O'REILLY®

Getting Started with Bazel

About the trainer

bmuschko

bmuschko

bmuschko.com

AUTOMATED ASCENT

automatedascent.com

Introduction to Bazel

Core Concepts, Project Structure and Lifecycle,
Using the Bazel Command Line

What is Bazel?

Open-source build automation tool

https://bazel.build/

- Evolved from Google-internal tool named Blaze
- Opinionated about code organization and modeling
- Main focus on monorepos, standardization, and fast execution speeds
- Depending on automation requirements, Bazel may be a good fit

Why Should I Use It?

Functional and non-functional features

- Declarative language
  - Build logic uses higher-level language called Starlark
  - Hides implementation complexities like compilation/linking
  - Runtime behavior can be fine-tuned

- Reproducibility
  - Sandboxed build execution by enforcing the definition of all of its dependencies
  - Includes execution environment

Why Should I Use It?

Functional and non-functional features

- Scalability
    - Focus is on projects with large codebases in monorepos
    - Fine-grained definition of modules (called *packages*)

- Parallel and distributed execution
    - Can execute its work in parallel (on a single machine)
    - Can execute its work in a distributed fashion (on multiple machines)

Why Should I Use It?

Functional and non-functional features

- Building polyglot projects
  - Support for different languages (e.g. Java, Go, Python, …)
  - Embraces modern toolchains and frameworks (Docker, Kubernetes, gRPC, ...)

- Extensibility
  - Fosters abstraction of build logic with *macros*
  - Reusability of build logic with *rules* for a wider audience

Runtime Installation Options

All major operating system are supported

- Installation instructions for Ubuntu Linux, MacOS, and Windows
- MacOSX distribution requires installation of XCode
- Windows distribution requires installation of Visual C++ Redistributable
- Container image available for Continuous Integration purposes

User-Friendly Launcher

Automatic installation of Bazel runtime

- Install the binary Bazelisk which is used for triggering the build
- The `.bazelversion` file defines the compatible Bazel version and is meant to be checked into version control
- Upon runtime, the tool downloads and installs the Bazel runtime and executes the build with it

Project Building Blocks

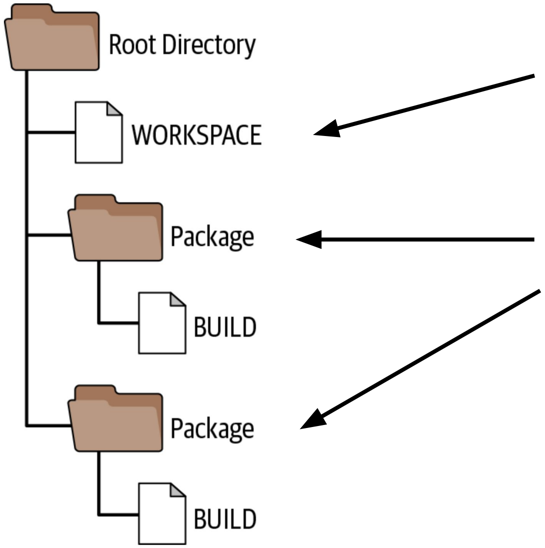
Two core concepts represented in every Bazel project

- **Workspace**
  - Represented by a `WORKSPACE` file in project root directory
  - Designates the directories containing source code
  - Defines external dependencies for specific language support

- **Package**
  - Represents a module containing software functionality that belongs together with specific visibility to other packages
  - Defined in a `BUILD` file located at the package directory-level

Project Structure

# of packages depends on functional code organization



Bazel enforces existence of a single workspace file

A Bazel package can be more fine-grained than a Java package

Build Logic Concepts

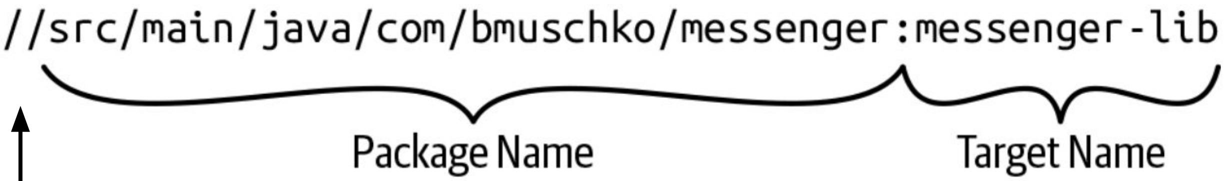
Important for applying reusable logic and executing it

- **Rule**
  - Defines executable logic, so-called actions
  - Usually requires definition of inputs and outputs
- **Target**
  - A package can contain a set of targets
  - Targets represent a file or a rule
  - Invoked from the command line

Executing a Target From the CLI

A label combines the package name and target name

```
$ bazel <command> <options> ...
```



```
//src/main/java/com/bmuschko/messenger:messenger-lib
```

// refers to root directory

Commonly-Used Commands

The daily bread and butter of developers

- `query`: Prints the dependency graph of a label.
- `build`: Builds the provided label. A target implements a “unit of work” like compiling source code, assembling artifacts etc.
- `test`: Executes the tests for a provided label. Builds the “code under test” and the test source code so it can be made available at test runtime.

Build Everything!

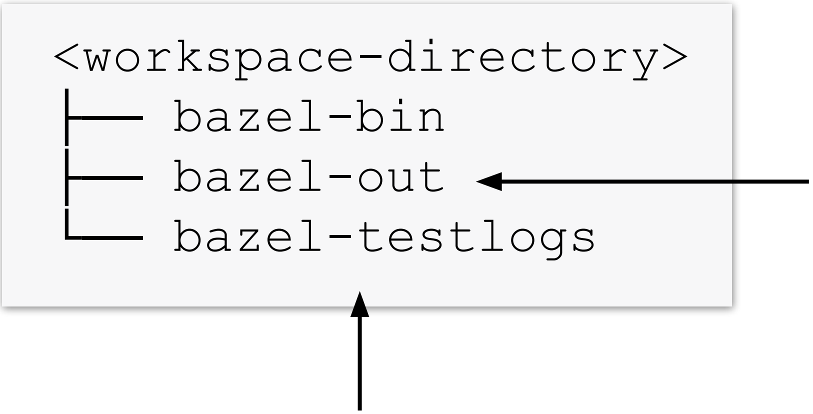
Compile/assemble/test for the whole project

- `bazel build //...`
- `bazel test //...`

Select all packages recursively
from the root directory

Output and Cache Directories

Not to be checked into version control!



```
<workspace-directory>
├── bazel-bin
├── bazel-out
└── bazel-testlogs
```

Built binary artifacts & artifacts retrieved from external locations

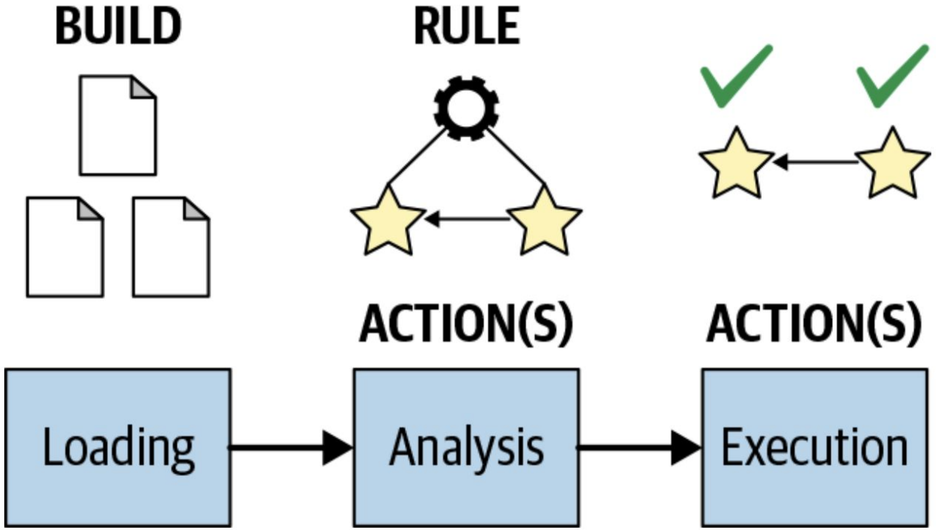
Test log files created by test runner

EXERCISE

“Hello World” in
Bazel

Lifecycle of a Bazel Build

Build executes as part of a three-part, phased process



Lifecycle Phases

Failure at any phase will stop build

- **Loading Phase:** Fast syntactic check of build logic.
- **Analysis phase:** Evaluation of build configuration and construction of the build execution graph.
- **Execution phase:** Runs the actions and distributes workload if configured.

Bazel Configuration File

Set common build configuration in a single location

- Stored in the file `.bazelrc` located in workspace directory and/or user home directory

- Bazel command + CLI option(s), grouping by using `--config`

```
build --show_timestamps
build:memcheck --strip=never --test_timeout=3600
```

Programming Language Rules

Common functionality + language-specific functionality

- **<language>_binary:** Builds an executable artifact.
- **<language>_library:** Builds an artifact containing reusable functionality.
- **<language>_test:** Executes tests for one or many packages.

Language-specific tutorials

Basic Automation for a Java Project

Exploring Java rules, Project Structure, Source Code Compilation, JAR assembly, IDE Support

Typical Java Rules

Common developer-focused rules for Java projects

- **java_binary:** Builds an executable JAR file.
- **java_library:** Builds a JAR file containing reusable functionality.
- **java_test:** Executes tests for one or many packages.

Reference of all Java rules

Simple Java Application Project

Single package project with a main class

```
.
├── BUILD
├── WORKSPACE
└── src
    └── main
        └── java
            └── com
                └── bmuschko
                    ├── HelloWorld.java
                    └── messenger
                        └── Messenger.java
```

Main class that prints a message to standard output

Instantiated by main class and method called from there

Modeling the Binary Package

Application is represented by a binary

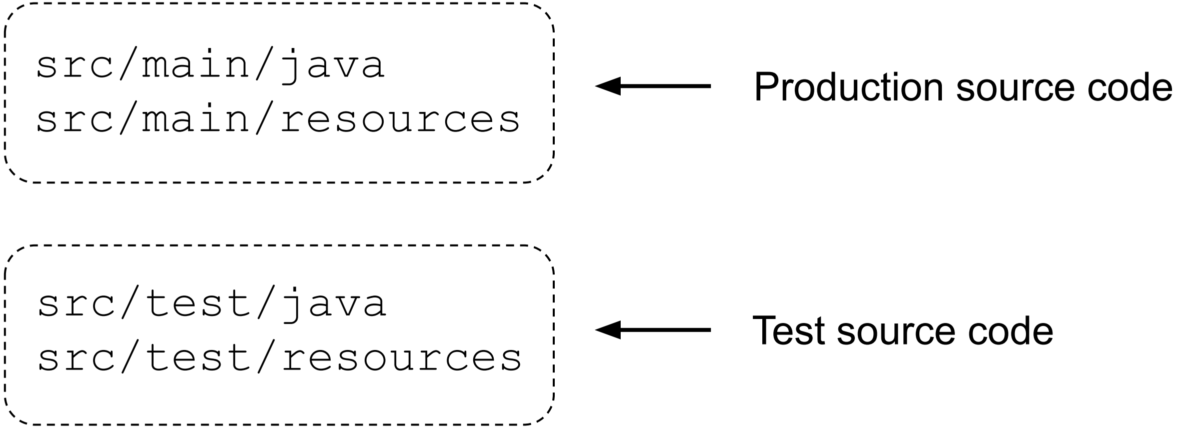
BUILD

```
java_binary(
    name = "hello-world",
    srcs = glob(["src/main/java/com/bmuschko/**/*.java"]),
    main_class = "com.bmuschko.HelloWorld",
)
```

Implemented by the java_binary rule

Standard Industry Conventions

*Does **not** prescribe standard directories like Maven/Gradle*



```
src/main/java
src/main/resources
```

Production source code

```
src/test/java
src/test/resources
```

Test source code

Modeling the Workspace

The application doesn’t define dependencies

WORKSPACE

```
<empty>
```

Java rules are built into Bazel runtime and therefore don’t need to be declared as dependency

Building hello-world Package

Package has been defined on the root-level of project

```
$ bazel build //:hello-world
...
INFO: Found 1 target...
Target //:hello-world up-to-date:
  bazel-bin/hello-world.jar
  bazel-bin/hello-world
INFO: Elapsed time: 23.491s, Critical Path: 4.23s
INFO: 3 processes: 2 darwin-sandbox, 1 worker.
INFO: Build completed successfully, 7 total actions
```

Produced artifact

The command `build` *builds* the specified target

Contents of Binary

Contains class files organized by package + manifest

```
$ jar tf bazel-bin/hello-world.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/bmuschko/
com/bmuschko/HelloWorld.class
com/bmuschko/messenger/
com/bmuschko/messenger/Messenger.class
```

Running the Application

Builds the artifact if it hasn't been built yet

```
$ bazel run //:hello-world
...
INFO: Found 1 target...
Target //:hello-world up-to-date:
  bazel-bin/hello-world.jar
  bazel-bin/hello-world
INFO: Elapsed time: 0.092s, Critical Path: 0.00s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
Hello World!
```

Standard output from main class

EXERCISE

Building an
Executable
Program in Java

Driving the Build from the IDE

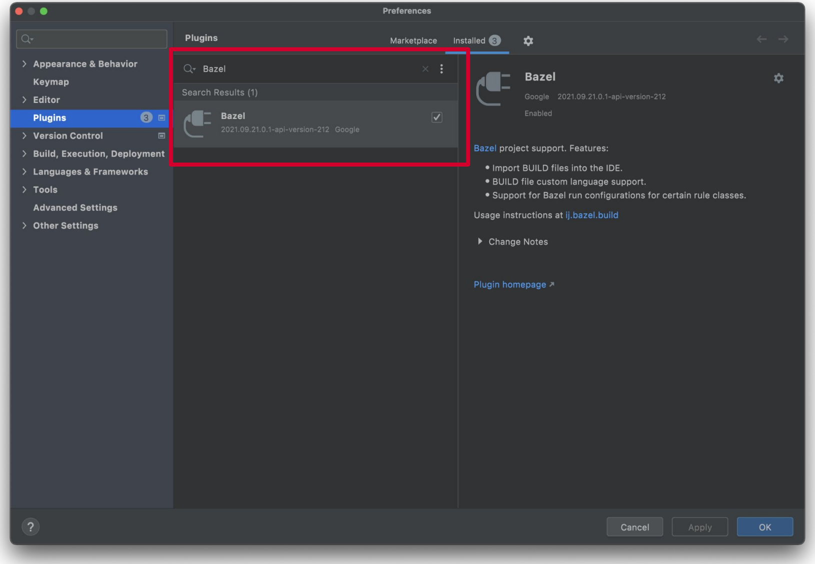
Auto-completion, syntax highlighting, running targets

IntelliJ plugin

VSCode plugin

Installing the IntelliJ Plugin

IntelliJ IDEA > Preferences... > Plugins



Bazel

Search Results (1)

Bazel ☑
2021.09.21.0.1-api-version-212 Google

Bazel
Google 2021.09.21.0.1-api-version-212
Enabled

Bazel project support. Features:

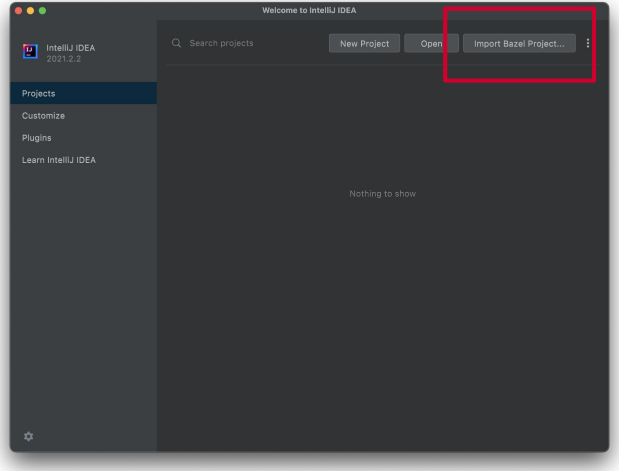
- Import BUILD files into the IDE.
- BUILD file custom language support.
- Support for Bazel run configurations for certain rule classes.

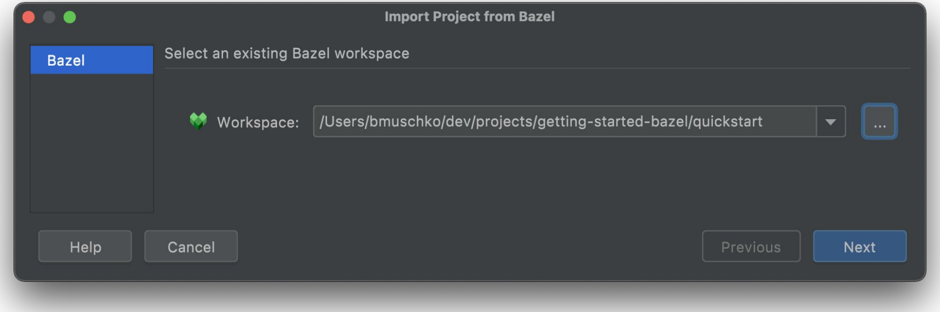
Usage instructions at ij.bazel.build

Search for “Bazel”
and install the plugin

Opening a Project

File > Open… > Import Bazel Project...



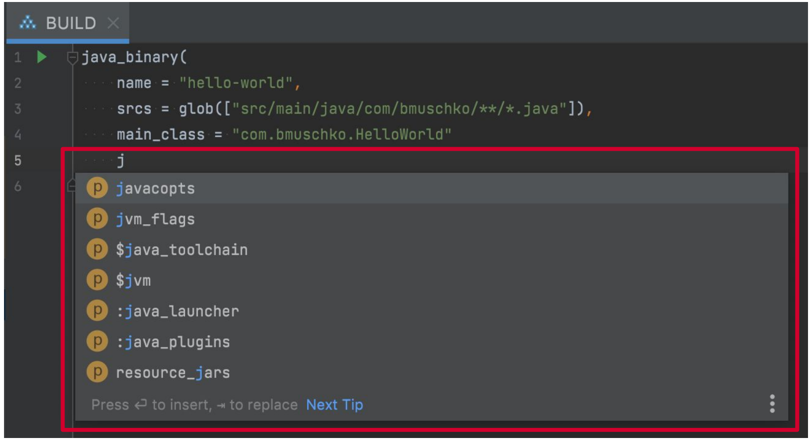


Select an existing Bazel workspace

Workspace: /Users/bmuschko/dev/projects/getting-started-bazel/quickstart

Auto-Completion in Bazel Files

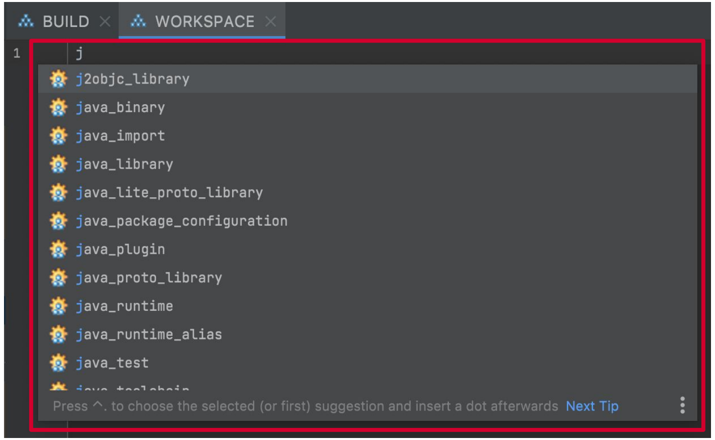
Within the Bazel file start typing



```
java_binary(
    name = "hello-world",
    srcs = glob(["src/main/java/com/bmuschko/**/*.java"]),
    main_class = "com.bmuschko.HelloWorld"
    j
```

- javacopts
- jvm_flags
- $java_toolchain
- $jvm
- :java_launcher
- :java_plugins
- resource_jars

Press ⏎ to insert, ⇥ to replace Next Tip



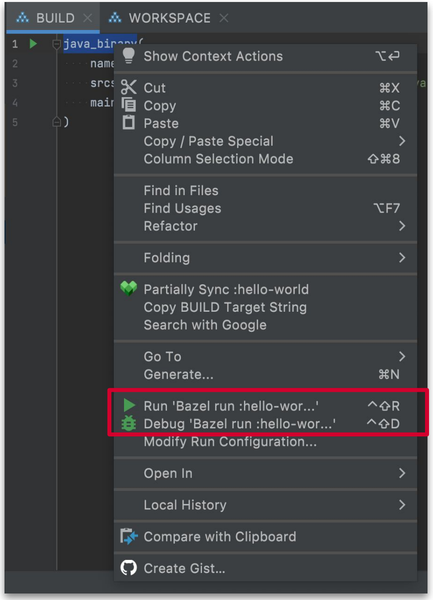
```
j
```

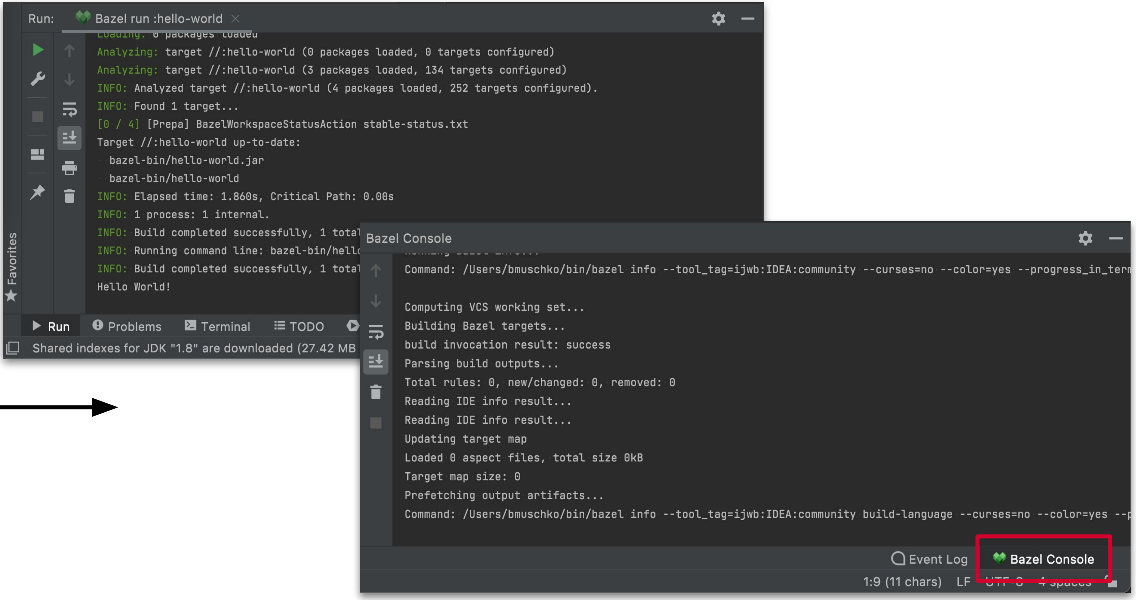
- j2objc_library
- java_binary
- java_import
- java_library
- java_lite_proto_library
- java_package_configuration
- java_plugin
- java_proto_library
- java_runtime
- java_runtime_alias
- java_test
- java_toolchain

Press ^. to choose the selected (or first) suggestion and insert a dot afterwards Next Tip

Executing Targets

Context menu on rule in BUILD file





```
Loading: 0 packages loaded
Analyzing: target //:hello-world (0 packages loaded, 0 targets configured)
Analyzing: target //:hello-world (3 packages loaded, 134 targets configured)
INFO: Analyzed target //:hello-world (4 packages loaded, 252 targets configured).
INFO: Found 1 target...
[0 / 4] [Prepa] BazelWorkspaceStatusAction stable-status.txt
Target //:hello-world up-to-date:
  bazel-bin/hello-world.jar
  bazel-bin/hello-world
INFO: Elapsed time: 1.860s, Critical Path: 0.00s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total
INFO: Running command line: bazel-bin/hello
INFO: Build completed successfully, 1 total
Hello World!
```

```
Command: /Users/bmuschko/bin/bazel info --tool_tag=ijwb:IDEA:community --curses=no --color=yes --progress_in_term

Computing VCS working set...
Building Bazel targets...
build invocation result: success
Parsing build outputs...
Total rules: 0, new/changed: 0, removed: 0
Reading IDE info result...
Reading IDE info result...
Updating target map
Loaded 0 aspect files, total size 0kB
Target map size: 0
Prefetching output artifacts...
Command: /Users/bmuschko/bin/bazel info --tool_tag=ijwb:IDEA:community build-language --curses=no --color=yes --p
```

EXERCISE

Installing and Using
Bazel IDE Support

Q & A

5 mins

BREAK

5 mins

Dependency Management and Automated Testing

Definition and Resolution of Dependencies, Writing and Executing Tests

Types of Dependencies

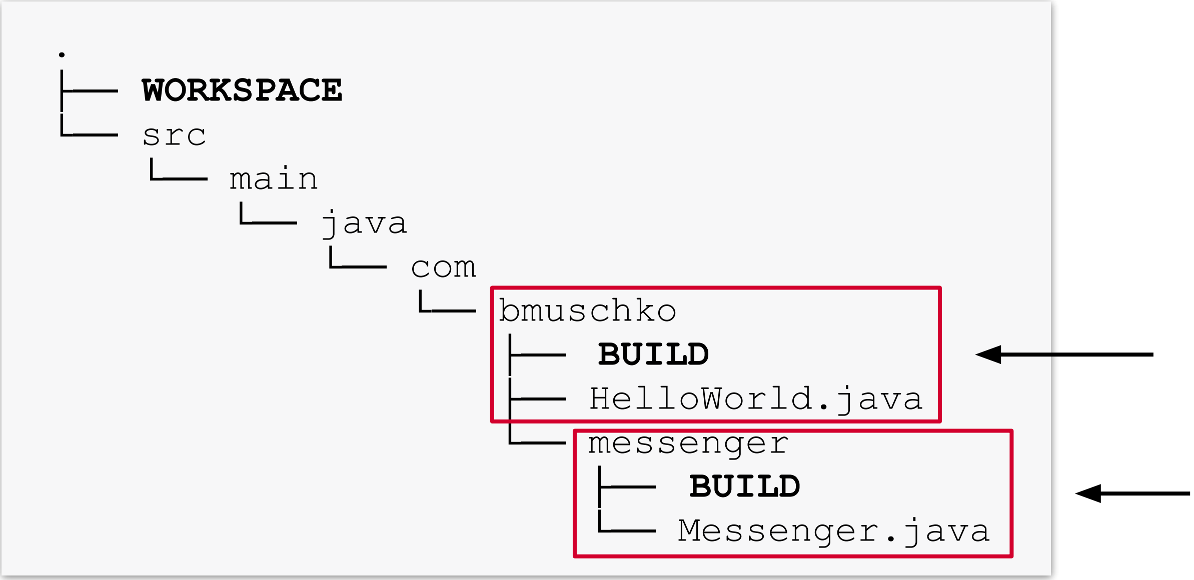
Two different use cases that can be combined

- **Package Dependencies:** One package depends on the produced output of another package e.g. the class files. Allows for more fine-grained definition of traditional functional modules.

- **External Dependencies:** Source code that lives in a package needs the API and/or implementation of an external library in the form of a Git repository, an archive accessible via HTTP or file in the local file system.

Multi-Package Project

Allows for fine-grained definition with dependencies



```
.
├── WORKSPACE
└── src
    └── main
        └── java
            └── com
                └── bmuschko
                    ├── BUILD
                    ├── HelloWorld.java
                    └── messenger
                        ├── BUILD
                        └── Messenger.java
```

Package A containing the main class

Package B containing classes used by package 1

Modeling the Library Package

Library that bundles class files represented as JAR file

BUILD

```
java_library(
     name = "messenger-lib",
     srcs = ["Messenger.java"]
)
```

Implemented by the java_library rule

Package Dependencies

Compile-time dependency on the messenger-lib target

BUILD

```
java_binary(
    name = "hello-world",
    srcs = ["HelloWorld.java"],
    main_class = "com.bmuschko.HelloWorld",
    deps = ["//src/main/java/com/bmuschko/messenger:messenger-lib"],
)
```

Failing to Resolve Dependency

By default, packages are isolated

```
$ bazel build //src/main/java/com/bmuschko:hello-world
...
ERROR: .../src/main/java/com/bmuschko/BUILD:1:12: in java_binary rule
//src/main/java/com/bmuschko:hello-world: target
'//src/main/java/com/bmuschko/messenger:messenger-lib' is not visible
from target '//src/main/java/com/bmuschko:hello-world'. Check the
visibility declaration of the former target if you think the dependency
is legitimate
ERROR: Analysis of target '//src/main/java/com/bmuschko:hello-world'
failed; build aborted: Analysis of target
'//src/main/java/com/bmuschko:hello-world' failed
```

Visibility of Targets

Targets of package cannot be used by other packages

BUILD

```
java_library(
    name = "messenger-lib",
    srcs = ["Messenger.java"],
    visibility = ["//src/main/java/com/bmuschko:__pkg__"]
)
```

Make “all rules in the package” available to assigned package

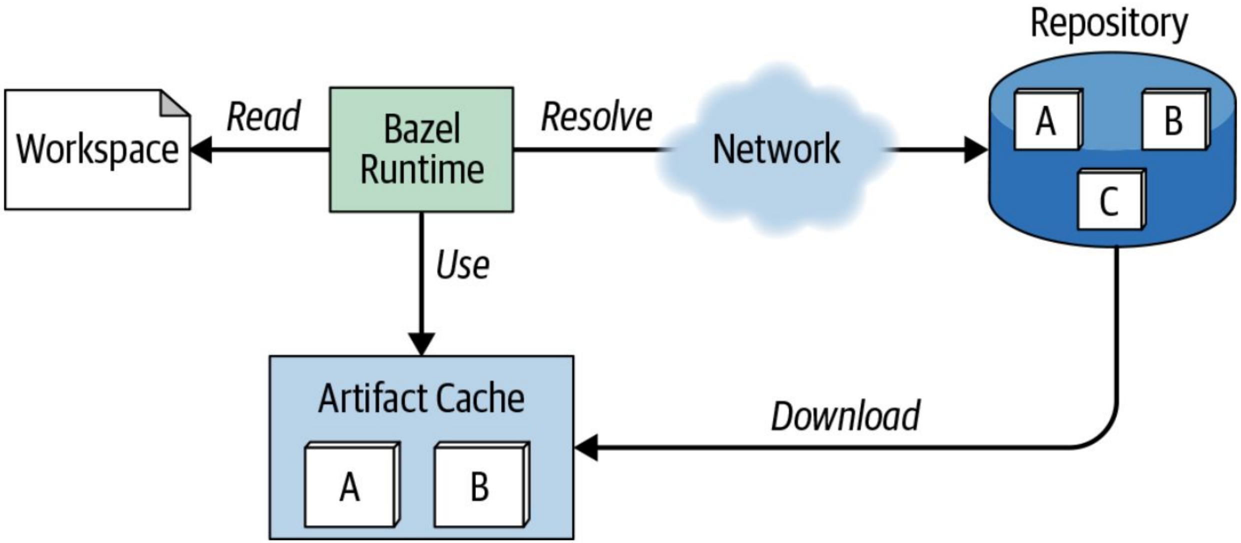
Building the Dependent Targets

Resolves declared dependencies and uses them

```
$ bazel build //src/main/java/com/bmuschko:hello-world
...
INFO: Found 1 target...
Target //src/main/java/com/bmuschko:hello-world
up-to-date:
  bazel-bin/src/main/java/com/bmuschko/hello-world.jar
  bazel-bin/src/main/java/com/bmuschko/hello-world
INFO: Elapsed time: 19.924s, Critical Path: 4.55s
INFO: 9 processes: 4 internal, 3 darwin-sandbox, 2 worker.
INFO: Build completed successfully, 9 total actions
```

External Library Dependencies

Artifacts live in repository and are downloaded to cache



Rules for JVM Dependencies

Functionality exists as rules on a GitHub repository

- Download rules archive with a specific tag and commit hash via HTTP. Load rules for usage.

- Define repositories used to resolve dependencies.

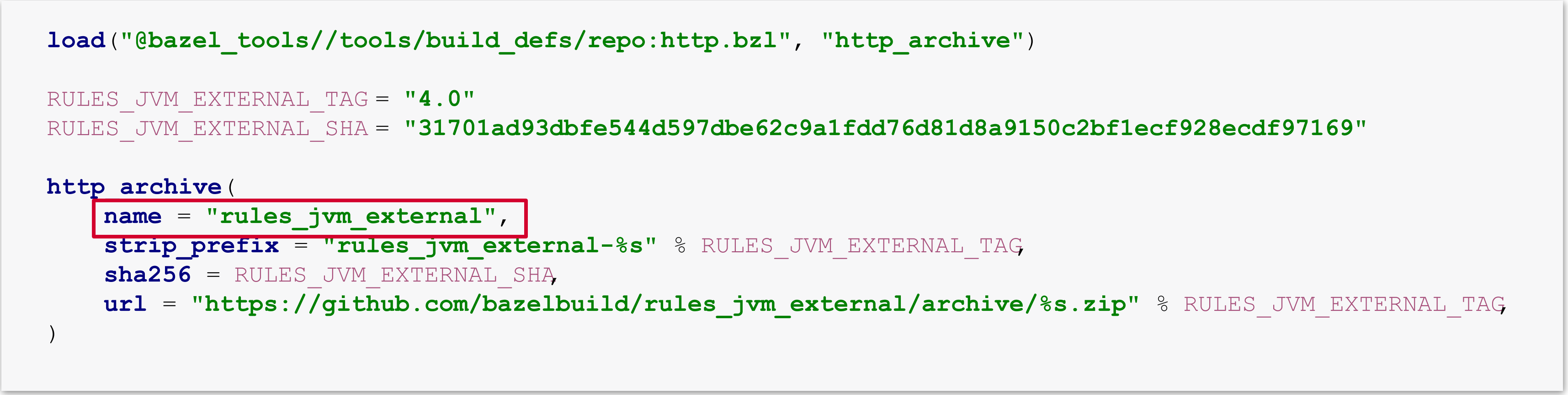
- Define dependencies with group, artifact ID, and version (GAV).

Implemented by rules_jvm_external

Declaring JVM Rules External

Load rules for consumption as HTTP archive

WORKSPACE



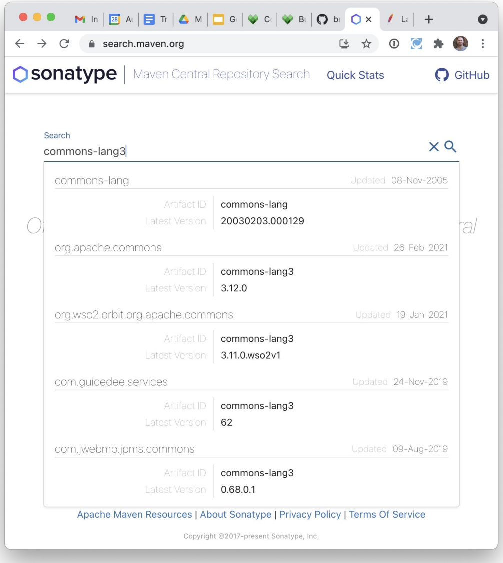
```
load("@bazel_tools//tools/build_defs/repo:http.bzl", "http_archive")

RULES_JVM_EXTERNAL_TAG = "4.0"
RULES_JVM_EXTERNAL_SHA = "31701ad93dbfe544d597dbe62c9a1fdd76d81d8a9150c2bf1ecf928ecdf97169"

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

Example JVM Dependency

Apache Commons Lang 3 - find via search.maven.org



search.maven.org

sonatype | Maven Central Repository Search   Quick Stats   GitHub

Search
commons-lang3

commons-lang   Updated 08-Nov-2005
Artifact ID commons-lang
Latest Version 20030203.000129

org.apache.commons   Updated 26-Feb-2021
Artifact ID commons-lang3
Latest Version 3.12.0

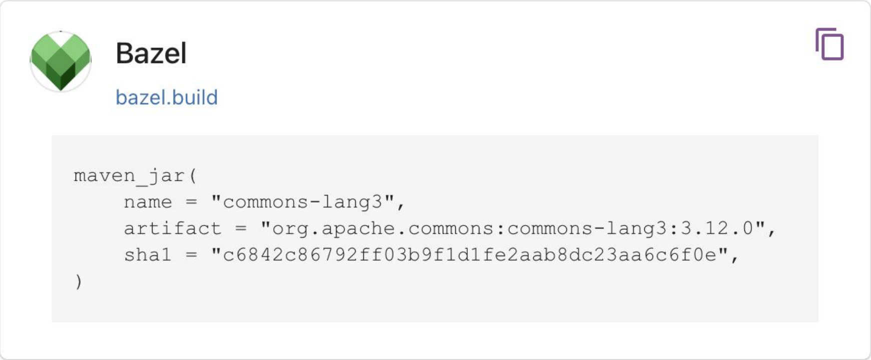
org.wso2.orbit.org.apache.commons   Updated 19-Jan-2021
Artifact ID commons-lang3
Latest Version 3.11.0.wso2v1

com.guicedee.services   Updated 24-Nov-2019
Artifact ID commons-lang3
Latest Version 62

com.jwebmp.jpms.commons   Updated 09-Aug-2019
Artifact ID commons-lang3
Latest Version 0.68.0.1

Apache Maven Resources | About Sonatype | Privacy Policy | Terms Of Service

Copyright ©2017-present Sonatype, Inc.



Bazel

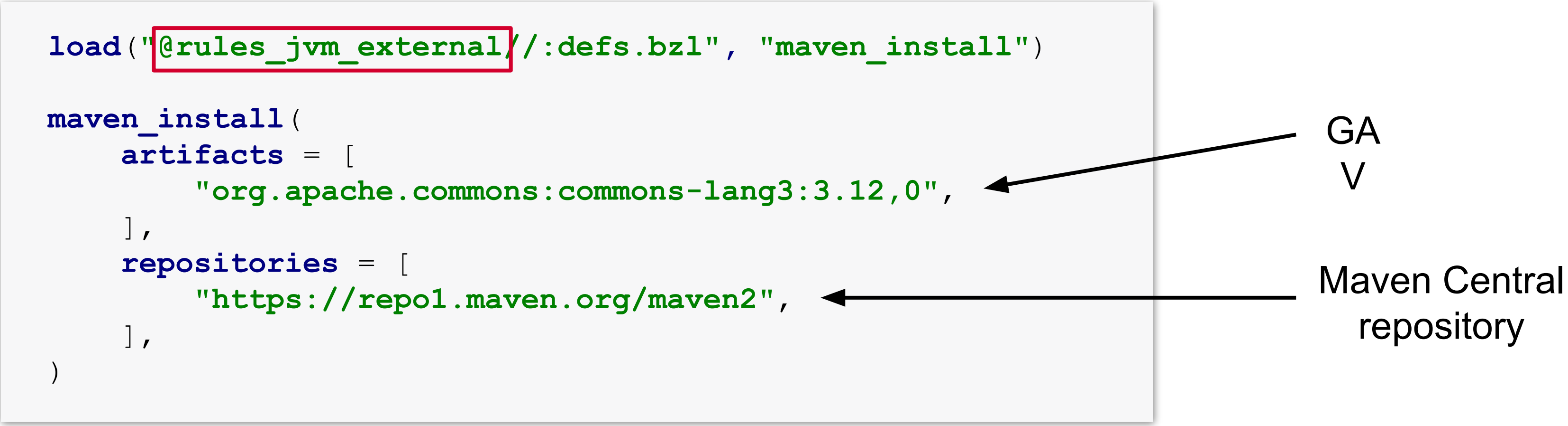
bazel.build

```
maven_jar(
    name = "commons-lang3",
    artifact = "org.apache.commons:commons-lang3:3.12.0",
    sha1 = "c6842c86792ff03b9f1d1fe2aab8dc23aa6c6f0e",
)
```

Definition of Dependency

Explicit declaration of GAVs and repositories

WORKSPACE



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

maven_install(
    artifacts = [
        "org.apache.commons:commons-lang3:3.12,0",
    ],
    repositories = [
        "https://repo1.maven.org/maven2",
    ],
)
```

GA
V

Maven Central
repository

Consuming a Maven Dependency

Dependencies can be scoped for compile or runtime

BUILD

```
java_library(
     name = "messenger-lib",
     srcs = ["Messenger.java"],
     visibility = ["//src/main/java/com/bmuschko:__pkg__"],
     deps = [
          "@maven//:org_apache_commons_commons_lang3"
     ]
)
```

Only the group and artifact IDs are required

Substitute non-alphanumeric characters with underscores

Import of External Library Class

Made available on the compilation classpath

Messenger.java

```java
package com.bmuschko.messenger;

import org.apache.commons.lang3.StringUtils;

public class Messenger {
    public String getMessage() {
        return StringUtils.upperCase("Hello World!");
    }
}
```

Import of class

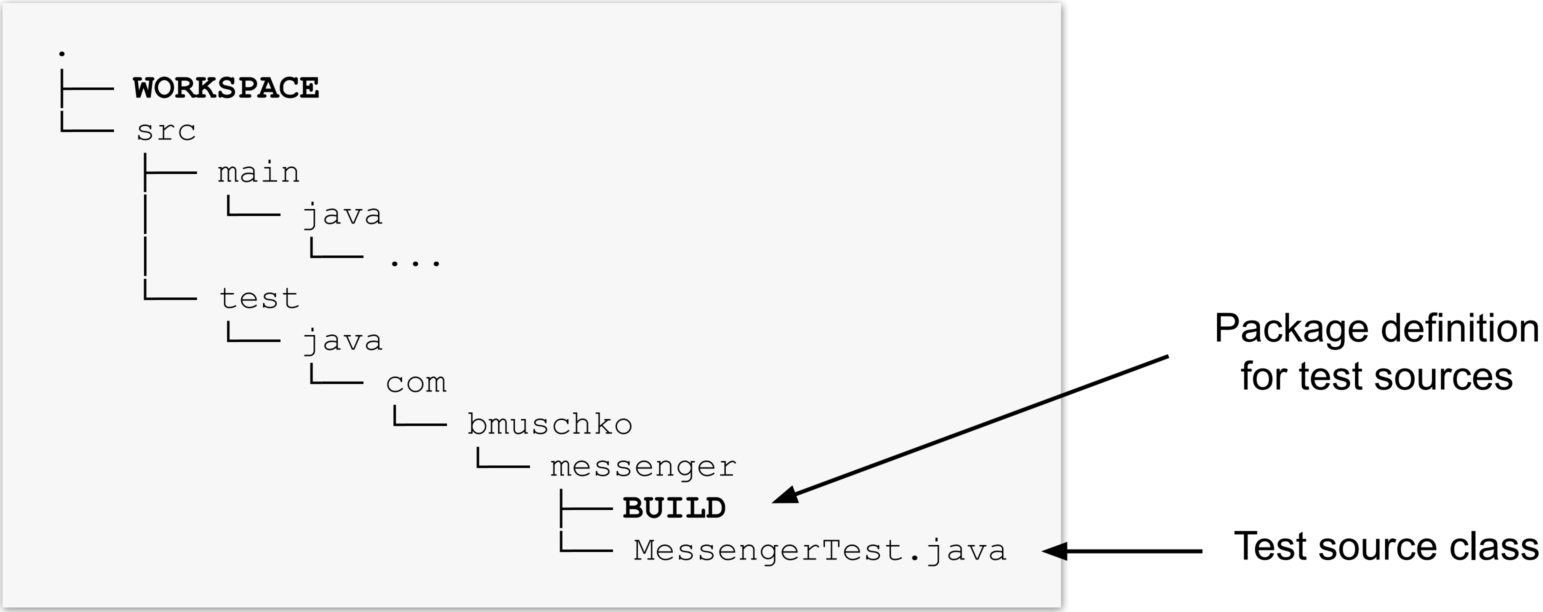
Usage of class

EXERCISE

Declaring a Dependency on a Package and an External Library

Separating Test Source Code

Distinguish different types of tests



```
.
├── WORKSPACE
└── src
    ├── main
    │   └── java
    │       └── ...
    └── test
        └── java
            └── com
                └── bmuschko
                    └── messenger
                        ├── BUILD
                        └── MessengerTest.java
```

Package definition for test sources

Test source class

Declaring Test Dependencies

Needs “code under test” and test framework libraries

BUILD

```
java_test(
    name = "messenger-test",
    srcs = ["MessengerTest.java"],
    test_class = "com.bmuschko.messenger.MessengerTest",
    deps = [
        "//src/main/java/com/bmuschko/messenger:messenger-lib",
        "@maven//:junit_junit"
    ]
)
```

Implemented by java_test rule

Executing Tests

Renders executed tests and their outcome on console

```
$ bazel test //src/test/java/com/bmuschko/messenger:messenger-test
...
INFO: Found 1 test target...
Target //src/test/java/com/bmuschko/messenger:messenger-test up-to-date:
  bazel-bin/src/test/java/com/bmuschko/messenger/messenger-test.jar
  bazel-bin/src/test/java/com/bmuschko/messenger/messenger-test
...
//src/test/java/com/bmuschko/messenger:messenger-test        PASSED in 0.5s

Executed 1 out of 1 test: 1 test passes.
```

Test Reporting

Java rules do not produce a human-readable report

```
bazel-testlogs
└── src
    └── test
        └── java
            └── com
                └── bmuschko
                    └── messenger
                        └── messenger-test
                            ├── ...
                            ├── test.log
                            └── test.xml
```

XML test results can be used for further processing

EXERCISE

Declaring the JUnit Dependency and Executing Tests

Publishing a Java Library

Sharing JAR for consumption from a binary repository

BUILD

```
load("@rules_jvm_external//:defs.bzl", "java_export")

java_export(
    name = "messenger-exported-lib",
    maven_coordinates = "com.bmuschko:messenger:1.0.0",
    pom_template = "pom.tmpl",
    srcs = glob(["*.java"]),
    deps = [
        "@maven//:org_apache_commons_commons_lang3"
    ]
)
```

GAV for library in Maven repository

POM information is not derived automatically

Q & A

5 mins

Outlook on Advanced Topics

An Introduction to In-Depth Features and Scenarios

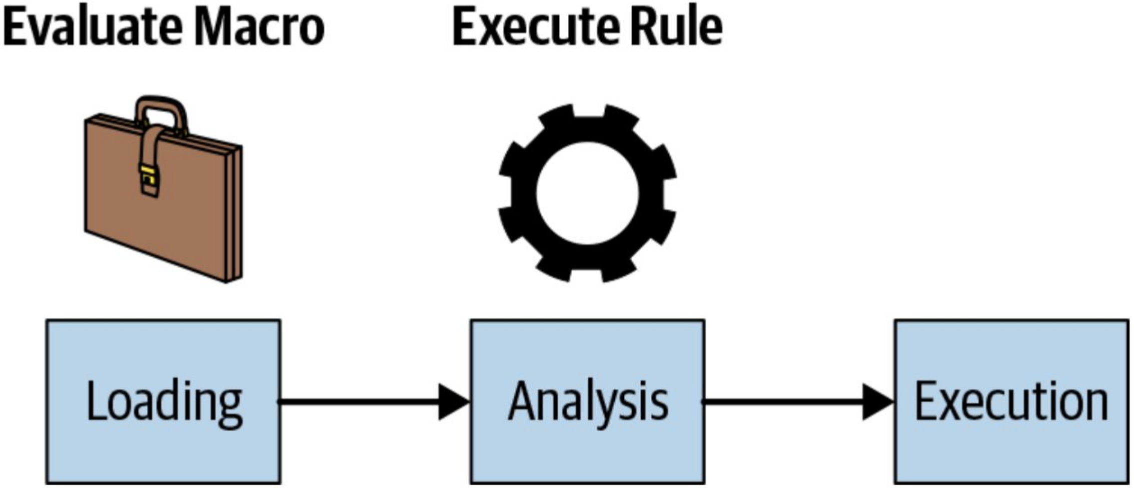
Extension Mechanisms

You can enhance the built-in Bazel capabilities

- **Rule:** Full control over Bazel’s internals, can configure other rules, and introduces elaborate features that are complex in nature.
- **Macros:** A way to better organize build logic within the same project e.g. call a rule with parameters you want to set by default.

When Are They Executed?

Invoked during a specific phase of the Bazel lifecycle



Evaluate Macro

Execute Rule

Loading

Analysis

Execution

Starlark Build Language

Implementing build scripts and extensions

- Dialect of Python 3 with restrictions e.g. access to filesystem.
- Achieve optimal build execution performance by supporting parallel and remote execution and to allow multithreaded processing of build logic.

More information on Starlark

Writing and Using a Macro

Lives in a file with a .bzl extension

```
.
├── WORKSPACE
├── macros
│   ├── BUILD
│   └── mymacro.bzl
└── src
    └── ...
```

BUILD

```
load("//macros:mymacro.bzl", "mymacro")

mymacro(
        ...
)
```

Call macro and configure it

Load macro

Macro Example

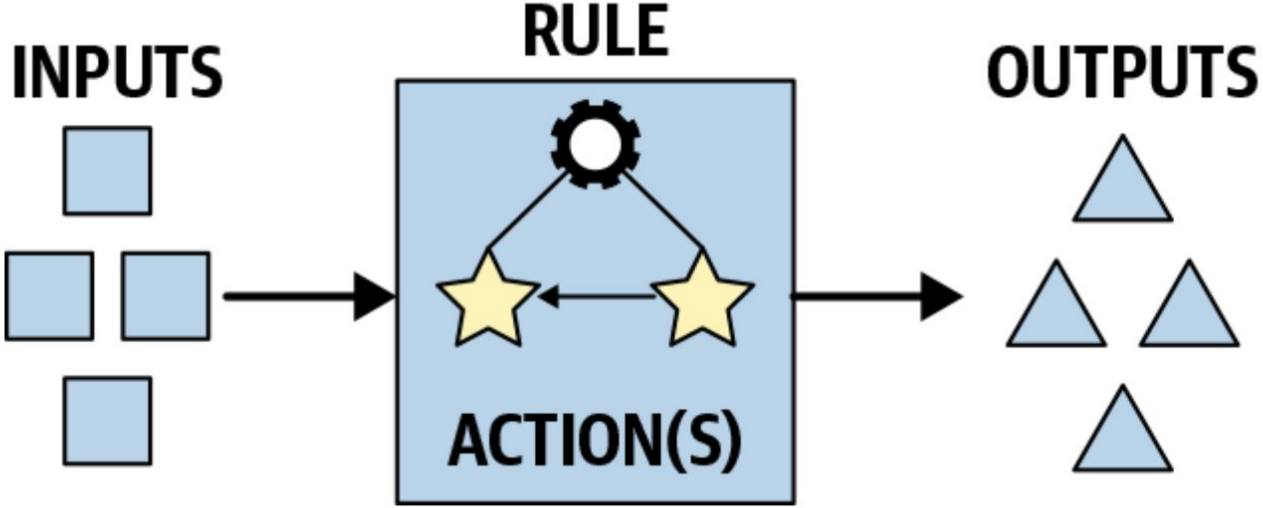
Configuring JUnit 5 to run tests

- Pre-configures the rule java_test
  - Defines JUnit Jupiter dependencies
  - Sets the main class for launching test execution
  - Declares default arguments

- Exposes end-user configuration options
  - Allows for providing additional compile-time and runtime-dependencies
  - Option for selecting specific test packages

Key Mechanics of a Rule

Inputs processed by actions that produce outputs

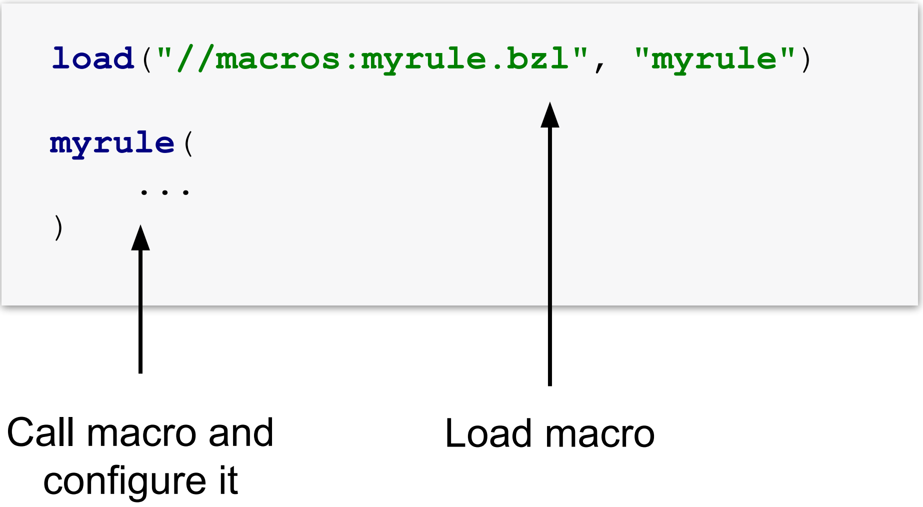


Writing and Using a Rule

Same organizational structure and usage as macro

```
.
├── WORKSPACE
├── rules
│   ├── BUILD
│   └── myrule.bzl
└── src
    └── ...
```

BUILD



```
load("//macros:myrule.bzl", "myrule")

myrule(
      ...
)
```

Call macro and configure it

Load macro

Rule Example

The rule java_library to create a Java library archive

- **Inputs:** The Java source files, dependencies, and compiler options

- **Actions:** Compiling the source code and packaging the class files into JAR file(s)

- **Outputs:** The Java archive containing the class files and a Java archive containing the source code

Remote Caching and Execution

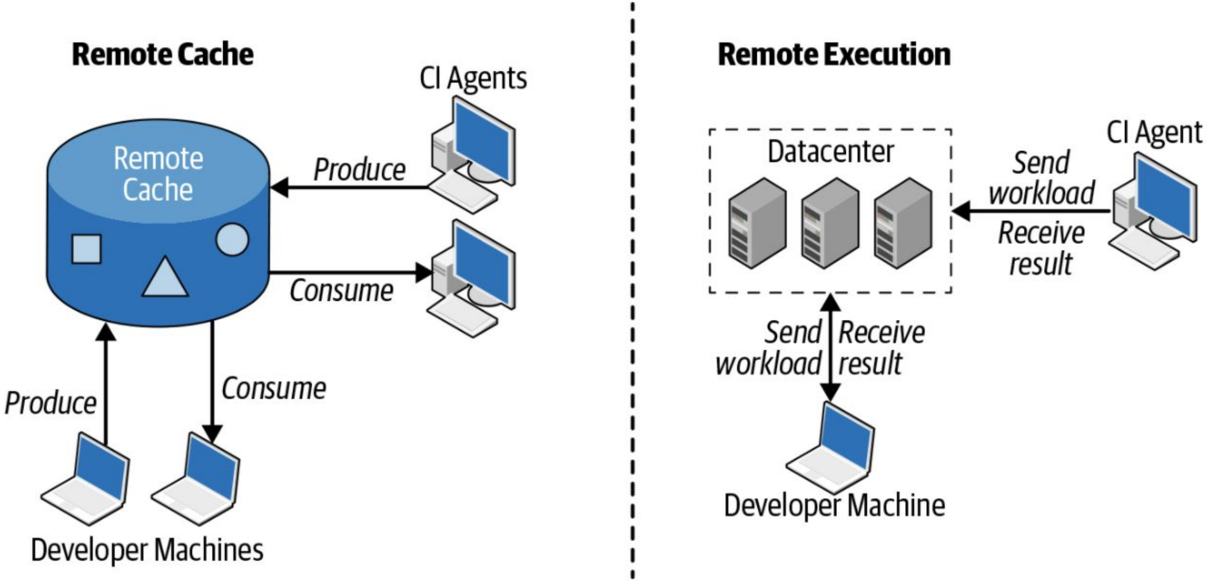
Faster build execution and feedback

- **Remote Caching:** Sharing and reusing build results across multiple, physically separated machines (e.g., developer machines and CI infrastructure)

- **Remote Execution:** Offloading build execution to high-performance computing nodes in a datacenter and using those results on the originating build machine

10,000 Foot View

Both concepts can and should be used together



Remote Caching

Share build outputs across multiple machines

- Based on the concept of a rule, hashes of input and outputs
- Reuses local cache result if existing or reaches out to remote server
- Uploads result if remote cache misses entry
- Remote cache can be used by developer machines or CI agents

Technical Implementation

Two step approach

- Stand up server that acts as the cache's backend
- Server options: nginx, bazel-remote, Google Cloud Storage
- Configure the Bazel build to use the remote cache via `--remote_cache` CLI option

Setup instructions for each solution

Remote Execution

Distribute build and test actions across multiple machines

- Motivation: developer machine doesn’t have to extremely powerful
- Faster builds by farming out build execution to remote machines
- Uses gRPC protocol for communication
- Free and commercial implementations

Continuous Integration (CI)

Trigger an automated build for every commit

- Integrates changes into master/main branch
- Fast feedback by executing the build
- Use the same build tool as on a developer machine
- Standardizes on Bazel runtime version used

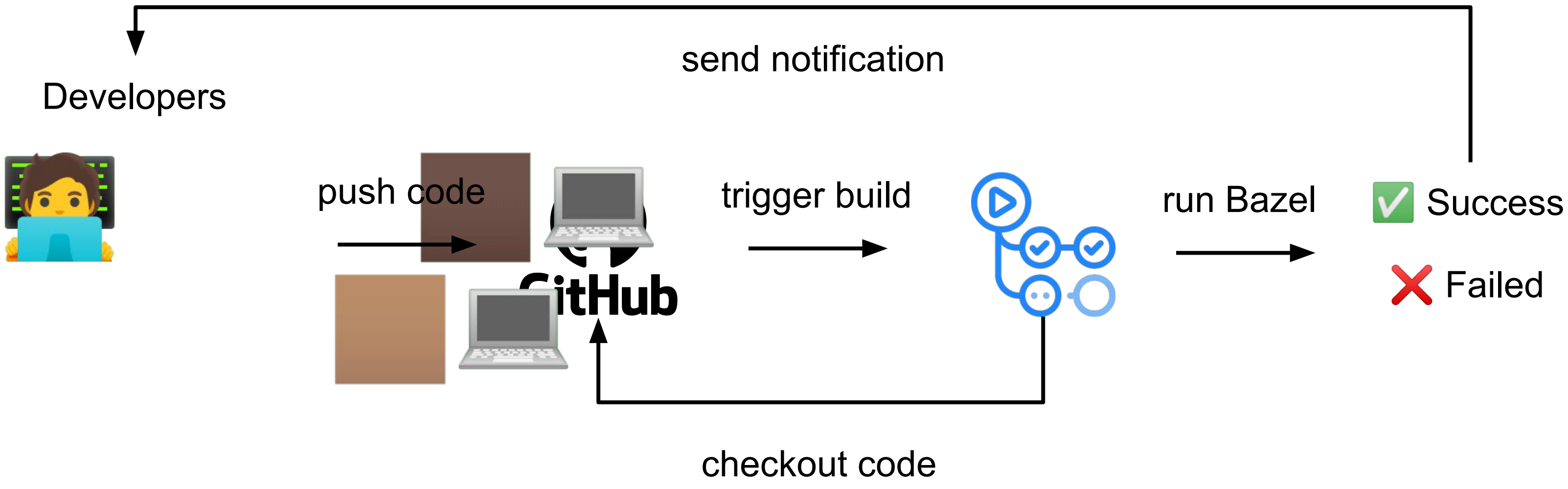
The CI Product GitHub Actions

Fully-integrated CI solution with GitHub repository

- Definition of build using a “configuration as code” approach
- Fast feedback by executing the build upon pushing a commit
- Use Bazelisk to standardize and bootstrap the Bazel runtime
- Use the same build tool and logic as on a developer machine

Basic Workflow

GitHub Actions reacts on an emitted repository event



Terminology

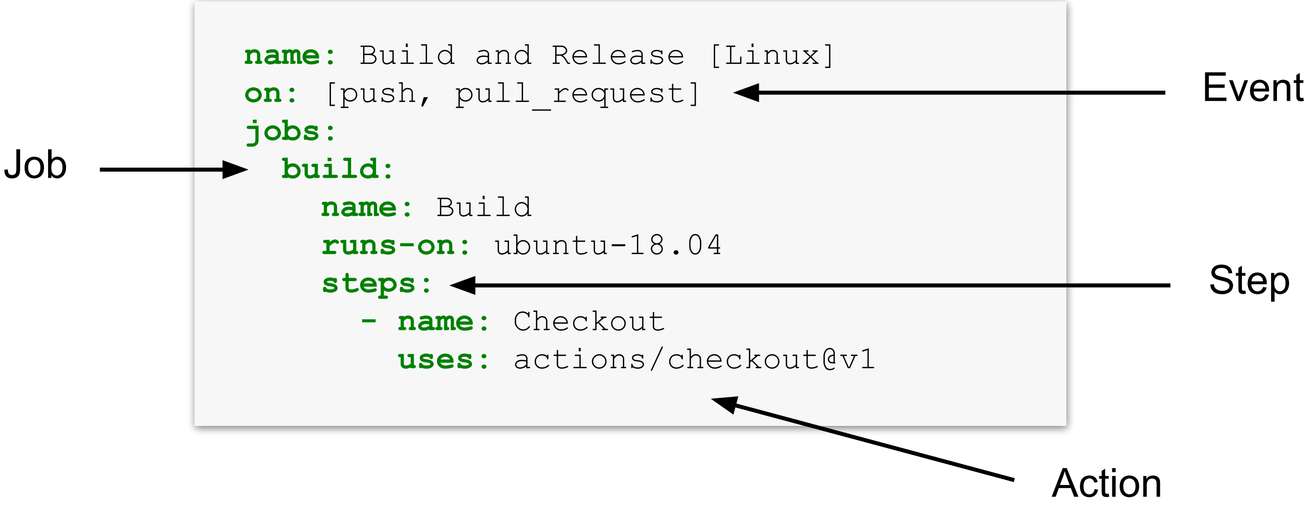
Essential for understanding a workflow definition

- **Event:** Repository activity that triggers a workflow
- **Job:** Set of steps that execute automation logic
- **Step:** Task that can run a command in a job
- **Action:** Reusable functionality provided by GitHub community

Typical Elements of Workflow File

Defines automation logic checked in GitHub repository

.github/workflows/build.yml



```yaml
name: Build and Release [Linux]
on: [push, pull_request]
jobs:
  build:
    name: Build
    runs-on: ubuntu-18.04
    steps:
      - name: Checkout
        uses: actions/checkout@v1
```

Event

Job

Step

Action

Using the Bazelisk Action

Downloads and uses Bazel runtime

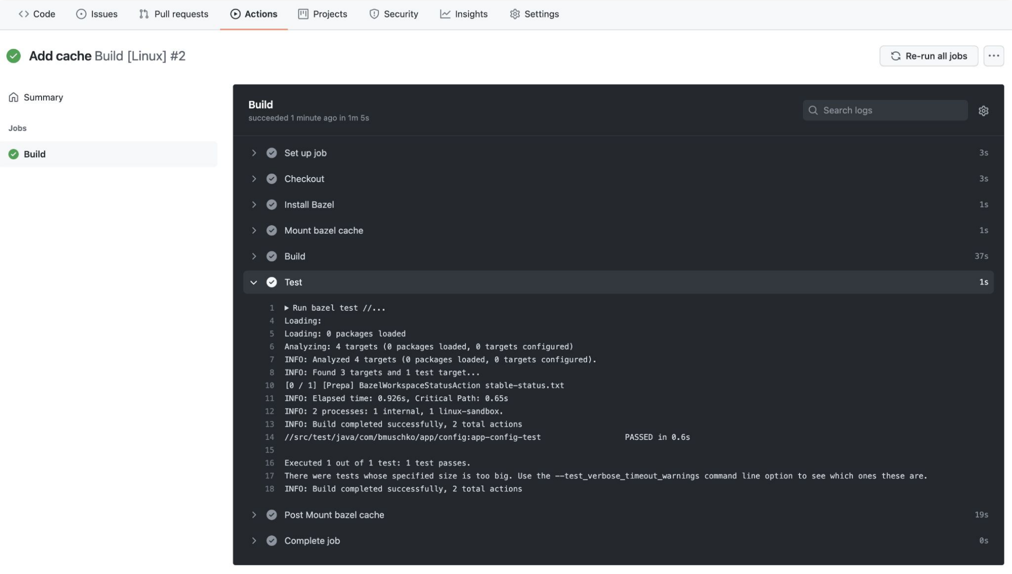
.github/workflows/build.yml

```yaml
steps:
- uses: actions/checkout@v2
- uses: bazelbuild/setup-bazelisk@v1
- name: Mount bazel cache  # Optional
  uses: actions/cache@v2
  with:
    path: "~/.cache/bazel"
    key: bazel
- run: bazel build //...
```

Setup Bazelisk Action page

Actions in the Repository

Click on “Actions” tab at the top



Add cache Build [Linux] #2

Build
succeeded 1 minute ago in 1m 5s

```
Set up job            3s
Checkout              3s
Install Bazel         1s
Mount bazel cache     1s
Build                 37s
Test                  1s
```

```
▶ Run bazel test //...
Loading:
Loading: 0 packages loaded
Analyzing: 4 targets (0 packages loaded, 0 targets configured)
INFO: Analyzed 4 targets (0 packages loaded, 0 targets configured).
INFO: Found 3 targets and 1 test target...
[0 / 1] [Prepa] BazelWorkspaceStatusAction stable-status.txt
INFO: Elapsed time: 0.926s, Critical Path: 0.65s
INFO: 2 processes: 1 internal, 1 linux-sandbox.
INFO: Build completed successfully, 2 total actions
//src/test/java/com/bmuschko/app/config:app-config-test                 PASSED in 0.6s

Executed 1 out of 1 test: 1 test passes.
There were tests whose specified size is too big. Use the --test_verbose_timeout_warnings command line option to see which ones these are.
INFO: Build completed successfully, 2 total actions
```

```
Post Mount bazel cache    19s
Complete job              0s
```

EXERCISE

Using GitHub
Actions for a Bazel
Project

Q & A

5 mins

Wrap Up

Summary and Lessons Learned

O'REILLY®

Thank you