Writing Bazel Rules

https://jayconrod.com/posts/106/writing-bazel-rules--simple-binary-rule
https://github.com/jayconrod/rules_go_simple

Jay Conrod
Software Engineer, EngFlow
@jayconrod

Introduction & History

- Previously, I worked on rules_go and Gazelle (2017–2021).
- Bazel had a lot of API churn, pre 1.0.0:
  constantly needed to rewrite and refactor.
- Documentation has references but lacked explanation.
- https://jayconrod.com/posts/106/writing-bazel-rules--simple-binary-rule
- https://github.com/jayconrod/rules_go_simple

- Prerequisites: a little experience using Bazel, Git, Python, your favorite editor.
  Go is used as an example, but no Go knowledge is needed.

Agenda

1. Concepts
2. Simple binary rule
3. Libraries, providers, depsets
4. Data, runfiles
5. Efficient execution
6. Repository rules
7. Toolchains
8. Module extensions

Agenda

1. **Concepts**
2. Simple binary rule
3. Libraries, providers, depsets
4. Data, runfiles
5. Efficient execution
6. Repository rules
7. Toolchains
8. Module extensions

EngFlow

Load statement for rule definition

```
load("@rules_go//go:def.bzl", "go_binary")
```

Rule

```
go_binary(
    name = "hello",
```

Label list attribute

```
    srcs = ["hello.go"],
```

String attribute

```
    importpath = "example.com/hello",
    deps = [
```

In same repo

```
        "//util",
```

In other repo

```
        "@org_golang_x_net//html",
    ],
)
```

Target

Concepts

- **Repo:** top-level directory containing source code and BUILD files. Defined by a MODULE.bazel or WORKSPACE file.
- **Package:** everything in a directory, defined by a BUILD file.
- **Target:** logical thing you can build. Has attributes, maybe output files.
- **Label:** string referring to a target, source file, or generated file.
  - **@repo//pkg/path:target** - complete form.
  - **//pkg/path:target** - omit @repo if in same repo.
  - **:target** - omit //pkg/path if in same package.
  - **target** - same as :target, conventionally only used for files
  - **//pkg/path** - same as **//pkg/path:path**

EngFlow

Concepts

- **Rule:** code used to define targets. Declares output files, actions. Written in Starlark. Called like a function, but the implementation runs later.
- **Action:** command with inputs and outputs. Usually cached. May run remotely or in a sandbox, preventing hidden dependencies.
- **Macro:** Starlark function called from BUILD file. Used to beautify target declarations or declare multiple targets. *Symbolic macros* are new.
- **Attribute:** named property of a rule (or target). Acts like an argument.

Why write rules?

- A rule takes input files, declares output files, and declares actions that may be used to generate them.
- For simple one-off actions, use `genrule` (maybe wrapped in a macro). `genrule` is a built-in generic rule that can run shell commands.
- Rules can be oriented around a logical goal and may use multiple actions, e.g., compile multiple files, or compile and link.
- Rules can return providers (metadata)
  and can consume providers from other rules.
- Rules can use configurable toolchains and can be platform-aware.

Agenda

1. Concepts
2. **Simple binary rule**
3. Libraries, providers, depsets
4. Data, runfiles
5. Efficient execution
6. Repository rules
7. Toolchains
8. Module extensions

github.com/jayconrod/rules_go_simple

- Install Go: https://go.dev/dl/
  - Make sure to add `bin` directory to your PATH. Try running **`go version`**.
- Checkout the repo:
  - **`git clone https://github.com/jayconrod/rules_go_simple`**
  - **`git switch v1_exercise`**
- Exercise: implement the go_binary rule
  - Everything marked **`# EXERCISE`**
  - `go_binary` declaration, `_go_binary_impl`, `go_compile`, `go_link`
- Test: **`bazel test //...`**
- Check: **`git diff v1_exercise v1`**

Review: simple binary rule

- Repo structure
  - Dependencies separate, in MODULE.bazel.
  - Internals separated from public definitions.
  - Shareable functions are separate from rules.
- Rule declaration using **`rule`** creates a callable rule object.
- Rule implementation is a regular function, called by Bazel during analysis.
- **`ctx`** gives us access to attributes.
- **`ctx.actions`** lets us declare output files and actions.
- **`return [DefaultInfo(...)]`**

More about Starlark

- Limited subset of Python. No modules, classes, exceptions, generators, floats, sets, async, ...
- No recursion or while loops. Not Turing complete.
  Avoid writing complicated code in Starlark.
- BUILD and MODULE.bazel files are even more limited.
- After loading a .bzl file, everything is **frozen**.
  - Language encourages an imperative, mutable style.
  - But files (and rules) can be evaluated in parallel without synchronization.
- Standalone interpreter: github.com/google/starlark-go

Load phase

Constructs the target graph.

- Load BUILD files (and imported .bzl files) for command-line arguments.
- Expand patterns in arguments to match specific targets.
- Recursively resolve labels, loading other BUILD files.

All this happens on the host machine, in parallel. Cached in memory.

Bazel phases: Load → Analysis → Execution

Analysis phase

Construct action graph.

- For each target in post-order, run the rule implementation.
- Rules **CAN**
  - declare output files and actions that produce them.
  - access attributes and file names from current target.
  - access "providers" (metadata) from attribute targets.
  - return providers for use by dependent targets.
- Rules **CANNOT**
  - read or write files directly.
  - run commands directly.
- Runs on the host machine, in parallel. Action graph cached in memory.

Bazel phases: Load → **Analysis** → Execution

Execution phase

Run commands to build stuff.

- Commands have arguments, environment, inputs, outputs.
- Several ways to run (`--spawn_strategy`, `--test_strategy`)
  - remote: on another machine.
  - worker: using a persistent worker.
  - sandboxed: on host machine with no access to other files.
  - local: in a temp directory. Useful for debugging, compatibility.
- Should be deterministic, ideally hermetic.
- Results cached on local disk or shared remote cache.

Bazel phases: Load → Analysis → Execution

Files and Targets

- **[`File`]**: represents an input or output file.
  - `ctx.actions.declare_file` creates a new File.
  - `ctx.files.<attr>` is a *flat* list of Files for a `label` or `label_list` attribute.
  - `ctx.file.<attr>` is a single file for a label attribute with `allow_single_file` set.
  - `ctx.executable.<attr>` is like `ctx.file.<attr>` but preferred for executables.
  - Properties: `basename`, `dirname`, `extension`, `path`, ...
  - Absolute path is not known during analysis.
- **[`Target`]**: represents a target in the graph created by load phase.
  - `ctx.attr.<attr>` is a single target (for `label` attribute)
    or list of targets (for `label_list` attribute).
  - Properties: `label`, `files`, ...
  - Rule can only find direct dependencies. No indirect or reverse dependencies.

Agenda

1. Concepts
2. Simple binary rule
3. **Libraries, providers, depsets**
4. Data, runfiles
5. Efficient execution
6. Repository rules
7. Toolchains
8. Module extensions

Library rule, depsets, providers

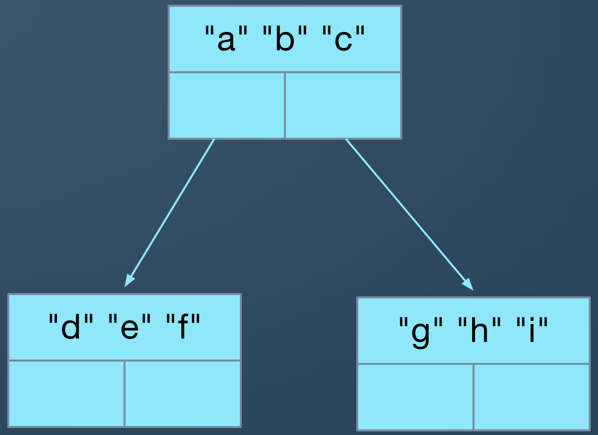
- We want to declare a bunch of `go_library` targets and allow dependencies among them.
- `go_binary` needs to know all *transitive* libraries in order to link.
- `go_library` should return a `GoLibraryInfo` provider with analysis metadata.
- **Provider:** a named struct with information about a target, returned by a rule, used by dependent rules.
- **depset:** efficient way of representing values without $O(n^2)$ time or space for construction or iteration. Will be used by our provider.

Library rule

- `git switch v2_exercise`
- Exercise: implement `go_library`, `go_binary`
- Test: **`bazel test //...`**
- Check: **`git diff v2_exercise v2`**

More about depsets

- Has **direct** list of values,
  and **transitive** list of depsets.
- Immutable. Contents must also be immutable.
- Iterate with `to_list()`
- Iteration order: `"default"`, `"postorder"`, `"preorder"`, `"topological"`.
- Internally uses a hash set to avoid repetition. Contents must be hashable.



Debugging tips

- `print("!!")`: adds `DEBUG:` line to Bazel console output.
- `--spawn_strategy=local`: if your action breaks in the sandbox (or on a remote worker) but works with this flag, look for undeclared dependencies.
- `--sandbox_debug`: leave sandbox directory behind.
- `--subcommands`: causes Bazel to print full commands.
  - cd to execution root directory. Are the input files what you expect?
  - Run the command. What happens?

Testing tips

- Simplest approach: bunch of targets in a test directory.
- Unit testing: `@bazel_skylib//lib:unittest.bzl`
    - `unittest` - for Starlark functions.
    - `analysistest` - for rules.
    - `loadingtest` - for macros.
    - `asserts` - assertion functions.
- Integration testing
    - `rules_bazel_integration_test`
    - Or write your own?

Agenda

1. Concepts
2. Simple binary rule
3. Libraries, providers, depsets
4. **Data, runfiles**
5. Efficient execution
6. Repository rules
7. Toolchains
8. Module extensions

Data and runfiles

- Bazel actions can run remotely or in a sandbox.
  They don't have access to input files they don't explicitly depend on.
  This also true for tests (special actions) and `bazel run` (special directory).
- Frequently we need configuration files, certificates, resources, test data, ...
- **runfiles:** set of data files available to anything that depends on a target.
- Construct with `ctx.runfiles`.
- Combine with `runfiles.merge`.
- Attach runfiles in `DefaultInfo` provider.
- Let users specify `runfiles` with `data` attribute.
- Access runfiles using dark magic.

Data and runfiles

- `git switch v3_exercise`
- Exercise: add `data` attribute,
  return runfiles from `go_binary` and `go_library`
- Test: **`bazel test //...`**
- Check: **`git diff v3_exercise v3`**

Accessing runfiles at run time

- Common case: use a relative path from repo root
- In arguments: use `$(execpath ...)`, `$(rootpath ...)`
  See [Predefined source/output path variables].
- Bazel has slightly different behavior for:
  - Tools within actions, tests, `bazel run`
  - Windows, other OSs
  - Main repo, external repos
- Most languages use a library like `@rules_go//go/tools/bazel`.
- Watch: [Runfiles and where to find them] by Fabian Meumertzheim

Agenda

1. Concepts
2. Simple binary rule
3. Libraries, providers, depsets
4. Data, runfiles
5. **Efficient execution**
6. Repository rules
7. Toolchains
8. Module extensions

Efficient execution

- Starlark is limited.
  - Cannot read or write files.
  - No recursion: difficult to process recursively structured data.
- Limitations are by design.
  - Load and analysis phases run on host machine, so they don't scale for huge repos.
  - Execution phase can run remotely; results can be stored in shared cache.
- Try to move complex logic into execution layer.
  - This usually means writing small "builder" tools, then depending on them from rules.
  - Avoid shell scripts except for bootstrapping: hermeticity and portability is hard.

Exercise: Efficient execution

- `git switch v4_exercise`
- Exercise: implement `go_tool_binary`, use in compile, link, test.
- Test: **`bazel test //...`**
- Check: **`git diff v4_exercise v4`**

Review: Efficient execution

- Use internal rules to build tools for execution phase.
  A tool can be written in your preferred language and has full I/O.
- Consider remote execution and portability when writing actions.
  - Avoid tiny actions that use the same inputs and always run together.
    This adds extra I/O overhead for remote execution.
  - Split actions if different parts can run in parallel on different machines.
  - Avoid shell scripts if you can. Very platform dependent.
- Implement a persistent worker when possible.

Passing information to actions

- `json.encode()` works on most values including structs, providers.
- `ctx.actions.write()`: write a file with given contents.
- `ctx.actions.args()`: creates an Args object.
  - Use with `ctx.actions.run`.
  - `Args.add()`: add fixed list of arguments.
  - `Args.add_all()`, `add_joined()`: add depset of arguments with custom formatting.
  - `Args.use_param_file`, `set_param_file_format`: spill arguments to a file.
    Very important for Windows due to command line length limit!
    Your tool must be able to read these and deal with shell quoting.

Profiling

- bazel clean
- Build with --profile=profile.gz
- bazel analyze-profile
- View with Perfetto.
- https://analyzer.engflow.com/ - gives recommendations for larger builds.

Agenda

1. Concepts
2. Simple binary rule
3. Libraries, providers, depsets
4. Data, runfiles
5. Efficient execution
6. **Repository rules**
7. Toolchains
8. Module extensions

Repository rules

- Defines a repo directory using Starlark code.
  Usually fetches an archive and writes BUILD files.
- Examples: `http_archive`, `git_repository`.
- Defined with `repository_rule`, uses `repository_ctx`.
- Full access to host: read/write files, execute commands.
- Prefer Bazel modules when available.
- Used to implement module extensions.

Exercise: repository rules

- `git switch v5_exercise`
- Exercise: go_download
- Test: **`bazel fetch @go_darwin_arm64//:README.md`**
- Check: **`git diff v5_exercise v5`**

Review: repository rules

- Powerful, but slow and usually non-hermetic.
- Evaluated during load phase, not part of the regular build.
- Minimize logic as much as possible: push toward execution phase.
- Prefer precompiled binaries over building tools inside repository rules.

Agenda

1. Concepts
2. Simple binary rule
3. Libraries, providers, depsets
4. Data, runfiles
5. Efficient execution
6. Repository rules
7. **Toolchains**
8. Module extensions

Insight: it's dependency injection

- Toolchain abstracts away platform-specific part of a rule set
  - Implementations for different machines: linux, windows, arm64, ...
  - Implementations for different tools: GCC, Clang, MSVC
- User registers toolchains, sets execution, target platforms.
- Bazel selects a toolchain that satisfies platform constraints.
- Rule requests a toolchain using its type.
  - Gets a provider object from the toolchain.
  - Provider contains implicit dependencies, functions, etc.

Concepts

- `platform`: description of where code can run, expressed as constraints
  - Host, execution, target platforms may all be considered.
- `constraint_value`: a fact about a platform (e.g., linux, x86_64)
- `constraint_setting`: a category of constraints values (OS, arch)
- `toolchain`: a target with a toolchain type, an implementation, and execution and target constraint values. Must be registered.
- `toolchain_type`: a symbol that uniquely identifies a set of toolchains.
- Toolchain implementation: a target that provides toolchain files, functions, and metadata to rules that request the toolchain.

Exercise: toolchains

- `git switch v5_exercise`
- Exercise: go_toolchain
- Test: **`bazel test //...`**
- Check: **`git diff v5_exercise v5`**

Review: toolchains

- Useful for dynamically selecting part of a rule implementation, based on execution and target platform.
- Users can:
  - Write `platform` targets listing constraints.
  - Set target platforms with `--platforms`.
  - Register execution platforms with `register_execution_platforms`.
  - Register toolchains with `register_toolchains` in MODULE.bazel.
- `--toolchain_resolution_debug=.` to see what happens.
- Rule authors can:
  - Depend on toolchains.
  - Use `transition` to depend on a target with a different configuration.

Agenda

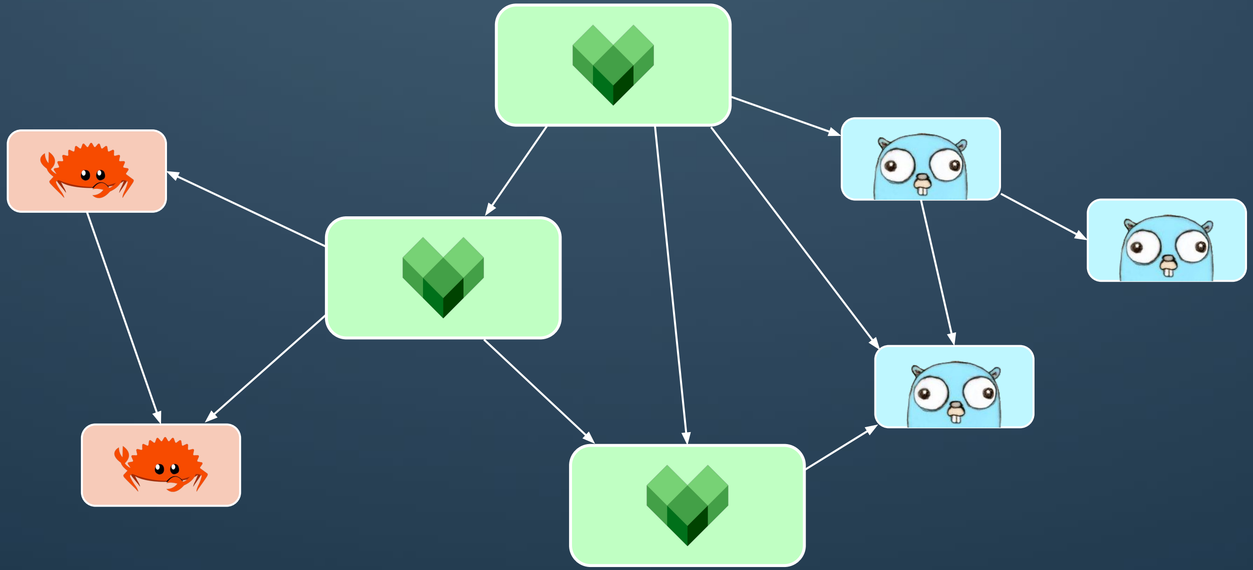
1. Concepts
2. Simple binary rule
3. Libraries, providers, depsets
4. Data, runfiles
5. Efficient execution
6. Repository rules
7. Toolchains
8. **Module extensions**

Problems

Earlier, we wrote the go_download rule and registered toolchains

1. `register_toolchains` forces download of each repo rule.
2. Multiple modules could call `register_toolchains`.
3. Every user lists all platforms they want to build for.

Bazel combines module graphs



Concepts

- As a polyglot tool, Bazel must integrate with other module systems.
- Module extension: a Starlark plugin to the module system
- *Tag class*: allows user to declare metadata in MODULE.bazel.
- *Implementation:* Starlark function.
  - Called once, globally.
  - Reads tags from all modules.
  - Reads files, runs commands, instantiates repository rules.

Usage in MODULE.bazel

```
# Load extension
go = use_extension("//:go.bzl", "go")

# Declare a tag
go.download(version = "1.25.0")

# Declare direct dependencies on repos (bazel mod tidy)
use_repo(go, "go_toolchains")

# Use symbols in repos
register_toolchains("@go_toolchains//:all")
```

Exercise: module extensions

- `git switch v6_exercise`
- Exercise: `go_ext.bzl`
- Test: **`bazel test //...`**
- Check: **`git diff v6_exercise v6`**

Review: module extensions

We didn't actually integrate with Go's module system.

```
go_deps = use_extension("@gazelle//:extensions.bzl", "go_deps")
go_deps.from_file(go_mod = "//:go.mod")
use_repo(
    go_deps,
    "com_github_bazelbuild_buildtools",
    "com_github_bmatcuk_doublestar_v4",
    "com_github_stretchr_testify",
    "org_golang_google_protobuf",
)
```

Review: toolchainization

- Goal: automatically configure and register toolchains.
  Don't download more than necessary.
- Collect tags across all MODULE.bazel files.
- Instantiate repo rule declaring all toolchains.
- Register toolchains in the rule set module.