Riot Bazel Training

Day 1, Lectures 3 & 4

Schedule

Day 1

1. ~~Introduction to Bazel~~
2. ~~Using Bazel~~
3. Building Java, Python, Protobufs
4. CLI and Tooling

flare.build

4. Building Java, Python, Protobufs with Bazel

flare.build

3.1
rules_jvm_external

Motivation

Rules jvm external make it easy to use any maven dependency, In this example we will. This is the only good way to get 3rd party dependencies from “maven central” or jcenter or other maven artifact repositories; in this case when they say maven they mean “java modules”.

By using **rules_jvm_external**, bazel downloads 3rd party JARs and lets you depend on them as deps in your libraries and correctly links them up in the classpath at compile time (and even lets you navigate to code in idea).

Any and all real-world JVM projects should use **rules_jvm_external** to get any 3rd party dep they use

Setting up WORKSPACE

```
RULES_JVM_EXTERNAL_TAG = "3.3"
RULES_JVM_EXTERNAL_SHA = "d85951a92c0908c80bd8551002d66cb23c3434409c814179c0ff026b53544dab"

http_archive(
   name = "rules_jvm_external",
   sha256 = RULES_JVM_EXTERNAL_SHA,
   strip_prefix = "rules_jvm_external-%s" % RULES_JVM_EXTERNAL_TAG,
   url =
     "https://github.com/bazelbuild/rules_jvm_external/archive/%s.zip" %
     RULES_JVM_EXTERNAL_TAG,
)
```

Import `maven_install()`

Note the location of maven_install.json

```
load("@rules_jvm_external//:defs.bzl", "maven_install")
load("@rules_jvm_external//:specs.bzl", "maven")

maven_install(
    name = "maven",
    artifacts = [], # maven dependencies come here
    maven_install_json = "//:maven_install.json",
    repositories = [
        "https://jcenter.bintray.com/",
        "https://maven.google.com",
        "https://repo1.maven.org/maven2",
    ],
)
```

- Private repositories are supported through HTTP Basic auth
  Eg: "`http://username:password@localhost:8081/artifactory/my-repository`",

Use `pinned_maven_install()`

Use pinned_maven_install to “pin” your downloaded and transitive dependencies versions

```
load("@maven//:defs.bzl", "pinned_maven_install")

pinned_maven_install()
```

Defining Dependencies – pom.xml to `maven_install()`

```
<dependency>
    <groupId>com.github.scopt</groupId>
    <artifactId>scopt_2.11</artifactId>
    <version>4.0.0-RC2</version>
</dependency>
```

Should be defined as following in `artifacts[]` of `maven_install()`:

```
artifacts = [
    "com.github.scopt:scopt_2.11:4.0.0-RC2",
],
```

Defining Dependencies:
pom.xml to `maven_install()`

```
<dependency>
      <groupId>com.github.scopt</groupId>
      <artifactId>scopt_2.11</artifactId>
      <version>4.0.0-RC2</version>
</dependency>
```

Alternatively, can be defined as maven.artifact, so we can gain more control over artifacts:

```
artifacts = [
      maven.artifact(
       "com.github.scopt",
       "scopt_2.11",
       "4.0.0-RC2",
       testonly = True,
    ),
],
```

Generate `maven_install.json`

Commands to work with maven dependencies:

Pin dependencies:

```
$ bazel run @maven//:pin
```

Change dependencies:

```
$ bazel run @unpinned_maven//:pin
```

Using Dependencies: pom.xml to BUILD.bazel

```
<dependency>
     <groupId>com.github.scopt</groupId>
     <artifactId>scopt_2.11</artifactId>
     <version>4.0.0-RC2</version>
</dependency>
```

Should be defined as following in target’s deps in BUILD.bazel:

```
"@maven//:com_github_scopt_scopt_2_11",
```

Lab

`rules_jvm_external`

Objective:

Build a java library, which will depend on `org.apache.commons.math3.complex.Complex`

Additionally, build and run tests for the library.

Functionally:

It takes 2 real numbers as input and returns a pretty printed complex number.

3.2

rules_python_external

Motivation

rules_python_external should be used as a **drop in replacement** for the python rules in all instances of use. It has the same API, but **addresses most of packaging issues** and a number of other things which currently broken in the official rules **bazelbuild/rules_python**

They solve

- Transitive dependency resolution
- Minimal runtime dependencies
- Support for spreading *purelibs*
- Support for namespace packages
- Fetches pip packages only for building Python targets
- Reproducible builds

Setting up WORKSPACE

```
rules_python_external_version = "0.1.5"

http_archive(
    name = "rules_python_external",
    sha256 = "",  # Fill in with correct sha256 of your COMMIT_SHA version
    strip_prefix = "rules_python_external-{version}".format(
        version = rules_python_external_version
    ),
    url = "https://github.com/dillon-giacoppo/rules_python_external/archive/v{version}.zip".format(
        version = rules_python_external_version
    ),
)

# Install the rule dependencies
load("@rules_python_external//:repositories.bzl", "rules_python_external_dependencies")
rules_python_external_dependencies()
```

Import pip_install()

```
load("@rules_python_external//:defs.bzl", "pip_install")

pip_install(
    name = "pip",
    requirements = "//:requirements.txt",
)
```

Python Dependencies – *Importing*

Adding the "pip install":

```
load("@rules_python_external//:defs.bzl", "pip_install")
pip_install(
    name = "pip",
    requirements = "//:requirements.txt",
)
```

Create requirements.txt file and define dependencies:

```
numpy==1.19.1
pandas==1.1.0
tensorflow==2.3.0
matplotlib==3.1.2
Pillow==7.2.0
```

Python Dependencies – *Referencing*

In order to reference and use the dependencies:

```
load("@pip//:requirements.bzl", "requirement")

py_binary(
   deps = [
       requirement("tensorflow"),
   ],
)
```

Lab

`rules_python_external`

Objective:

Build a python application, which depends on tensorflow, numpy and other data science libraries. Additionally, build and run test for the application.

Functionally:

Using a pre-trained keras model, application takes as an input the image from test set and outputs prediction and expected tag for the image.

flare.build

3.3. Using Protobuf for Java and Python with Bazel

Motivation

Protocol buffers are Google's **language-neutral**, platform-neutral, extensible mechanism for serializing structured data.

Once declared in ***.proto** files, they can be compiled to specific languages. Set of rules supporting protobufs in bazel is called **rules_proto**.

However, as we will see further, rules_proto don't support all languages, so we will also look at **rules_proto_grpc** as an example of external rules we will use to support proto in python.

Setting up WORKSPACE

```
http_archive(
    name = "rules_proto",
    sha256 = "602e7161d9195e50246177e7c55b2f39950a9cf7366f74ed5f22fd45750cd208",
    strip_prefix = "rules_proto-97d8af4dc474595af3900dd85cb3a29ad28cc313",
    urls = [
"https://mirror.bazel.build/github.com/bazelbuild/rules_proto/archive/97d8af4dc474595af3900dd85cb3a29ad28c
c313.tar.gz",
"https://github.com/bazelbuild/rules_proto/archive/97d8af4dc474595af3900dd85cb3a29ad28cc313.tar.gz",
    ],
)

load("@rules_proto//proto:repositories.bzl", "rules_proto_dependencies", "rules_proto_toolchains")

rules_proto_dependencies()

rules_proto_toolchains()
```

Using `rules_proto` rules

```
load("@rules_proto//proto:defs.bzl", "proto_library")

proto_library(
    name = "sample_proto",
    srcs = [":sample.proto"],
)
```

- `proto_library` outputs compiled protobuf *.bin, which is used in language-specific proto libraries
- You may notice that it takes a while to build it for the first time, this is because bazel had to pull a protobuf compiler itself.

Using Generated Code

```
load("@rules_java//java:defs.bzl", "java_proto_library")

java_proto_library(
    name = "sample_proto_java",
    deps = [":sample_proto"],
)
```

Once protobuf file if compiled, we can proceed and define it as dependency to `java_proto_library` target “`sample_proto_java`”. This code needs to reside in proto repository, since it can be reused by multiple java repositories.

Using Generated Code, ctd.

```
java_binary(
   name = "sample",
   srcs = ["sample.java"],
   deps = [
       "//src/main/proto:sample_proto_java",
   ],
)
```

Now `sample_proto_java` can be used in java_binary to provide access to generated proto code.

Complications with Python Proto

`java_proto_library` is a part of `rules_java`, but Bazel doesn't support python proto by default. As a result python rules for working with compiled proto should be pulled from external source.

To learn which proto rules are included in Bazel by default:
see Build Encyclopedia.

Setting up WORKSPACE *– Python*

```
http_archive(
   name = "rules_proto_grpc",
   sha256 = "5f0f2fc0199810c65a2de148a52ba0aff14d631d4e8202f41aff6a9d590a471b",
   strip_prefix = "rules_proto_grpc-1.0.2",
   urls = ["https://github.com/rules-proto-grpc/rules_proto_grpc/archive/1.0.2.tar.gz"],
)

load("@rules_proto_grpc//:repositories.bzl", "rules_proto_grpc_repos", "rules_proto_grpc_toolchains")

rules_proto_grpc_toolchains()

rules_proto_grpc_repos()
```

Using Generated Code

```
load("@rules_proto_grpc//python:defs.bzl", "python_proto_library")

python_proto_library(
    name = "sample_proto_python",
    deps = [":sample_proto"],
)
```

Similar to java, now we need to use python_proto_library to make compiled proto available.

Using Generated Code, ctd.

```
py_binary(
    name = "sample",
    srcs = ["sample.py"],
    deps = [
        "//src/main/proto:sample_proto_python",
    ],
)
```

Now `sample_proto_python` can be used in `py_binary` target to provide access to generated proto code.

Package & Target Visibility

By default, targets can depend only to targets in the same package, since proto is typically a separate repo, it's important to reiterate on the concept of visibility. Visibility can be defined on package or target level:

```
package(default_visibility = ["//visibility:private"])
load("@build_bazel_rules_typescript//:defs.bzl", "ts_library")

ts_library(
      //omitted
)

python_proto_library(
   name = "sample_proto_python",
   visibility = ["//src/main/python/sample:__pkg__"],
   deps = [":sample_proto"],
)
```

Package & Target Visibility

`["//visibility:public"]`	Anyone can use this. This visibility should be considered to be a public API and should not be used unless we do intend to expose a public API from workspace/repo.
`["//visibility:private"]`	Only targets in this package can use this
`["//some/demo_package:__pkg__", "//other/package:__pkg__"]`	Only targets in some/package and other/package have access to this
`["//my_project:__subpackages__", "//other:__subpackages__"]`	Only targets in packages project or other or in one of their sub-packages have access to this
`["//some/demo_package:my_package_group"]`	A package group is a named set of package names

Lab

Using Protobuf for Java and Python with Bazel

Objective:

Compile **protobuf.proto** definition and use it in **java** and **python** code

4. CLI & Tooling

flare.build

4.1. Build Querying

Asking Bazel Questions – *Query Types*

`query`	queries target graph, the output of loading phase
`sky query`	an alternative implementation of query
`cquery`	queries configured target graph (correctly handles select())
`aquery`	queries action graph
`genquery`	general bazel rule to run queries and save result to a file

Useful Queries

- List all packages in a workspace

```
bazel query '//...' --output package
```

- List all rules in a workspace

```
bazel query 'kind(rule, //...)' --output label_kind
```

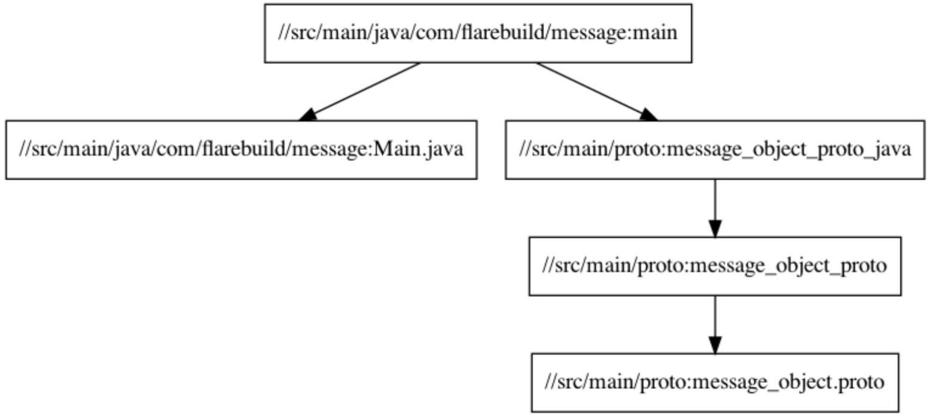
- Find all dependencies of //packages/core

```
bazel query "deps(//packages/core)"
```

- More queries in labs...

Java Proto Example Dependencies

```
$ bazel query 'deps(//src/main/java/com/flarebuild/message:main)' --notool_deps
--noimplicit_deps --output graph | dot -Tpng > /tmp/test.png
```



//src/main/java/com/flarebuild/message:main
//src/main/java/com/flarebuild/message:Main.java
//src/main/proto:message_object_proto_java
//src/main/proto:message_object_proto
//src/main/proto:message_object.proto

flare.build

4. 2 Execution Log & Profiling

Execution Log

The execution log can be used to list **all Bazel's executed actions,** along with all inputs and outputs. Useful to collect analytics or, for example, it may be helpful to troubleshoot remote cache hits (see Chapter 8).

Currently, Bazel supports 3 types of flags to produce log files:

```
bazel build //your:target --execution_log_json_file=/tmp/log.json

bazel build //your:target --execution_log_binary_file=/tmp/log.bin

bazel build //your:target --experimental_execution_log_file=/tmp/log.txt
```

Execution Log

- This is an example of information available in execution log. It contains extensive info about executed actions:
    - Inputs, outputs, whether action result was cached and much more
    - Full log format is described in the protobuf scheme called 'spawn.proto'

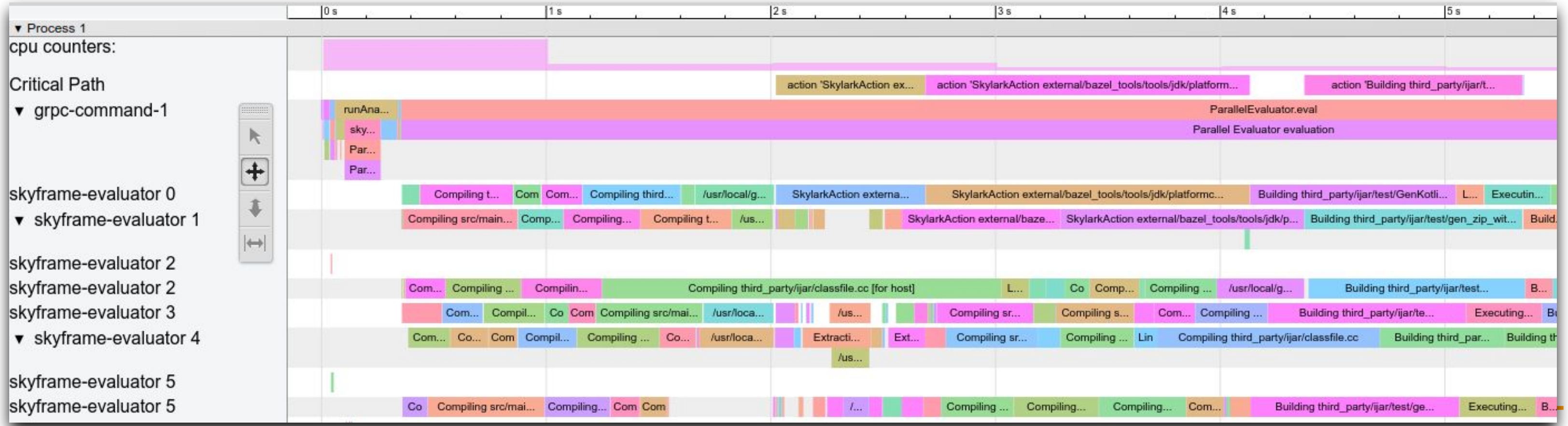
```
command_args: "bazel-out/darwin-opt/bin/external/libjpeg_turbo/_objs/jpeg/jcmaster.o"
environment_variables {
 name: "APPLE_SDK_PLATFORM"
 value: "MacOSX"
}//omitted
inputs {
 path: "bazel-out/darwin-opt/bin/external/libjpeg_turbo/jconfig.h"
 digest {
   hash: "317659a520922996ac746b3c589c441148eccfacdf58e66bc1100593b65ec3c5"
   size_bytes: 1985
   hash_function_name: "SHA-256"
   }
}//omitted
listed_outputs:
"bazel-out/darwin-opt/bin/external/libjpeg_turbo/_objs/jpeg/jcmaster.o"
remotable: true
cacheable: true
progress_message: "Compiling external/libjpeg_turbo/jcmaster.c"
mnemonic: "CppCompile"
actual_outputs {
 path: "bazel-out/darwin-opt/bin/external/libjpeg_turbo/_objs/jpeg/jcmaster.o"
 digest {
   hash: "7ba7e5fc36d77d7f8a17fd1285db462a69c054482998625461fcef5f055752cb"
   size_bytes: 8572
   hash_function_name: "SHA-256"
   }
}
runner: "remote cache hit"
remote_cache_hit: true
```

Profiling

Profiling is useful for finding build and test bottlenecks

Bazel writes a JSON profile which can be later opened with Chrome

```
$ bazel build //.... --profile=/tmp/profile.gz
```



0 s
1 s
2 s
3 s
4 s
5 s
Process 1
cpu counters:
Critical Path
grpc-command-1
skyframe-evaluator 0
skyframe-evaluator 1
skyframe-evaluator 2
skyframe-evaluator 2
skyframe-evaluator 3
skyframe-evaluator 4
skyframe-evaluator 5
skyframe-evaluator 5
action 'SkylarkAction ex...
action 'SkylarkAction external/bazel_tools/tools/jdk/platform...
action 'Building third_party/ijar/t...
runAna...
ParallelEvaluator.eval
Parallel Evaluator evaluation
Compiling third_party/ijar/classfile.cc [for host]
SkylarkAction external/bazel_tools/tools/jdk/platformc...
Building third_party/ijar/test/GenKotli...
Building third_party/ijar/test/gen_zip_wit...
Compiling third_party/ijar/classfile.cc
Building third_party/ijar/test/ge...

4. 3 Command-line Flags (options)

Command-line Flags – *Introduction*

```
bazel [build|run|test|query] [flags] -- [target patterns]
```

- Reference Documentation: https://bit.ly/bazel-cli-ref

Useful Flags

`--keep_going`	Sometimes it is useful to try to build as much as possible even in the face of errors. This option enables that behavior, and when it is specified, the build will attempt to build every target whose prerequisites were successfully built, but will ignore errors.
`--verbose_failures`	This option causes Bazel's execution phase to print the full command line for commands that failed. This can be invaluable for debugging a failing build.
`--sandbox_debug`	Useful for debug sandboxed build invocation, sandbox state will not be erased after call
`--[no]use_action_cache`	On clean ci build, better set to false, it can save memory & disk space

Useful Flags, Ctd.

`--[no]remote_upload_local_results`	Better set to false, it will not upload possibly huge local files (like java platform classpath jars) to remote cache.
`--remote_download_minimal`	Do not download intermediate results from remote cache
`--output_base=dir`	Override the default output directory (which will be placed into /var/tmp). Useful to debug output artefacts produced by the build.
`--[no]build`	Causes the build to stop before executing the build actions, returning zero iff the package loading and analysis phases completed successfully; this mode is useful for testing those phases.

4. 4 Tooling

.bazelrc

For **project-specific options**, use the configuration file your

`<workspace>/.bazelrc` (see bazelrc format).

Bazel looks for optional configuration files in the following locations, in the order shown below:

- `/etc/bazel.bazelrc`
- `%workspace%/.bazelrc` - in workspace root
- `$HOME/.bazelrc`
- `--bazelrc=file cmd line flag`

flare.build

.bazelrc, ctd.

- You can load another RC file with **`import`** and **`try-import`**

```
try-import %workspace%/user.bazelrc
```

- You can add default options for commands with

```
build --default_option1
build --default_option2
...
```

- You can add option groups which are enabled by a shorthand switch **`--config=group_name`**

```
build:group_name --default_option3
build:group_name --default_option4
...
```

Useful Tools

- **Bazelisk**
  - Bazel Launcher, provides a way to use specific Bazel version (with **`.bazelversion`** file in a workspace)

- **Buildifier**
  - A formatting tool for bazel BUILD and .bzl files, can be executed as a **run** rule

Buildifier

Buildifier applies standard formatting to the named Starlark files.

- Can run directly on a command line or as a Bazel Target
- Can warn about any inconsistencies, or auto-fix some of them

CLI Usage:

```
buildifier [-d] [-v] [-r] [-diff_command=command] [-help]
  [-multi_diff] [-mode=mode] [-lint=lint_mode] [-path=path] [files...]
```

Example:

```
$ find . -name 'BUILD*' -exec buildifier {} \;
```

Buildifier

Buildifier also has a corresponding Bazel rule you can invoke:

Bazel BUILD file usage:

```
load(
     "@com_github_bazelbuild_buildtools//buildifier:def.bzl",
     "Buildifier"
)

buildifier(name = "buildifier-lint-fix", lint_mode = "fix")

buildifier(name = "buildifier-lint-warn", lint_mode = "warn")
```

Example:

```
$ bazel run :buildifier-lint-fix
```

flare.build

4.5 Nuking

Useful Nuking Commands

- `bazel clean`

  — delete all outputs in dist/bin

- `bazel clean --expunge`

  — same as above plus deleting external repos

- `bazel shutdown`

  — stop the persistent workers, useful to save resources

  (also might be needed to drop cached credentials to remote cache)

Lab

Querying & profiling & cache & configuration

Objectives

Query build targets and their dependencies, check profiling, compile and save build artifacts into a local disk cache, configure shorthand for common options

Schedule

~~Day 1~~ ✓

1. ~~Introduction to Bazel~~
2. ~~Using Bazel~~
3. ~~Building Java, Python, Protobufs~~
4. ~~CLI and Tooling~~

Schedule

Day 2

5. Starlark, Genrules, Macros
6. Writing Rules
7. Platforms and Toolchains
8. Remote Features, Packaging, Deployment