Bazel Workshop

Introduction

The goal of this workshop is

- to familiarize you with Bazel
- to teach you how to use it effectively as your build system when you're working on your code.

The project you're working on contains components written in different languages: C++ and Rust.

Right now you may still be using the respective native build tools for each of these languages.

As we will see today, Bazel offers a few improvements over those:

- It supports large scale projects.
- It supports multiple programming languages.
- It can manage tests and workflows.
- It can help obtainaing reproducible builds.
- And more...

That being said ...

- It may take some getting used to.
- Some of your workflows or tooling may need to be adapted to fit within Bazel.

Bazel makes the assumption that you will want to manage all your transitive dependencies with it.

It works well by fully embracing this approach.

Setup

The environment for this workshop is provided in a Docker image with Bazel pre-installed which we've provided.

Additionally, you must have the accompanying git repository cloned.

```
https://github.com/tweag/bazel-workshop.git
```

Once in the repository you can build the image

```
$ docker build . -t workshop
```

Getting Started

To run the image with the example projects mounted you can run

```
$ docker run \
    -v$REPO/workspace:/home/user/workspace \
    -it workshop:latest \
    bash
```

where `<REPO>` is the path in which you cloned the git repository for the workshop.

Bazel basic commands

For the rest of the presentation, we will be demonstarting things on the `01_cpp` project.

We can build one or multiple targets in our project using the `bazel build` command.

```
bazel build //...
```

`//...` is a pattern for all targets in the repository. We'll introduce patterns and labels in detail later.

Bazel is also a test system.

We can ask Bazel to run some or (in this case) all of our tests:

```
bazel test //...
```

We can also directly run executable targets:

```
bazel run //:hello-cc
```

As previously, `//:hello-cc` is the **label** of the executable target.

Relation to the C++, Rust tools?

How does Bazel relate to the compilers and build tools that you may be already familiar with?

Bazel provides an abstraction on top of these tools.

With Bazel you can define all your build targets, C++, Rust, or otherwise, in a common language.

Bazel will then invoke the appropriate compilers and tools to build your code.

Ruleset

The way that Bazel knows how to do so is by so called "rulesets".

- they implement the logic for calling compilers, linking, and in general anything required to build a certain language.

They can be either builtin (for example C++ rules) or written as extension (for example `rules_rust`).

Anatomy of a Bazel project

Let's analyse the files in our minimal Bazel project, located in `~/workshop/01_cpp`.

```
> cd ~/workshop/01_cpp
> tree
├── bazel-* -> ...
├── BUILD
├── greeter
│   ├── greeter.cpp
│   ├── greeter.h
│   └── greeter_test.cpp
├── hello-cc
│   └── main.cpp
└── WORKSPACE
```

`WORKSPACE`

Every Bazel project has **one** `WORKSPACE` file.

It marks the root of the repository.

It is also where external dependencies are declared.
We'll look at this file a bit later.

`bazel-*` (documentation)

These are convenience symlinks generated by Bazel.

- `bazel-bin` and `bazel-out` provide access to generated files, e.g. compiled binaries.

```
bazel build //:hello
ls -1 bazel-bin
./bazel-bin/hello-cc
...
```

- `bazel-testlogs` provides access to test logs.
- `bazel-01_cpp` (generally `bazel-<dirname>`) provides access to the "execroot". This is where `bazel run` executes commands.

Source files (e.g. `greeter.cpp`)

Your project's source files. These have Bazel labels assigned to them just as targets and generated files do.

E.g. `//greeter:greeter.h` addresses a source file, while `//:greeter` addresses a C++ binary target.

`BUILD` files

Bazel provides a language to describe build targets and their dependencies.

Let's look at the "BUILD" file for our simple project - `~/workshop/01_cpp/BUILD`.

```
load("@rules_cc//cc:defs.bzl",
    "cc_binary", "cc_library")
# Define a C++ library target.
cc_library(
    name = "greeter",
    srcs = ["greeter/greeter.cpp"],
    hdrs = ["greeter/greeter.h"],
    deps = [ ],
    # We define the visibilty explicitly,
    # so it can be accessed from other packages.
    # https://bazel.build/concepts/visibility
    visibility = ["//visibility:public"],
)
# Define a C++ executable target (binary).
cc_binary(
    name = "hello-cc",
    srcs = ["hello-cc/main.cpp"],
    deps = [
      ":greeter",
    ],
)
```

```
# Define a C++ test suite.
cc_test(
    name = "tests",
    srcs = ["greeter/greeter_test.cpp"],
    deps = [
        "//:greeter",
        "@com_google_googletest//:gtest_main",
    ],
)
```

Each BUILD file marks a **package** and describes the **targets** of this package.

The BUILD file we're looking at exists at the top-level, so it marks the package `//`.

A nested BUILD file, e.g. `some/dir/BUILD.bazel`, marks a nested package, e.g. `//some/dir`.

Targets in a package are addressed with labels such as `//:hello-cc` or `//some/dir:some-target`.

We will promptly discuss labels in detail.

Bazel's configuration language is called Starlark. Its syntax is heavily "inspired" by Python.

BUILD files are meant to be declarative:

- They declare build targets and their metadata.

As such, BUILD files only allow a subset of Starlark. BUILD files are not meant to contain complex logic.

Bazel targets (documentation)

Bazel targets

Bazel understands different **kinds** of targets

In this case we see `cc_library`, `cc_binary`, and `cc_test`.

Each target is usually defined by a **rule**.

So `//:hello-cc` is the label of a target of kind `cc_binary`, which is defined by the `cc_binary` rule.

Bazel targets

While this example only contains C++ targets...

It is completely fine and actually common to have targets for **different languages**,
along with cross-dependencies between them.

We will see an example with multiple languages later.

Labels (documentation)

Labels

Labels are names that refer to targets.

Many labels can refer to the same target, but each label refers to at most one target. (i.e. there is a function `Label -> Target`)

The most common label would look something like `@repo//path/to/package:target`. Let's take it apart.

Labels

@repo//path/to/package:target

`repo` refers to the repository we're building.

This is optional and if omitted, defaults to the current repo.

This is useful when we only want to build **external** dependencies.

Labels

@repo//**path/to/package**:target

`path/to/package` is the path to the package we're building.

Spelled out, this is a directory with a path `path/to/package` with a `BUILD` file in it, relative to the repository root, marked by the `WORKSPACE` file.

Labels

@repo **//** path/to/package:target

`//` designates the root of the repository we're building.

This is optional. If we omit it, Bazel assumes we're referring to a path relative to the current directory.

```
cd path/to
bazel build package:target
```

does the same thing as

```
bazel build //path/to/package:target
```

Labels

@repo//path/to/package:**target**

`target` is the target we want to refer to

It must be available in the `BUILD` file for the package we're building, i.e. from `path/to/package/BUILD.bazel`

There is some special label syntax, some of which we'll explore in the next section.

`bazel query`ing targets (tutorial, documentation)

Bazel provides tools to query the build targets defined in a project, their metadata, and their interdependencies.

All targets

We can query for all targets in the top-level package.

```
bazel query //:all
```

We also saw the `...` target earlier. For our `01_cpp` example, these coincide, but in general, `...` refers to all targets in **all packages** in the workspace.
(In truth, it's a bit more subtle than that. You can read more about targets in the Bazel docs on targets.)

All files

We can also query for all ruls **and** all files (source or generated) in the top-level package.

```
bazel query //:*
```

Specific target

Let's query for some details of the `hello` target.

```
bazel query //:hello-cc
```

Get the kind

```
bazel query //:hello-cc --output label_kind
```

Querying for dependencies

Finally, as a small showcase of a more "real" query, here's how you could list all the C++ dependencies of a project.

```
bazel query 'deps(//:hello-cc)' --noimplicit_deps
```

The `--noimplicit_deps` flag excludes some things which are considered "implicit dependencies", not specified by the BUILD file but instead added by the Bazel (such as compilers)
We exclude them, since they would heavily obfuscate the result here.

It's documented here

We can also output a format suitable for consumption by graphviz

```
bazel query 'deps(//:hello-cc)' --noimplicit_deps --output
```

`--output build` (documentation)

Output information in "the format of a BUILD file".

Useful for seeing the result of expansions and macros.

```
bazel query //:hello-cc --output build
```

Fast and Correct

Bazel's tagline is that it offers fast and correct builds.

It aims to provide fast builds by aggressively caching outputs, parallelizing the build.

It also provides advanced features, like remote execution, to scale to very large projects.

Let's look at Bazel's caching features first.

Caching

Bazel has extensive caching features.

- It can keeps track of which targets are affected or unaffected by a change. So it won't rebuild a target that didn't change. This is similar to what tools like `make` provide.
- It can cache build outputs on disk or remotely and re-use them when a previously run action is requested again with the same inputs.
  For example, if you switch to a feature branch and then back to the main branch, then you will not have to rebuild everything from scratch.
- It caches successful test results. So you don't have to rerun a test case if none of its inputs changed.

```
export CACHE=~/.cache/bazel/disk
```

- Run `bazel clean --expunge` to force a rebuild.
- Run `bazel build //:hello-cc --disk_cache=$CACHE`
- Change `main` in `main.cpp`.
- Run `bazel build //:hello-cc --disk_cache=..` again.
- Undo the change in `main.cpp`.
- Run `bazel build //:hello-cc --disk_cache=..` once more.
  This time Bazel should use the cached results and print a message about "disk cache hit".

Sandboxing

- Caching, parallelism, remote execution rely on **correct** builds. Meaning all dependencies of a target must be declared, so that Bazel knows to build those first.

- Correct builds can be difficult to ensure.
  For example in a system like `make` it is possible to forget to declare a dependency.
  What's worse, build actions might still run successfully because the dependency just happens to already be built before its dependees due to how actions are scheduled.
- Such build descriptions are brittle: Changing parallelism or some other part of the build graph might change this ordering, and suddenly a seemingly unrelated target fails to build.

- Bazel strives to enforce correctness by enforcing hermeticity.
  A build must declare all its dependencies, otherwise it will not succeed. It does that by sandboxing build actions and only exposing declared dependencies to a target.
- Note that these mechanisms only operate on the level of sources and artifacts from within the project.
  Bazel, in its default configuration, does not go so far to isolate common system dependencies like system libraries or the C compiler.
  This level of isolation can be achieved, but requires additional setup, e.g. Nix, or sandboxfs, or Docker images.

Let's see sandboxing in action.

- Take the `//:hello-cc` target and remove its dependency on `//:greeter` in the BUILD file.
- Try to build the target and see it fail.
- Use the Bazel flags `--subcommands --sandbox_debug` to inspect Bazel's sandbox directory of a failed action.

```
bazel clean --expunge
bazel build //:hello-cc --subcommands --sandbox_debug
```

The `--sandbox_debug` flag makes Bazel **not** delete the created sandbox dirs after building, which is useful for debugging (documentation).
It also prints some of these directories, which would be rather hard to find otherwise.

```
DIR=~/.cache/bazel/_bazel_user/57368225eee2fa94d64fb430d40
find "$DIR" -iname '*greeter*'
find -L bazel-out -iname '*greeter*'
```