flare.build

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

flare.build

Day 1
Getting to Know Bazel

flare.build

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.

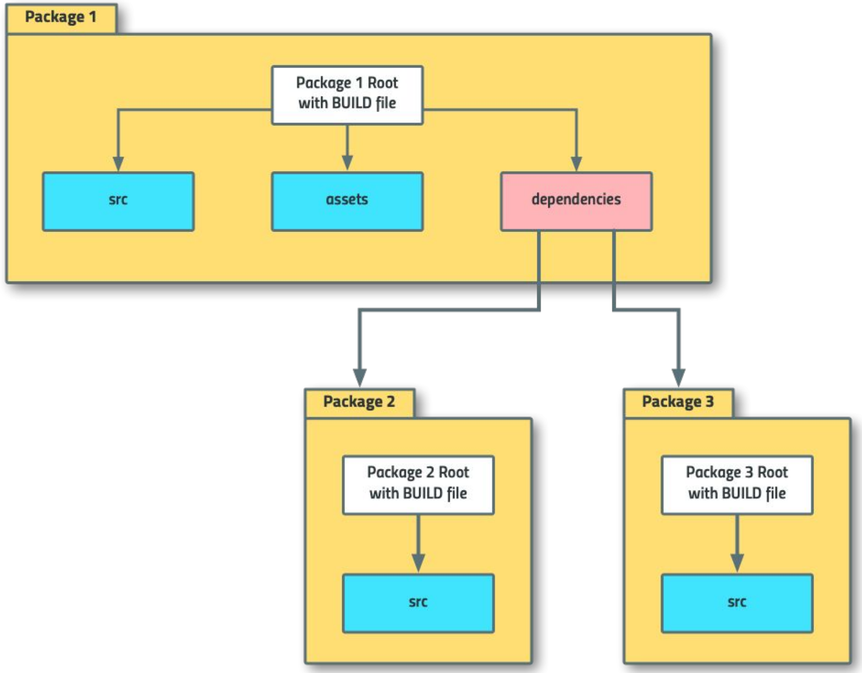
The file can be empty, and only one **WORKSPACE** file is allowed per repository.

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 Bazil 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.


Package 1
Package 1 Root with BUILD file
src
assets
dependencies
Package 2
Package 2 Root with BUILD file
src
Package 3
Package 3 Root with BUILD file
src

flare.build

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.

flare.build

1.2 Core Concepts

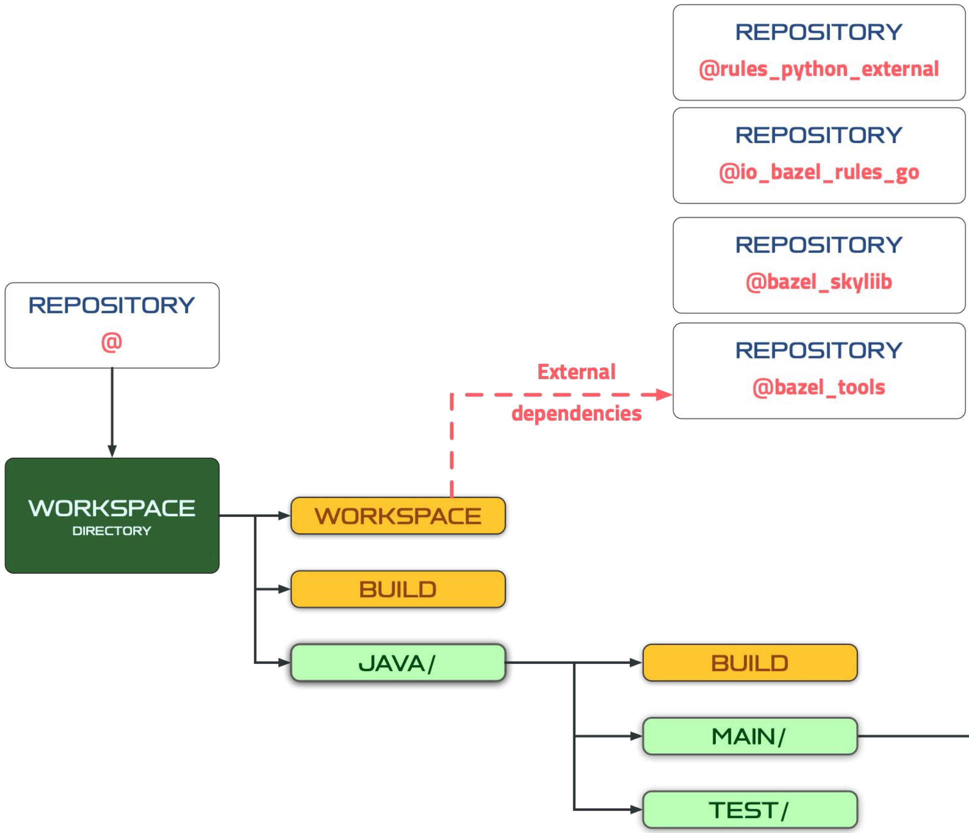
REPOSITORIES & WORKSPACES

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.

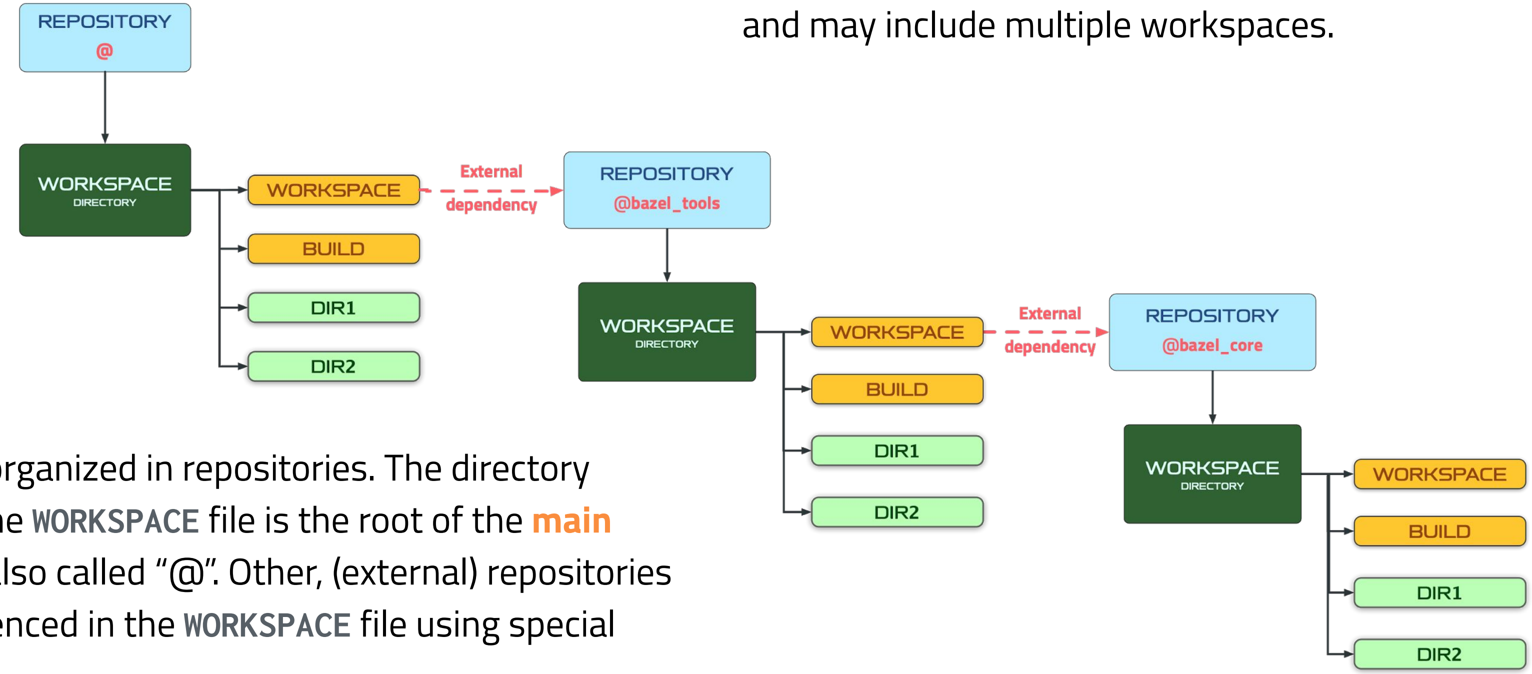
Directories containing file `WORKSPACE` are considered the root of a repository, and are referenced as "@"



flare.build

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”,
)
```

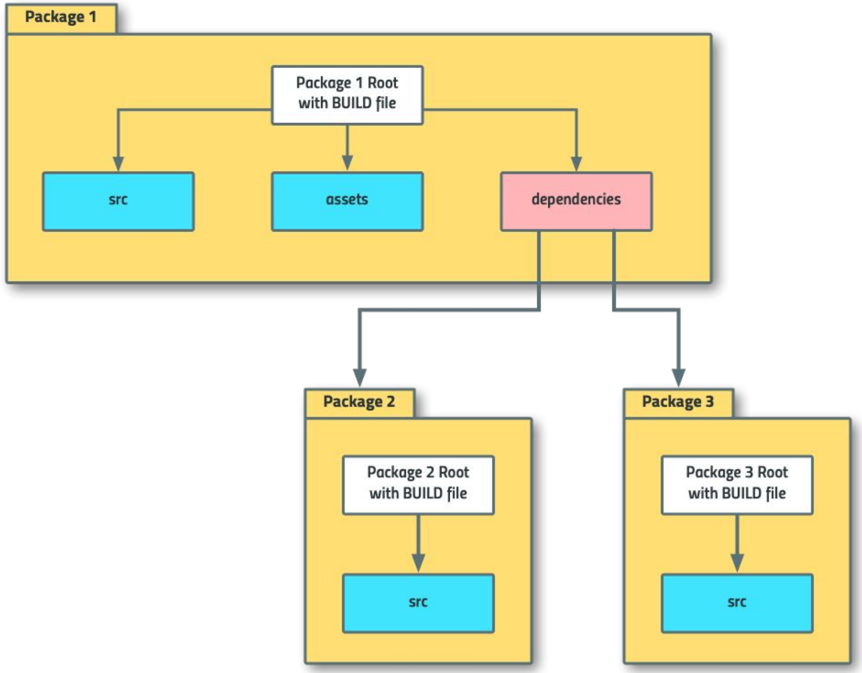
PACKAGES

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 1
Package 1 Root with BUILD file
src
assets
dependencies
Package 2
Package 2 Root with BUILD file
src
Package 3
Package 3 Root with BUILD file
src

PACKAGE EXAMPLE

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

```
src
└── my
    └── app
        ├── BUILD
        ├── app.cc
        ├── data
        │   └── input.txt
        └── tests
            ├── BUILD
            └── test.cc
```

- 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.

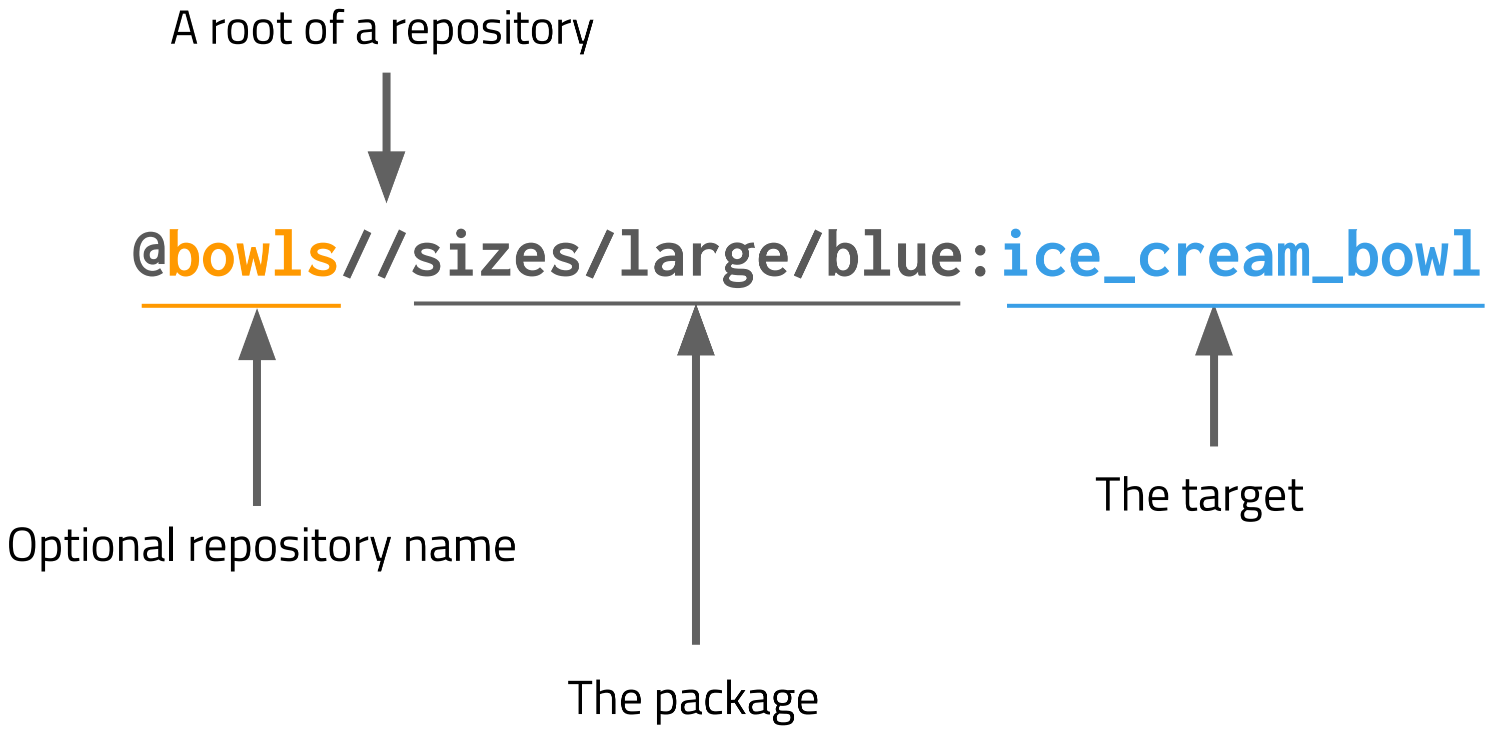
- Label is also a **proper Class** in Bazel and **can be instantiated** and passed around as an argument.

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



```
@bowls//sizes/large/blue:ice_cream_bowl
```

flare.build

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**

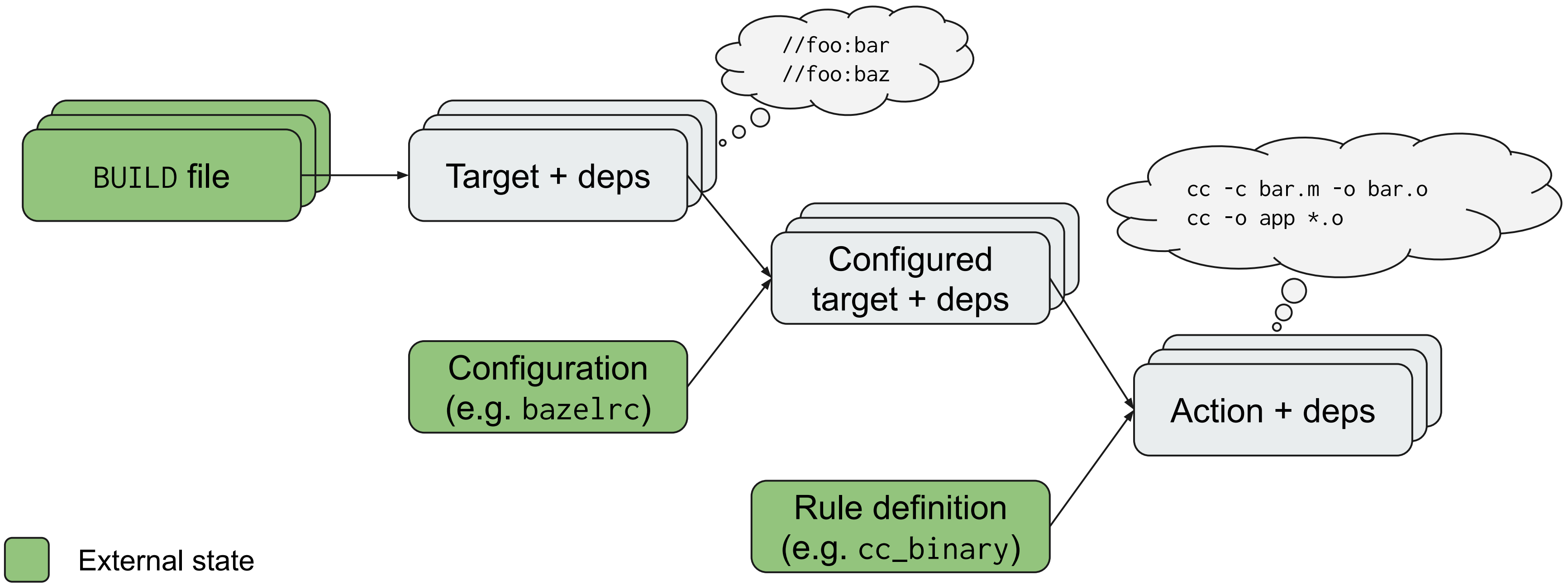
Rules & Actions

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.

flare.build

Build Phases



🟩 External state

⬜ In-memory graph maintained by Bazel

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
```

flare.build

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 asan 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.

flare.build

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.

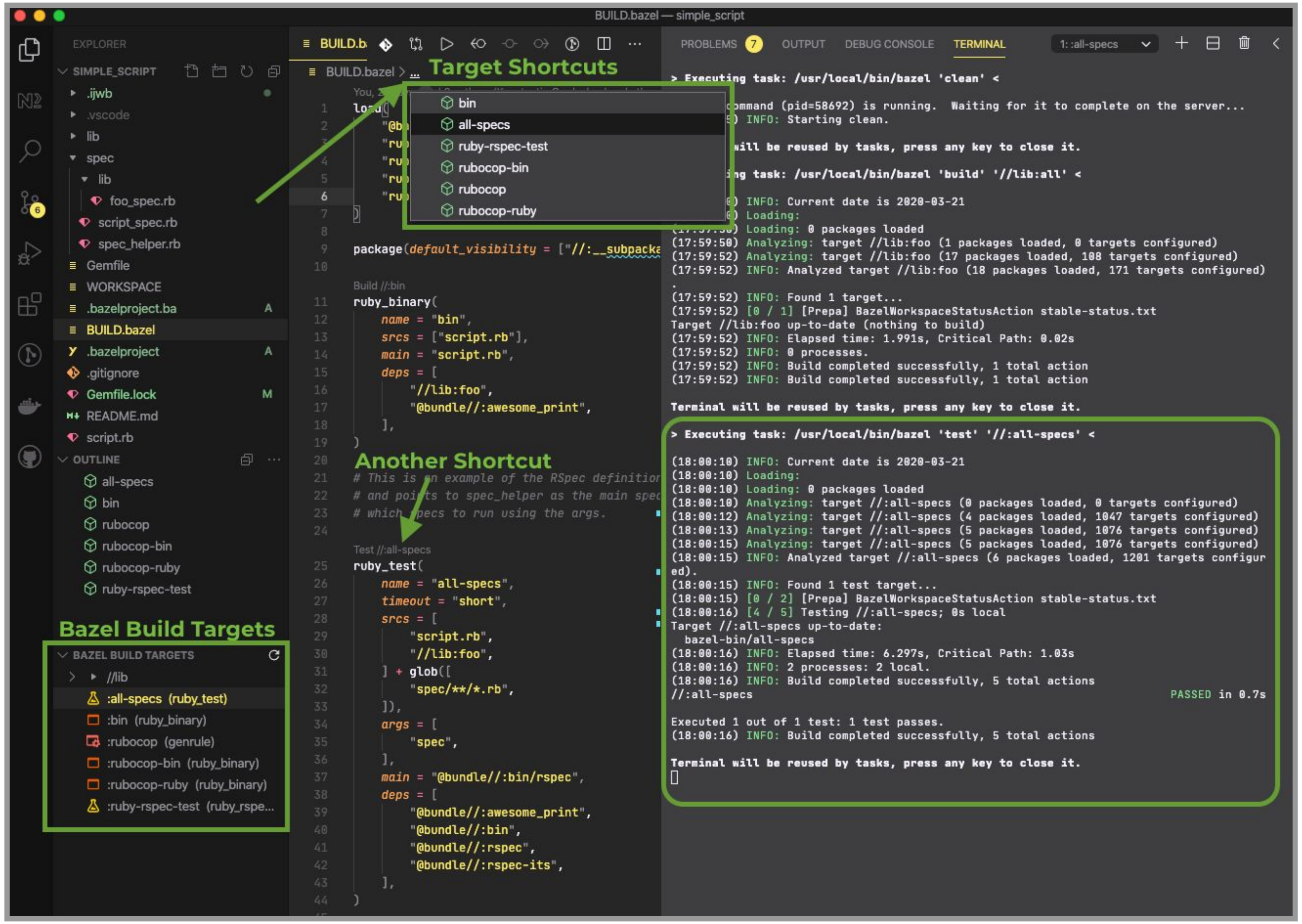
Configurable attributes, commonly known as select(), are a Bazel feature that lets users toggle the values of BUILD rule attributes at the command line.

```
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()

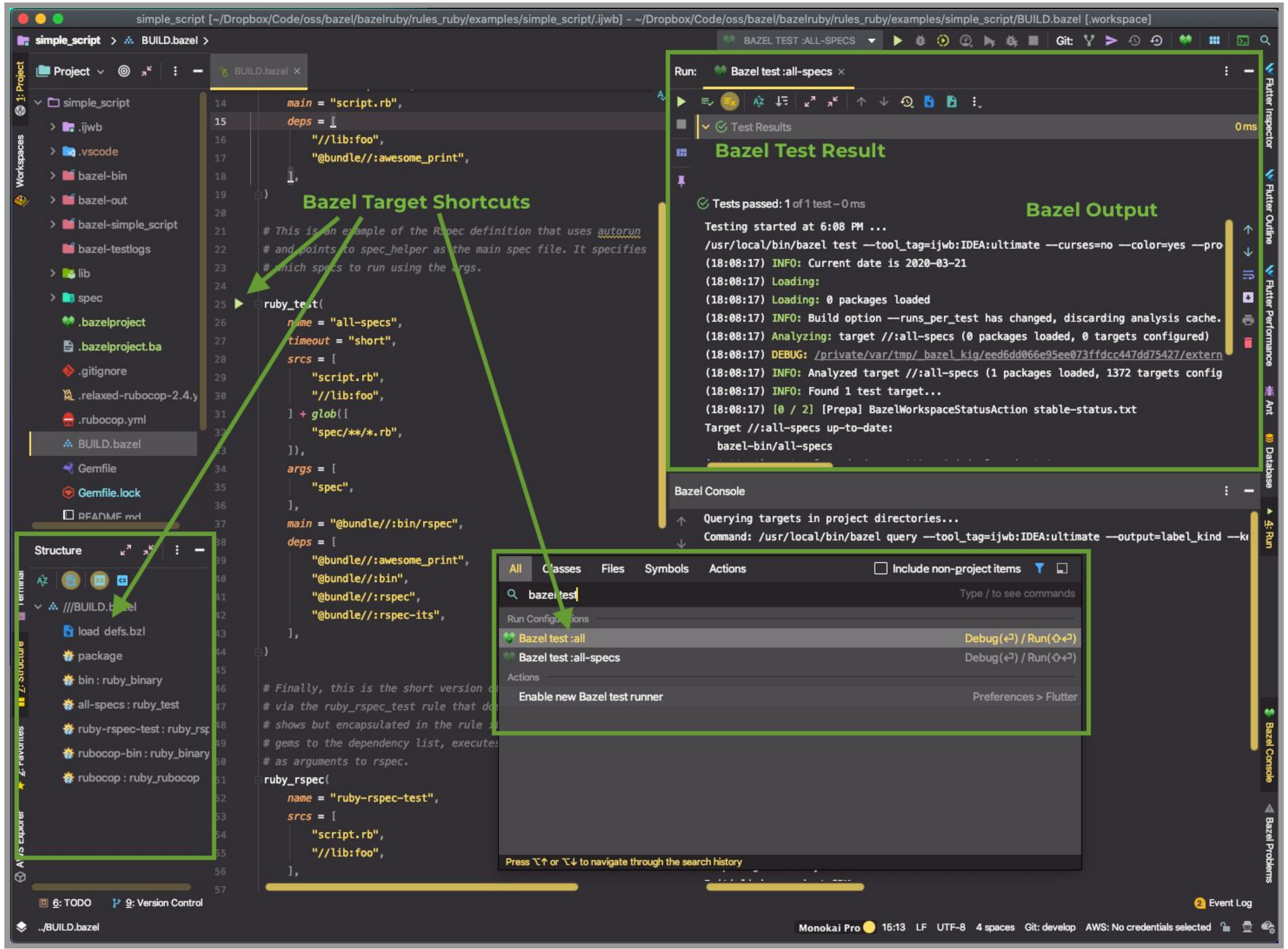
IDE Integration: VSCode



```
load(
    "@b
    "ru
    "ru
    "ru
    "ru
)

package(default_visibility = ["//:__subpacka

ruby_binary(
    name = "bin",
    srcs = ["script.rb"],
    main = "script.rb",
    deps = [
        "//lib:foo",
        "@bundle//:awesome_print",
    ],
)

# This is an example of the RSpec definition
# and points to spec_helper as the main spec
# which specs to run using the args.

ruby_test(
    name = "all-specs",
    timeout = "short",
    srcs = [
        "script.rb",
        "//lib:foo",
    ] + glob([
        "spec/**/*.rb",
    ]),
    args = [
        "spec",
    ],
    main = "@bundle//:bin/rspec",
    deps = [
        "@bundle//:awesome_print",
        "@bundle//:bin",
        "@bundle//:rspec",
        "@bundle//:rspec-its",
    ],
)
```

```
> Executing task: /usr/local/bin/bazel 'clean' <

ommand (pid=58692) is running.  Waiting for it to complete on the server...
5) INFO: Starting clean.

will be reused by tasks, press any key to close it.

ing task: /usr/local/bin/bazel 'build' '//lib:all' <

0) INFO: Current date is 2020-03-21
0) 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 configur
ed).
(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


simple_script [~/Dropbox/Code/oss/bazel/bazelruby/rules_ruby/examples/simple_script/.ijwb] - ~/Dropbox/Code/oss/bazel/bazelruby/rules_ruby/examples/simple_script/BUILD.bazel [.workspace]
simple_script > BUILD.bazel >
BAZEL TEST :ALL-SPECS
Project
simple_script
.ijwb
.vscode
bazel-bin
bazel-out
bazel-simple_script
bazel-testlogs
lib
spec
.bazelproject
.bazelproject.ba
.gitignore
.relaxed-rubocop-2.4.y
.rubocop.yml
BUILD.bazel
Gemfile
Gemfile.lock
README.md
Structure
//BUILD.bazel
load defs.bzl
package
bin : ruby_binary
all-specs : ruby_test
ruby-rspec-test : ruby_rsp
rubocop-bin : ruby_binary
rubocop : ruby_rubocop
Bazel Target Shortcuts
Run: Bazel test :all-specs
Test Results 0 ms
Bazel Test Result
Bazel Output
Bazel Console
All Classes Files Symbols Actions
Include non-project items
bazel test
Type / to see commands
Run Configurations
Bazel test :all Debug(⏎) / Run(⇧⏎)
Bazel test :all-specs Debug(⏎) / Run(⇧⏎)
Actions
Enable new Bazel test runner Preferences > Flutter
Press ⌥↑ or ⌥↓ to navigate through the search history
6: TODO
9: Version Control
Event Log
../BUILD.bazel
Monokai Pro 15:13 LF UTF-8 4 spaces Git: develop AWS: No credentials selected

```
    main = "script.rb",
    deps = [
        "//lib:foo",
        "@bundle//:awesome_print",
    ],
)

# This is an example of the RSpec definition that uses autorun
# and points to spec_helper as the main spec file. It specifies
# which specs to run using the args.

ruby_test(
    name = "all-specs",
    timeout = "short",
    srcs = [
        "script.rb",
        "//lib:foo",
    ] + glob([
        "spec/**/*.rb",
    ]),
    args = [
        "spec",
    ],
    main = "@bundle//:bin/rspec",
    deps = [
        "@bundle//:awesome_print",
        "@bundle//:bin",
        "@bundle//:rspec",
        "@bundle//:rspec-its",
    ],
)

# Finally, this is the short version o
# via the ruby_rspec_test rule that do
# shows but encapsulated in the rule i
# gems to the dependency list, executes
# as arguments to rspec.
ruby_rspec(
    name = "ruby-rspec-test",
    srcs = [
        "script.rb",
        "//lib:foo",
    ],
```

Tests passed: 1 of 1 test – 0 ms

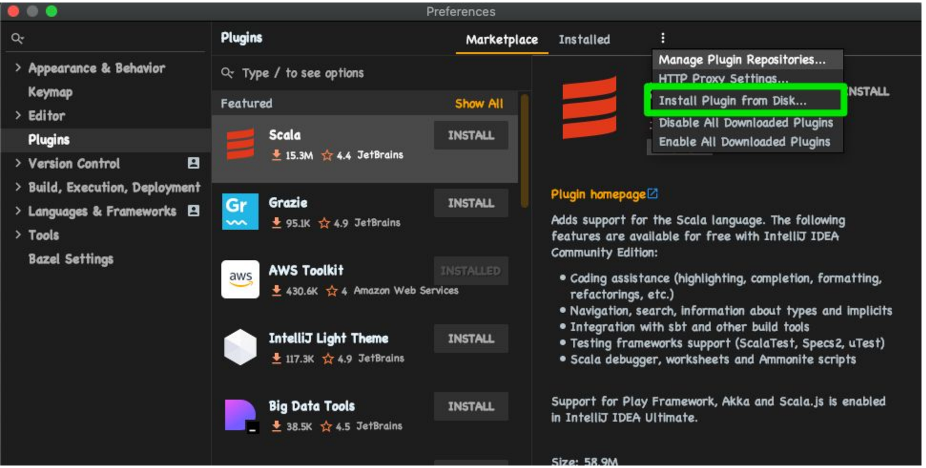
```
Testing started at 6:08 PM ...
/usr/local/bin/bazel test --tool_tag=ijwb:IDEA:ultimate --curses=no --color=yes --pro
(18:08:17) INFO: Current date is 2020-03-21
(18:08:17) Loading:
(18:08:17) Loading: 0 packages loaded
(18:08:17) INFO: Build option --runs_per_test has changed, discarding analysis cache.
(18:08:17) Analyzing: target //:all-specs (0 packages loaded, 0 targets configured)
(18:08:17) DEBUG: /private/var/tmp/_bazel_kig/eed6dd066e95ee073ffdcc447dd75427/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] [Prepa] BazelWorkspaceStatusAction stable-status.txt
Target //:all-specs up-to-date:
  bazel-bin/all-specs
```

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

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



Adds support for the Scala language. The following features are available for free with IntelliJ IDEA Community Edition:

- Coding assistance (highlighting, completion, formatting, refactorings, etc.)
- Navigation, search, information about types and implicits
- Integration with sbt and other build tools
- Testing frameworks support (ScalaTest, Specs2, uTest)
- Scala debugger, worksheets and Ammonite scripts

Support for Play Framework, Akka and Scala.js is enabled in IntelliJ IDEA Ultimate.

Size: 58.9M

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

flare.build

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