**Bazel**

Bazel is an open-source build and test tool similar to Make, Maven, and Gradle.

It uses a human-readable, high-level build language.

Bazel supports projects in multiple languages and builds outputs for multiple platforms.

Bazel supports large codebases across multiple repositories, and large numbers of users.

Bazel offers the following advantages:

**High-level build language.** Bazel uses an abstract, human-readable language to describe the build properties of your project at a high semantical level.

**Bazel is fast and reliable.** Bazel caches all previously done work and tracks changes to both file content and build commands. build is possible in a highly parallel and incremental fashion.

**Bazel is multi-platform.** Bazel runs on Linux, macOS, and Windows. Bazel can build binaries and deployable packages for multiple platforms.

**Bazel scales.** Bazel maintains agility while handling builds with 100k+ source files. It works with multiple repositories.

**Bazel is extensible.** Many languages are supported, and you can extend Bazel to support any other language or framework.

## **How do I use Bazel?**

To build or test a project with Bazel, you typically do the following:

1. **Set up Bazel.** Download and install Bazel.
2. **Set up a project workspace**, which is a directory where Bazel looks for build inputs and BUILD files, and where it stores build outputs.
3. **Write a BUILD file**, which tells Bazel what to build and how to build it.

You write your BUILD file by declaring build targets using Starlark, a domain-specific language.

A build target specifies a set of input artifacts that Bazel will build plus their dependencies, the build rule Bazel will use to build it, and options that configure the build rule.

A build rule specifies the build tools Bazel will use, such as compilers and linkers, and their configurations. Bazel ships with a number of build rules covering the most common artifact types in the supported languages on supported platforms.

1. **Run Bazel** from the command line. Bazel places your outputs within the workspace.

In addition to building, you can also use Bazel to run tests and query the build to trace dependencies in your code.

## **How does Bazel work?**

When running a build or a test, Bazel does the following:

1. **Loads** the BUILD files relevant to the target.
2. **Analyzes** the inputs and their dependencies, applies the specified build rules, and produces an action graph.
3. **Executes** the build actions on the inputs until the final build outputs are produced.

Since all previous build work is cached, Bazel can identify and reuse cached artifacts and only rebuild or retest what’s changed. To further enforce correctness, you can set up Bazel to run builds and tests hermetically through sandboxing, minimizing skew and maximizing reproducibility.

### **What is the action graph?**

The action graph represents the build artifacts, the relationships between them, and the build actions that Bazel will perform. Bazel can track changes to file content as well as changes to actions, such as build or test commands, and know what build work has previously been done. The graph also enables you to easily trace dependencies in your code.

# Best practices for Bazel

The overall goals are:

* To use fine-grained dependencies to allow parallelism and incrementality.
* To keep dependencies well-encapsulated.
* To make code well-structured and testable.
* To create a build configuration that is easy to understand and maintain.
* A project should always be able to run bazel build //... and bazel test //... successfully on its stable branch.
* You may declare third party dependencies:

1. Either declare them as remote repositories in the WORKSPACE file.
2. Or put them in a directory called third\_party/ under your workspace directory.

## **.bazelrc, the Bazel configuration file**

To avoid specifying unchanged options for every build (and other commands), you can specify options in a configuration file. .bazelrc file is a text file with a line-based grammar. Empty lines and lines starting with # (comments) are ignored. Each line contains a sequence of words, which are tokenized according to the same rules as the Bourne shell.

# C++ and Bazel

### **BUILD files**

* Each BUILD file should contain one [cc\_library](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html" \l "cc_library) rule target per compilation unit in the directory.
* granularize your C++ libraries as much as possible to maximize incrementality and parallelize the build.
* If there is a single source file in srcs, name the library the same as that C++ file’s name. This library should contain C++ file(s), any matching header file(s), and the library’s direct dependencies. For example:

cc\_library(

name = "mylib",

srcs = ["mylib.cc"],

hdrs = ["mylib.h"],

deps = [":lower-level-lib"]

)

* Use one cc\_test rule target per cc\_library target in the file. Name the target [library-name]\_test and the source file [library-name]\_test.cc. For example, a test target for the mylib library target shown above would look like this:

cc\_test(

name = "mylib\_test",

srcs = ["mylib\_test.cc"],

deps = [":mylib"]

)

# Building a C++ Project

### **Set up the workspace**

Before you can build a project, you need to set up its workspace. A workspace is a directory that holds your project’s source files and Bazel’s build outputs. It also contains files that Bazel recognizes as special:

* The WORKSPACE file, which identifies the directory and its contents as a Bazel workspace and lives at the root of the project’s directory structure,
* One or more BUILD files, which tell Bazel how to build different parts of the project.
* To designate a directory as a Bazel workspace, create an empty file named WORKSPACE in that directory.

When Bazel builds the project, all inputs and dependencies must be in the same workspace. Files residing in different workspaces are independent of one another unless linked.

### **Understand the BUILD file**

A BUILD file contains several different types of instructions for Bazel. The most important type is the build rule, which tells Bazel how to build the desired outputs, such as executable binaries or libraries. Each instance of a build rule in the BUILD file is called a target and points to a specific set of source files and dependencies. A target can also point to other targets.

cc\_binary(

name = "hello-world",

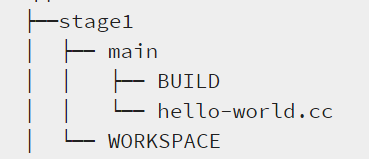
srcs = ["hello-world.cc"],

)

In the above example, the hello-world target instantiates Bazel’s built-in cc\_binary rule. The rule tells Bazel to build a self-contained executable binary from the hello-world.cc source file with no dependencies.

The attributes in the target explicitly state its dependencies and options. While the name attribute is mandatory, many are optional. For example, in the hello-world target, name is self-explanatory, and srcs specifies the source file(s) from which Bazel builds the target.

### **Build the project**



bazel build //main:hello-world

 //main: part is the location of our BUILD file relative to the root of the workspace, and hello-world is what we named that target in the BUILD file.

To test the output,

bazel-bin/main/hello-world

The above command runs the .exe file for hello-world.

### **The dependency graph**

To generate a text representation of the dependency graph (run the command at the workspace root)

bazel query --notool\_deps --noimplicit\_deps "deps(//main:hello-world)" \

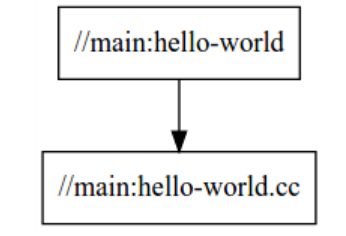
--output graph

The above command tells Bazel to look for all dependencies for the target //main:hello-world and format the output as a graph.

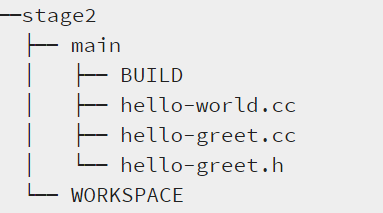
**Another method**:

bazel query --notool\_deps --noimplicit\_deps "deps(//main:helloworld)" --output graph>sample.dot

dot -Tpng sample.dot -o myDemo\_dot.png



### **Specify multiple build targets**



The build file for main.

cc\_library(

name = "hello-greet",

srcs = ["hello-greet.cc"],

hdrs = ["hello-greet.h"],

)

cc\_binary(

name = "hello-world",

srcs = ["hello-world.cc"],

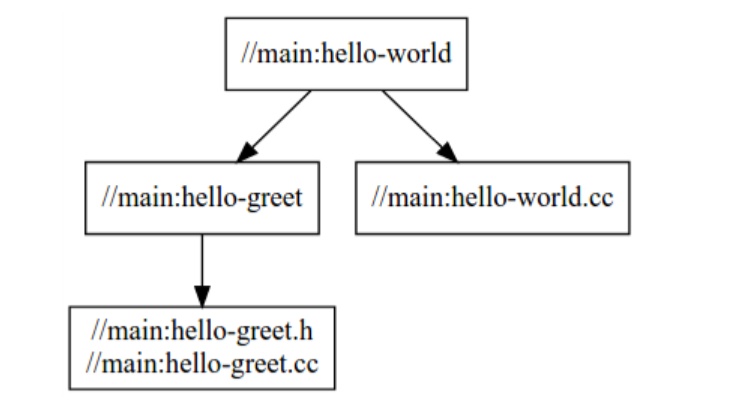
deps = [

":hello-greet",

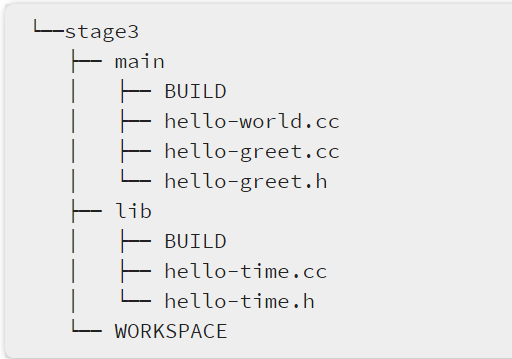
],

)

With this BUILD file, Bazel first builds the hello-greet library (using Bazel’s built-in cc\_library rule), then the hello-world binary. The deps attribute in the hello-world target tells Bazel that the hello-greet library is required to build the hello-world binary.



### **Use multiple packages**



The stage3 workspace now contains two packages, lib and main, as both directories contain BUILD file.

lib/BUILD

cc\_library(

name = "hello-time",

srcs = ["hello-time.cc"],

hdrs = ["hello-time.h"],

visibility = ["//main:\_\_pkg\_\_"],

)

main/BUILD

cc\_library(

name = "hello-greet",

srcs = ["hello-greet.cc"],

hdrs = ["hello-greet.h"],

)

cc\_binary(

name = "hello-world",

srcs = ["hello-world.cc"],

deps = [

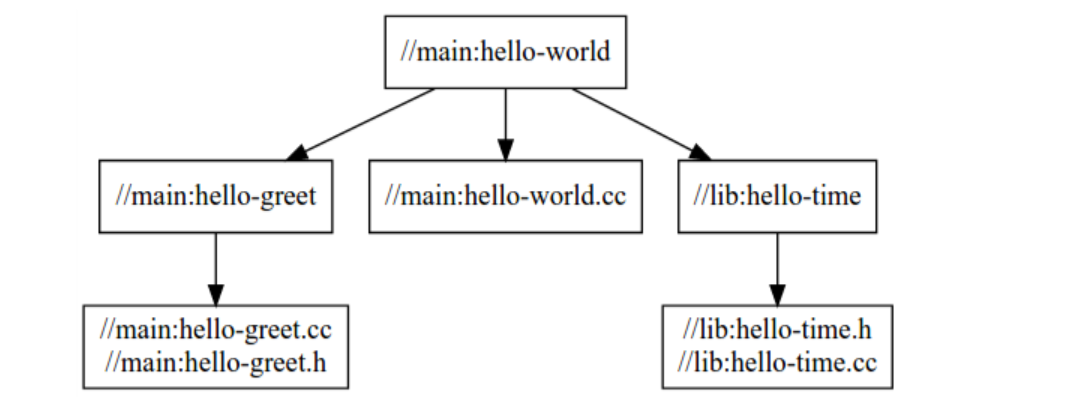
":hello-greet",

"//lib:hello-time",

],

)

As you can see, the hello-world target in the main package depends on hello-greet   in main and the hello-time target in the lib package (hence the target label //lib:hello-time)



NOTE:

//:target-name implies that the target is in the root or Workspace.

# Common C++ Build Use Cases

## **Including multiple files in a target**

You can include multiple files in a single target with glob.

cc\_library(

name = "build-all-the-files",

srcs = glob(["\*.cc"]),

hdrs = glob(["\*.h"]),

)

With this target, Bazel will build all the .cc and .h files it finds in the same directory as the BUILD file that contains this target.

## **Using transitive includes**

If a file includes a header, then the file’s rule should depend on that header’s library. Conversely, only direct dependencies need to be specified as dependencies. For example, suppose sandwich.h includes bread.h and bread.h includes flour.h. sandwich.h doesn’t include flour.h

cc\_library(

name = "sandwich",

srcs = ["sandwich.cc"],

hdrs = ["sandwich.h"],

deps = [":bread"],

)

cc\_library(

name = "bread",

srcs = ["bread.cc"],

hdrs = ["bread.h"],

deps = [":flour"],

)

cc\_library(

name = "flour",

srcs = ["flour.cc"],

hdrs = ["flour.h"],

)

Here, the sandwich library depends on the bread library, which depends on the flour library.

## **Adding include paths**

└── my-project

├── legacy

│   └── some\_lib

│   ├── BUILD

│   ├── include

│   │   └── some\_lib.h

│   └── some\_lib.cc

└── WORKSPACE

Build file

cc\_library(

name = "some\_lib",

srcs = ["some\_lib.cc"],

hdrs = ["include/some\_lib.h"],

copts = ["-Ilegacy/some\_lib/include"],

)

## **Including external libraries**

Gtest

Suppose you are using [Google Test](https://github.com/google/googletest). You can use one of the repository functions in the WORKSPACE file to download Google Test and make it available in your repository:

load("@bazel\_tools//tools/build\_defs/repo:http.bzl", "http\_archive")

http\_archive(

name = "gtest",

url = "https://github.com/google/googletest/archive/release-1.7.0.zip",

sha256 = "b58cb7547a28b2c718d1e38aee18a3659c9e3ff52440297e965f5edffe34b6d0",

build\_file = "@//:gtest.BUILD",

strip\_prefix = "googletest-release-1.7.0",

)

Then create gtest.BUILD, a BUILD file used to compile Google Test.  The gtest.BUILD would look like this:

cc\_library(

name = "main",

srcs = glob(

["src/\*.cc"],

exclude = ["src/gtest-all.cc"]

),

hdrs = glob([

"include/\*\*/\*.h",

"src/\*.h"

]),

copts = ["-Iexternal/gtest/include"],

linkopts = ["-pthread"],

visibility = ["//visibility:public"],

)

Now cc\_ rules can depend on @gtest//:main.

## **Writing and running C++ tests**

For example, we could create a test ./test/hello-test.cc such as:

#include "gtest/gtest.h"

#include "lib/hello-greet.h"

TEST(HelloTest, GetGreet) {

EXPECT\_EQ(get\_greet("Bazel"), "Hello Bazel");

}

Then create ./test/BUILD file for your tests:

cc\_test(

name = "hello-test",

srcs = ["hello-test.cc"],

copts = ["-Iexternal/gtest/include"],

deps = [

"@gtest//:main",

"//main:hello-greet",

],

)

Note that in order to make hello-greet visible to hello-test, we have to add "//test:\_\_pkg\_\_", to the visibility attribute in ./main/BUILD.

Now you can use bazel test to run the test.

bazel test test:hello-test

# C++ rules for Bazel by Starlark

To add the C++ rules repository maintained by Starlark, add the following to your WORKSPACE file:

load("@bazel\_tools//tools/build\_defs/repo:http.bzl", "http\_archive")

http\_archive(

name = "rules\_cc",

urls = ["https://github.com/bazelbuild/rules\_cc/archive/TODO"],

sha256 = "TODO",

)

Then, in your BUILD files, import and use the rules (Not only cc\_library, others can also be added):

load("@rules\_cc//cc:defs.bzl", "cc\_library")

cc\_library(

...

)

# rules\_boost -- Bazel build rules for Boost

To use these rules, add the following to your WORKSPACE file:

load("@bazel\_tools//tools/build\_defs/repo:git.bzl", "git\_repository")

git\_repository(

name = "com\_github\_nelhage\_rules\_boost",

commit = "1e3a69bf2d5cd10c34b74f066054cd335d033d71",

remote = "https://github.com/nelhage/rules\_boost",

shallow\_since = "1591047380 -0700",

)

load("@com\_github\_nelhage\_rules\_boost//:boost/boost.bzl", "boost\_deps")

boost\_deps()

You can then use libraries in deps through the @boost repository, for example @boost//:algorithm.