Documentation

Using Bazel

Build Encyclopedia

Overview (/versions/master/docs/be/overview.html)

Concepts

Rules

Extensions

Overview (/versions/master/docs/skylark/concepts.html)

Language (/versions/master/docs/skylark/language.html)

Macros (/versions/master/docs/skylark/macros.html)

Rules (/versions/master/docs/skylark/rules.html)

Depsets (/versions/master/docs/skylark/depsets.html)

Aspects (/versions/master/docs/skylark/aspects.html)

Repository rules (/versions/master/docs/skylark/repository_rules.html)

Challenges of writing rules (/versions/master/docs/rule-challenges.html)

Reference (/versions/master/docs/skylark/lib/skylark-overview.html)

Examples (/versions/master/docs/skylark/cookbook.html)

Packaging rules (/versions/master/docs/skylark/deploying.html)

Documenting rules (https://skydoc.bazel.build)

Style guide for BUILD files (/versions/master/docs/skylark/build-style.html)

Style guide for bzl files (/versions/master/docs/skylark/bzl-style.html)

Getting Started with Bazel

Setup

Use the installation instructions (/docs/install.html) to install a copy of Bazel on your machine.

Using a Workspace

All Bazel builds take place in a *workspace* (/docs/build-ref.html#workspaces), a directory on your filesystem that contains source code for the software you want to build, as well symbolic links to directories that contain the build outputs (for example, `bazel-bin` and `bazel-out`). The location of the workspace directory is not significant, but it must contain a file called `WORKSPACE` in the top-level directory; an empty file is a valid workspace. The `WORKSPACE` file can be used to reference external dependencies (/docs/external.html) required to build the outputs. One workspace can be shared among multiple projects if desired.

```
$ touch WORKSPACE
```

Creating a Build File

To know which targets can be built in your project, Bazel inspects `BUILD` files. They are written in Bazel's build language which is syntactically similar to Python. Usually they are just a sequence of declarations of rules. Each rule specifies its inputs, outputs, and a way to compute the outputs from the inputs.

The rule probably most familiar to people who have used `Makefile`s before (as it is the only rule available there) is the genrule (/docs/be/general.html#genrule), which specifies how the output can be generated by invoking a shell command.

```
genrule(
  name = "hello",
  outs = ["hello_world.txt"],
  cmd = "echo Hello World > $@",
)
```

The shell command may contain Make variables (/docs/be/make-variables.html).

Using the above `BUILD` file, you can ask Bazel to generate the target.

```
$ bazel build :hello
.
INFO: Found 1 target...
Target //:hello up-to-date:
  bazel-genfiles/hello_world.txt
INFO: Elapsed time: 2.255s, Critical Path: 0.07s
```

We note two things. First, targets are normally referred to by their label (/docs/build-ref.html#labels), which is specified by the name (/docs/be/general.html#genrule.name) attribute of the rule. (Referencing them by the output file name is also possible, but this is not the preferred way.) Second, Bazel puts the generated files into a separate directory (the `bazel-genfiles` directory is actually a symbolic link) so as not to pollute your source tree.

Rules may use the output of other rules as input, as in the following example. Again, the generated sources are referred to by their label.

```
genrule(
  name = "hello",
  outs = ["hello_world.txt"],
  cmd = "echo Hello World > $@",
)

genrule(
  name = "double",
  srcs = [":hello"],
  outs = ["double_hello.txt"],
  cmd = "cat $< $< > $@",
)
```

Finally, note that, while the genrule (/docs/be/general.html#genrule) might seem familiar, it usually is *not* the best rule to use. It is preferrable to use one of the specialized rules (/docs/be/overview.html#rules) for various languages.

Next Steps

Next, check out the tutorial on building java (/docs/tutorial/java.html) or C++ (/docs/tutorial/cpp.html) programs.