First look: Bazel Build System

Google has an interesting build system for their internal applications and projects called Blaze. Bazel is the open-source version of that project.

According to its documentation, Bazel offers the following advantages:

- **High-level build language**. Bazel uses an abstract, human-readable language to describe the build properties of your project. Unlike other tools, Bazel operates on the concepts of libraries, binaries, scripts, and data sets, shielding you from the complexity of writing individual calls to tools such as compilers and linkers
- **Bazel is fast and reliable**. Bazel caches all previously done work and tracks changes to both file content and build commands. This way, Bazel knows when something needs to be rebuilt, and rebuilds only that.
- **Bazel is multi-platform**. Bazel is cross-platform (Linux, Mac, and Windows). Bazel can build binaries and deployable packages for multiple platforms, including desktop, server, and mobile, from the same project
- **Bazel scales**. Bazel maintains agility while handling builds with 100k+ source files. It works with multiple repositories and user bases in the tens of thousands
- **Bazel is extensible**. Bazel supports many languages, and you can extend Bazel to support any other language or framework

The idea is that we trade using older versions of specific software that is added to our repository for reproducible, fast, multi-language builds.

If you worked with Make in the early days of the web or when trying to compile C/C++ code the idea behind Bazel will look familiar.

In the next sections, we'll discuss the following items

- Is Bazel the tool for your project?
- Bazel installation via Homebrew for the Mac
- Basic configuration for our setup

In a follow-up post, we'll look at toolchains and build files for different languages.

Is Bazel the right tool for the project?

Before we jump into installing and configuring Bazel we should take a step back and ask ***if Bazel is the right tool for the project*** and what are the alternatives.

Bazel is the open-source version of Google's Blaze build system used to build everything in their huge monorepo from C++ to Go to iOS and Android.

Bazel is best for medium to larger projects that use multiple languages in a monorepo architecture; for example, a front-end written in a framework like Angular, React, or Vue or some other Typescript/Javascript-based architecture, and a front end written in Java or Go.

You have to be careful in evaluating the tradeoffs between the complexity of learning a new tool and, potentially, migrating your code to a monorepo architecture, versus the advantages it provides by keeping all your code in one repository and the faster builds it generates.

Installing Bazel on macOS with Homebrew

As documented in Installing using Homebrew the installation process is simple:

> If you don't have it installed, then you must install Homebrew first. The command is:

```
/bin/bash -c "$(curl -fsSL \
https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

This will make the `brew` command available. You can then install Bazel by running:

```
brew install bazel
```

Verify a successful install with the following command:

```
bazel --version
```

This should return a version like `bazel 3.7.0-homebrew`. If it doesn't return a similar string or reports `bazel command not found` or a similar error then the installation didn't work. Try installing it again.

Configuring Bazel

Bazel uses its own DSL, Starlark for its configuration files and build-time rules. This may take a little time as it is similar, but not exactly the same as, Python.

Global Configuration

To use the same Bazel settings for the project, create a `bazel.rc` file at the root of the Bazel workspace. These settings will work in all projects in the workspace, regardless of the language.

> Adding it to the workspace will check the file into version control and

propagate it to others working on the project as well as the CI system.

```
################################
# Directory structure          #
################################

# Artifacts are typically placed in a directory called "dist"
# Be aware that this setup will still create a bazel-out symlink in
# your project directory, which you must exclude from version control and
# editor's search path.
build --symlink_prefix=dist/

################################
# Output                       #
################################

# A more useful default output mode for bazel query, which
# prints "ng_module rule //foo:bar" instead of just "//foo:bar".
query --output=label_kind

# By default, failing tests don't print any output, it's logged to a
# file instead.
test --test_output=errors

################################
# Typescript                   #
################################
# Make TypeScript compilation fast, by keeping a few
# copies of the compiler running as daemons, and cache
# SourceFile ASTs to reduce parse time.
build
  --strategy=TypeScriptCompile=worker
```

Note how we install both a Bazel-specific version and the tool itself. The @bazel package contains Bazel specific information

Target Configuration Examples

The following example, taken from Chapter 18 of Software Engineering at Google illustrates Blaze / Bazel concepts. We'll discuss them in more detail along with specifics for each type of content we're building later in the post.

```
java_binary(
  name = "MyBinary",
  srcs = ["MyBinary.java"],
  deps = [
  ":mylib",
  ],
)

java_library(
  name = "mylib",
  srcs = ["MyLibrary.java", "MyHelper.java"],
  visibility = ["//java/com/example/myproduct:__subpackages__"],
  deps = [
    "//java/com/example/common",
    "//java/com/example/myproduct/otherlib",
    "@com_google_common_guava_guava//jar",
  ],
)
```

In Bazel, BUILD files define targets — the two types of targets in the example are `java_binary` and `java_library`. Every target corresponds to an artifact that can be created by the system: binary targets produce binaries that can be executed directly, and library targets produce libraries that can be used by binaries or other libraries.

> Every target has a set of attributes:
>
> - `name` defines how we reference the target on the command line and from other targets
> - `srcs` define the source files that must be compiled to create the artifact for the target
> - `deps` define other targets that must be built before this target and linked

into it There are three different types of dependencies

- Within the same package (MyBinary's dependency on `:mylib`)
- A different package in the same source hierarchy (mylib's dependency on `//java/com/example/common`)
- A third-party artifact outside of the source hierarchy (mylib's dependency on `@com_google_common_guava_guava//jar`

Each source hierarchy is called a workspace and is identified by the presence of a special WORKSPACE file at the root.

Like with Ant, users perform builds using Bazel's command-line tool. To build the MyBinary target, a user would run `bazel build :MyBinary`. Upon entering that command for the first time in a clean repository, Bazel would do the following:

1. Parse every BUILD file in the workspace to create a graph of dependencies among artifacts.
2. Use the graph to determine the transitive dependencies of MyBinary; that is, every target that MyBinary depends on and every target that those targets depend on, recursively.
3. Build (or download for external dependencies) each of those dependencies, in order. Bazel starts by building each target that has no other dependencies and keeps track of which dependencies it still needs to build for each target
    1. As soon as all of a target's dependencies are built, Bazel starts building that target. This process continues until every one of MyBinary's transitive dependencies has been built.
4. Build MyBinary to produce a final executable binary that links in all of the dependencies that were built in step 3

Bazel uses the concept of toolchains to ensure reproducible builds. A toolchain contains a set of tools and other properties defining how a type of target is built on a particular platform.

For larger projects or projects with large teams of engineers, Bazel provides remote caching and remote execution. They are mentioned here for completeness but they are not required if all you need is to build on your local machine.