

# Riot Bazel Training

Day 1, Lectures 1 & 2



## Course Outline

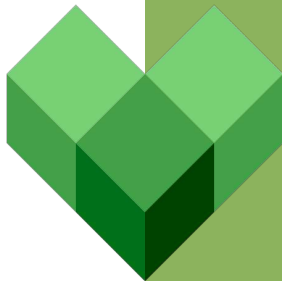
In the next two days you will learn how to use Bazel to its fullest.

- **The first day will be focused on Using Bazel** — we'll talk mostly about existing rules, features and languages. There are four lectures covering this track.
- **During the second day we will talk about extending Bazel**, in lectures 5-8. This will require a look under the hood, leading inevitably to Starlark programming. Let us know if you get stuck anywhere. We are here to help.

# Schedule

## Day 1

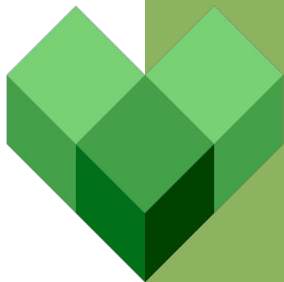
1. Introduction to Bazel
2. Using Bazel
3. Building Java, Python, Protobufs
4. CLI and Tooling



# Schedule

## Day 2

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

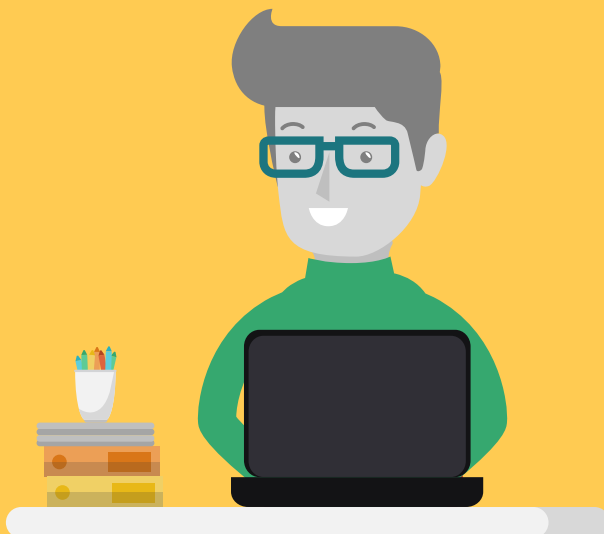


# Day 1

# Getting to Know Bazel



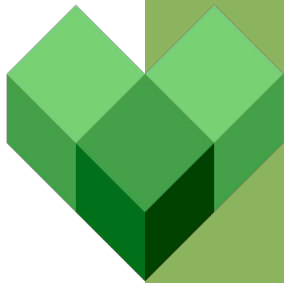
# 1. Introduction to Bazel



# What is Bazel?

Bazel is an open-source build and test tool similar to Make, Maven, and Gradle. Bazel supports projects in multiple languages and builds outputs for multiple platforms. It can handle codebases containing thousands of applications, used by thousands of developers daily.

- While Bazel supports many popular programming languages out the box, it can also **be extended with Starlark** — a Python-subset scripting language used in **BUILD** and **WORKSPACE** files and the extension libraries, i.e files ending with \*.bzl
- Today we'll learn how to use Bazel.
- Tomorrow we'll focus on extending Bazel.



# Was Bazel the answer, what would be the question?

The question would likely be: “How do I get my software to build in a highly consistent, always correct, and extremely fast way? Oh, and I should probably also mention that I have ten thousand developers working on a Petabyte-sized mono repo and committing an average of five commits per second.” Right. Glad we cleared that up.

Pause for a moment to appreciate this massive achievement of software engineering. Because — and this is exactly what Bazel does — it is a bit like magic.



Luckily, using Bazel is much much easier than writing Bazel, and over the next two days we hope you will feel comfortable with Bazel to the extent that you are not only able to confidently use it within your projects, but it becomes your preferred method of building software of any kind.





# Bazel

{Fast, Correct} - Choose two

# 1.1 Bazel in a Nutshell



---

# Bazel Quick Intro



Bazel works best when you are using a single large repository of source code that may contain several applications, potentially written in different languages.

**You tell Bazel where the root of your workspace is by placing a text file with the name `WORKSPACE` there.**

The file can be empty, and only one **WORKSPACE** file is allowed per repository.

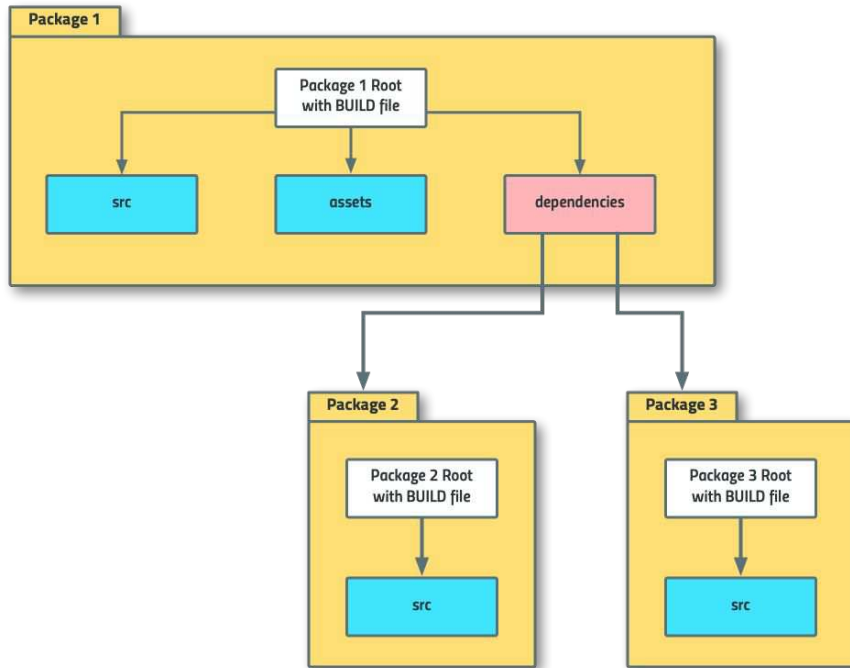
Inside application folders you may or may not place a **BUILD** file – a similar text file that defines rules relative to the directory it's in.

NOTE: A directory with a BUILD file is called a **Package** in Bazel terminology, just like the directory with a WORKSPACE file is called a **Repository**.



## Packages & BUILD files

- To enable Basil in any source code repository, you must create one **WORKSPACE** file at the root, and as many **BUILD** files in each source folder/sub-folders as necessary.
- Inside the **BUILD** files you define **build targets** in your repo by invoking various **rules** (which are basically build functions).
- Build targets are the **smallest unit you can declare** a dependency on from someplace else.
- There are rules that know how to build executable binaries and libraries using Java, C++, Go. Other rules know how to run tests, build docker images, etc.



# Targets, and Target's Targets



While defining your targets **you must also declare all of the dependencies** the target needs.

A really important design constraint of Bazel is this:

- External dependencies (interpreters, SDKs, libraries, etc) i.e — anything that's not inside the source tree, **can only be declared and loaded via the WORKSPACE file**
- **WORKSPACE** file may invoke Bazel functions that actually perform the download of external resources.
- **BUILD** files can not declare or fetch any external resources.

---

# Bazel Quick Facts

Bazel is written in Java and requires a valid JDK installed.

- When you run Bazel commands in a workspace, Bazel will **start a server on the background, anchored to that specific workspace.**
- If you have another workspace, and you run Bazel commands there, **another server will be booted**, potentially with a completely different configuration as defined by the other workspace.
- It's important to understand that **multiple background Bazel servers rarely interact with one another**, and that they are attached to the workspace they were started in.



# 1.2 Core Concepts



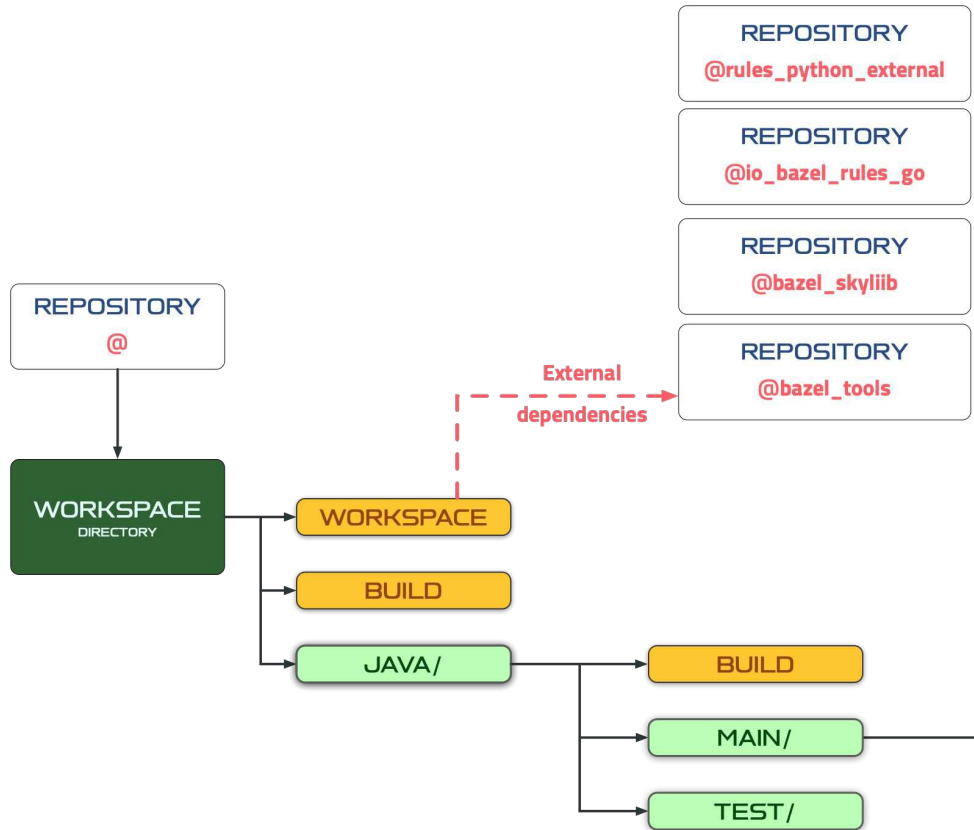
# REPOSITORIES & WORKSPACES

**Repository** is any directory on your filesystem that contains the source files for the software you want to build, as well as symbolic links to directories that contain the build outputs.

**Bazel Workspace** is a directory that **has a text file named WORKSPACE at its root, which may be empty**

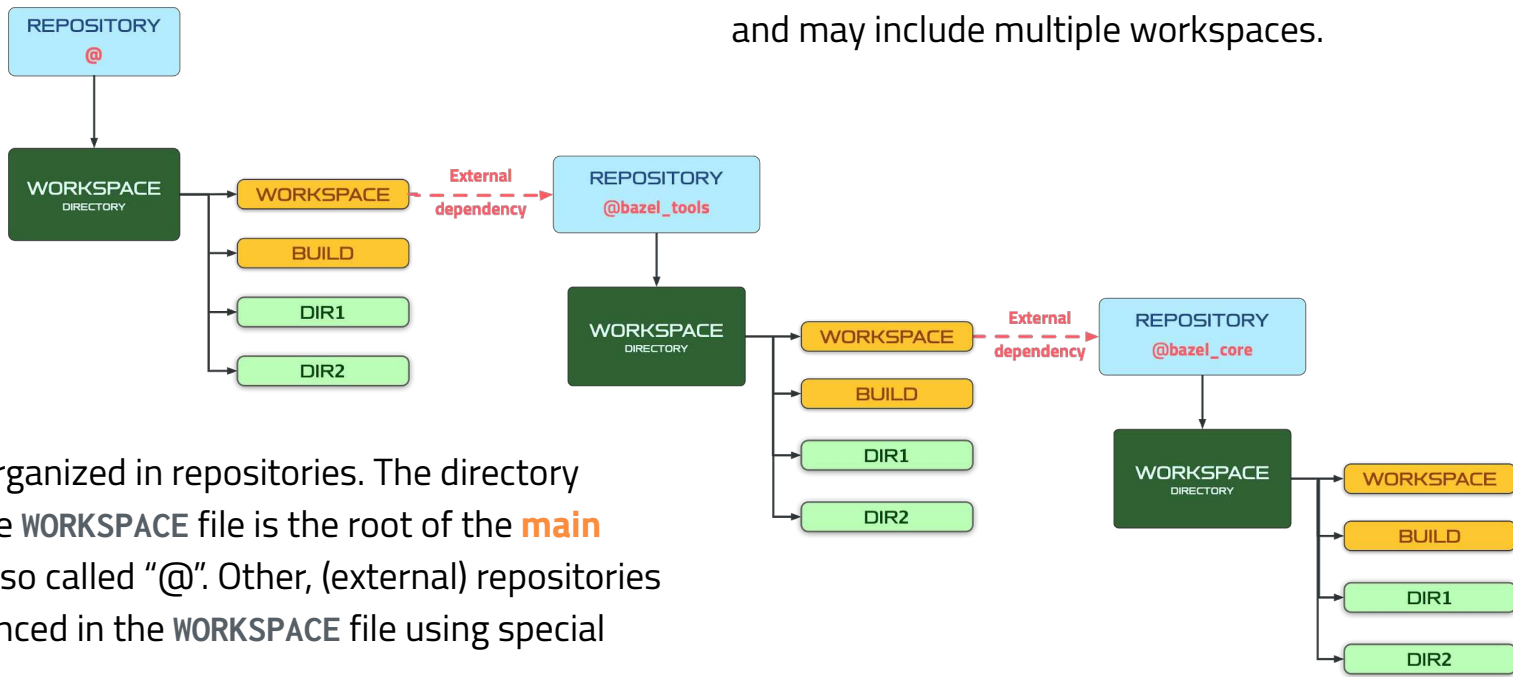
- **Nested workspaces** are discouraged, although may be required sometimes. When Bazel, running in one workspace, finds a subdirectory containing a **WORKSPACE** file, Bazel skips that directory entirely.

**Directories containing file WORKSPACE are considered the root of a repository, and are referenced as “@”**



## REPOSITORIES & WORKSPACES, ctd.

Workspaces and repositories are very similar, but there is a slight difference: workspaces must have a **WORKSPACE** file at its root, while repositories do not, and may include multiple workspaces.



The code is organized in repositories. The directory containing the **WORKSPACE** file is the root of the **main repository**, also called "@". Other, (external) repositories can be referenced in the **WORKSPACE** file using special syntax.

## WORKSPACE file – *External Dependencies*

**The workspace rules** are responsible for downloading, installing, and using **any and all external dependencies** your project might have.

NOTE: As external repositories are repositories themselves, they often contain a **WORKSPACE** file as well. However, these additional **WORKSPACE** files are ignored by Bazel. In particular, repositories dependent upon transitively are not added automatically to the current repository hierarchy, but their components are made available via the **load** statement in the **BUILD** files. It's conceptually similar to **TypeScript import** statement.



# Workspace example

```
workspace(name = "donut_project")
```

```
load("http.bzl", "http_archive")
```

```
http_archive(  
    name = "flour",  
    urls = ["https://github.com/flour.tar.gz"],  
    sha256 = "123456789",  
)
```



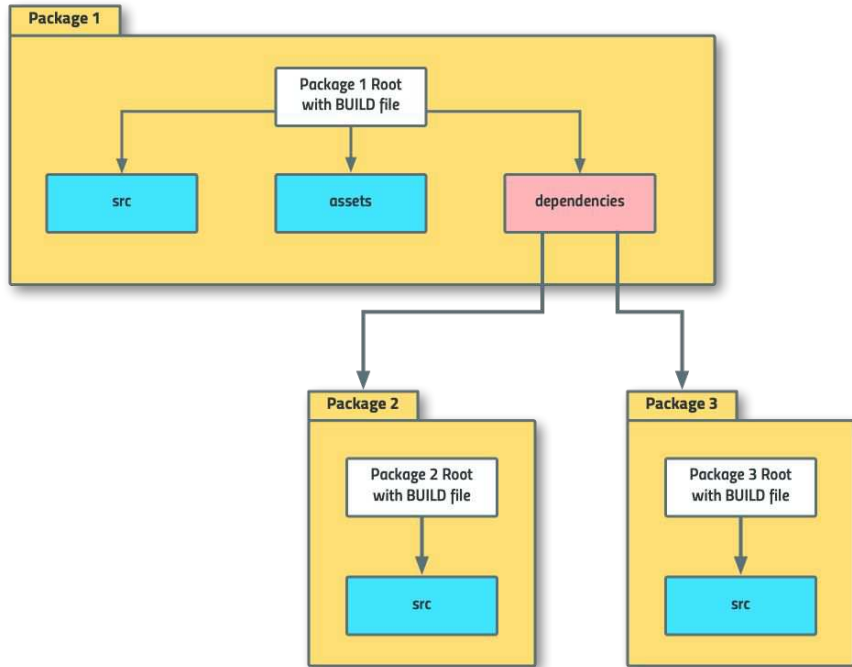
# PACKAGES

The primary unit of code organization within a repository is the **package**.

*A package is a collection of related files and a specification of the dependencies among them.*

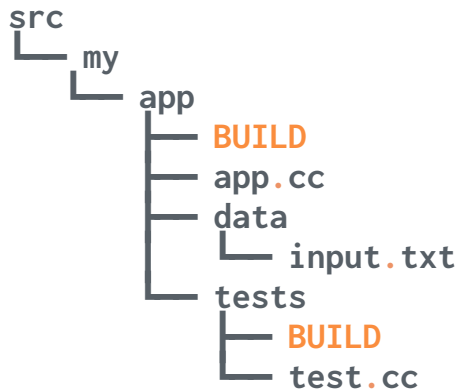
Practically, a package is defined as a directory containing a file named **BUILD** or **BUILD.bazel**, residing beneath the top-level directory in the workspace.

A package includes all files in its directory, plus all subdirectories beneath it, except those which themselves contain a **BUILD** file.



## PACKAGE EXAMPLE

For example, in the following directory tree (presumed to be nested inside a Workspace):



- There are two packages, `my/app`, and the subpackage `my/app/tests`.
- Note that `my/app/data` is not a package, but a directory belonging to package `my/app`.

# BUILD file Example

```
# flavors/BUILD

java_binary(  
    name = "chocolate",  
    srcs = ["Chocolate.java"],  
    main_class = "flavors.Chocolate",  
)
```



# Targets

- A BUILD file (and a corresponding **package**) consists of zero or more targets.
- Each call to a build rule returns no value but has the side effect of defining a new target; this is called **instantiating the rule**.
- There three kinds of targets:
  - **Files**
  - **Rules**
  - **Package Groups**



---

## Targets – *Files*

Files are further divided into two kinds:

- **Source files** are usually written by the efforts of people, and checked in to the repository.
- **Generated files**, sometimes called derived files, are not checked in, but are generated by the build tool from source files according to specific rules.



# Target *Visibility*

An important property of all **Rules** is that *the files generated by a rule always belong to the same package as the rule itself.*

In other words — *it is not possible to generate files into another package.*

It is not uncommon for a rule's inputs to come from another package, though.

**Package groups** are sets of packages whose purpose is to limit accessibility of certain rules. Package groups are defined by the `package_group` function. They do not generate or consume files.

They have two properties:

1. the list of packages they contain
2. and their name.



## Labels – A Target's Name

In Bazel labels refer to:

- They refer to a **name of a target**, and a particular syntax used to reference it.
- Label is also a **proper Class** in Bazel and **can be instantiated** and passed around as an argument.



A good way to think of a label is that of a **postal address**: Label provides a public address for all targets.

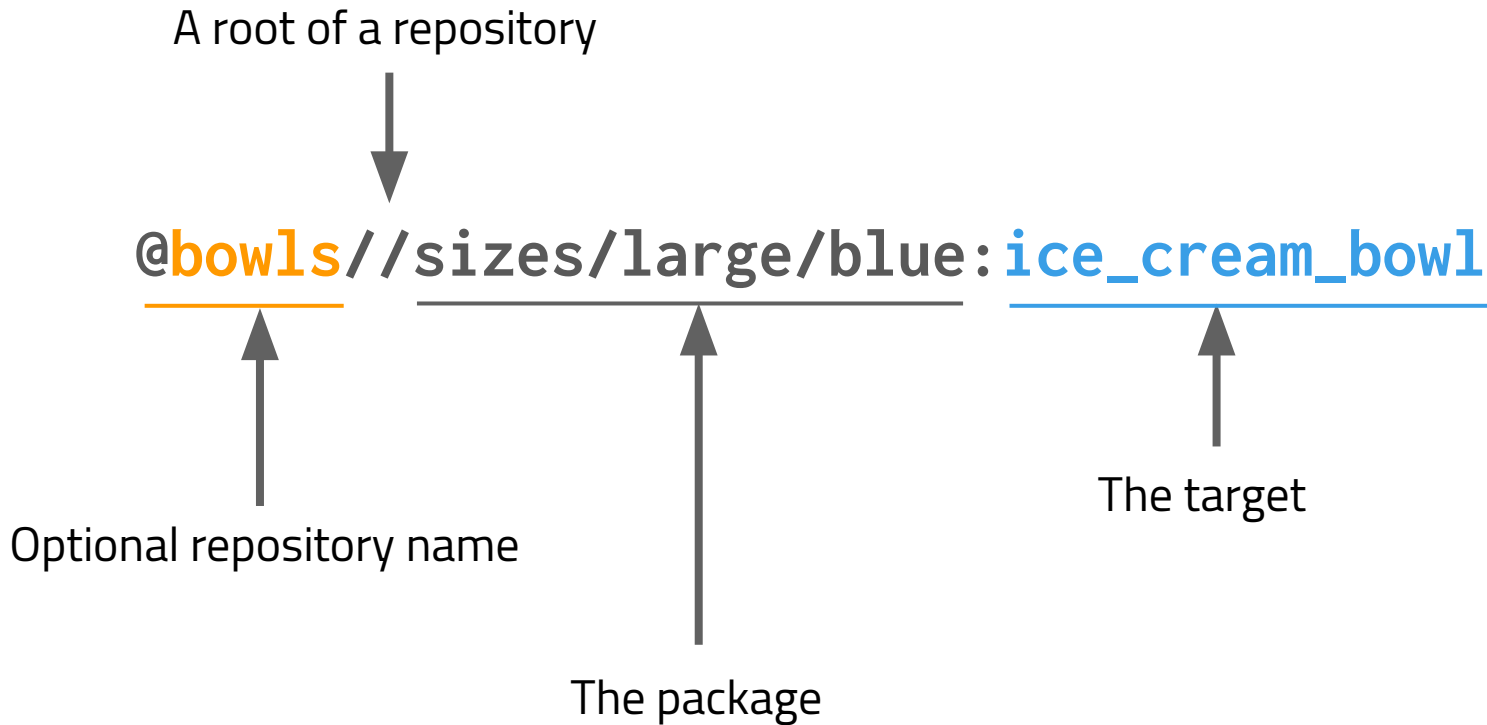


Not all labels are accessible from another package because of the visibility constraints.

**Labels uniquely identify** each target within the current package, current workspace, and even across multiple repositories. Structurally, a package contains zero or more targets.



# Anatomy of a Label



# Labels — *Canonical Form of a Label*

Target's name is a label, and a typical label in canonical form looks like this:

```
@myrepo//my/app/main:app_binary
```

In the typical case that a label refers to the same repository it occurs in, the repository name may be **left out**.

So, inside @myrepo this label is usually written as

```
//my/app/main:app_binary
```

Each label has two parts, a package name (`my/app/main`) and a target name (`app_binary`).

**Every label uniquely identifies a target.**

Labels sometimes appear in other forms; when the colon is omitted, the target name is assumed to be the same as the last component of the package name, so these two labels are equivalent:

```
//my/app:app
```

```
//my/app
```

```
//:app
```



## Labels ctd. — *Labels within a BUILD file*

**Within a BUILD file**, the package-name part of the label may be omitted, and optionally the colon too. So within the BUILD file for package my/app (i.e. `//my/app:BUILD`), the following "relative" labels are all equivalent:

```
//my/app:app  
//my/app  
:app  
app
```

NOTE: It is a matter of convention that the colon is omitted for files, but retained for rules, but it is not otherwise significant.

Similarly, **within a BUILD file**, files belonging to the package may be referenced by their unadorned name relative to the package directory:

```
generate.cc  
testdata/input.txt
```



# Labels, ctd.— *Main Repo & Querying*

- Labels starting with **@//** are references to the main repository, which will still work even from external repositories.
- Therefore **@//a/b/c** is different from **//a/b/c** when referenced from an external repository.
- The former refers back to the main repository, while the latter looks for **//a/b/c** in the external repository itself.
- This is especially relevant when writing rules in the main repository that refer to targets in the main repository, and will be used from external repositories.



With **query** command you can easily introspect and successfully reason about functionality of any part of a mono repo, written in any language.

Since labels are language independent, they offer a unique interface into application subsystems but from the build perspective, documenting code organization, instead of the code API.



# Rules & Actions

- Bazel **Rules** are made up of many **Actions**.
- Actions take a **set (which can be empty) of input files and generate a (non-empty) set of output files**. The set of input and output files must be known during the analysis phase.
- The output of an action **must only depend on the explicitly stated inputs**. All actions should be hermetic, isolated from all but explicit dependencies
- Rules are defined in **starlark**



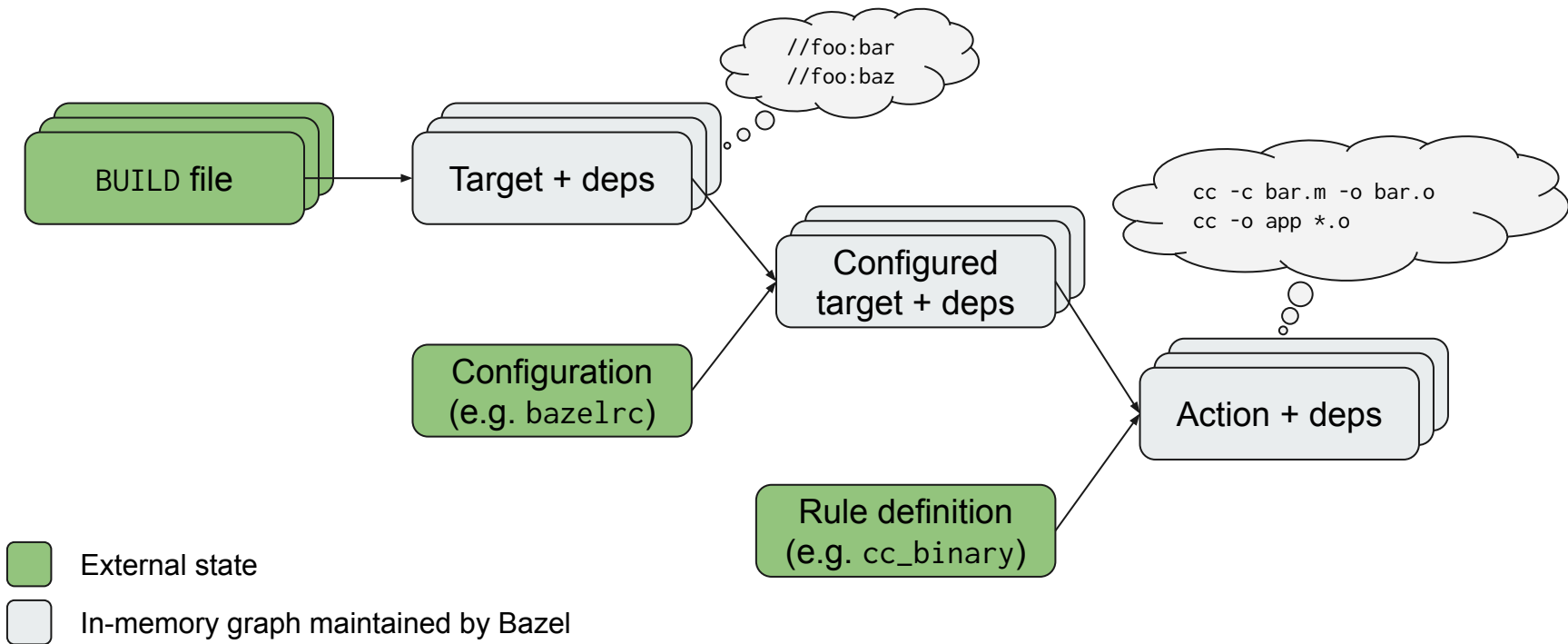
# Rules & Actions

## Bazel Build Phases:

1. **Loading Phase:** Parse and instantiate all the rules, create the Target Graph
2. **Analysis Phase:** Calculate the Action Graph, and compute hashes of inputs to look up in the cache and see what needs to be rebuilt.
3. **Execution Phase:** Run the minimum number of actions to recompute the final result.



# Build Phases



# Rules & Actions: pkg\_tar

The action to create a tarball in **pkg\_tar** looks like this:

```
ctx.actions.run(  
    executable = ctx.executable.build_tar,  
    inputs = file_inputs + ctx.files.deps + [arg_file],  
    arguments = ["--flagfile", arg_file.path],  
    outputs = [ctx.outputs.out],  
)
```

- **ctx** is the context of the rule.
- **ctx.actions** is how you register an action to a rule
- **ctx.actions.run** is an action that calls a script
- **executable** is a reference to the build\_tar script to be run.
- **inputs** are all files needed to run this script.
- **arguments** are sent to the executable.
- **outputs** are all files this action generates.



# Hermeticism

For something to be hermetic means “**impervious to external influence**”.

It follows that for a build system to be hermetic, it should be “**sealed airtight**”, i.e. unaffected by external (to the build) influences.

---



# Rules & Actions: Hermeticism

Explicit inputs & outputs = speed & correctness. How?

1. **Caching** — local or remote, for each artifact and result of an action
2. **Parallel Remote Execution** — ability to spread the build actions across a build farm
3. **Minimum work to rebuild** — Bazel computes the list of nodes that must be rebuilt using the declared inputs, and the directed action graph it compiles during the analysis phase
4. **Deterministic builds** — build steps produce the same result (with an identical SHA) regardless of when they run, or which machine they run on
5. Conversely, build steps produce a ***new result*** every time their ***inputs change***
6. Build steps that are hermetic can be **cached and reused**, with their content's SHA acting as a Cache Key



# Rules & Actions: Sandboxing

Bazel uses **Sandboxing** to *enforce hermeticity*:

- Starlark is prohibited from arbitrary I/O
- Processes are run with sandbox-exec on macOS and limited privileges (no network, etc)
- Tools are run in isolation to ensure they're only operating on declared inputs, and undeclared outputs don't affect future actions



# Directed Acyclic Graphs (DAGs) & Bazel

- During the loading phase, Bazel computes a target dependency graph which is used in the analysis phase
- During analysis, Bazel computes the action graph
- A Merkle tree is created:
  - Files are the leaf nodes and are digested using their corresponding content;
  - Directories are the tree nodes and are digested using digests from their subdirectories and children files
  - SHA256 hashing is often used but this can be customized
  - Merkle tree nodes are immutable. Any change in a node would alter its identifier and thus affect all the ascendants in the DAG, essentially creating a different DAG



# Caching

- All inputs and outputs are hashed and accessible by the digest of the content itself.
- This is often referred to as CAS: **Content Addressable Storage**.
- Thanks to this mechanism, it's possible to implement shared, distributed build caches that greatly reduce build times for developers and CI machines





# You're in good company!

- **BMW** saw 10x speed up in their unit test times and 12x speed up in build times with bazel remote cache and execution.
- **Braintree**: they've been suffering from slow and sluggish Gradle builds and they report various improvements after switching to bazel: half test time, 10x clean builds, etc.
- **Databricks** reports increased build stability, 10x decrease in time. They also successfully use bazel for k8s deployments
- **Dataform**: 6x faster CI builds.
- **Google/OSS**:
  - Angular test times reduced from 1 hour to 15 minutes
  - Android Studio: better scalability allows them to run full tests at every commit
  - Tensorflow: average build times decreased by 80%; Dedicated CI hardware reduced by 80%;
- **LinkedIn** migrated 2 iOS applications and saw ~50% incremental build time decrease and 6x clean build time decrease. They are only using caching, so when they turn on remote execution, they are likely to see even better results.
- **Pinterest** iOS build time dropped from 4 minutes to under a minute, sometimes as low as 30 seconds.
- **Redfin** saw 10x build speed up when they migrated from maven to bazel.
- **Stripe** is happy about build reproducibility, they have also seen 3x reduction in their build times.
- **SpaceX**: reproducibility of the build is what matters most for them.
- **Wix**: 90% reduction in build time.

# First Lab - Repo Setup

- Download and extract [labs repo](#)
  - or clone with `git clone https://github.com/flarebuild/training-labs-templates.git`
- Run `scripts/setup` (if bazelisk isn't installed and in your path)
- If you prefer to use **VSCode** —
  - download it, install,
  - and make sure you have the `code` shortcut configured/installed
    - Press `⇧⌘P` (Command-Shift-P) to bring up the "Show All Commands" drop down, and search for "code"
    - Select `Shell Command: Install 'code' in your PATH` and press ENTER.
  - Then run:  
`cd .vscode && make install`





## 2. Using Bazel

# Invoking Bazel

Bazel is implemented as a client-server application



- The Bazel system is implemented as a long-lived server process.
  - This allows it to perform many optimizations not possible with a batch-oriented implementation, such as caching of BUILD files, dependency graphs, and other metadata from one build to the next.
  - It massively improves the speed of incremental builds, and allows different commands, such as build and query to share the same cache of loaded packages, making queries very fast.

---

# Bazel Client & Server

## Client

- When you run bazel, you're running the client
- The client finds the server based on the output base, which by default is determined by the path of the base workspace directory and your user id
- If the client cannot find a running server instance, it starts a new one

## Server

- If you build in multiple workspaces, you'll have multiple output bases and thus multiple Bazel server processes
- Multiple users on the same workstation can build concurrently in the same workspace because their output bases will differ (different users)
- The server process will stop after a period of inactivity (3 hours, by default, which can be modified using the startup option `--max_idle_secs`).



# Persistent Workers

- In addition to the long-running Bazel server, many compilers are also implemented using Bazel's persistent worker mechanism to ensure no time is wasted starting up slow-booting compilers.
- This is important because in Bazel, every action is invoked in a separate process, meaning thousands of calls to javac for a large application.
- Adding a persistent worker for javac increased build times for java applications by approximately 4x.
- Controlling the spawn strategy of workers can be accomplished via the flag `--strategy`.



---

# Rules In Depth

## A refresher:

A rule specifies the **relationship between inputs and outputs**, and the steps to build the outputs.

Rules can be of one of many different kinds or classes.

Rules typically produce compiled executables and libraries, test executables and other supported outputs as described in the Build Encyclopedia.

---



---

## Rules In Depth: Names

- Every rule has a **name**, specified by the name **attribute**, of type **string**.
- The name must be a syntactically valid target name.
- In some cases, the name is somewhat arbitrary, and more interesting are the names of the files generated by the rule; this is true of genrules. In other cases, the name is significant: for binary and test rules, for example, the rule name determines the name of the executable produced by the build.



---

## Rules In Depth: Attributes

- Every rule has a set of **attributes**; the applicable attributes for a given rule, and the significance and semantics of each attribute are a function of the rule's class.
- Each attribute has a **name** and a **type**.
- Some of the common types an attribute can have are integer, label, list of labels, string, list of strings, output label, list of output labels.
- Most rules define 3 types of dependencies: **src**, **deps**, and **data**, as well as additional common attributes shared by all rules on top of the rule's unique attributes.



# Rule Types

- **Workspace Rules** (also known as **Repository Rules**)
- Rules used in BUILD files can be of the following three broad types:
  - **Build Rules**  
— rules that create build artifacts
  - **Run Rules**  
— rules that execute binaries and scripts
  - **Test Rules**  
— rules that run tests
- **Native rules** are rules built into Bazel itself, such as **cc\_binary**, or **java\_library** (but see a comment in the following slides).
- **Non-native rules** are the ones that require an external download, and activation (via the WORKSPACE file)



---

# Workspace Rules

- Workspace rules are the only place in the Bazel workspace where you can **define, download, and install external dependencies**.
- Using `git pull` of any commit or a branch, or via **https**, or even `ftp/sftp` — it's possible to fetch an external dependency and verify its **sha256sum**.



# Build Rules

Build rules are rules that are instantiated in BUILD files and describe targets corresponding to source files in the workspace—typically compiler invocations, etc.

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

java_binary(
    main_class = "com.flarebuild.hello.Hello",
    name = "hello",
    srcs = ["Hello.java"],
)
```

This **java\_binary** rule defines a target with a name “hello” which takes source code as an input and produces a jar when invoked with `bazel build`. It’s also an executable rule which can be run with `bazel run`.



# Native Rules

**Native rules** are shipped with the Bazel binary and are always available in BUILD files without a load statement.



In the case of language-specific rules, some of which were previously native, the current trend is to move them to individual packages as can be seen with java and C++ rules which are now loaded from packages.



# Native Rule: Filegroup

```
filegroup(  
    name = "foo_bar",  
    srcs = [  
        "foo.txt",  
        "bar.txt",  
    ],  
)
```

Use **filegroup** to give a convenient name *to a collection of targets*. These can then be referenced from other rules.

Using filegroup is encouraged instead of referencing directories directly. The latter is unsound since the build system does not have full knowledge of all files below the directory, so it may not rebuild when these files change.

When combined with glob, filegroup can ensure that all files are explicitly known to the build system.



---

## Native Rule: Alias

As the name suggests, you can use this rule to create a new name for an existing rule. This can be a rule in the same package, or in another package.

```
alias(  
    name = "foobar_alias",  
    actual = ":foo_bar",  
)
```



# Native Rule: Genrule

A genrule rule generates one or more files using a user-defined Bash command.

**General rules** are typically generic build rules that you can use if there's no specific rule for the task.

```
genrule(  
    name = "create_baz",  
    outs = ["baz.txt"],  
    srcs = [],  
    cmd_bash = "echo baz > $@"  
)
```

This genrule invokes the bash command specified, creating the defined output. If the bash command invoked didn't create a file, this would fail to build. Note that Bazel expands `$@` into the single output (baz.txt, in the expected path). More on this later.



# ConfigSetting & Select

**Configurable attributes**, commonly known as `select()`, are a Bazel feature that lets users toggle the values of BUILD rule attributes at the command line.

```
config_setting(  
    name = "bar_config",  
    values = {  
        "define": "word=bar",  
    },  
)  
  
config_setting(  
    name = "baz_config",  
    values = {  
        "define": "word=baz",  
    },  
)  
  
filegroup(  
    name = "foo_bar_or_baz",  
    srcs = [  
        "foo.txt",  
    ] + select({  
        ":bar_config": ["bar.txt"],  
        ":baz_config": ["baz.txt"],  
        "//conditions:default": ["bar.txt"],  
    })),  
)
```

```
$ bazel build //:foo_bar_or_baz --define word=bar
```



# ConfigSetting & Select, ctd.

**Configurable attributes**, commonly known as `select()`, are a Bazel feature that lets users toggle the values of BUILD rule attributes at the command line.

```
config_setting(  
    name = "bar_config",  
    values = {  
        "define": "word=bar",  
    },  
)
```

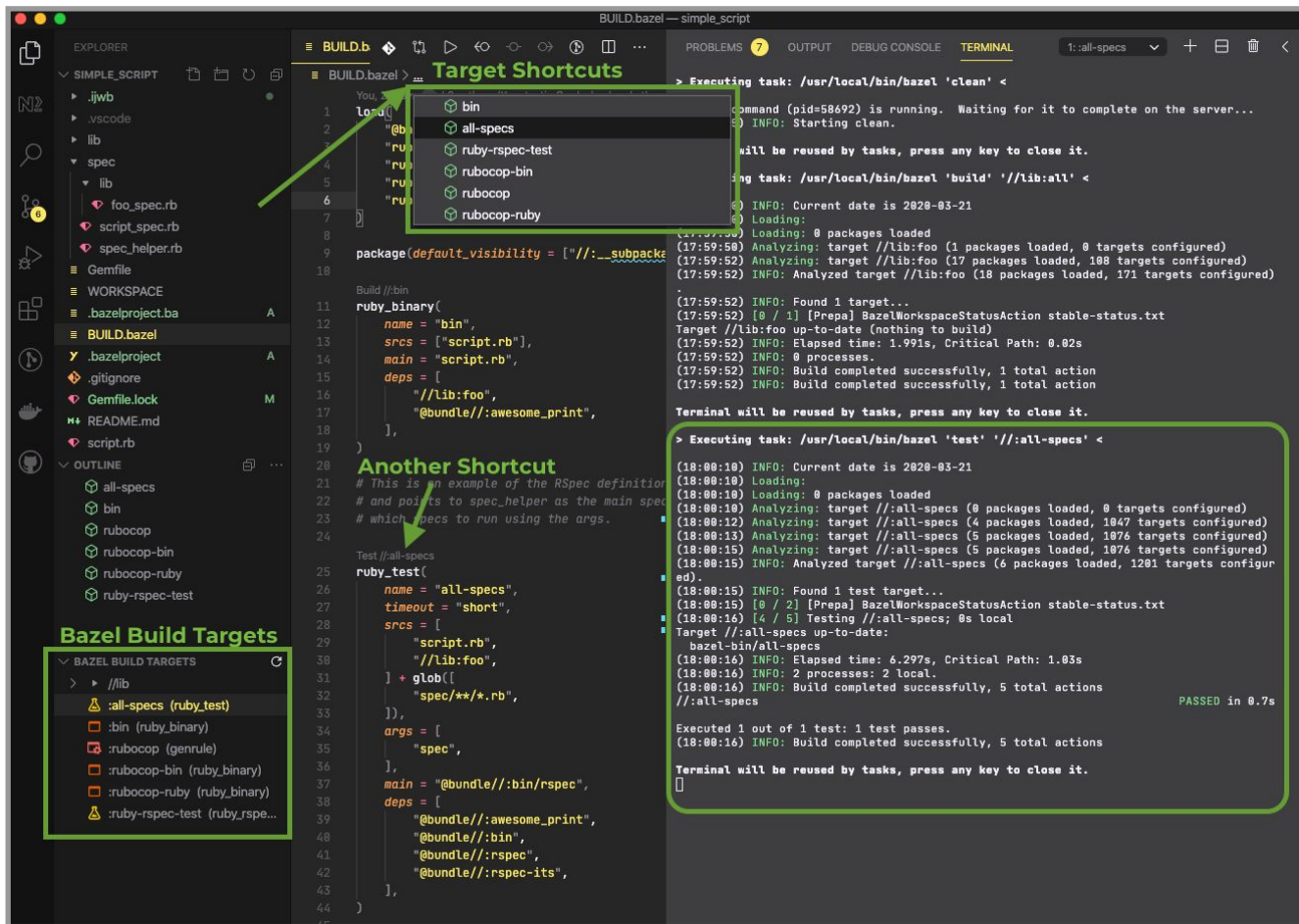
```
config_setting(  
    name = "baz_config",  
    values = {  
        "define": "word=baz",  
    },  
)
```

More examples:

- `bazel build //:foo_bar_or_baz --define word=bar`
  - **bar.txt** is included in srcs of the filegroup
- `bazel build //:foo_bar_or_baz --define word=bar`
  - **baz.txt** is included
- `bazel build //:foo_bar_or_baz`
  - **bar.txt** is included, as it's the default of the `select()`



# IDE Integration: VSCode



The screenshot displays the VS Code interface with a `BUILD.bazel` file open. The file contains a `ruby_binary` target and a `ruby_test` target. Annotations include a green box around the `all-specs` target in the Explorer, a green box around the `all-specs` target in the BUILD file, and a green box around the `Test //all-specs` line. A green arrow points from the `all-specs` target in the Explorer to the `all-specs` target in the BUILD file. Another green arrow points from the `Test //all-specs` line to the `all-specs` target in the BUILD file. A green box highlights the `Target Shortcuts` menu, which lists targets like `bin`, `all-specs`, `ruby-rspec-test`, `rubocop-bin`, `rubocop`, and `rubocop-ruby`. The terminal shows the execution of `bazel clean` and `bazel build` commands, followed by a `bazel test` command. The test results show that the `all-specs` target passed.

**Target Shortcuts**

- bin
- all-specs
- ruby-rspec-test
- rubocop-bin
- rubocop
- rubocop-ruby

**Another Shortcut**

**Bazel Build Targets**

- all-specs
- bin
- rubocop
- rubocop-bin
- rubocop-ruby
- ruby-rspec-test

**BUILD.bazel**

```
1 load("@bazel_tools//tools/build_defs/repo:http", http)
2
3 You,
4
5 "ruby",
6 "script_spec.rb",
7 "script_helper.rb",
8
9 package(default_visibility = ["//:...", subpackages])
10
11 Build //bin
12 ruby_binary(
13     name = "bin",
14     srcs = ["script_spec.rb"],
15     main = "script_spec.rb",
16     deps = [
17         "//lib:foo",
18         "@bundle//:awesome_print",
19     ],
20 )
21
22 # This is an example of the RSpec definition
23 # and points to spec_helper as the main spec
24 # which specs to run using the args.
25
26 Test //all-specs
27 ruby_test(
28     name = "all-specs",
29     timeout = "short",
30     srcs = [
31         "script_spec.rb",
32         "//lib:foo",
33     ] + glob([
34         "spec/**/*.rb",
35     ]),
36     args = [
37         "spec",
38     ],
39     main = "@bundle//:bin/rspec",
40     deps = [
41         "@bundle//:awesome_print",
42         "@bundle//:bin",
43         "@bundle//:rspec",
44         "@bundle//:rspec-its",
45     ],
46 )
```

**Terminal**

```
> Executing task: /usr/local/bin/bazel 'clean' <
command (pid=58692) is running. Waiting for it to complete on the server...
INFO: Starting clean.
will be reused by tasks, press any key to close it.
> Executing task: /usr/local/bin/bazel 'build' '//lib:all' <
INFO: Current date is 2020-03-21
Loading:
Loading: 0 packages loaded
(17:59:50) Analyzing: target //lib:foo (1 packages loaded, 0 targets configured)
(17:59:52) Analyzing: target //lib:foo (17 packages loaded, 108 targets configured)
(17:59:52) INFO: Analyzed target //lib:foo (18 packages loaded, 171 targets configured)
(17:59:52) INFO: Found 1 target...
(17:59:52) [0 / 1] [Prepa] BazelWorkspaceStatusAction stable-status.txt
Target //lib:foo up-to-date (nothing to build)
(17:59:52) INFO: Elapsed time: 1.991s, Critical Path: 0.02s
(17:59:52) INFO: 0 processes.
(17:59:52) INFO: Build completed successfully, 1 total action
(17:59:52) INFO: Build completed successfully, 1 total action
Terminal will be reused by tasks, press any key to close it.

> Executing task: /usr/local/bin/bazel 'test' '//all-specs' <
(18:00:10) INFO: Current date is 2020-03-21
(18:00:10) Loading:
(18:00:10) Loading: 0 packages loaded
(18:00:10) Analyzing: target //all-specs (0 packages loaded, 0 targets configured)
(18:00:12) Analyzing: target //all-specs (4 packages loaded, 1047 targets configured)
(18:00:13) Analyzing: target //all-specs (5 packages loaded, 1076 targets configured)
(18:00:15) Analyzing: target //all-specs (5 packages loaded, 1076 targets configured)
(18:00:15) INFO: Analyzed target //all-specs (6 packages loaded, 1201 targets configured)
(18:00:15) INFO: Found 1 test target...
(18:00:15) [0 / 2] [Prepa] BazelWorkspaceStatusAction stable-status.txt
(18:00:16) [4 / 5] Testing //all-specs; 0s local
Target //all-specs up-to-date:
bazel-bin/all-specs
(18:00:16) INFO: Elapsed time: 6.297s, Critical Path: 1.03s
(18:00:16) INFO: 2 processes: 2 local.
(18:00:16) INFO: Build completed successfully, 5 total actions
//all-specs
PASSED in 0.7s

Executed 1 out of 1 test: 1 test passes.
(18:00:16) INFO: Build completed successfully, 5 total actions
Terminal will be reused by tasks, press any key to close it.
```



# IDE Integration: IDEA

**Bazel Target Shortcuts**

```

14  main = "script.rb",
15  deps = [
16      "-//lib:foo",
17      "@bundle//:awesome_print",
18  ],
19  )
20
21  # This is an example of the RSpec definition that uses autogrun
22  # and points to spec_helper as the main spec file. It specifies
23  # which specs to run using the args.
24
25  ruby_test(
26      name = "all-specs",
27      timeout = "short",
28      srcs = [
29          "script.rb",
30          "-//lib:foo",
31      ] + glob([
32          "spec/**/*.rb",
33      ]),
34      args = [
35          "spec",
36      ],
37      main = "@bundle//:bin/rspec",
38      deps = [
39          "@bundle//:awesome_print",
40          "@bundle//:bin",
41          "@bundle//:rspec",
42          "@bundle//:rspec-its",
43      ],
44  )
45
46  # Finally, this is the short version of the rule that
47  # via the ruby_rspec_test rule that
48  # shows but encapsulated in the rule.
49  # gems to the dependency list, executed
50  # as arguments to rspec.
51  ruby_rspec(
52      name = "ruby-rspec-test",
53      srcs = [
54          "script.rb",
55          "-//lib:foo",
56      ],
57  )

```

**Bazel Test Result**

Tests passed: 1 of 1 test - 0ms

**Bazel Output**

```

Testing started at 6:08 PM ...
/usr/local/bin/bazel test --tool_tag=ijwb:IDEA:ultimate --curses=no --color=yes --pro
(18:08:17) INFO: Current date is 2020-03-21
(18:08:17) Loading:
(18:08:17) Loading: 0 packages loaded
(18:08:17) INFO: Build option --runs_per_test has changed, discarding analysis cache.
(18:08:17) Analyzing: target //:all-specs (0 packages loaded, 0 targets configured)
(18:08:17) DEBUG: /private/var/tmp/_bazel_kig/ced6dd866e95ee073fddcc447dd75427/extern
(18:08:17) INFO: Analyzed target //:all-specs (1 packages loaded, 1372 targets config
(18:08:17) INFO: Found 1 test target...
(18:08:17) [0 / 2] [Prepal] BazelWorkspaceStatusAction stable-status.txt
Target //:all-specs up-to-date:
  bazel-bin/all-specs

```

**Bazel Console**

```

Querying targets in project directories...
Command: /usr/local/bin/bazel query --tool_tag=ijwb:IDEA:ultimate --output=label_kind --k

```

**Bazel Target Shortcuts**

All	Classes	Files	Symbols	Actions	Include non-project items
bazel test					
Type / to see commands					
Run Configuration					
+ Bazel test:all					
+ Bazel test:all-specs					
Actions					
Enable new Bazel test runner					
Preferences > Flutter					

Press ⌘↑ or ⌘↓ to navigate through the search history

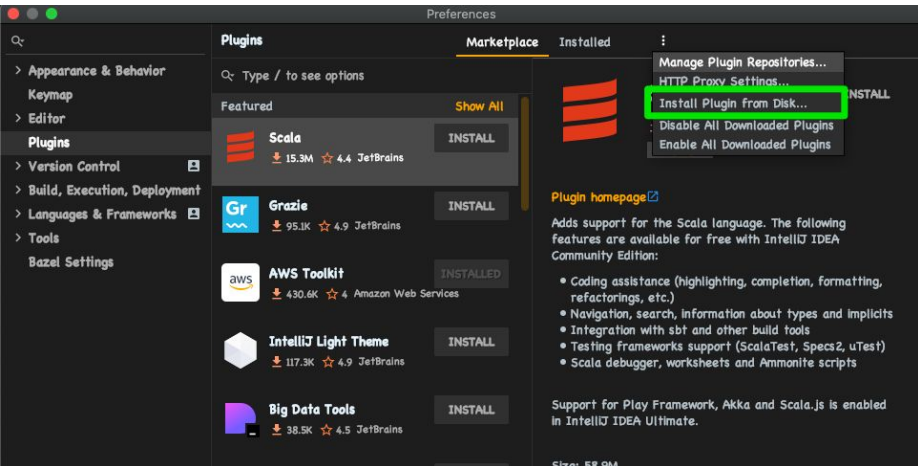


# IDE Integration: IDEA

- The plugin offers Bazel support in the IDE, but in order to enable it, a project must first be "imported" as a Bazel project; this creates a **.bazelproject** "project view" file in .ijwb project folder. This is commonly in gitignore, and a shared project view file is typically committed in **project/.bazelproject**.
- If you're starting with a shared **.bazelproject**, you'll want to be sure to import this file; the contents will then be cloned to your local **.bazelproject** copy.
- There's a wide range of things that can be done in this **.bazelproject** file to enhance the experience of using Bazel in the IDE outside of what is possible with Bazel configuration and tags themselves, and there are additional IDEA-only tags that can be applied to targets to change plugin behavior per target using Bazel's built in tag mechanism.



# Installing Bazel IntelliJ IDEA Plugin



The simplest way to install the plugin is to find it in the **Plugin Marketplace**, and install from there.

However, sometimes the plugin is behind the most recent version of the IntelliJ IDE, and in these cases you have two choices:

- Rollback IDE to a previous version
- Or, build the plugin from sources <https://bit.ly/bazel-idea-build>



---

# IDEA Plugin Features

The plugin allows you to:

- Compile your project and get navigable compile errors in the IDE.
- Run lint from within the IDE with navigable issues.
- Support for IDEA run configurations for any executable Bazel rule.
- Run tests from within the IDE by right-clicking on methods/classes, with deep JUnit integration
- **BUILD** file and .bzl (starlark) language support.
- CTRL/CMD click to navigate to targets
- Live templates & autocompletion/intellisense in BUILD & bzl files for all rules, custom or native
- Synchronization of source with Bazel, allowing proper imports, autocompletion, and syntax highlighting in 8+ supported languages



---

# Lab: labs/lab2.md

## Lab 2.1 - Bazel Hello World

1. Create a BUILD file for hello world java application
2. Create a BUILD file for hello world python application

## Lab 2.2 - PySpark

- Create BUILD and bzl files for PySpark/dataproc



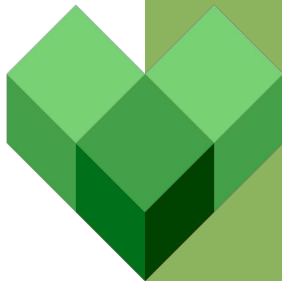
# Riot Bazel Training

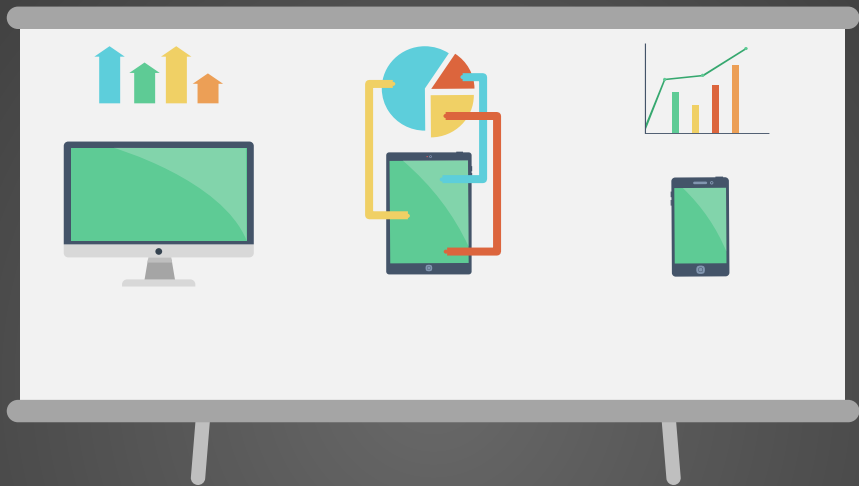
Day 1, Lectures 3 & 4

# Schedule

## Day 1

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





## 4. Building Java, Python, Protobufs with Bazel

# 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

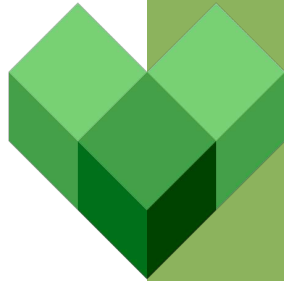
### Objective:

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

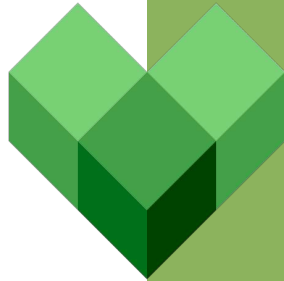
## rules\_python\_external

### 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.



## 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/97d8af4dc474595af3900dd85cb3a29ad28cc313.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 is 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

<code>["//visibility:public"]</code>	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.
<code>["//visibility:private"]</code>	Only targets in this package can use this
<code>["//some/demo_package:__pkg__", "//other/package:__pkg__"]</code>	Only targets in some/package and other/package have access to this
<code>["//my_project:__subpackages__", "//other:__subpackages__"]</code>	Only targets in packages project or other or in one of their sub-packages have access to this
<code>["//some/demo_package:my_package_group"]</code>	A package group is a named set of package names



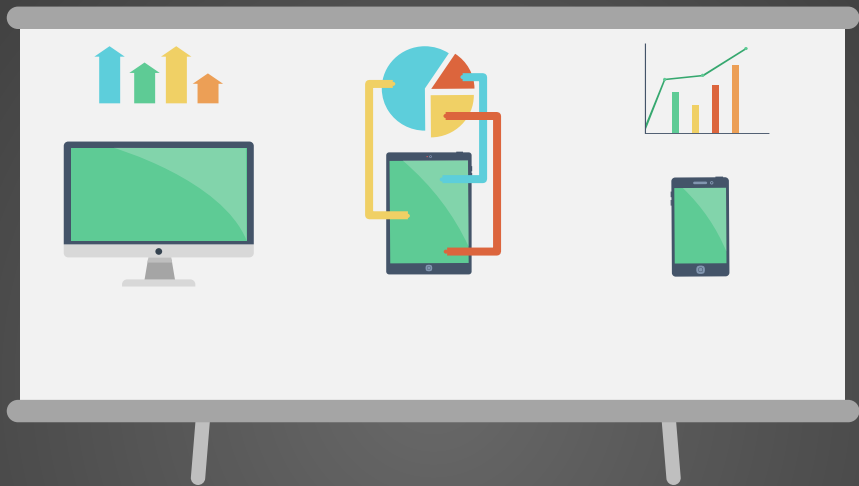
# Lab

## Using Protobuf for Java and Python with Bazel

Objective:

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





## 4. CLI & Tooling

## 4.1. Build Querying

## Asking Bazel Questions – *Query Types*

<code>query</code>	queries target graph, the output of loading phase
<code>sky query</code>	an alternative implementation of query
<code>cquery</code>	queries configured target graph (correctly handles select() )
<code>aqquery</code>	queries action graph
<code>genquery</code>	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

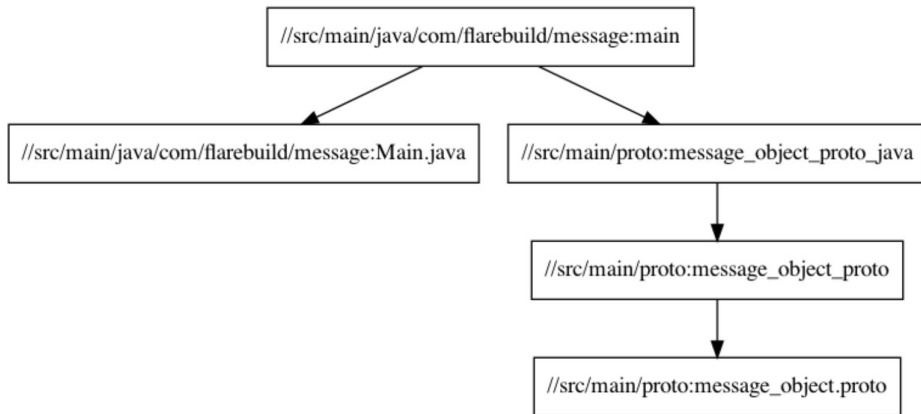
```
bazel query "deps(//packages/core)"
```

- 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
```



## 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
```

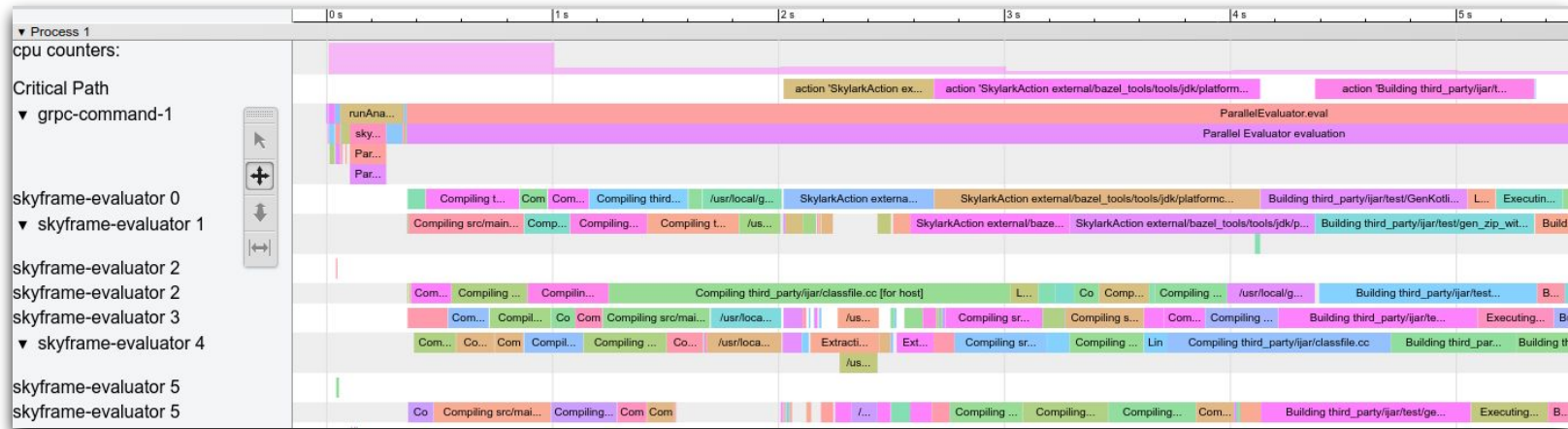


# Execution Log

- 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
```





## 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.

<code>--[no]remote_upload_local_results</code>	Better set to false, it will not upload possibly huge local files (like java platform classpath jars) to remote cache.
<code>--remote_download_minimal</code>	Do not download intermediate results from remote cache
<code>--output_base=dir</code>	Override the default output directory (which will be placed into /var/tmp). Useful to debug output artefacts produced by the build.
<code>--[no]build</code>	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



## .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
```



## 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

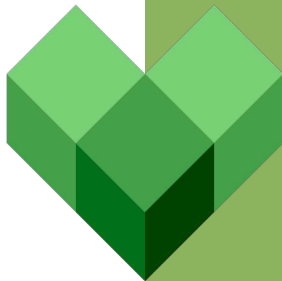


# Schedule

Day 1



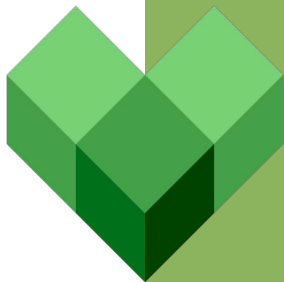
1. ~~Introduction to Bazel~~
2. ~~Using Bazel~~
3. ~~Building Java, Python, Protobufs~~
4. ~~CLI and Tooling~~



# Schedule

## Day 2

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



The background features a large, stylized graphic of two interlocking cubes. The cubes are rendered in a 3D perspective, with the top faces being a light green and the bottom faces being a darker green. They are set against a solid, medium-green background.

# Riot Bazel Training

Day 2, Lectures 5 & 6

# Day 2

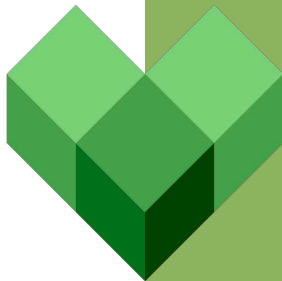
## Extending Bazel



# Schedule

~~Day 1~~

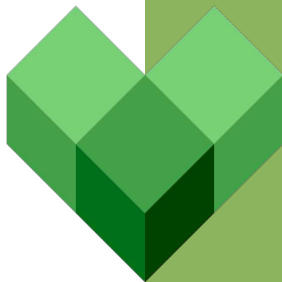
- ~~1. Introduction to Bazel~~
- ~~2. Using Bazel~~
- ~~3. Building Java, Python, Protobufs~~
- ~~4. CLI and Tooling~~



# Schedule

## Day 2

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





## 5. Genrules, Starlark, Macros

# Bazel's Power Lies in Extensibility

Does Bazel support my favorite programming language \_\_\_\_?

- If the answer is “yes”, — great!
- If the answer is “no”? Should we give up and go back to the dreaded `Makefile`? `CMakeFile.txt`? `build.xml`?
- Anyway, what does it take to support a new language in Bazel? What about a compiled language? What about a new interpreted language?



## 5.1 General Rules — genrules

# Genrules – Introduction

**Genrules** are generic build rules that you can use if there's no specific rule for the task.

Genrules are suitable if you need to use bash scripts or one-liners to produce some output from your inputs or sources.

Below genrule creates an output file, but has no inputs.

```
genrule(  
    name = "make_data",  
    outs = ["data.txt"],  
    cmd = "echo foo bar baz > $@",  
)
```



# Inputs/outputs Make Variables

- Note how CMD attribute uses make variables substitutions with `$@`
- `$@` is an alias for single-file output, instead of `$(OUTS)`  
`$<` is an alias for single-file input, instead of `$(SRCS)`
- There are others available: `$(@D)`, `$(RULEDIR)`, `$(location)`, `$(output_name)`, `$(execpath)`, etc.
- For CMD attribute you can use a shell one-liner, or provide a bash script, or any any other binary target.



## Example with a py\_binary Target:

Note the **\$location** helper – it will substitute path for the actual executable at runtime.

```
py_binary(  
    name = "unfiltered_data",  
    srcs = ["main.py"],  
    main = "main.py",  
)  
  
genrule(  
    name = "filtered_data",  
    srcs = [":input"],  
    outs = ["output.txt"],  
    cmd = "$(location :unfiltered_data) $< > $@",  
    tools = [":unfiltered_data"],  
)
```



# Escape Characters

Note that \$ characters need to be escaped with \$\$ in CMD.  
So, in order to invoke a shell command containing dollar-signs such as:

```
ls $(dirname $x)
awk '!a[$0]++'
```

You must escape it like this in Starlark:

```
ls $$$(dirname $$x)
awk '!a[$$0]++'
```



# Hermeticity Considerations

- Avoid non-hermetic rules or functions that do “improper” things like downloads external files, put timestamps in the outputs, generate random data, and so on.
- As a rule of thumb, a **genrule** should never access the network.
- If, for some reason, non-hermetic usage is unavoidable (for instance, versioning the release before publishing a package, or time-stamping the production deployment), at the very least please make sure you don’t have any downstream dependent targets (in other words, this target is **the final step in the build**).



---

## Other Considerations

- **Do not use absolute paths in scripts.**
- **Do not write to STDOUT** (echo something), except for debugging purposes.
- **Do not create symlinks and directories**, Bazel will not preserve them. Exception: /tmp writes for intermediate data.
- **Do not write to srcs!** This is a bad idea in general, and usually it is best to treat srcs as read-only (which they will be in a sandboxed environment, for example).



# Footgun

```
genrule(  
    name = "do_not_do_this",  
    srcs = ["src.txt"],  
    outs = ["out.txt"],  
    cmd = "echo 'garbage' > $<",  
)
```

What's happening here?

- Reminder: **\$<** is a make variable replacement for **SRCS**, if it is a single file (otherwise Bazel throws an error)
- So the command **"echo ... > \$<"** is attempting to write to a source file!



# Genrule Limitations

- Windows concerns — <https://docs.bazel.build/versions/master/windows.html>
- Genrule executes bash commands, so it is not cross-platform. In case your Workspace will be used on Windows without MSYS2, genrules will not work (same is applicable for sh\_binary or sh\_test rules).
- If you need to support Windows, there are several options:
  - First, you can provide several scripts for different platforms:  
`cmd_bash + cmd_bat / cmd_ps`.
    - Invocation priority: `cmd_ps` → `cmd_bat` → `cmd_bash` → `cmd`.
  - Second, you can rewrite genrules using `native_binary()` from `bazel-skylib`, wrapped in macros.



## 5.2 Starlark — Pythonesque Elucidation

## Decades Ago: A Kilobyte of History



"Blaze Fortnite"

Many years ago, code at Google was built using **Makefiles**.

As other people noticed, Makefiles don't scale well with a large code base.

A temporary solution was to generate Makefiles using Python scripts, where the description of the build was stored in BUILD files containing calls to the Python functions.

But this solution was way too slow, and the bottleneck was Make.

The project Blaze (later open-sourced as Bazel) was started in 2006.

It used a simple parser to read the BUILD files (supporting only function calls, list comprehensions and variable assignments).

When Blaze could not directly parse a BUILD file, it **used a preprocessing step that ran the Python interpreter** on the user BUILD file to generate a simplified BUILD file.

The output was used by Blaze.

---



## Decades Ago: *A Megabyte of History, ctd.*

Eventually, **slow Make and Python scripts**, as well as “just too much mutual baggage” of both platforms, prompted a development of new backwards-compatible language, custom-fitted as a build configuration language.



# Starlark Was Born – *Python3, ~~Crippled~~Improved*

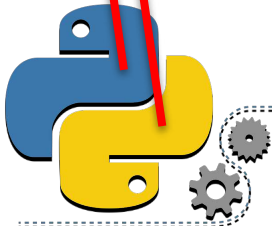


"Starlord", an inventor of "Starlark" and  
all around super awesome dude

- And just like that, **Starlark** was born.
- As a strict subset of Python, it is a **dynamically typed language** with high-level data types, first-class functions with lexical scope, and garbage collection.
- **Independent Starlark threads execute in parallel**, so Starlark workloads scale well on parallel machines.
- Starlark is a small and simple language with a familiar and highly readable syntax (unless you are a Rubyist, with a special affinity for Betamax).



## Years Ago: *Two Bits of History, ctd.*



- In the current iteration of Bazel, the **Python preprocessing step has been removed.**
- But the Python syntax stuck around, and **Stalark became the only extension language of Bazel.**
- Towards the end of 2010s, thanks in part to its success in the academia, **Python became the most popular interpreted language ever.** This development sealed the choice of Starlark as the configuration language for the builds.
- Several other build tools (e.g. **Buck, Pants, and Please**) have all adopted Starlark as the build configuration language.



---

## Starlark, ctd.

- **A Starlark interpreter is embedded within Bazel.** Starlark was designed from the ground up to be embeddable within larger applications.
- Bazel uses Starlark both for its BUILD and WORKSPACE files, and for its macro language, through which **Bazel is extended with custom logic to support new languages and compilers.**
- NOTE: There are multiple implementations of Starlark.



# Star-who?

Starlark's syntax is inspired by Python3.

```
def fizz_buzz(n):  
    """  
    Print Fizz Buzz numbers  
    from 1 to n.  
    """  
    for i in range(1, n + 1):  
        s = ""  
        if i % 3 == 0:  
            s += "Fizz"  
        if i % 5 == 0:  
            s += "Buzz"  
        print(s if s else i)  
  
fizz_buzz(20)
```



# Starlark & Parallelism

During the loading phase, Bazel first evaluates the leaves of the dependency graph (i.e. the files that have no dependencies) in **parallel**.

It will load the other files as soon as their dependencies have been loaded, which means **the evaluation of BUILD and .bzl files is interleaved**.

This also means that the order of the load statements doesn't matter at all.

**Each file is loaded at most once.** Once it has been evaluated, its definitions (the global variables and functions) are cached. Any other file can access the symbols through the cache.

**Once the definitions of a file are cached, they are made read-only**, i.e. you can iterate on an array, but not modify its elements. You may create a copy and modify that, though.



# Starlark, Not Python



- Global variables are **immutable**.
- **for statements are not allowed at the top-level.** Use them within functions instead. In BUILD files, you may use list comprehensions.
- **if statements are not allowed at the top-level.** However, if expressions can be used: `first = data[0] if len(data) > 0 else None`.
- Deterministic order for iterating through Dictionaries.
- **Recursion is not allowed.**
- Int type is limited to 32-bit signed integers. Overflows will throw an error.
- **Modifying a collection during iteration is an error.**
- Except for equality tests, comparison operators `<`, `<=`, `>=`, `>`, etc. are not defined across value types. In short: `5 < 'foo'` will throw an error and `5 == "5"` will return false.
- In tuples, a trailing comma is valid only when the tuple is between parentheses, e.g. write `(1,)` instead of `1,`.
- Dictionary literals cannot have duplicate keys. For example, this is an error: `{"a": 4, "b": 7, "a": 1}`.
- Strings are represented with double-quotes (e.g. when you call `repr`).
- Strings aren't iterable.



# Unsupported Python Features

- Implicit string concatenation (use explicit + operator)
- Chained comparisons (e.g.  $1 < x < 5$ )
- class (see struct function)
- import (see load statement)
- while, yield
- float and set types
- generators and generator expressions
- lambda and nested functions
- is (use == instead)
- try, raise, except, finally (see fail for fatal errors)



# Where Does Starlark Code Go?

- **BUILD** files register targets via making calls to rules.
- **\*.bzl** files provide definitions for constants, rules, macros, and functions
- Native functions and native rules are
  - global symbols in **BUILD** files
  - While **\*.bzl** files need to load them using the **native** module
- There are two syntactic restrictions in BUILD files:
  - declaring functions is illegal, and
  - **\*args** and **\*\*kwargs** arguments are not allowed.



## 5.3 Macros

# Macros

- Macros are best suitable for reducing repetitive code
- Macros in a nutshell is just a Starllark function which could call rules (and other macros) with desired arguments
- It could also be used to combine several rules invocations within one logical block
- Macro contents will be parsed during the loading phase and replaced with rules defined in it, so execution phase will only see rules and arguments



# Macro Example

Simple example for macro that does nothing except copying input to the output using genrule invocation

```
def copy_data(name, src, out = "", **kwargs):  
    native.genrule(  
        name = name,  
        srcs = [src],  
        outs = [name + ".txt"],  
        cmd = "cat $< > $@",  
        **kwargs  
    )
```



# Macro Example

- Note **genrule** is invoked as **native.genrule** in macro. Same will work for all native rules (i.e. rules that don't need to be loaded with `load(...)` statement).
- All macros must have a name argument.
- Macro could be reused across Workspace with different arguments.
- Macros could have an optional visibility argument.
- Create macros in `.bzl` files and use them in your BUILD files.

```
load(":defs.bzl", "my_macro")
```

```
my_macro(  
    arg1 = "one",  
    arg2 = "two"  
)
```



# Macro Example

To check how macro will work after evaluation, use query with `--output=build` on the target using this macro:

```
bazel query --output=build //src/package:target_using_macro
```

If you want to throw an error within macro, use `fail()` statement:

```
def my_macro(name):  
    fail("Don't call me maybe")
```



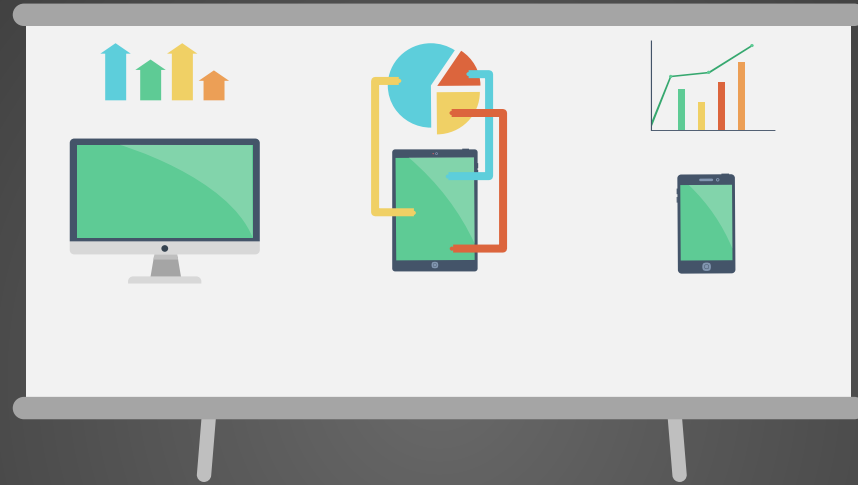
# Lab 5

## Genrules & Macros

### Objectives:

- Create genrules to transform CSV files and write macros to ease the process.
- Add tests to verify data transformed as expected.





## 6. Writing Rules

# Rules

**Workspace Rules** modify the workspace and as such are instantiated in the **WORKSPACE**.

A commonly encountered version of a workspace rule is a “repository rule” which makes repositories available to the BUILD files within the primary workspace.

Examples workspace rules: **http\_archive**, **bind**, **local\_repository**.

In a lab after this lecture we will write our own custom workspace rule.

**Build Rules** are instantiated in **BUILD files**, and fit the description of a rule above; given inputs, they run some actions and create outputs. `Java_library`, `python_binary`, `pkg_zip`, etc are examples of Build Rules.



# 6.1. Workspace Rules

# Hello world repository rule

```
def _hello_repo_impl(ctx):  
    ctx.file("hello.txt", ctx.attr.message)  
    ctx.file("BUILD.bazel", 'exports_files(["hello.txt"])')  
  
hello_repo = repository_rule(  
    implementation = _hello_repo_impl,  
    attrs = {  
        "message": attr.string(  
            mandatory = True,  
        ),  
    },  
)
```

1. Call a global function **repository\_rule()**
2. Each repository rule must have **implementation** function
3. Implementation function gets ctx parameter, which is a [repository ctx](#) object.
4. Repository ctx has specific API lets you download files, execute commands, and access the file system. **file()** generates a file in the repository directory with the provided content.



# Registering repository rule

## WORKSPACE

```
load("//:deps.bzl", "hello_repo")
```

```
hello_repo(  
    name = "hello",  
    message = "Hello, world!",  
)
```

Since it's a repository rule, it is instantiated from **WORKSPACE**.



# Important things to remember

- Repository rules are evaluated during the **loading phase**, rather than the analysis phase. This means repository rules cannot create actions or depend on files created by actions.
- The **repository\_ctx API** provides rules with direct access to the host system. Take extra care to stay hermetic: do not let information from the host system slip into the build (watch out for directory names, environment variables, or timestamps)



## 6.2. Build rules

# Hello world Build Rule

```
def _hello_world_impl(ctx):
    out = ctx.actions.declare_file(ctx.label.name + ".cc")
    ctx.actions.expand_template(
        output = out,
        template = ctx.file.template,
        substitutions = [{"NAME": ctx.attr.username}],
    )
    return [DefaultInfo(files = depset([out]))]

hello_world = rule(
    implementation = _hello_world_impl,
    attrs = {
        "username": attr.string(default = "unknown person"),
        "template": attr.label(
            allow_single_file = [".cc.tpl"],
            mandatory = True,
        ),
    },
)
```

1. Call a global function **rule()**
2. Each rule must provide an **implementation** function
3. Implementation function gets ctx parameter, which is a rule **ctx** object.
4. Rules context can be used to:
  - access attribute values and obtain handles on declared input and output files;
  - call **actions**
  - pass information to other targets that depend on this one, via **providers**.



# Actions

In our example, `hello_world` rule defines `declare_file()` and `expand_template()` actions. Some other important actions from `ctx.actions` API:

- `ctx.actions.run`, to run an executable.
- `ctx.actions.run_shell`, to run a shell command.
- `ctx.actions.write`, to write a string to a file.

```
def _hello_world_impl(ctx):  
    out = ctx.actions.declare_file(ctx.label.name + ".cc")  
    ctx.actions.expand_template(  
        output = out,  
        template = ctx.file.template,  
        substitutions = {"{NAME}": ctx.attr.username},  
    )  
    return [DefaultInfo(files = depset([out]))]
```

An action describes **how to generate a set of outputs from a set of inputs**, for example “run gcc on `hello.c` and get `hello.o`”. When an action is created, Bazel doesn’t run the command immediately. It registers it in a graph of dependencies, because an action can depend on the output of another action (e.g. in C, the linker must be called after compilation).

In the **execution phase**, Bazel decides which actions must be run and in which order.



# Attributes

Rule context provides an API used to access rule **attributes**.

There are two special kinds of attributes:

- Dependency attributes, such as **attr.label** and `attr.label_list`, declare a dependency from the target that owns the attribute to the target whose label appears in the attribute's value. This kind of attribute forms the **basis of the target graph**.
- Output attributes, such as `attr.output` and `attr.output_list`, declare an output file that the target generates. Although they refer to the output file by label, they do not create a dependency relationship between targets. Output attributes are used relatively rarely, in favor of other ways of declaring output files that do not require the user to specify a label.

```
def _hello_world_impl(ctx):
    out = ctx.actions.declare_file(ctx.label.name +
    ".cc")
    ctx.actions.expand_template(
        output = out,
        template = ctx.file.template,
        substitutions = [{"NAME": ctx.attr.username}],
    )
    return [DefaultInfo(files = depset([out]))]

hello_world = rule(
    implementation = _hello_world_impl,
    attrs = {
        "username": attr.string(default = "unknown
person"),
        "template": attr.label(
            allow_single_file = [".cc.tpl"],
            mandatory = True,
        ),
    },
)
```



# Providers

Providers are pieces of information that a rule exposes to other rules that depend on it and it's **the only mechanism** to exchange data between rules.

This data can include output files, libraries, parameters to pass on a tool's command line, etc.

It is analogous to a function's return value and can be thought of as part of a **rule's public interface**

In our example, **DefaultInfo** provider is returned. It is a provider that gives general information about a target's direct and transitive files. Every rule type has this provider, even if it is not returned explicitly by the rule's implementation function.

**Outputs** are be passed along in providers to make them available to a target's consumers.

```
def _hello_world_impl(ctx):  
    out = ctx.actions.declare_file(ctx.label.name + ".cc")  
    ctx.actions.expand_template(  
        output = out,  
        template = ctx.file.template,  
        substitutions = {"{NAME}": ctx.attr.username},  
    )  
    return [DefaultInfo(files = depset([out]))]
```



## Providers, cont.

Since `provider` is just a named struct that contains information about a rule, we can define our own provider. This is done by calling the `provider` function.

```
MyProvider = provider(  
  doc = "My custom provider",  
  fields = {  
    "foo": "A foo value",  
    "bar": "A bar value",  
  },  
)
```



# Depset

Depsets are a specialized data structure for efficiently collecting data across a target's transitive dependencies.

The main feature of depsets is that they support a **time- and space-efficient merge operation**, whose cost is independent of the size of the existing contents.

```
def _hello_world_impl(ctx):  
    out = ctx.actions.declare_file(ctx.label.name + ".cc")  
    ctx.actions.expand_template(  
        output = out,  
        template = ctx.file.template,  
        substitutions = {"{NAME}": ctx.attr.username},  
    )  
    return [DefaultInfo(files = depset([out]))]
```



# Using Hello world Build Rule

Instantiation in BUILD file:

hello.bzl:

```
hello_world(  
    name = "hello",  
    username = "Alice",  
    template = "file.cc.tpl",  
)  
  
cc_binary(  
    name = "hello_bin",  
    srcs = [":hello"],  
)
```

```
hello_world = rule(  
    implementation = _hello_world_impl,  
    attrs = {  
        "username": attr.string(default = "unknown person"),  
        "template": attr.label(  
            allow_single_file = [".cc.tpl"],  
            mandatory = True,  
        ),  
    },  
)
```

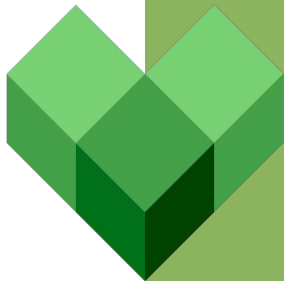


# Lab 6

## Workspace & Build Rules

### Objectives:

Create a repo rule and a build rule which enable Bazel-native python code generation



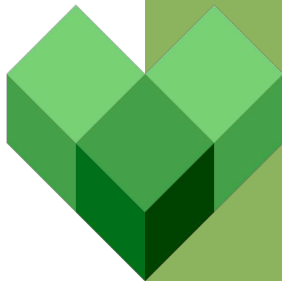
# Riot Bazel Training

Day 2, Lectures 7 & 8

# Schedule

## Day 2

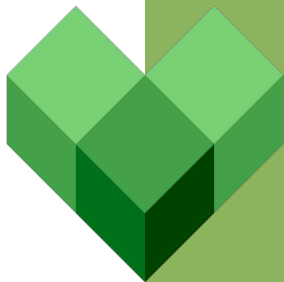
1. Introduction to Bazel
2. Using Bazel
3. Building Java, Python, Protobufs
4. CLI and Tooling



# Schedule

## Day 2

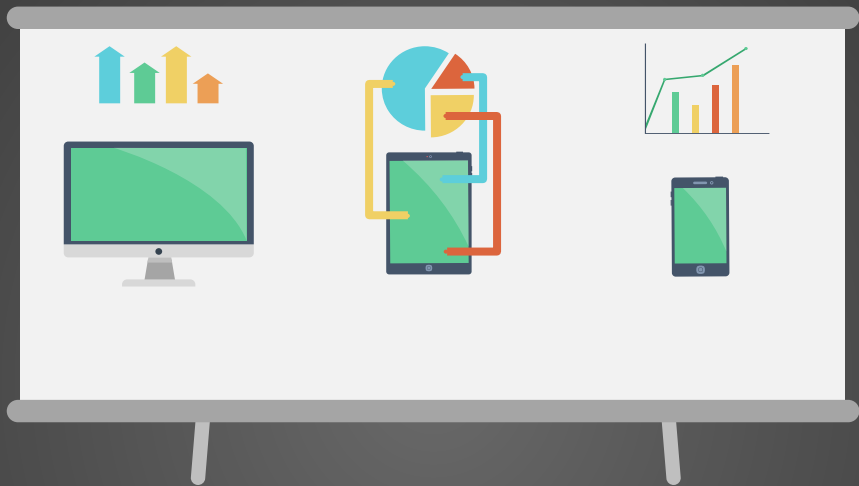
5. Starlark, Genrules, Macros
6. Writing Rules
7. Platforms and Toolchains
8. Remote Features, Packaging, Deployment



# Day 2

## Extending Bazel





## 7. Platforms and Toolchains

Bazel can build and test code **on a variety of hardware**, operating systems, and system configurations, using **many different versions of build tools such as linkers and compilers.**



To help manage this complexity, Bazel has a concept of **constraints** and **platforms**, which exist to serve exactly *one* purpose: to tell Bazel how to select the correct **toolchain** for a given action.



# What are Toolchains?

Toolchains are a standard way for rule authors to decouple their rule logic from platform-based selection of tools.

## Platforms

- Specified at the command line
- Are made up of a few constraints

## Constraints

- Specify values for machine properties
- Assist bazel in selecting the correct toolchain



# 7.1 Platforms

# Platforms

Bazel recognizes three roles that a platform may serve:

- **Host:** the platform on which Bazel itself runs.
  - **Execution:** a platform on which build tools execute build actions to produce intermediate and final outputs.
  - **Target:** a platform on which final output resides and executes.
- A platform is a collection of **constraint\_values**.
  - A **constraint\_value** is a property of a machine, for example, the OS.
  - A **toolchain** definition declares the **constraint\_values** of machines it can build for (target platform) as well as the machines it can run on (execution platform).



# Built-in Constraints & Platforms

- The Bazel team maintains a repository with constraint definitions for the most popular CPU architectures and operating systems.
- These are located at <https://github.com/bazelbuild/platforms>.
- Additionally, Bazel ships with a special platform definition: **@local\_config\_platform//:host**.
- This is the autodetected host platform value - represents the autodetected platform for the system Bazel is running on.



# Platforms – *Defining Constraints*

## Step 1.

Define a **constraint\_setting** and some values

```
constraint_setting(name = "my_constraint")
```

```
constraint_value(  
  name = "my_constraint_val_1",  
  constraint_setting = ":my_constraint",  
)
```

```
constraint_value(a  
  name = "my_constraint_val_2",  
  constraint_setting = ":my_constraint",  
)
```



# Platforms – *Defining the Platform*

## Step 2.

Add a platform which uses this constraint

```
platform(  
    name = "my_linux_x86",  
    constraint_values = [  
        "@platforms//os:linux",  
        "@platforms//cpu:x86_64",  
        ":my_constraint_val_1",  
    ],  
)
```

## Step 3.

Usage @ CLI:

```
bazel build //:my_target --platforms=//:my_linux_x86
```



## 7.2 Toolchains

---

# Toolchains

Toolchains allow Bazel to **properly select the build tool or compiler** to use when evaluating a rule (such as `java_binary` or `my_binary`).

Toolchains are relied on by custom build rule definitions and are decoupled for ease of use.



## Toolchains – *Defining a Toolchain (1 & 2)*

### Step 1.

Define the toolchain type in a BUILD file

```
toolchain_type(  
    name = "my_toolchain",  
)
```

### Step 2.

Define an **InfoProvider** to hold our constraint value (in a .bzl file)

```
MyInfo = provider(  
    fields = [  
        "my_attribute",  
    ],  
)
```



# Toolchains – *Defining a Toolchain (3A & 3B)*

## Step 3A

Define an implementation function for our rule:

```
def _my_toolchain_impl(ctx):  
    return [platform_common.ToolchainInfo(  
        myinfo = MyInfo(  
            my_attribute = ctx.attr.my_attribute,  
        ),  
    )]
```

## Step 3B

Define a rule providing **ToolchainInfo** returning our **InfoProvider** by calling the implementation function.

```
my_toolchain = rule(  
    implementation = _my_toolchain_impl,  
    attrs = {  
        "my_attribute": attr.string(  
            mandatory = True,  
        ),  
    },  
    provides = [platform_common.ToolchainInfo],  
)
```



# Toolchains — *Defining a Toolchain 4*

## Step 4

Create a build rule which uses the toolchain

```
def _my_library_impl(ctx):  
    info = ctx.toolchains["//:my_toolchain"].myinfo  
    print(info.my_attribute)  
    # ...
```

```
my_library = rule(  
    implementation = _my_library_impl,  
    attrs = {  
        # ...  
    },  
    toolchains = ["//:my_toolchain"]  
)
```



# Toolchains — *Defining a Toolchain 5*

## Step 5

Instantiate the toolchain in the **BUILD** file

```
my_toolchain(  
    name = "my_toolchain_impl",  
    my_attribute = "some_value",  
)
```

```
toolchain(  
    name = "my_toolchain",  
    toolchain_type = "//:my_toolchain",  
    target_compatible_with = [  
        "//:my_constraint_val_1",  
    ],  
    toolchain = ":my_toolchain_impl",  
)
```

```
my_library(  
    name = "example",  
    # ...  
)
```



## Toolchains – *Defining a Toolchain 6*

### Step 6

Register in the **WORKSPACE**:

```
register_toolchains(  
    "://:my_toolchain",  
)
```

### Step 7

Invoke via a Bazel command:

```
bazel build //:example --platforms=//:my_linux_x86
```



## Toolchains – *Defining a Toolchain 8*

Invoking `bazel build //:example --platforms=//:my_linux_x86` will result in the selection of the toolchain implementation which aligns with `my_constraint_val_1` as defined in the platform.

When the `my_library` target (with label `//:example`) is built, the custom rule implementation will receive a `MyInfo` provider populated with `my_attribute="some_value"`





## Lab 7

# Platforms & Toolchains

### Objective:

Create a custom toolchain, controlled by **constraints** and **platforms**, which is utilized by a **build rule** which invokes a custom compiler included in the repository



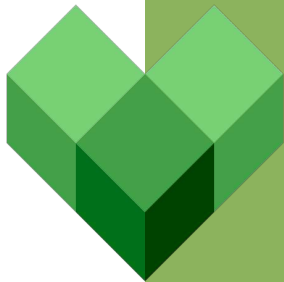


## 8. Caching, Remote Features, Packaging, Deployment

# Caching & Remote Features

One word — Speed.

Unleashing the true power of Bazel  
through Remote Cache and Remote  
Execution



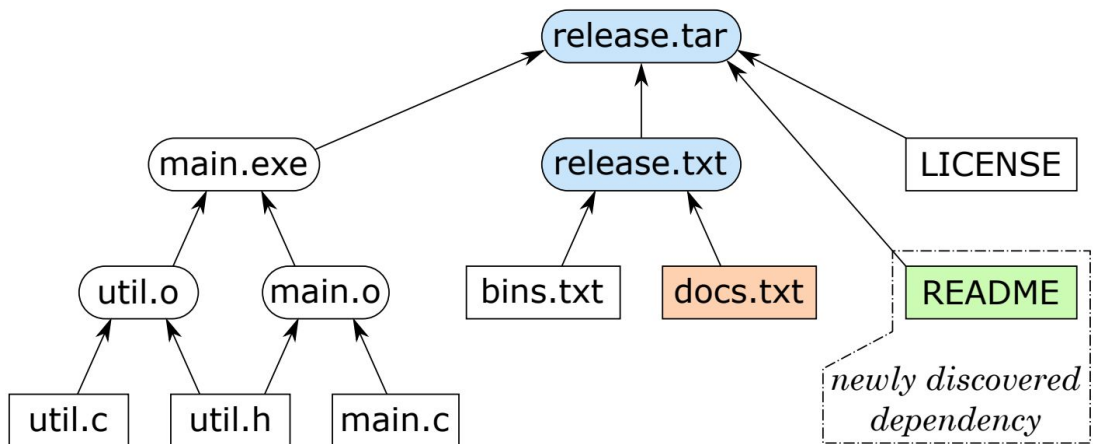
# 8.1 Local Caching

# There Are Two Cache Types:

1. A local cache
2. And a remote cache



# Smart Caching – *Rebuild Only What Changed*



Source: [Build Systems à la Carte](#)



# Caching in Bazel — *Speeding up the build*

## Local Cache — Implementations

1. **In-memory**

Bazel keeps parts of the analysis cache in memory for fast incremental builds

2. **On-disk**

Typically at `~/.cache/bazel`

3. **Or, on an external disk** cache (via `--disk_cache`)

This method allows for sharing of the artifacts between WORKSPACES as well as fast builds **even after bazel clean.**



# Caching in Bazel

## Local Cache — Disadvantages

On a **fresh checkout** or when sources have changed considerably, Bazel must build the entire dependency graph for a given target in order to populate the local cache.

- This can **take quite a long time** for large projects (hours!)
- Results in **redundant rebuilds** across a group of engineers working in the same codebase



## 8.2 Remote Caching

# Caching in Bazel

## Remote Cache is...

1. Shared across many developers
2. Configured with `--remote_cache=[your-cache-url]`
3. Can be a **HTTP**-based caches:
  - a. Nginx with WEBDAV
  - b. GCS & AWS S3 blob storage
4. Can be a **gRPC**-based:
  - a. **Bazel-remote** — an OSS gRPC based cache
  - b. **Flare® Cache** - a blazing-fast managed cache SaaS



---

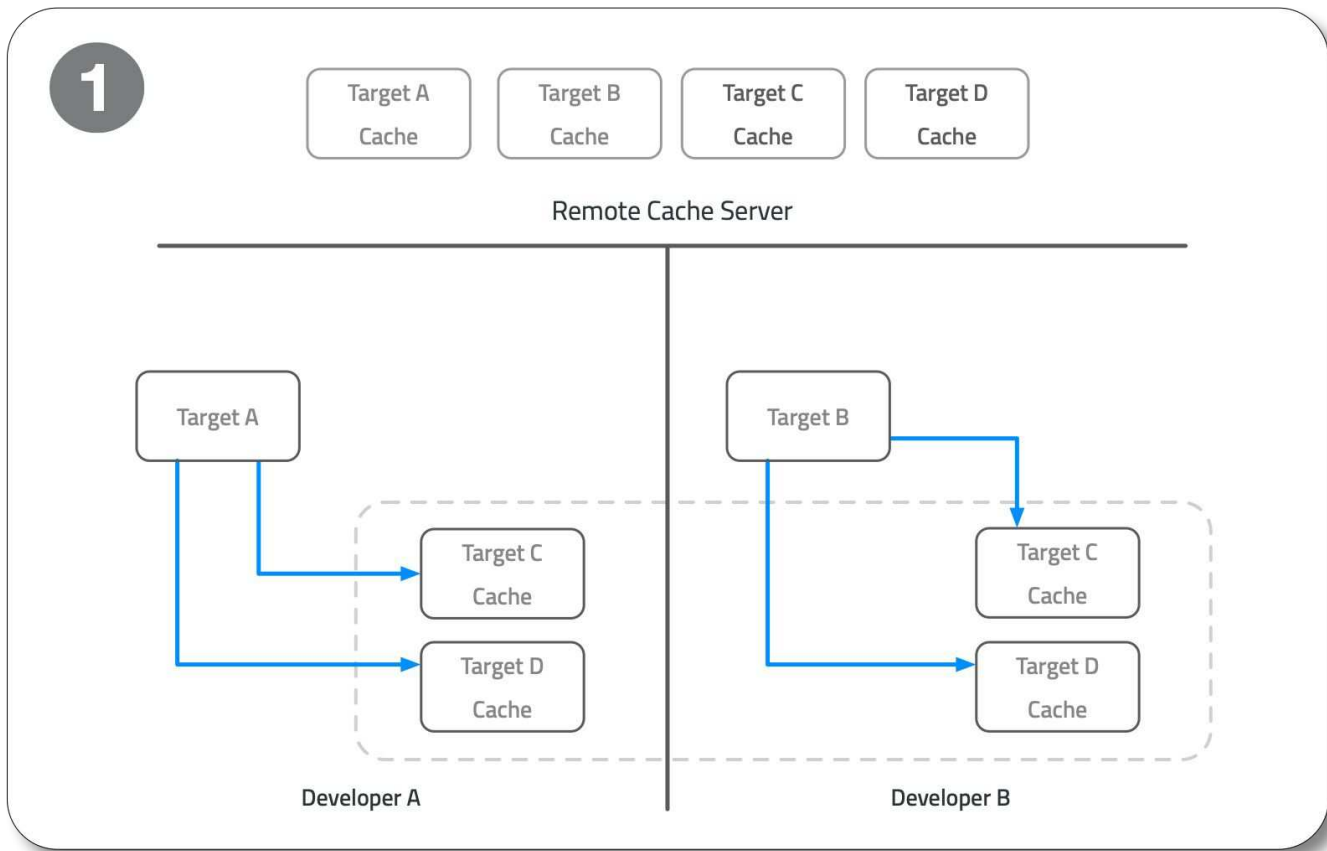
# Bazel Cache

## Remote Cache Examples

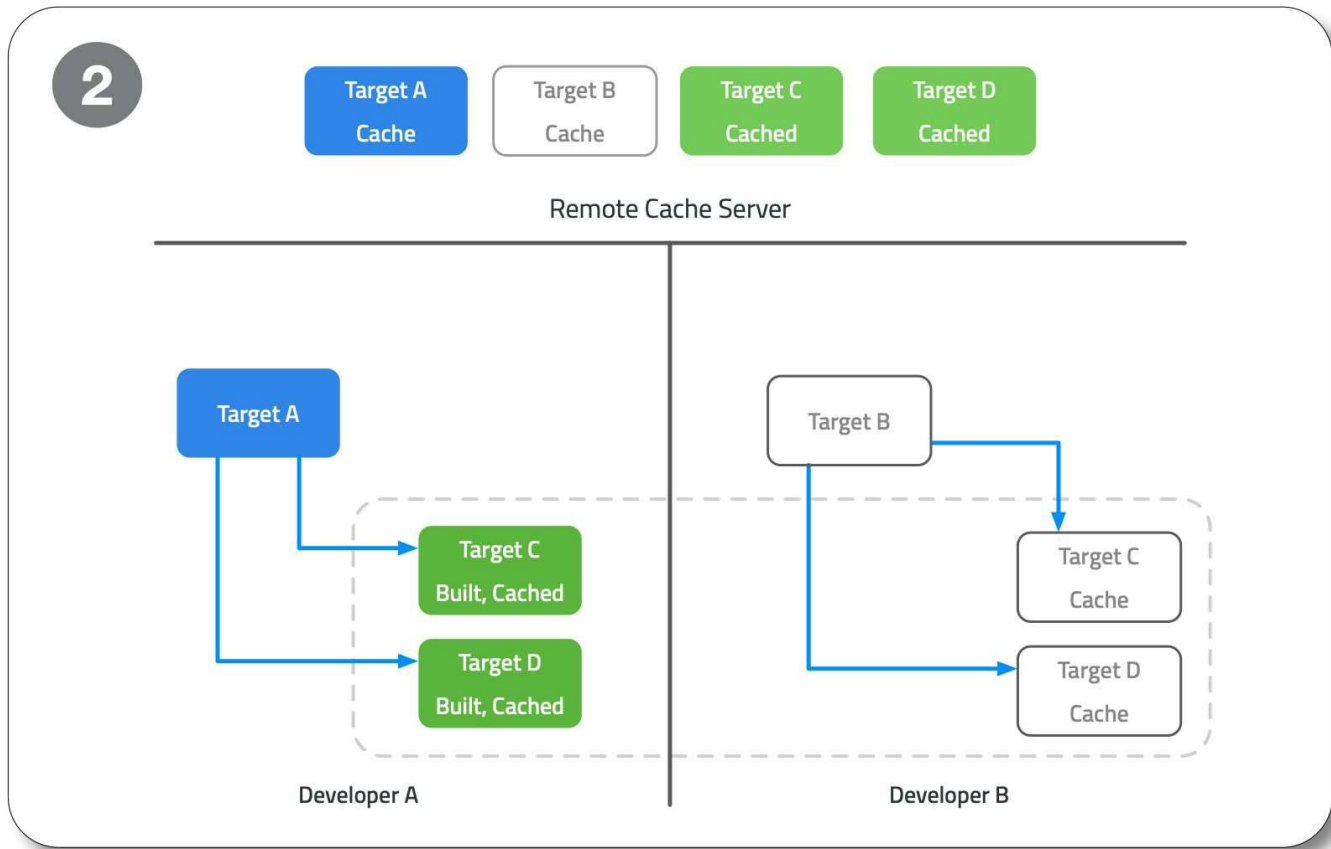
- In the next few slides we demonstrate visually the propagation of a locally built and cached target to a remote cache.
- Two developers are working on different projects, but they both share two dependencies: **C & D**.



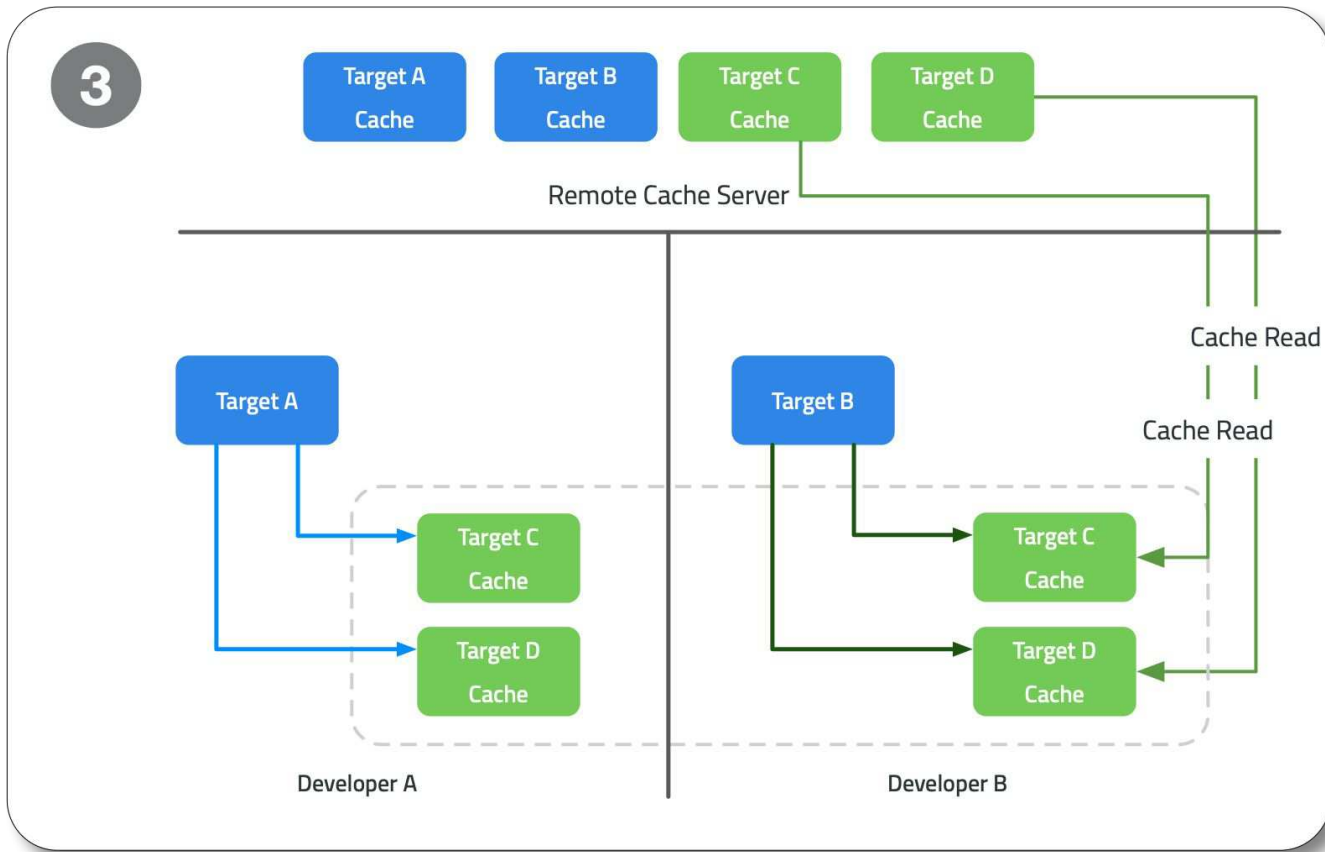
# Remote Caching — *Nothing is built yet*



# Remote Caching — *Developer A Builds Her Targets*



# Remote Caching — *Developer B fetches C&D from the cache*



---

# Bazel Cache

## Remote Cache Conclusions

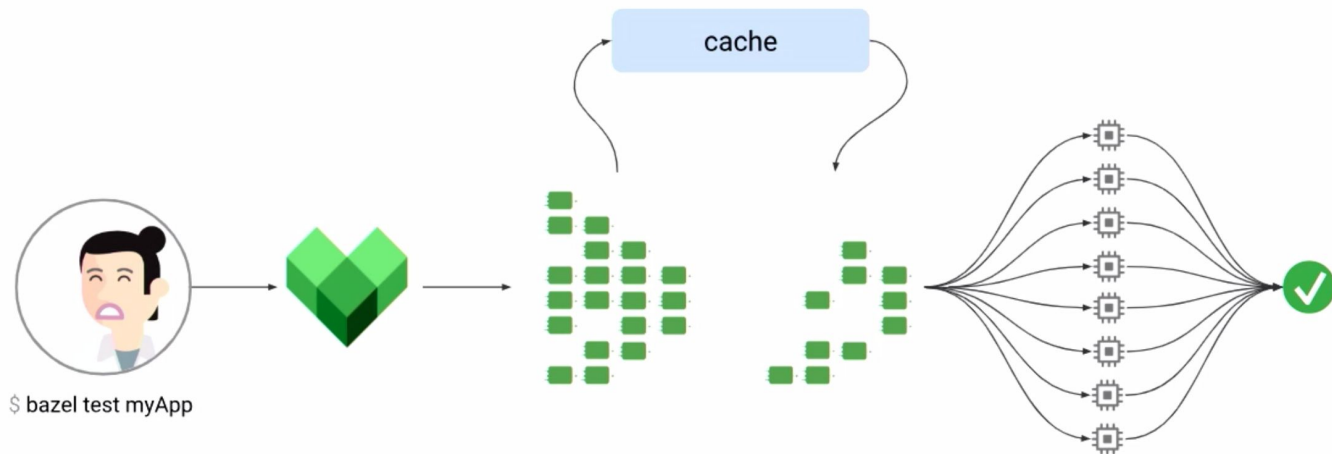
- Remote Cache is particularly effective **on very large projects** with **thousands of developers**; in fact, probably necessary
- New Engineers starting up are able have **near-instant local builds** because all artifacts are already cached on CI
- Ensures that only the **absolute minimum work is performed** each time a target must be rebuilt
- **Is a key ingredient** in the “secret sauce” that makes Bazel such a fast and effective build system for complex projects.



## 8.3. Remote Execution Features

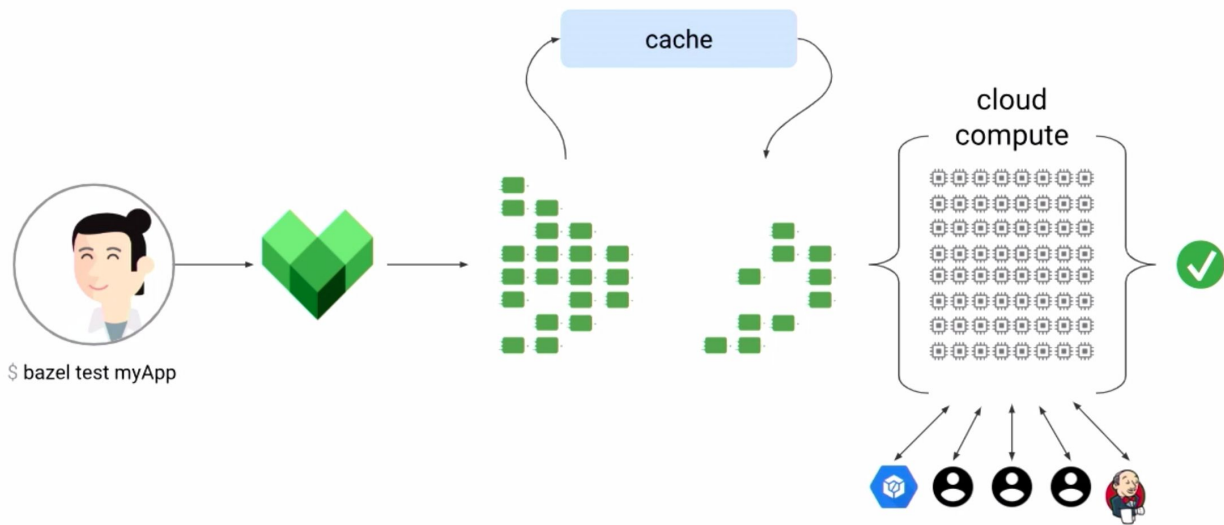
# Local Execution / Local Caching

Developer machines are over-provisioned, until they actually need to do some work.



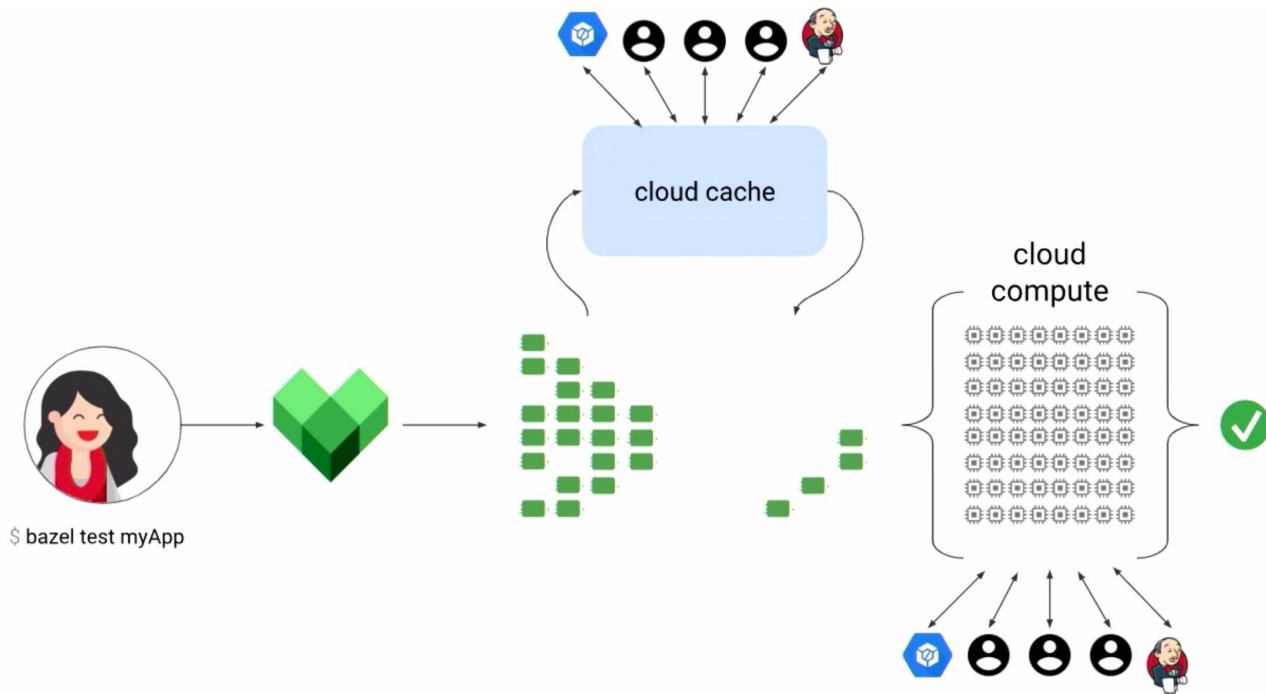
# Remote Execution / Local Caching

The only hard limit on the number of cloud nodes used for build parallelization is the topology of your build graph!



# Remote Execution / Remote Caching

Remote cache makes **everyone's** build faster: **very small subset** of actions needs to be rebuilt.



# Remote Features: Takeaways

- **Remote cache:** only rebuild the build sub-graph, affected by your code change, all other artifacts are downloaded from remote cache pre-populated by CI.
- **Remote execution:** once the minimal subset of actions is defined, execute them with maximum parallelism possible for your build graph.
- Using remote cache without remote execution is able to **bring 10x speed up** to the build, while using remote execution without remote cache will not.



## 8.4. Packaging & Distribution

# Packaging Rules

[https://github.com/bazelbuild/rules\\_pkg](https://github.com/bazelbuild/rules_pkg)

`rules_pkg` can be used to create zip or tar archives from a set of inputs (which can be outputs of other rules, such as compilers).

- `pkg_zip` — the most basic rule in this package
- `pkg_tar` — provides the most flexibility thanks to `strip_prefix`
- `pkg_deb` — often takes a `pkg_tar` as input



- `pkg_zip` tends to **flatten folder** structures in a way that is undesirable for most programs.
- `pkg_tar` is able to work around this, but if a zip with a particular structure is desired, one of the alternatives that follows is recommended.





# Example: Packaging Python

```
# src/main/python/hello/BUILD:
```

```
load("@rules_python//python:defs.bzl", "py_binary")
```

```
py_binary(  
    name = "hello",  
    srcs = ["hello.py"],  
)
```

Usage:

- Run the program:  
`bazel run //src/main/python/hello:hello`
- Create a zip:  
`bazel build //src/main/python/hello:hello --build_python_zip`





## Example: Packaging Java

```
# src/main/java/com/flarebuild/hello/BUILD:

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

java_binary(
    main_class = "com.flarebuild.hello.Hello",
    name = "hello",
    srcs = ["Hello.java"],
)
```

Usage:

- Run the program:  
`bazel run //src/main/java/com/flarebuild/hello:hello`
- Create a jar:  
`bazel build //src/main/java/com/flarebuild/hello:hello_deploy.jar`





# Packaging Maven

There are a variety of ways to deploy artifacts to a maven repository:

- via the built-in pom file generator,
- using a third-party maven build rule, or
- writing a rule from scratch on top of the maven binary, implementing requirements specifically to your organization.

The implementation of a custom maven rule is out of scope here, but the following resources are a good place to start for those interested in deploying artifacts to maven repositories:

- Bazel Distribution:  
<https://github.com/graknlabs/bazel-distribution>
- Pom Gen:  
<https://github.com/salesforce/pomgen>
- Bazel's default pom file support via [bazel-common](#)





---

# Packaging With Docker

[https://github.com/bazelbuild/rules\\_docker](https://github.com/bazelbuild/rules_docker)

**rules\_docker** repository enables Bazel to create Docker images with our binary targets baked right in. This is the recommended approach in modern projects.

The following rules are available:

- `container_image`
- `container_bundle`
- `container_import`
- `container_load`
- `container_pull`
- `container_push`





# container\_pull

- A **WORKSPACE rule** which allows pulling down an image from an external docker registry.
- It's *image* target can later be used as a base image in other docker rules.
- Corresponding image labels will reference the image as, eg. “@debian10//image” and “@java\_distroless//image”

```
container_pull(  
    name = "base",  
    registry = "gcr.io",  
    repository = "my-project/my-base",  
    # 'tag' is also supported,  
    # but digest is encouraged for reproducibility.  
    digest = "sha256:deadbeef",  
)
```





# container\_pull, ctd.

```
container_pull(  
    name = "debian10",  
    # tag = "10.5-slim",  
    digest = "sha256:e0a33348ac8cace6b4294885e6e0bb57ecdfe4b6e415f1a7f4c5da5fe3116e02",  
    registry = "index.docker.io",  
    repository = "library/debian",  
)
```

```
container_pull(  
    name = "java_distroless",  
    # tag = "11",  
    digest = "sha256:19ebdd790a1cd1592036644543c50f6b2d133e631ae090460701089ab0962d41",  
    registry = "gcr.io",  
    repository = "distroless/java",  
)
```





# container\_image

`container_image` is the most basic packaging rule that builds a Docker image:

```
# src/main/docker/BUILD:
```

```
container_image(  
    name = "hello_image",  
    # References container_pull  
    # rule from the WORKSPACE file  
    base = "@debian10//image",  
    cmd = [  
        "cat",  
        "/data/hello.txt",  
    ],  
    directory = "/data",  
    files = ["data/hello.txt"],  
)
```

An equivalent **Dockerfile**:

```
FROM debian:10.5-slim  
ADD data/hello.txt /data/hello.txt  
CMD cat /data/hello.txt
```





# container\_image, ctd.

A more useful example of `container_image`: binary produced by a target from another package.

```
# src/main/docker/BUILD:
```

```
container_image(  
    name = "java_hello_image",  
    # References container_pull from WORKSPACE (above)  
    base = "@java_distroless//image",  
    entrypoint = [  
        "/usr/bin/java",  
        "-cp",  
        "hello.jar",  
        "com.flarebuild.hello.Hello",  
    ],  
    files = ["//src/main/java/com/flarebuild/hello:hello.jar"],  
)
```

```
# An equivalent Dockerfile:
```

```
FROM gcr.io/distroless/java:11  
ADD bazel-bin/src/main/java/com/flarebuild/hello/hello.jar /hello.jar  
ENTRYPOINT ["/usr/bin/java", "-cp", "hello.jar", "com.flarebuild.hello.Hello"]
```





# container\_push

An executable rule that pushes a Docker image to a Docker registry on bazel run.

```
# src/main/docker/BUILD:
```

```
container_push(  
    name = "push_hello",  
    format = "Docker",  
    image = ":hello_image",  
    registry = "index.docker.io",  
    repository = "${image_repository}/hello_image",  
    tag = "{BUILD_TIMESTAMP}",  
    tags = ["manual"],  
)
```

Usage (requires preliminary *docker login*):

```
bazel run //src/main/docker:push_hello \  
    --define image_repository=<your docker hub id here>
```





# lang\_image

Language rules (f.e. `py_image`, `java_image`, `cc_image`) are based on `container_image` and produce the same outputs.

As the documentation states:

“The idea behind these rules is to make containerizing an application built via a `lang_binary` rule as simple as changing it to `lang_image`”

```
# src/main/python/hello/BUILD:
```

```
py_image(  
    name = "hello_image",  
    srcs = ["hello.py"],  
    # Need to specify main  
    # explicitly because the  
    # name is different  
    main = "hello.py",  
)
```





# Loading Docker Toolchain

```
load("@bazel_tools//tools/build_defs/repo:http.bzl", "http_archive")

# Download the rules_docker repository at release v0.14.4
http_archive(
    name = "io_bazel_rules_docker",
    sha256 = "4521794f0fba2e20f3bf15846ab5e01d5332e587e9ce81629c7f96c793bb7036",
    strip_prefix = "rules_docker-0.14.4",
    urls = [
        "https://github.com/bazelbuild/rules_docker/releases/download/v0.14.4/rules_docker-v0.14.4.tar.gz"
    ],
)

load(
    "@io_bazel_rules_docker//repositories:repositories.bzl",
    container_repositories = "repositories",
)

container_repositories()

load("@io_bazel_rules_docker//repositories:deps.bzl", container_deps = "deps")

container_deps()
```

```
load(
    "@io_bazel_rules_docker//repositories:pip_repositories.bzl",
    "pip_deps"
)

pip_deps()

load(
    "@io_bazel_rules_docker//container:container.bzl",
    "container_pull",
)

container_pull(
    name = "java_base",
    # 'tag' is also supported, but
    # digest is encouraged for reproducibility.
    digest = "sha256:deadbeef",
    registry = "gcr.io",
    repository = "distroless/java",
)
```





# Configuring Docker Toolchain

- Call `docker_toolchain_configure` to override the default docker toolchain configuration.
- This call should be placed BEFORE the call to "`container_repositories`" below to actually override the default toolchain configuration.
- Note this is only required if you actually want to call `docker_toolchain_configure` with a custom attr.

```
load(  
  "@io_bazel_rules_docker//toolchains/docker:toolchain.bzl",  
  docker_toolchain_configure = "toolchain_configure",  
)  
  
docker_toolchain_configure(  
  name = "docker_config",  
  client_config = "<enter absolute path to your docker config directory here>",  
  docker_flags = [  
    "--tls",  
    "--log-level=info",  
  ],  
  docker_path = "<enter absolute path to the docker binary (in the remote exec env) here>",  
  gzip_path = "<enter absolute path to the gzip binary (in the remote exec env) here>",  
  gzip_target = "<enter absolute path (i.e., must start with repo name @...//:...) to an executable gzip target>",  
  xz_path = "<enter absolute path to the xz binary (in the remote exec env) here>",  
)
```



## 8.5. Continuous Integration

---

## Best Practices for CI

- Use the same VM/container image for running bazel
- Use build cache
  - can be remote cache
  - can be local cache preserved by CI system
  - only CI should have write access to the cache to prevent spoiling from hosts with another toolchain versions
- CI shouldn't use tools other than **bazel build**, **bazel test**, **bazel run** so every engineer may replicate CI processes locally with bazel
- Preserve bazel repository cache to avoid re-downloading external dependencies.



# GitHub Actions workflow example

```
# .github/workflows/build.yml:
```

## steps:

- **uses:** actions/checkout@v2
- **name:** Restore repository cache  
**uses:** actions/cache@v2  
**with:**
  - path:** repository\_cache/
  - key:** bz1-r-cache-\${{ hashFiles('WORKSPACE') }}
- **name:** Install .bazelrc for CI  
**run:** |
  - cat <<EOF > .bazelrc.user
  - build --repository\_cache repository\_cache/
  - build --remote\_cache=grpc://remote.cache.tld
  - EOF
- **name:** Bazel Build  
**uses:** docker://gcr.io/cloud-builders/bazel:3.4.1  
**run:** |
  - bazel build //...:all

Notes about Github Actions configuration shown here:

1. **We activate Github Actions caching** in order to be able to cache external dependencies between builds.
2. We supply Bazel with **CLI flag overrides** via the file `.bazelrc.user` which we are loading (if exists) from `.bazelrc`.
3. The flags redirect Bazel's cache to the directory that will be **included in the Action Cache** once the build succeeds.
4. By using `.bazelrc.user` we ensure that **all Bazel invocations will be using the same disk cache**.
5. We use a specific docker image with the exact Bazel version and set of host toolchains for bazel steps



## 8.6. Bazel “In Pictures”

# “Bazel in Pictures” — A Diagram Repo

<https://github.com/bazelruby/bazel-in-pictures>

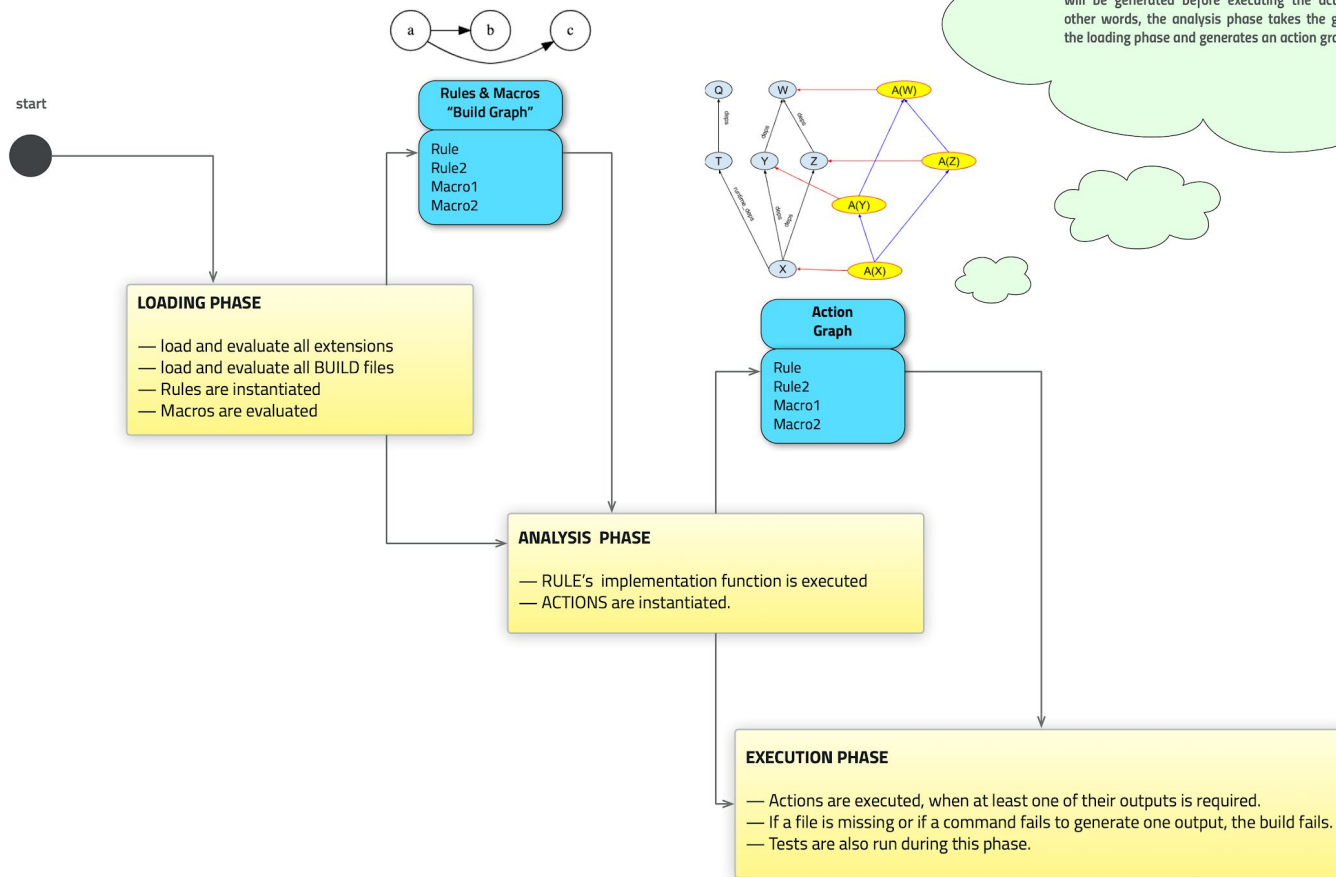


An open source project that aims to create helpful diagrams that aid in learning and understanding Bazel for both beginners, and Custom Rules Authors.

The project is work in progress.



# Build Phases



An action describes how to generate a set of outputs from a set of inputs, e.g. "run gcc on hello.c and get hello.o".

It is important to note that we have to list explicitly which files will be generated before executing the actual commands. In other words, the analysis phase takes the graph generated by the loading phase and generates an action graph.

# Rules

## Data Model

### rule()

Creates a new rule, which can be called from a BUILD file or a macro to create targets. Rules must be assigned to global variables in a .bzl file; the name of the global variable is the rule's name.

Test rules are required to have a name ending in `_test`, while all other rules must not have this suffix. (This restriction applies only to rules, not to their targets.)

```
implementation(function)
test(bool)=False
attrs(dict)=None
outputs(dict)=None
executable(bool)=False
output_to_genfiles(bool)=False
fragments(sequence)=[]
host_fragments(sequence)=[]
_skyllark_testable(bool)=False
toolchains(sequence<string>)=[]
doc(string)="
*"
provides(sequence<Provider>)
exec_compatible_with(sequence<String>)
analysis_test(bool)=False
build_setting(BuildSetting)=None
cfg=None
```

path
basename → string
dirname → path
exists → bool
get_child → path
readdir → list
realpath → path

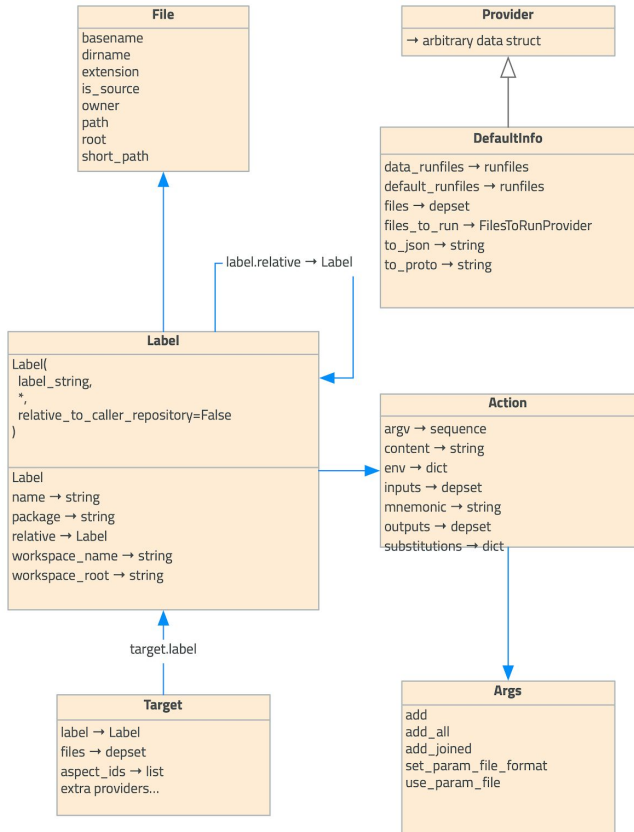
root
path

rule_attributes
attr
executable
file
files
kind

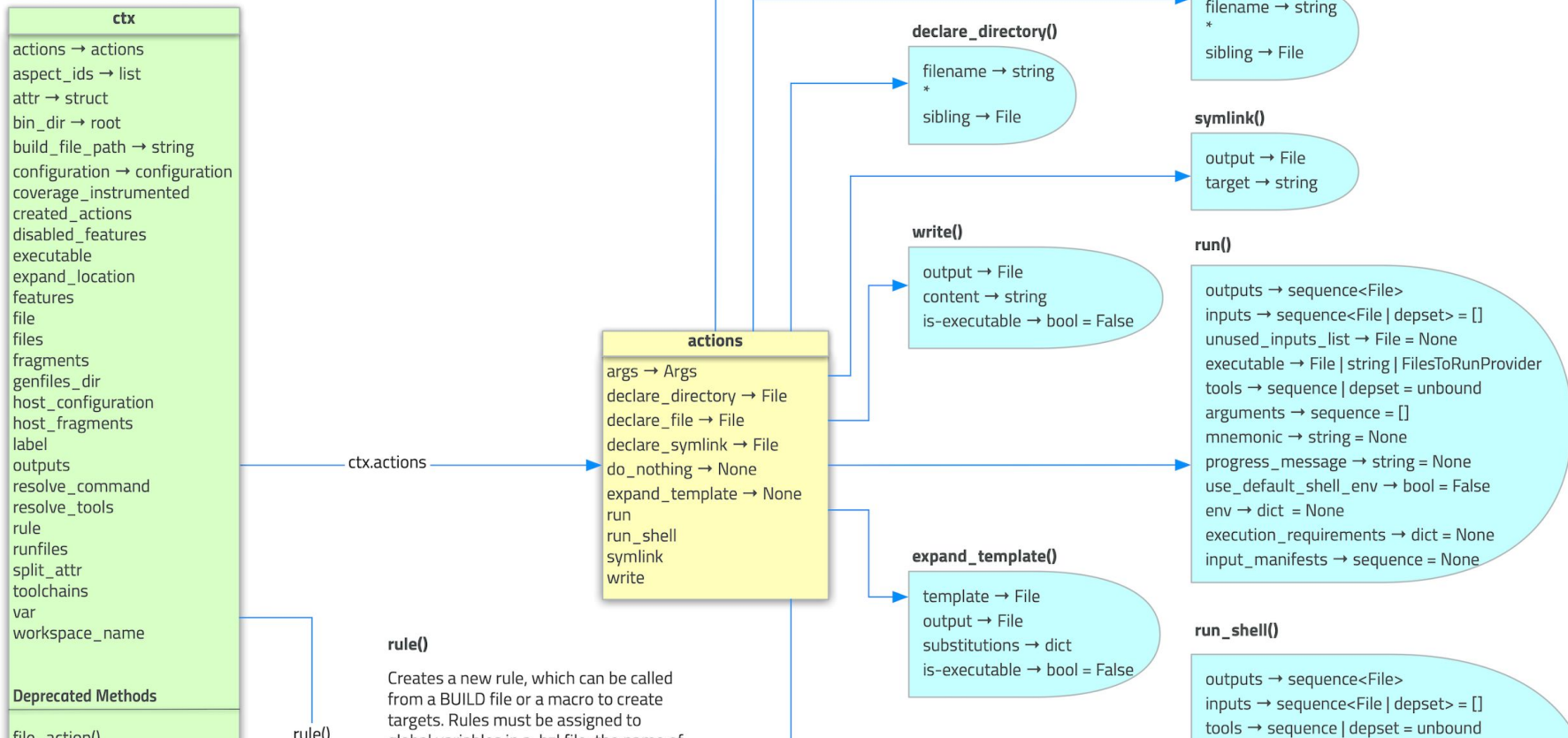
attr
bool
int
int_list
label
label_keyed_string_dict
label_list
output
output_list
string
string_dict
string_list
string_list_dict

runfiles
empty_filenames → depset
files → depset
merge → runfiles
root_symlinks → depset
symlinks → depset

depset
depset(
x=None,
order="default",
*,
a=None,
transitive=None,
items=[]
)
to_list → list



# Build Execution Context (1 of 2)



- file
- files
- fragments
- genfiles\_dir
- host\_configuration
- host\_fragments
- label
- outputs
- resolve\_command
- resolve\_tools
- rule
- runfiles
- split\_attr
- toolchains
- var
- workspace\_name

#### Deprecated Methods

- file\_action()
- action() → **Action**
- build\_setting\_value()
- default\_provider()
- empty\_action()
- expand\_make\_variables()
- new\_file()
- template\_action()

ctx.actions

**actions**

- args → Args
- declare\_directory → File
- declare\_file → File
- declare\_symlink → File
- do\_nothing → None
- expand\_template → None
- run
- run\_shell
- symlink
- write

#### expand\_template()

- template → File
- output → File
- substitutions → dict
- is-executable → bool = False

#### run\_shell()

- outputs → sequence<File>
- inputs → sequence<File | depset> = []
- tools → sequence | depset = unbound
- arguments → sequence = []
- mnemonic → string = None
- command → string
- progress\_message → string = None
- use\_default\_shell\_env → bool = False
- env → dict = None
- execution\_requirements → dict = None
- input\_manifests → sequence = None

#### rule()

Creates a new rule, which can be called from a BUILD file or a macro to create targets. Rules must be assigned to global variables in a .bzl file; the name of the global variable is the rule's name.

Test rules are required to have a name ending in `_test`, while all other rules

rule()

```

implementation(function)
test(bool)=False
attrs(dict)=None
outputs(dict)=None
executable(bool)=False
output_to_genfiles(bool)=False
fragments(sequence)=[]
host_fragments(sequence)=[]
_skylark_testable(bool)=False
toolchains(sequence<string>)=[]
doc(string)=""
*
provides(sequence<Provider>)
exec_compatible_with(sequence<String>)
analysis_test(bool)=False
build_setting(BuildSetting)=None
cfg=None
  
```

## Build Execution Context (2/2)

# Repository Context (1 of 2)

## repository\_ctx

```
attr → struct
name → string
os → repository_os
delete(string path) → bool
download()
download_and_extract()
execute()
extract()
file()
patch(patch_file, strip = 0)
path(path) → path
read(path) → string
report_progress(string status)
symlink(string from, string to)
template()
```

Returns the path of the corresponding program or None if there is no such program in the path.

```
which(string program)
```

## configuration

```
coverage_enabled
default_shell_env
host_path_separator
test_env
```

## download()

Downloads a file to the output path for the provided url and returns a struct containing a hash of the file with the fields sha256 and integrity.

```
url(string(s))
output(string | Label | path)
sha256=(string)
executable(bool) = False
allow_fail(bool) = False
canonical_id(string) = ""
auth(dict) = {}
*,
integrity(string) = ""
```

## template()

Generates a new file using a template. Every occurrence in template of a key of substitutions will be replaced by the corresponding value. The result is written in path. An optional executable argument (default to true) can be set to turn on or off the executable bit.

```
path(string | Label | path)
template(string | Label | path)
substitutions(dict) = {}
executable(bool) = True
```

## download\_and\_extract()

Downloads a file to the output path for the provided url, extracts it, and returns a struct containing a hash of the downloaded file with the fields sha256 and integrity.

```
url(string(s))
output(string | Label | path)
sha256=(string)
executable(bool) = False
allow_fail(bool) = False
canonical_id(string) = ""
auth(dict) = {}
*,
integrity(string) = ""
type(string) = ""
stripPrefix(string) = ""
```

## extract()

Extract an archive to the repository directory.

```
archive(string | Label | path)
output(string | Label | path) = ""
stripPrefix(string) = ""
```

**report\_progress**(string status)  
**symlink**(string from, string to)  
**template()**

Returns the path of the corresponding program or None if there is no such program in the path.

**which**(string program)

#### configuration

coverage\_enabled  
default\_shell\_env  
host\_path\_separator  
test\_env

**integrity**(string) = ""

#### template()

Generates a new file using a template. Every occurrence in template of a key of substitutions will be replaced by the corresponding value. The result is written in path. An optional executable argument (default to true) can be set to turn on or off the executable bit.

path(string | Label | path)  
template(string | Label | path)  
substitutions(dict) = {}  
executable(bool) = True

#### file()

Generates a file in the repository directory with the provided content.

path(string | Label | path)  
content(string) = ""  
executable(bool) = True  
legacy\_utf8(bool) = True

**integrity**(string) = ""  
**type**(string) = ""  
**stripPrefix**(string) = ""

#### extract()

Extract an archive to the repository directory.

archive(string | Label | path)  
output(string | Label | path) = ""  
stripPrefix(string) = ""

#### execute()

Executes the command given by the list of arguments. The execution time of the command is limited by timeout (in seconds, default 600 seconds). This method returns an exec\_result structure containing the output of the command. The environment map can be used to override some environment variables to be passed to the process.

arguments(sequence)  
timeout(int)=600  
environment(dict)={}  
quiet(bool)=True  
working\_directory=(string) = ""

## Repository Context (2 of 2)

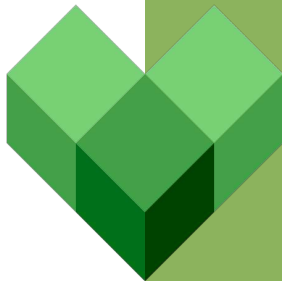


## Lab 8

# Remote Features & Packaging

### Objectives:

- Try out remote features
- Build, push, and pull a docker image



# THE END.

Thanks!

Your friends @ Flare.Build