Welcome

Introduction to Bazel

Hello

About me...

- Independent software engineer, trainer, and author
- Specialized in CI/CD, cloud-native application development, DevSecOps
- Runner, Cyclist, Hiker

Prerequisites

This course assumes you:

- Have a basic understanding of build automation concepts.
- Know how to program in the Java language and have been exposed to the ecosystem.

Please Note:

Hands-on programming experience in other languages will work. We are not teaching a language here but rather the automation aspects with Java as the sample language.

Please message me in Chat so we can get you to the right resource.

Why study this subject?

- This course teaches concepts and usage of Bazel for beginners.
- Your organization is using Bazel to build software components with Java.
- You need to translate your knowledge of Maven to Bazel.

We teach over 400 technology topics

You experience our impact on a daily basis!

My pledge to you

I will...

- Make this interactive
- Ask your questions
- Ensure everyone can speak
- Use an on-screen timer

Objectives

At the end of this course you will be able to:

- You'll be able to execute Bazel from the CLI and the IDE.
- You'll be able to model and build Java-based projects with Bazel.
- You'll gain an understanding of advanced concepts in Bazel.

Agenda

- Introduction to Bazel
- Basic usage of Bazel in a Java project
- Dependency management, testing, and publishing
- Outlook on advanced topics

How we’re going to work together

- We’ll build the Bazel knowledge from the ground up.
- Features will be explained with the help of a simple Java-based project.
- You’ll practice the usage of Bazel with the help of exercises. You can find those exercises in a Github repository bmuschko/getting-started-with-bazel. Please clone the repository so you have them available locally.

Introduction to Bazel

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

DevelopIntelligence
A PLURALSIGHT COMPANY

What is Bazel?

Open-source build automation tool

- 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?

Declarative Language

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

Why Should I Use It?

Reproducibility

- Sandboxed build execution by enforcing the definition of all of its dependencies
- Includes execution environment needed to perform the work (toolchains)

Why Should I Use It?

Incrementality

- Bazel keeps tracks of inputs and outputs for a build step
- Inputs and outputs are kept in a database including the timestamp
- A build step is only executed if any inputs/outputs and/or their timestamps change
- Bazel does not require a “clean” build

Why Should I Use It?

Scalability

- Focus is on projects with large codebases in monorepos
- Fine-grained definition of modules (called *packages*)
- Microservices can be built equally as well

Why Should I Use It?

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?

Building polyglot projects

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

Why Should I Use It?

Extensibility

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

Runtime Installation Options

All major operating systems 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

The workspace file

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

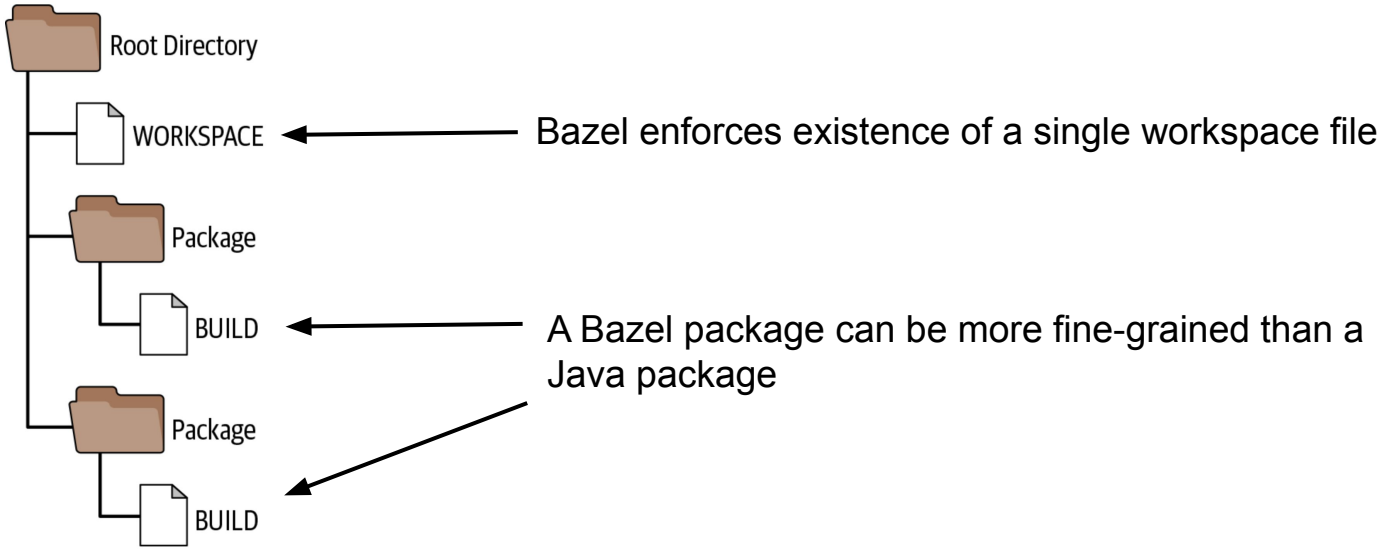
Project Building Blocks

The package file

- 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



Build Logic Concepts

Rule: Reusable, more complex logic

- Defines executable logic, so-called actions
- Usually requires definition of inputs and outputs
- Example: Building a WAR file in Java configurable by different attributes

Build Logic Concepts

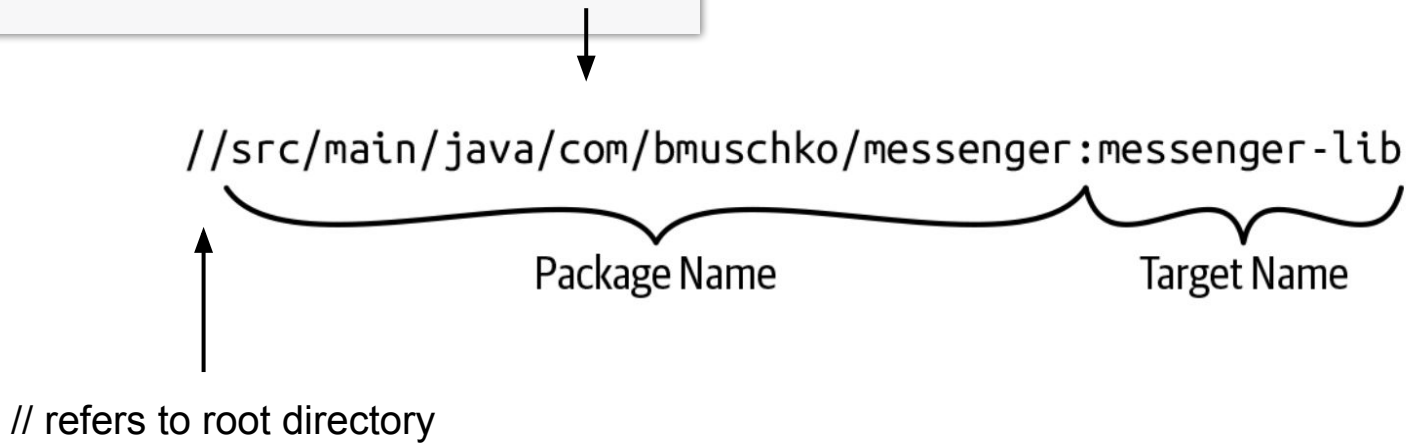
Target: Elements in a package

- A package can contain a set of targets
- Targets represent a file or a rule
- Invoked from the command line by name
- Often times an instantiation of a rule
- Example: A target named `war` configuring a rule that knows how to build a Java web archive

Executing a Target From the CLI

A label combines the package name and target name

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



Commonly-Used Commands

The daily bread and butter of developers

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

Build Everything!

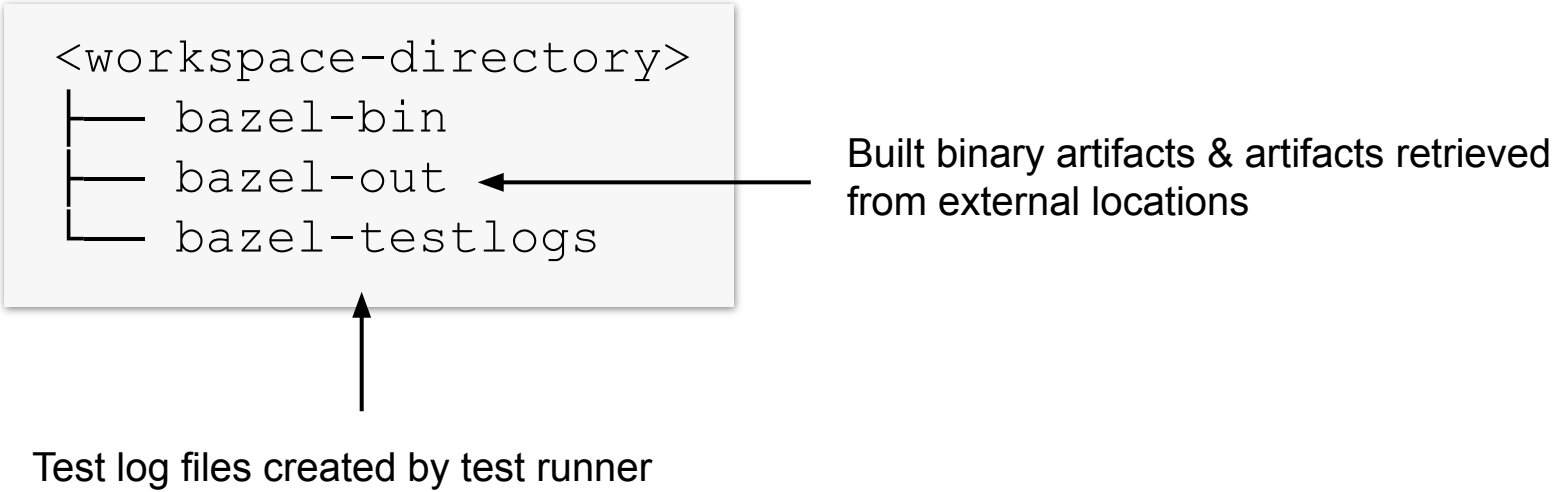
Compile/assemble/test for the whole project

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

Select all packages recursively from the root directory

Output and Cache Directories

Not to be checked into version control!

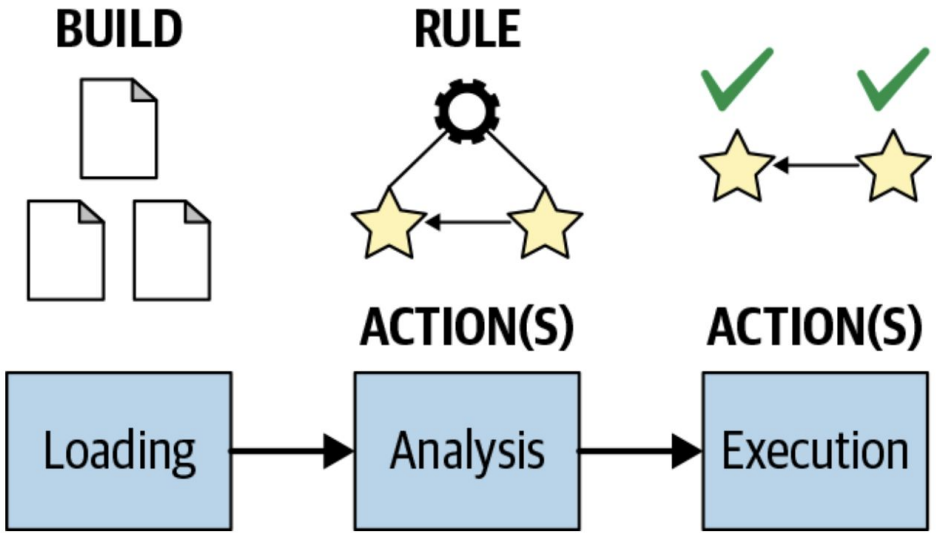


Exercise

“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.

Q & A

Basic Automation for a Java Project

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

DevelopIntelligence
A PLURALSIGHT COMPANY

Typical Java Rules

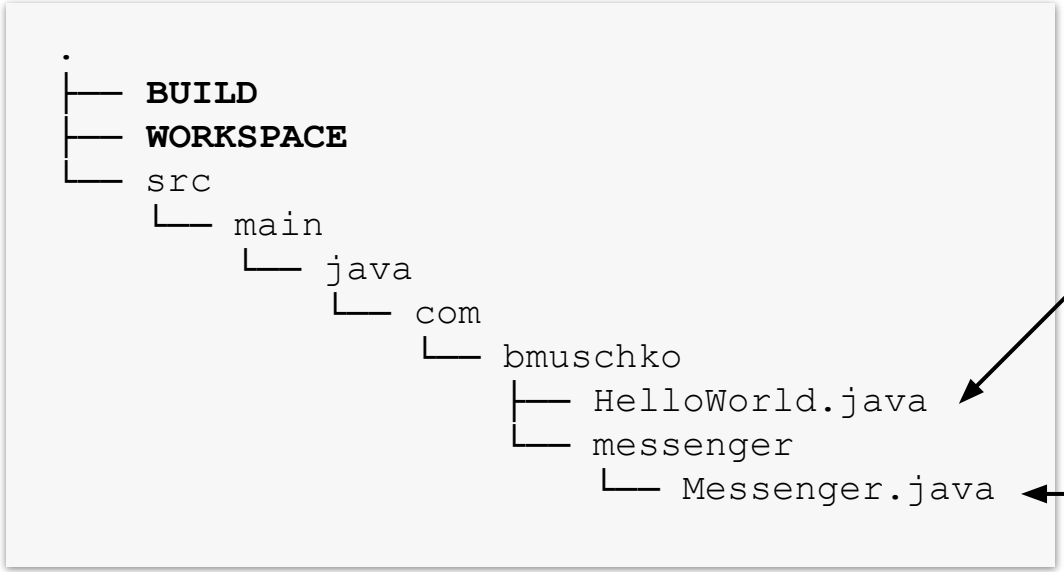
Common developer-focused rules for Java projects

- **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



Modeling the Binary Package

Application is represented by a binary

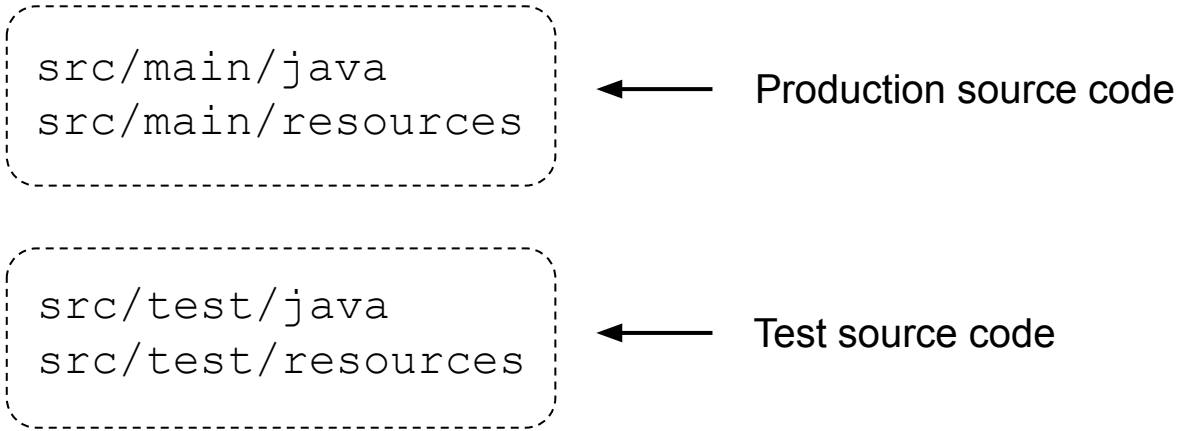
BUILD

```
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



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

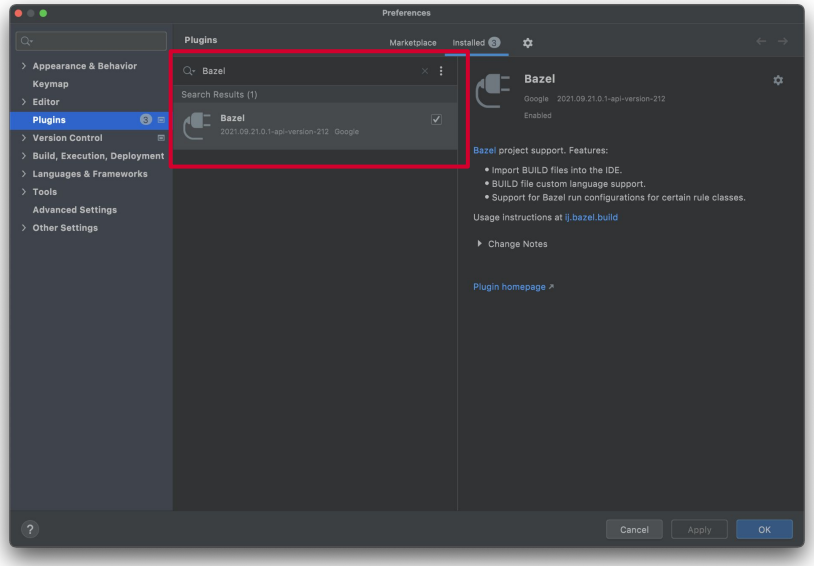
IntelliJ plugin

VSCode extension

Eclipse plugin

Installing the IntelliJ Plugin

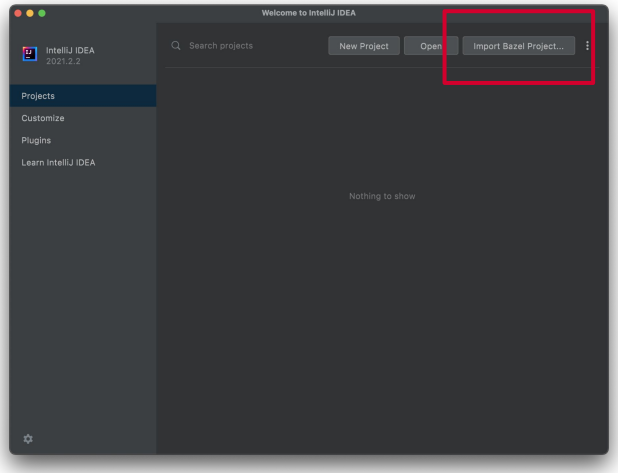
IntelliJ IDEA > Preferences... > Plugins

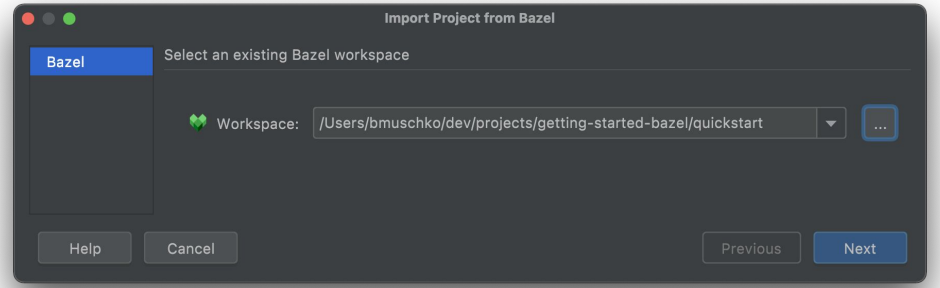


Search for “Bazel”
and install the plugin

Opening a Project

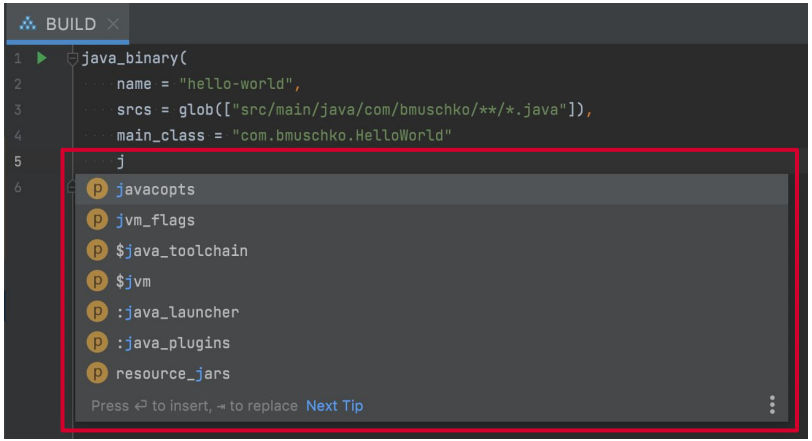
File > Open… > Import Bazel Project...

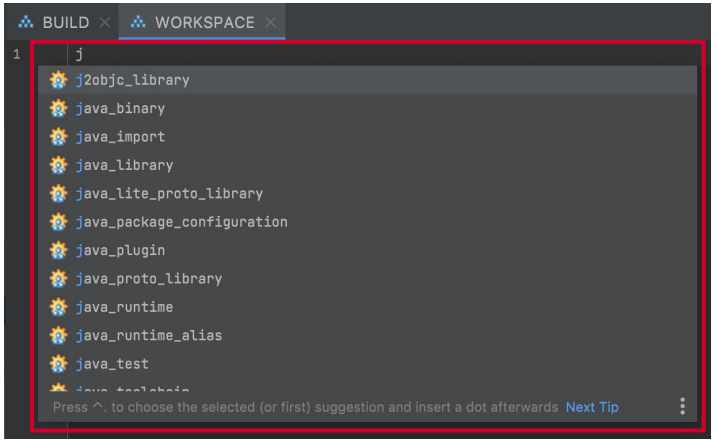




Auto-Completion in Bazel Files

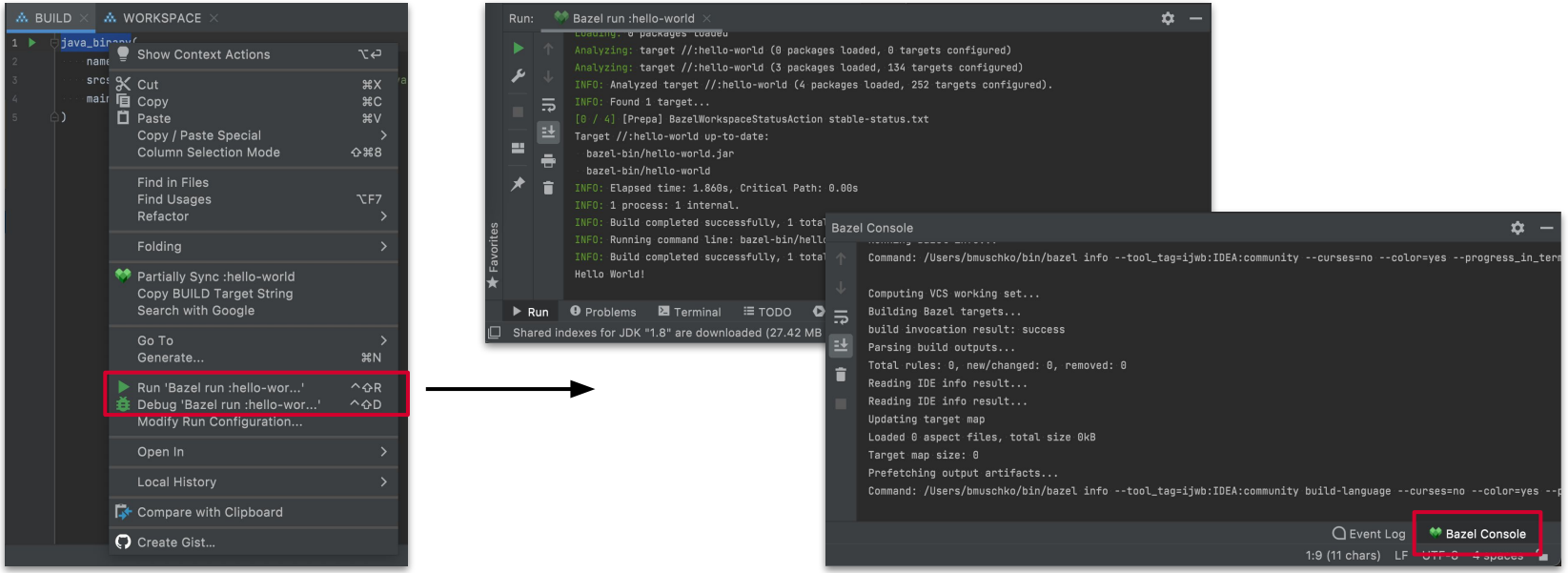
Within the Bazel file start typing





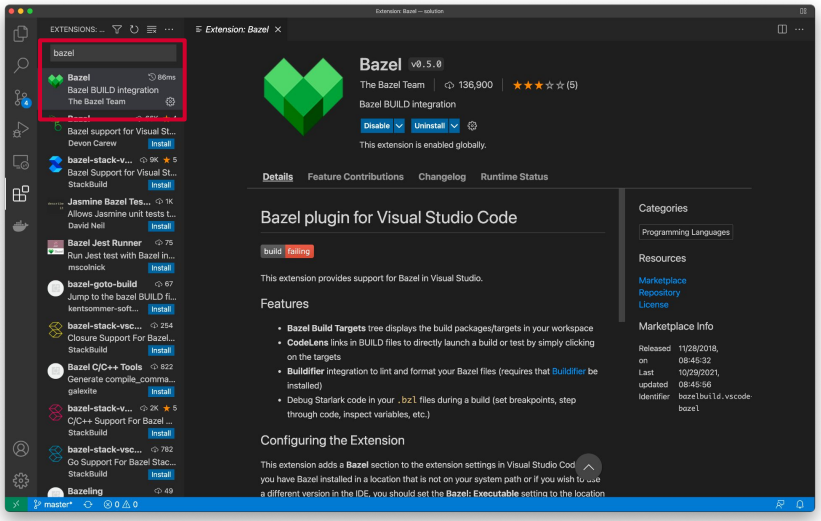
Executing Targets

Context menu on rule in BUILD file



Installing the VSCode Extension

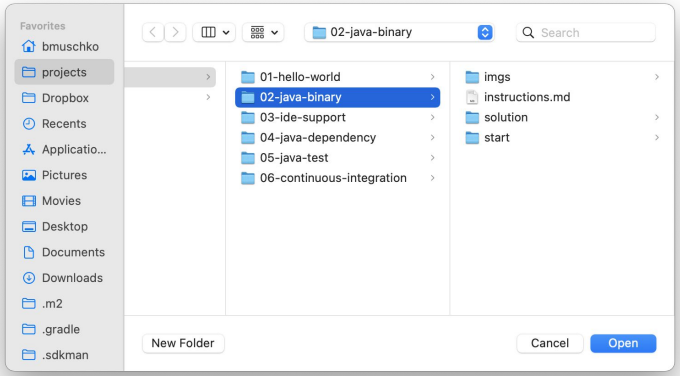
Code > Preferences > Extensions



Search for “Bazel” and install the extension

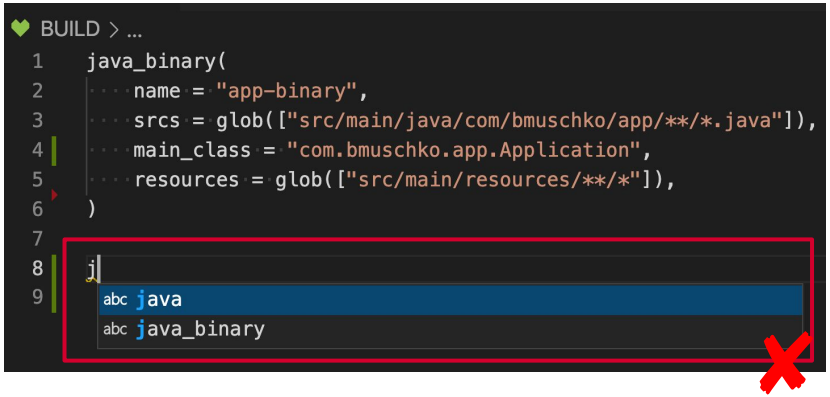
Opening a Project

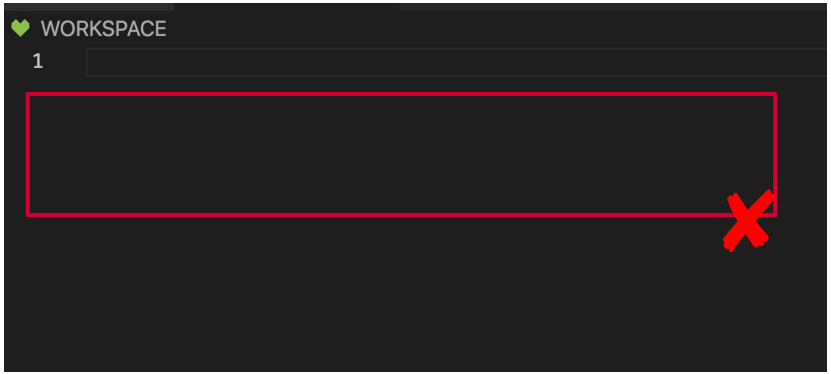
File > Open…



Auto-Completion in Bazel Files

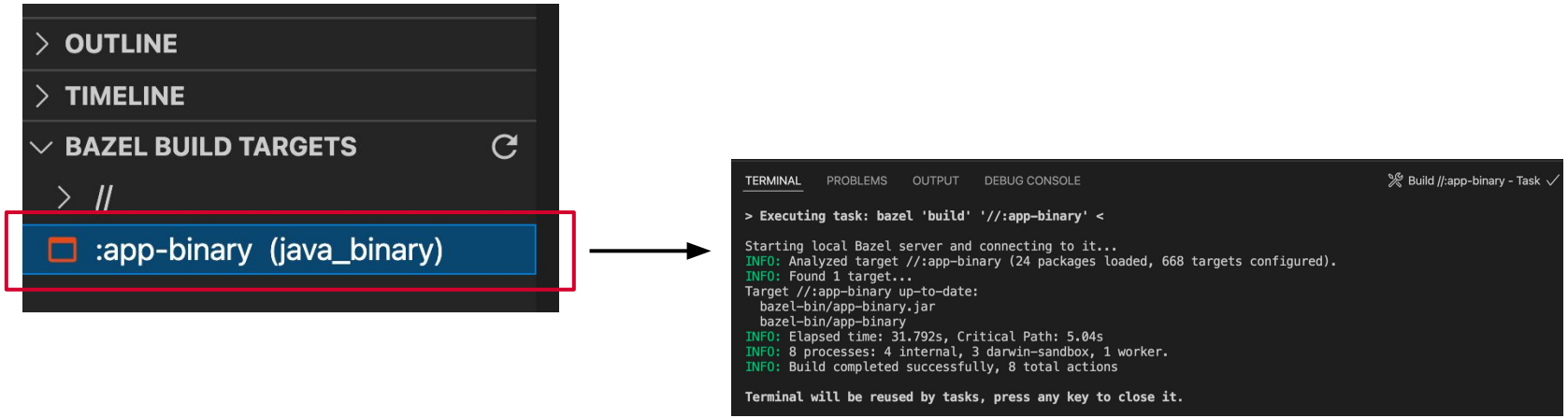
Unfortunately, not working correctly





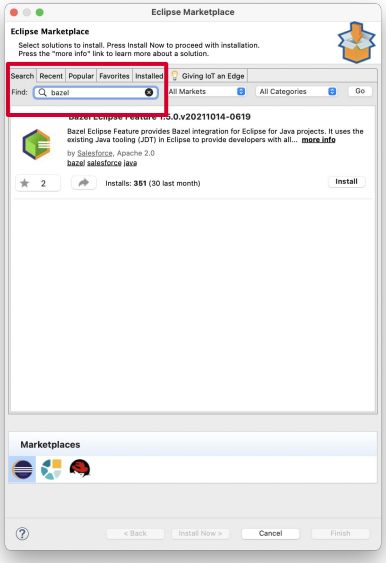
Executing Targets

Context menu on BAZEL BUILD TARGETS



Installing the Eclipse Plugin

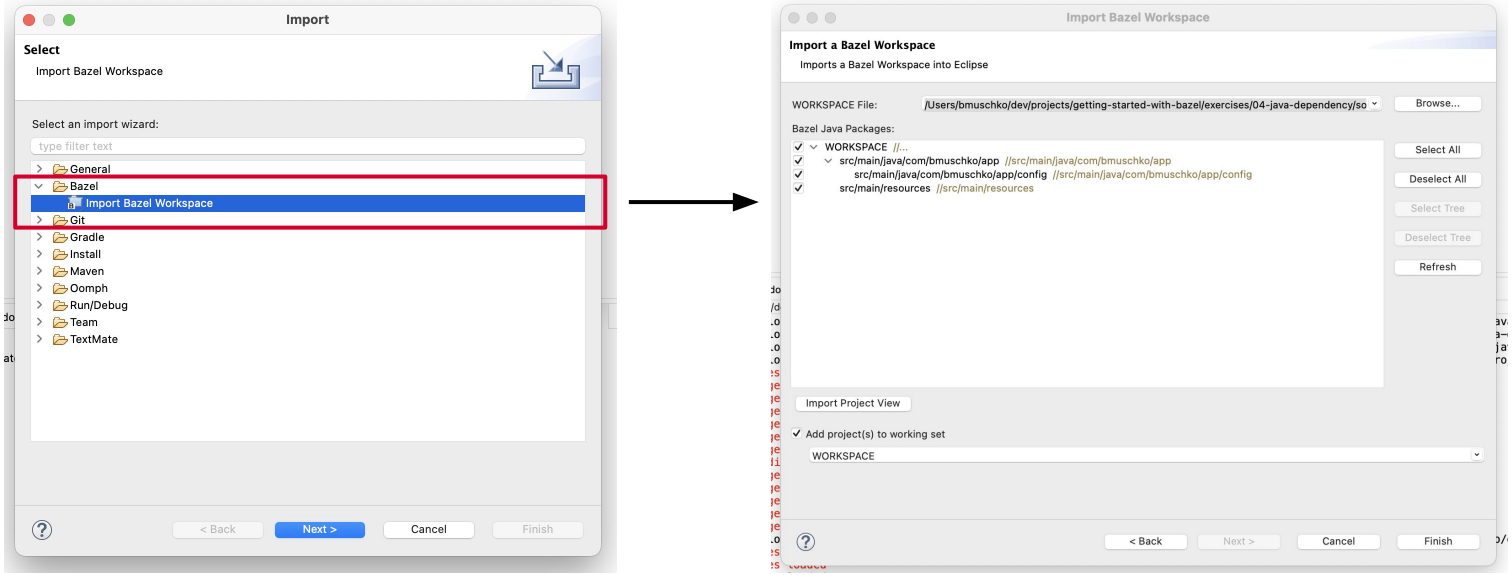
Help > Eclipse Marketplace… > Find



Search for “Bazel” and install the plugin

Opening a Project

File > Import… > Bazel > Import Bazel Workspace



Auto-Completion in Bazel Files

Not available as feature

BUILD | WORKSPACE

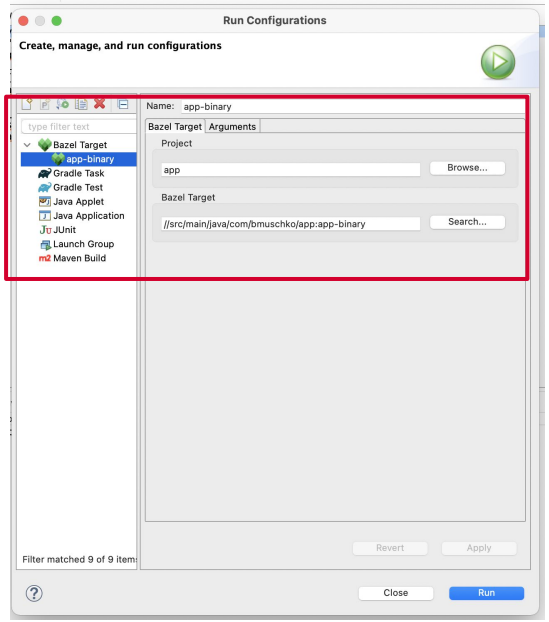
```
java_library(
    name = "app-lib",
    srcs = glob(["*.java"]),
    visibility = ["//src/main/java/com/bmuschko/app:__pkg__"],
    deps = [
        "@maven//:org_apache_commons_commons_configuration2"
    ],
    runtime_deps = [
        "//src/main/resources:app-resources",
        "@maven//:commons_beanutils_commons_beanutils"
    ]
)
```

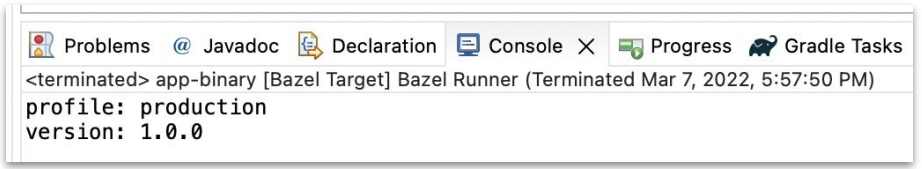
BUILD | WORKSPACE

```
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,
)

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

maven_install(
    artifacts = [
        "org.apache.commons:commons-configuration2:2.7",
        "commons-beanutils:commons-beanutils:1.9.2",
    ],
    repositories = [
        "https://repo1.maven.org/maven2",
    ],
)
```

Executing Targets

Run > Run Configurations…





Exercise

Installing and Using Bazel IDE Support

Q & A

Dependency Management and Automated Testing

Definition and Resolution of Dependencies, Writing and Executing Tests

DevelopIntelligence
A PLURALSIGHT COMPANY

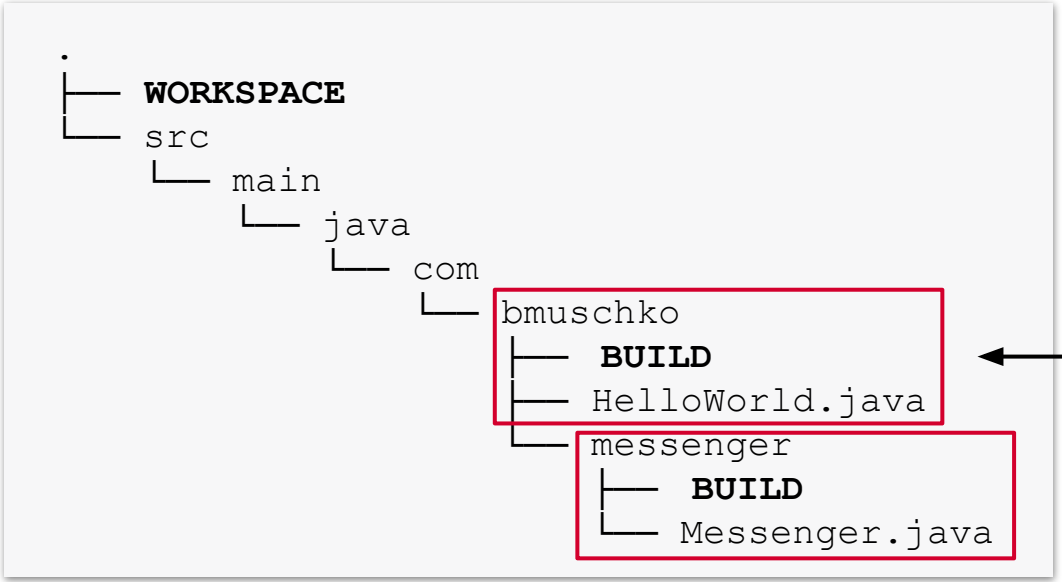
Types of Dependencies

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



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

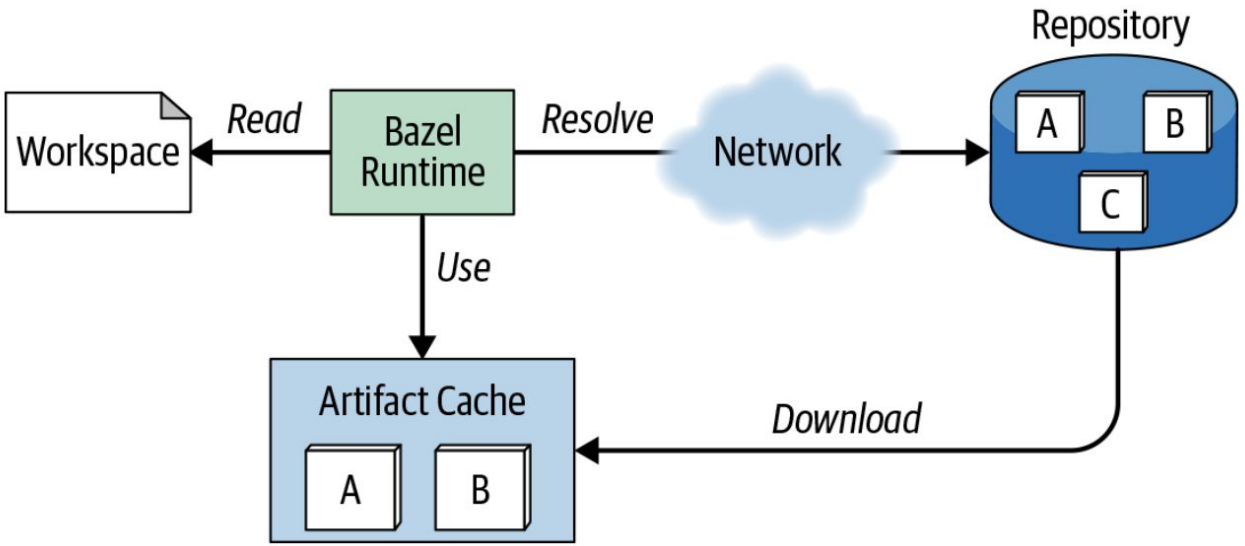
```
$ bazel build //src/main/java/com/bmuschko:hello-world
...
INFO: Found 1 target...
Target //src/main/java/com/bmuschko:hello-world
up-to-date:
  bazel-bin/src/main/java/com/bmuschko/hello-world.jar
  bazel-bin/src/main/java/com/bmuschko/hello-world
INFO: Elapsed time: 19.924s, Critical Path: 4.55s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

Exercise

Declaring Package Dependencies

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

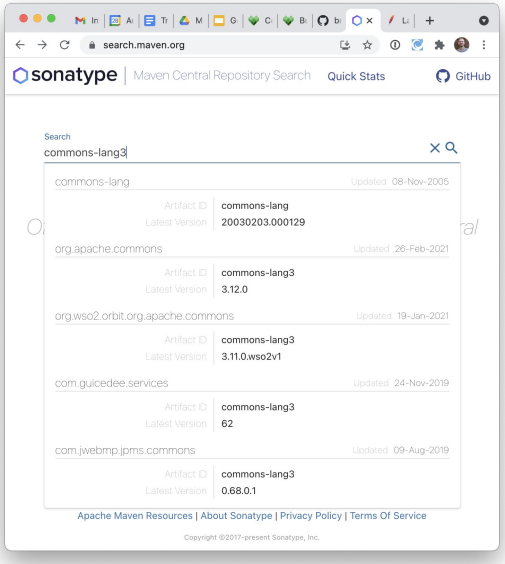
Load rules for consumption as HTTP archive

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



Bazel

bazel.build

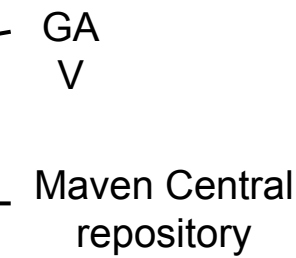
```
maven_jar(
    name = "commons-lang3",
    artifact = "org.apache.commons:commons-lang3:3.12.0",
    sha1 = "c6842c86792ff03b9f1d1fe2aab8dc23aa6c6f0e",
)
```

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",
    ],
)
```



Consuming a Maven Dependency

Dependencies can be scoped for compilation 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

Consuming a Maven Dependency

Dependency notation for consumption

WORKSPACE

```
org.apache.commons:commons-lang3:3.12.0
```

Group ID/Artifact ID/Version

BUILD

```
@maven//:org_apache_commons_commons_lang3
```

Group ID/Artifact ID

Import of External Library Class

Dependencies can be scoped for compile or runtime

Messenger.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

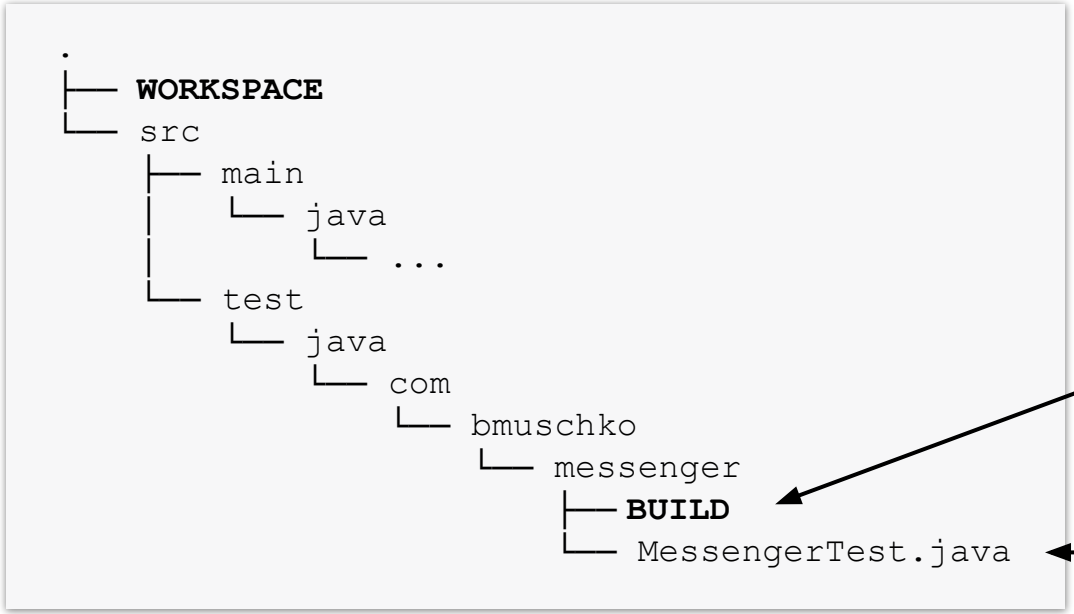
Usage of class

Exercise

Declaring a Dependency on External Libraries

Separating Test Source Code

Distinguish different types of tests



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

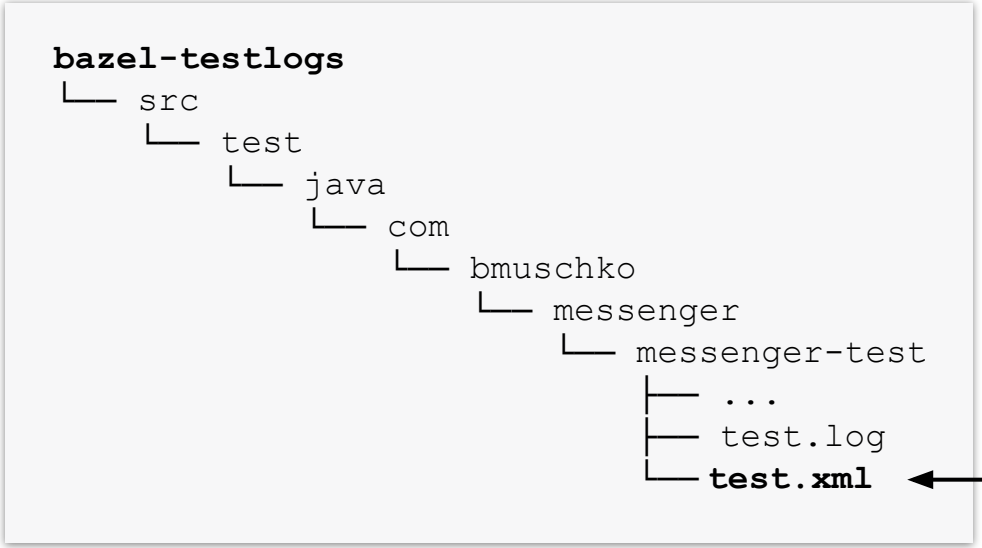
Executing Tests

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



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

Publishing a Java Library

Command for publishing to the Maven Local directory

```
$ bazel run --define "maven_repo=file://$HOME/.m2/repository"
//src/main/java/com/bmuschko/messenger:messenger-lib.publish
```

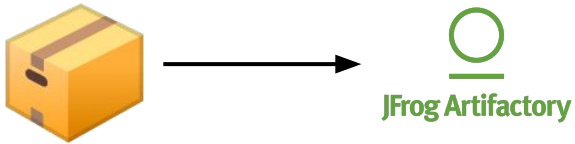
Auto-generated and required

`//$HOME/.m2/repository`

Publishing a Java Library

Command for publishing to a binary repository

```
$ bazel run --stamp
  --define "maven_repo=https://localhost:8081/artifactory/libs-release"
  --define "maven_user=admin"
  --define "maven_password=password"
  --define gpg_sign=true
  //src/main/java/com/bmuschko/messenger:messenger-lib.publish
```



https://localhost:8081/artifactory/libs-release

Exercise

Publishing a Java library to Maven Local

Q & A

Outlook on Advanced Topics

An Introduction to In-Depth Features and Scenarios

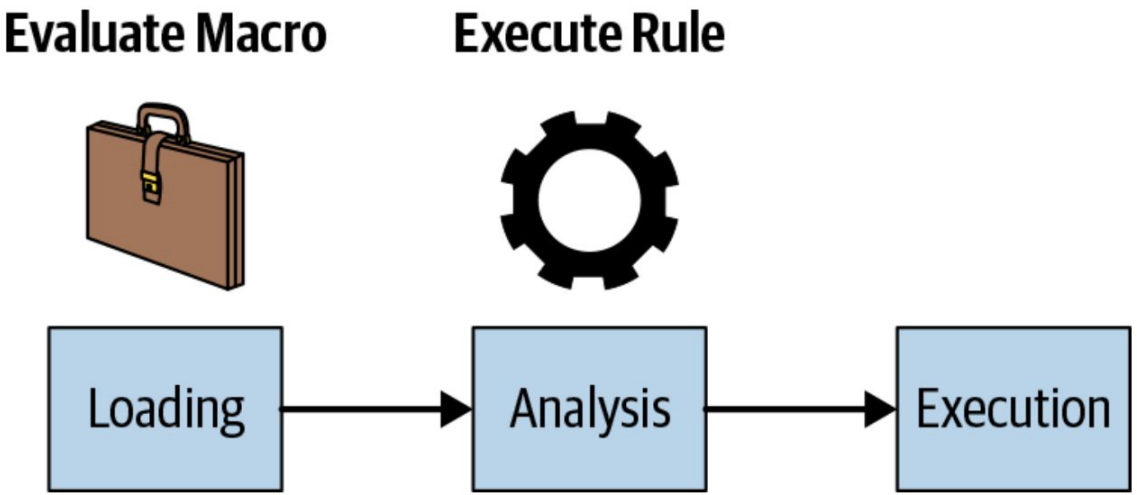
Extension Mechanisms

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



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

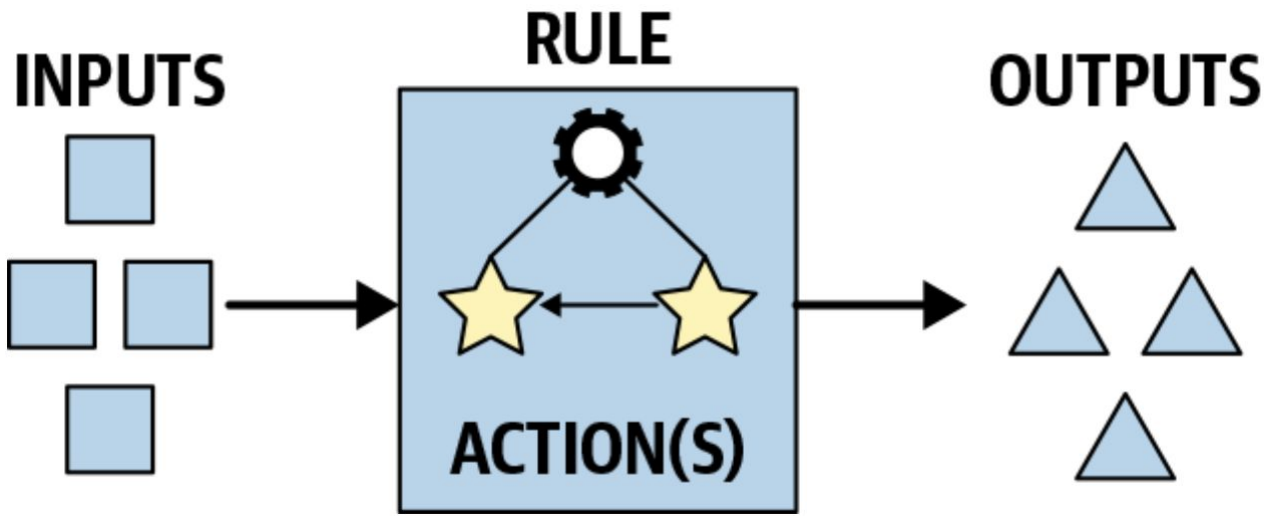
Macro Example

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("//rules:myrule.bzl", "myrule")

myrule(
    ...
)
```

Call rule and configure it

Load rule

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

The genrule

A genrule is a rule that produces at least a single output

- The action invokes a bash command
- Inputs are optional
- To be used when no more specialized rule exists or if you want to execute a simple command
- See documentation for a full description of parameters

Writing and Using a genrule

Implementation can happen directly in BUILD file

BUILD

```
genrule(
    name = "env"
    outs = ["env.txt"],
    cmd_bash = "env >$@",
)
```

The specific type of rule

Shortcut to reference output file

Genrule Example

Substituting strings a in a text file

- **Inputs**: A text file to be modified
- **Actions**: Replacing certain terms or placeholders in the text file
- **Outputs**: A copy of the input text file with selectively replaced content

Exercise

Writing and executing a genrule

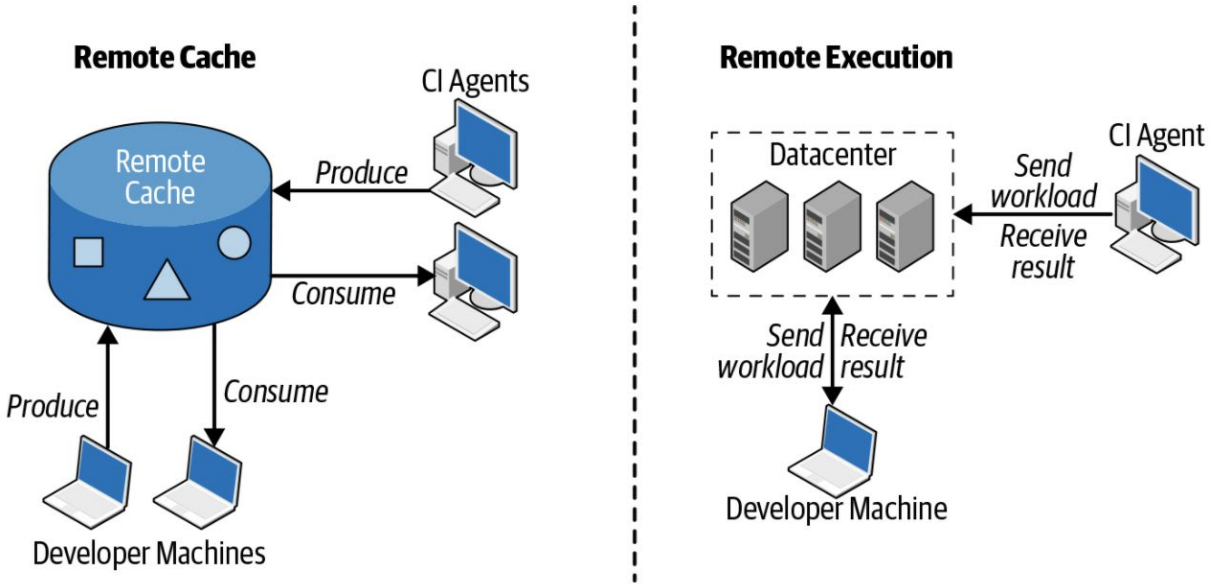
Remote Caching and Execution

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

Workspace Status Information

By default, Bazel keeps track of build meta information

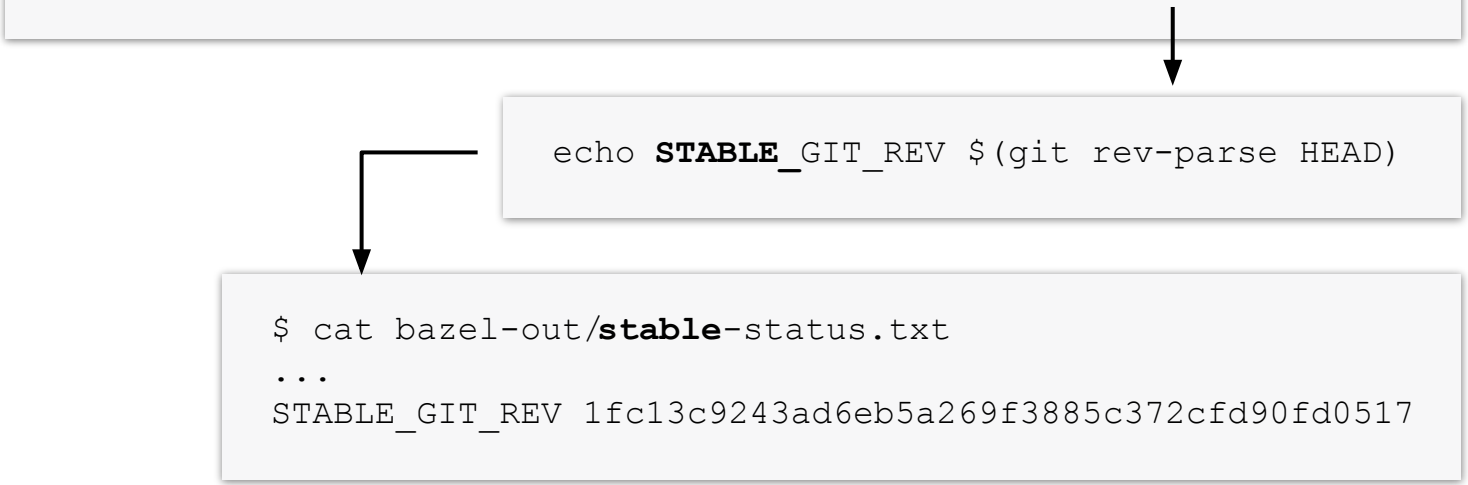
```
$ cat bazel-out/stable-status.txt
BUILD_EMBED_LABEL
BUILD_HOST ascent.lan
BUILD_USER bmuschko
```

```
$ cat bazel-out/volatile-status.txt
BUILD_TIMESTAMP 1646934566
```

Workspace Status Information

You can generate and append custom information with a script

```
$ bazel build //... --workspace_status_command=custom-metadata.sh
```



Build Stamping

A released artifact may want to include build information

- The CLI option `--stamp` embeds build information in binaries
- Usually performed on a packaging rule
- The functionality is not consistently implemented across common rules
- Including volatile information (e.g. a timestamp) can cause a rebuild and therefore redo a lot of work
- A genrule can be used to react to the `--stamp` option and perform custom handling

Bazel Query

Analyze build dependencies by an expression

- Performed by `query` command
- Requires you to provide an expression
- Wide range of options available to form a query
- Reference the query guide for more information

Common Queries

Transitive closure of dependencies: `deps`

```
$ bazel query "deps(//src/main/java/com/bmuschko/app/config:app-lib)"
//src/main/java/com/bmuschko/app/config:app-lib
//src/main/resources:app-resources
//src/main/resources:application.properties
//src/main/java/com/bmuschko/app/config:PropertiesConfigurationReader.java
//src/main/java/com/bmuschko/app/config:ConfigurationReader.java
...
```

Use case: “Give me the dependencies consumed by this target”

Common Queries

Transitive closure of source dependencies: `deps`

```
$ bazel query 'kind("source file",
  deps(//src/main/java/com/bmuschko/app/config:app-lib))'
  --noimplicit_deps
//src/main/resources:application.properties
//src/main/java/com/bmuschko/app/config:PropertiesConfigurationReader.java
//src/main/java/com/bmuschko/app/config:ConfigurationReader.java
```

Use case: "Give me the source files of the dependencies consumed by this target"

Common Queries

Build files for packages: `buildfiles(deps)`

```
$ bazel query
  "buildfiles(deps(//src/main/java/com/bmuschko/app/config:app-lib))"
  --noimplicit_deps
//src/main/resources:BUILD
//src/main/java/com/bmuschko/app/config:BUILD
```

Use case: “Give me the build files that of the dependencies consumed by this target”

Common Queries

Finding reverse dependencies: `rdeps`

```
$ bazel query "rdeps(//...,
  //src/main/java/com/bmuschko/app/config:app-lib)"
//src/main/java/com/bmuschko/app:app-binary
//src/main/java/com/bmuschko/app/config:app-lib
```

Use case: “Find the downstream dependencies that are potentially affected by a change to this target”

Java Toolchains

Java rules provide two toolchains

- Toolchain 1: Compiling Java sources: `--java_language_version`
- Toolchain 2: Executing and testing Java binaries: `--java_runtime_version`
- The default toolchain is based on Java 11

```
$ bazel run --java_language_version=17 --java_runtime_version=remotejdk_17
  //:app-binary
```

Java Toolchains

Discovering available Java runtime toolchains

```
$ bazel query 'kind(java_runtime, @bazel_tools//tools/jdk:all)'
@bazel_tools//tools/jdk:current_java_runtime
@bazel_tools//tools/jdk:jdk_8
@bazel_tools//tools/jdk:remote_jdk11
@bazel_tools//tools/jdk:remotejdk_15
@bazel_tools//tools/jdk:remotejdk_16
@bazel_tools//tools/jdk:remotejdk_17
```

Java Toolchains

Configuring JVM and Java compiler flags

- `--jvmopt`: JVM runtime options
- `--javacopt`: Java compiler options

```
$ bazel run --java_language_version=remotejdk_11 --java_language_version=11
  --jvmopt="--enable-preview" --javacopt="--enable-preview" //:app-binary
```

Java Toolchains

Configuring toolchains in .bazelrc

.bazelrc

```
build --java_language_version=17
run --java_runtime_version=remotejdk_17
```

Java Toolchains

Defining additional toolchains

- A custom execution toolchain allows for defining a JVM from a local or remote repository including its version, OS, and architecture
- A custom compilation toolchain allows for defining for a specific JDK including specific features
- See documentation for more information

Q & A

Thank you!

If you have additional questions,
please reach out to me at:
benjamin.muschko@automatedascent.com

DevelopIntelligence
A PLURALSIGHT COMPANY