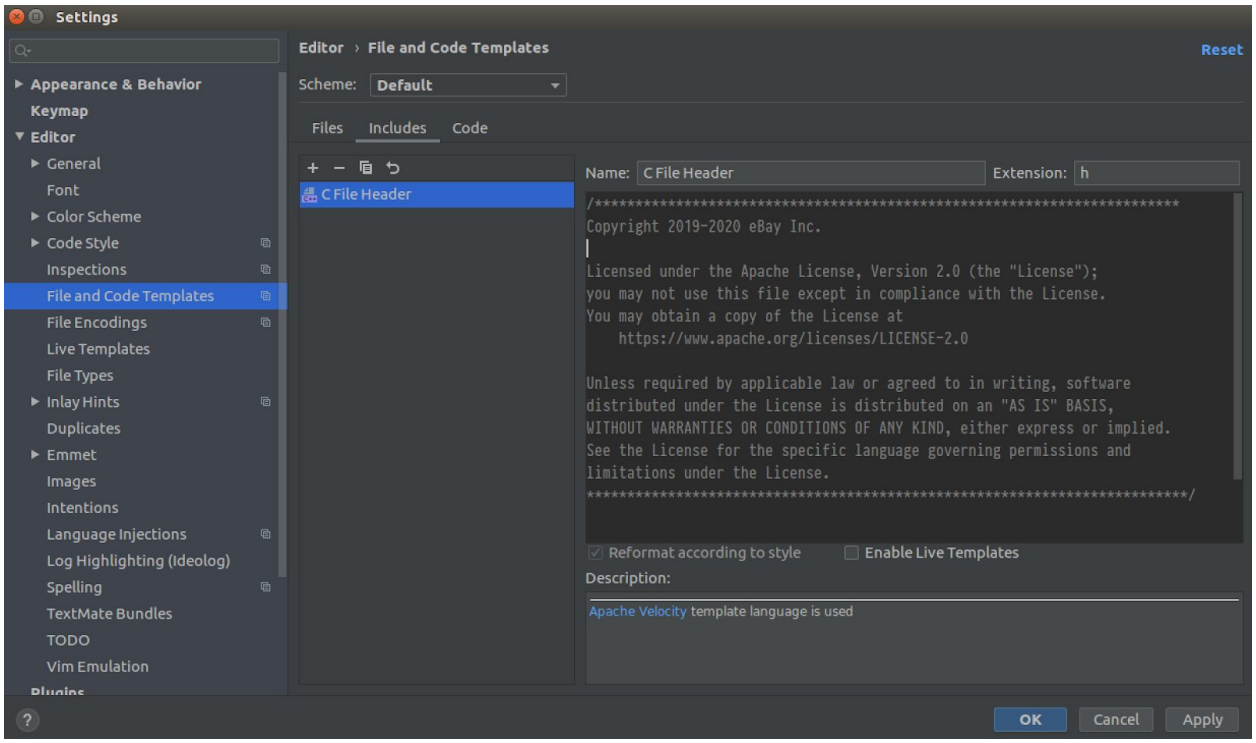


Figure 8: Copyright

/***

Copyright 2019-2020 eBay Inc.

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software

distributed under the License is distributed on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and

limitations under the License.

**********/

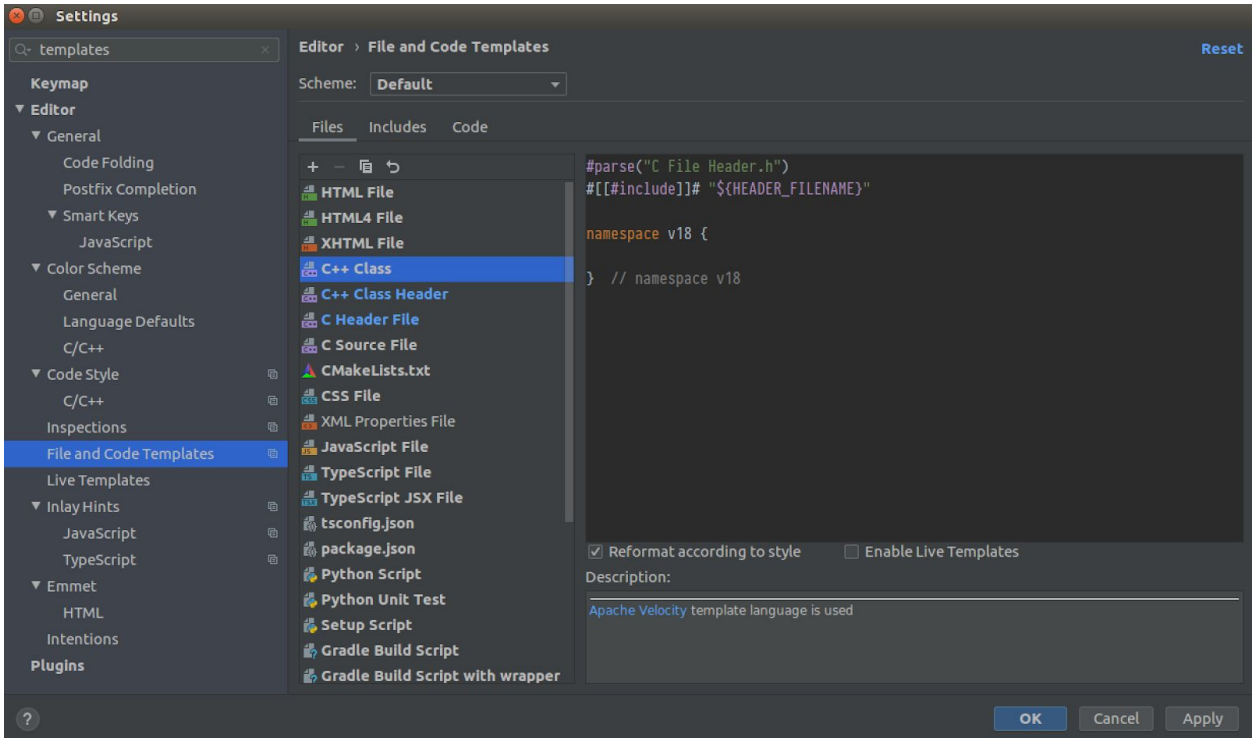


Figure 9: C++ Class Template

#parse("C File Header.h")

#[[#include]]# "${HEADER_FILENAME}"

namespace gringofts {

} // namespace gringofts

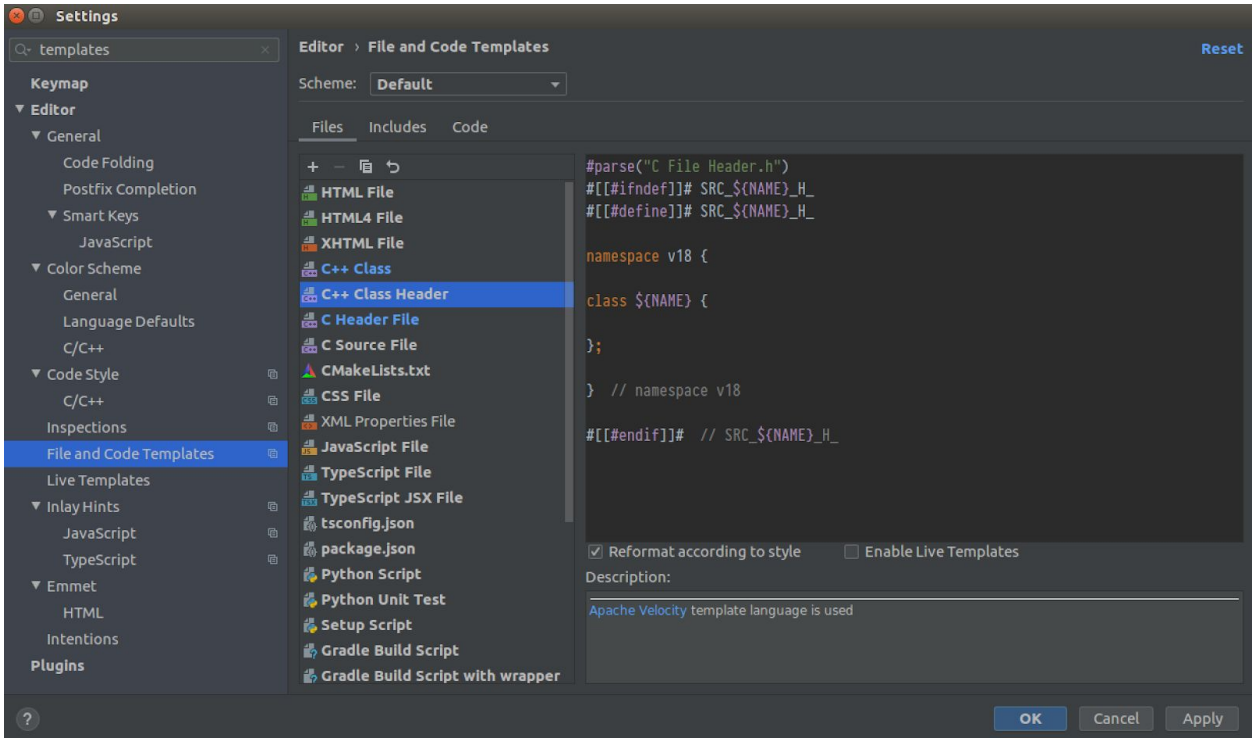


Figure 10: C++ Class Header Template

```
#parse("C File Header.h")
#[[#ifndef]]# SRC_${NAME}_H_
#[[#define]]# SRC_${NAME}_H_
namespace gringofts {
class ${NAME} {
};
} // namespace gringofts
#[[#endif]]# // SRC_${NAME}_H_
```

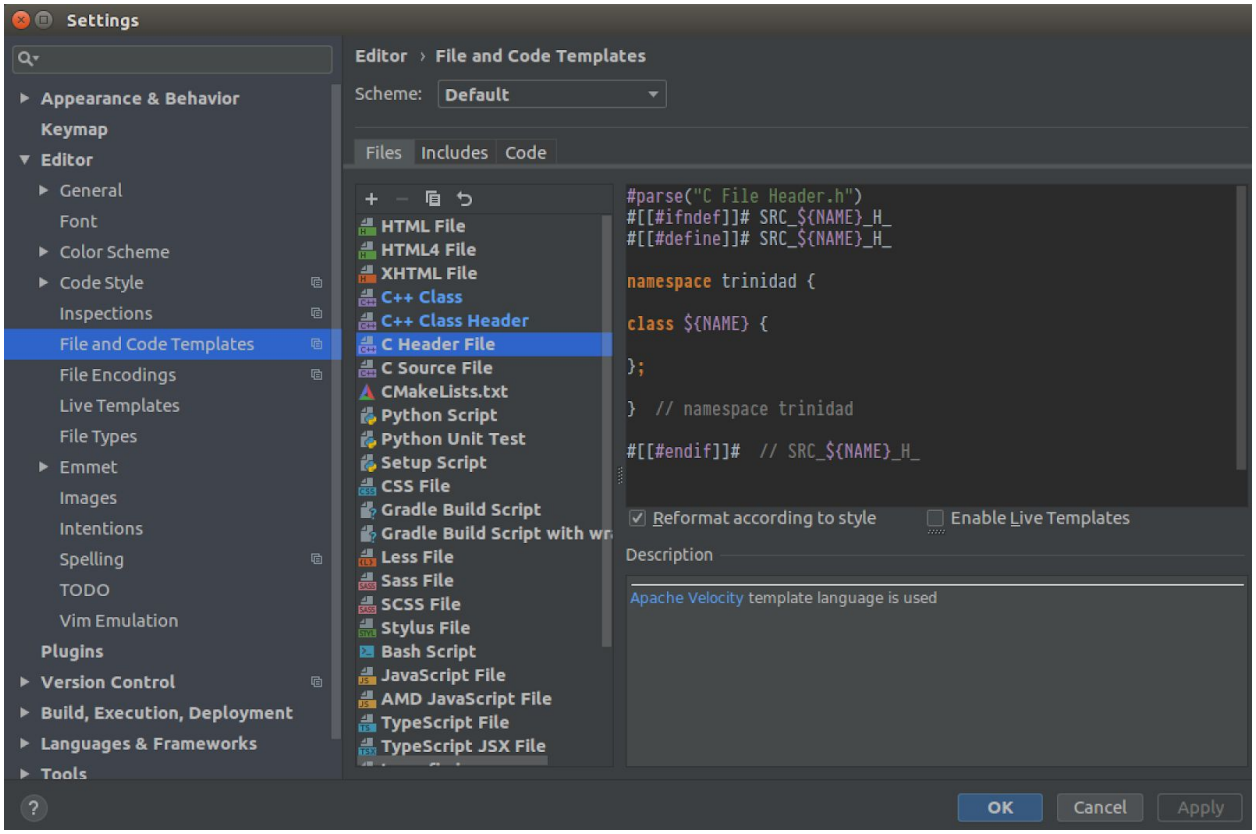


Figure 11: C Header File Template

```
#parse("C File Header.h")
#[[#ifndef]]# SRC_${NAME}_H_
#[[#define]]# SRC_${NAME}_H_
namespace gringofts {
class ${NAME} {
};
} // namespace gringofts
#[[#endif]]# // SRC_${NAME}_H_
```