Extending the design of WORKSPACE files

Klaus Aehlig, Dmitry Lomov
Status: IN REVIEW
Date: 2018-03-22
Last update: 2018-04-12

Extending the ideas of "A Brave New World for the Bazel WORKSPACE File", we propose the following design for freezing WORKSPACE dependencies.

Background

While the semantics of workspace rules allows for arbitrary behaviour, in practice, workspace rules are used only for two purposes:

- discovering and fetching external source code, and
- inspecting the execution platform (for local builds, this is the local machine) to find tools and libraries.

Those two use cases have very different update characteristics. The result of discovering the precise versions of external code can be shared between different machines and architectures; after this discovery, the result of fetching is strictly a function of discovery result. Discovering tools and libraries on the execution platform, on the other hand, have to be rerun on every execution platform; in particular, for local execution, the outcome of those rules cannot be shared as they contain paths specific for this very machine. But at least, inspection of the local machine is comparably cheap and does not require network access. These different characteristics suggest to handle those two kind of rules differently; obviously, this presupposes that rules can declare of which kind they are.

For external sources, it is often desirable to fix specific versions in order to have a reproducible build. However, those versions of external sources need to be updated regularly. Doing so by hand is cumbersome and mechanical work that can better be done by a machine.

In fact, even today, a lot of WORKSPACE files are generated by tools like bazel_deps. As those tools typically are restricted to a particular package manager for a particular language, we expect more of those tools to come into existence. For bazel to support multi-language builds, some coordination between those tools is necessary, and WORKSPACE files is bazel's interface for orchestrating these tools.

Even when not fixing the versions of external sources, it still is desirable to know after the fact against which version a binary was built or a test was run. This information is, e.g., used to determine in a CI system which updates broke a test and do proper bisecting from there.

Proposed Design

We propose the following extension to the current semantics of WORKSPACE files.

Source-like and configure-like workspace rules

The `repository_rule` constructor gets a new parameter called `configure_like` with default value `False`. So repository rules are considered to be source-like, unless they declare themselves as configure-like.

Source-like rules promise to have a way to produce reproducible output; in particular, their output has to be platform independent. They can do so by either being reproducible themselves, maybe with changed parameters,or by expanding to a list of reproducible rules. The latter is intended for rules interacting with package managers; the rules can read a description file, compute the transitive closure of dependencies, and expand to a list of packages (architecture independent (e.g., JVM languages), or one per architecture) that can all be fetched and cached independently. Another use case is recursive workspaces (once we introduce them, a separate design); here a recursive repository rules expands to a flat list of non-recursive repositories.

Configure-like rules may depend on the machine they are executed, but promise to be somewhat cheap and only inspect the machine they are executed on. Also, as these rules need to be run more often, we do not allow source-like rules to access file from "repositories" generated by configure-like rules.

WORKSPACE.resolved and WORKSPACE.configure

We introduce two new files, WORKSPACE.resolved and WORKSPACE.configure, usually located in the workspace directory; if the workspace directory is read-only, of symlink-prefix is `/`, they are generated in the output base. Both files weakly depend on the WORKSPACE file in the following sense. If either of the two files WORKSPACE.resolved or WORKSPACE.configure is missing, they are both generated from WORKSPACE. However, if present (either in the workspace directory or in the output base) they are used, even if the WORKSPACE file has changed. An INFO level message will indicate if the files are regenerated from WORKSPACE or a (and which) preexisting file is used.

WORKSPACE.resolved

The WORKSPACE.resolved is used to contain the authoritative information about external sources. It is generated when not present and needed by the requested command (fetch, query, build, test, run), or when a regeneration is explicitly requested by the (new) command `bazel sync`; a `bazel sync` will also remove all artifacts generated by a configure-like rules.

Generation of `WORKSPACE.resolved` file

The WORKSPACE.resolved file is generated by executing, and recording the return values of, all source-like rules in WORKSPACE file.

Every source-like rule reports the reproducible repositories it expands to by its return value:

1. If it is `None`, that signifies that the rule is already reproducible and it will expand to itself with its arguments unchanged.
2. If the return value is a list, it has to be the list of repository calls it expands to (i.e., bzl file, exported name, positional arguments, keyword arguments).
3. If the return value is a dict, then the rule expands to itself, but with the keyword arguments replaced by this dict.
4. If the return value is a string, that string will be parsed as a Skylark value, and then interpreted according to the rules 2-3. This is for the convenience of rules that essentially call out to external tools; if the external tool is aware of bazel's format, they can just return stdout ofthe call to the external tool.

The effect of execution of the source-like rule is:

- in cases 2-4, bazel will execute the repository calls as returned by the source rule implementations
- bazel will checksum the resulting directories
- bazel will record the repository calls and checksums in WORKSPACE.resolved files

The Format of `WORKSPACE.resolved` file

A valid WORKSPACE.resolved file contains a single Skylark value which is a list containing one entry for each source-like repository rule executed during the evaluation of the WORKSPACE file. The WORKSPACE.resolved file is generated (e.g., on a run of `bazel sync`) by evaluating the WORKSPACE file and recording the return values for every source-like repository rule executed.

For every evaluated source-like rule, the corresponding entry in WORKSPACE.resolved fil contains the following information:

- The full invocation of the originating rule in the WORKSPACE file (only workspace rules can produce entries in WORKSPACE.resolved and every workspace rule has a list of attributes; the value of all these attributes is recorded).

- The list of reproducible repository rules the workspace rule expanded to. Each entry consists of
    - a precise description of the call (i.e., bzl file, exported name, list of positional arguments, and dict of keyword arguments, as above), and
    - a checksum of the repository it produced.

If an already existing WORKSPACE.resolved file is used, it will be interpreted as if we had all the repository calls the various repository rules expanded to after each other, with duplicate values (literally, i.e., same bzl file and exported name, same arguments) removed. If such a rule is called (e.g., because of a checked-in WORKSPACE.resolved) the checksum of the produced repository is verified. A mismatch is either a warning or an error depending on the (new) flag `--{no,}repository_checksum_mismatch_is_error`. To allow a smooth transition from the current state, the default value for this flag is initially to warn only, but will default to fail in later releases after a transition period.

The reason we choose to have WORKSPACE.resolved a value, rather than an imperative program, is to allow tools to more easily read and interpret the file. Such potential tools can do so, e.g., to try to interpolate between two WORKSPACE.resolved files to support bisecting which external dependency update broke a test.

An example

WORKSPACE rule

```
git_repository(
   name = "build_bazel",
   remote = "https://github.com/bazelbuild/bazel.git",
)
```

determines a floating dependency on HEAD of bazelbuild/bazel github repo.
At "bazel sync" time, the return value of that rule would be:

```
[{ "rule_class" : "@bazel_tools//:git.bzl%git_repository",
   "attrs" : {
      "name" : "build_bazel",
      "remote" : ""https://github.com/bazelbuild/bazel.git",
      "commit": "7ed032e7da2245062040e20879400955fe775616",
      "shallow_since": "2017-10-06"}]
```

This invocation will be recorded in WORKSPACE.resolved as follows:

```
{ "original_rule_class" : "@bazel_tools//:git.bzl%git_repository",
  "original_attrs" : {
     "name" : "build_bazel",
     "remote" : "https://github.com/bazelbuild/bazel.git",
  },
  "repos" :
     [{ "rule_class" : "@bazel_tools//:git.bzl%git_repository",
        "attrs" : {
           "name" : "build_bazel",
           "remote" : "https://github.com/bazelbuild/bazel.git",
           "commit": "7ed032e7da2245062040e20879400955fe775616",
           "shallow_since": "2017-10-06"},
        "output_tree_hash": "108a92703be6e094998d345c464d6a66d9059bd7" }

     ]
}
```

WORKSPACE.configure

During a `bazel sync` (including implicit ones, e.g., when build is requested by no WORKSPACE.resolved is present) all the calls to configure-like workspace rules are recorded (as calls, in an imperative format) in the WORKSPACE.configure file.

A new command, `bazel configure` will re-execute the WORKSPACE.configure file unconditionally (considering all rules present there as invalidated). This can be used to adapt to a changed execution platform.

Rules in WORKSPACE.configure for the local machine are only invalidated on explicit request (calling `bazel configure`). Their outputs are placed in a known location (available via `bazel info`), and the users have the possibility to fix any wrong guesses by a configure-like rule before starting the build. This use case will be explicitly supported.

As an extension of this design (to follow up in a subsequent document), "bazel configure" might also run the configuration autodetection on a remote machine or on a different execution platform. The results of such execution (contents of repository directory) can then be made available to bazel, and stored keyed by execution platform identifier. With sufficient isolation (e.g. container images) these result can be made fully reproducible.

Partial "bazel sync"

"bazel sync" command will support partial operation:

```
  bazel sync <repo_name>....
```

The intention of this command is just sync one of the repos in WORKSPACE file, leaving the others intact. We propose the following semantics of this operation.

1. bazel reads the current WORKSPACE.resolved file
2. bazel executes WORKSPACE file, proceeding as normal with one difference:
    a. for all repository rules whose names are on the command line, they are executed as normal
    b. for all repository rules whose names are *not* on the command line:
        i. bazel searches for their invocation in current WORKSPACE.resolved
        ii. if the invocation of the rule is recorded in the current WORKSPACE.resolved and has the exact same arguments as in the current execution, the rule is not re-executed and the result is taken from the current WORKSPACE.resolved file
        iii. otherwise the rule is executed as normal
3. bazel overwrites the WORKSPACE.resolved file with the result of this execution.

This simple procedure allows us to correctly track dependencies between repos:

```
 git_repository(name = "foo", remote = "...")
 git_repository(name = "fux", remote = "...")
load("@foo//:version.bzl", "version")
other_repo(name = "bar", v = version)
```

When the user invokes "bazel sync foo", foo call will be re-executed in any case, "fux" call will not be re-executed, and "bar" call will only be re-executed if the version number loaded from "@foo//:version.bzl" has changed.