Project Dependencies and Structuring the Build

Klaus Aehlig

Oct 13, 2025

Intro

- I have some experience with software build ...
  - I use software build tools since 1998 and package-build tools since 2007
  - I work on build tools since 2016
    - SWE in the bazel team 2016–2020
    - Leading development of a new build tool from scratch 2020–2025 open-sourced as `https://github.com/just-buildsystem/justbuild` (not to be confused with other build tools with the same name)

Intro

- I have some experience with software build ...
  - I use software build tools since 1998 and package-build tools since 2007
  - I work on build tools since 2016
    - SWE in the bazel team 2016–2020
    - Leading development of a new build tool from scratch 2020–2025 open-sourced as `https://github.com/just-buildsystem/justbuild` (not to be confused with other build tools with the same name)

... and observed one (of several) re-occurring themes

For every non-trivial software project you need libraries, tools (compilers, code generators, linters ...), and/or frameworks for wich you are not upstream.

In fact, those dependencies are moving faster and faster.

Intro

- I have some experience with software build ...
  - I use software build tools since 1998 and package-build tools since 2007
  - I work on build tools since 2016
    - SWE in the bazel team 2016–2020
    - Leading development of a new build tool from scratch 2020–2025
      open-sourced as `https://github.com/just-buildsystem/justbuild`
      (not to be confused with other build tools with the same name)

... and observed one (of several) re-occurring themes

> *For every non-trivial software project you need libraries, tools (compilers, code generators, linters ...), and/or frameworks for wich you are not upstream.*

In fact, those dependencies are moving faster and faster.

- Let's look at (some of the) approaches to deal with this situation
  (from the perspective of a build tool)

Ignore

The most simple approach (for the build tool) is to ignore the problem.

Ignore

The most simple approach (for the build tool) is to ignore the problem.

- Just take it from host/`PATH`/...
  - Gets you into the "Works on my machine" problem

Ignore

The most simple approach (for the build tool) is to ignore the problem.

- Just take it from host/`PATH`/...
  - Gets you into the "Works on my machine" problem
  - However, this is the standard interface for package building
    (where the tool sets up the "host environment" in a well-defined way)
    $\rightsquigarrow$ every OSS project has to support this mode of building
    ... just not as the default for delopment!

Ignore

The most simple approach (for the build tool) is to ignore the problem.

- Just take it from host/`PATH`/...
  - Gets you into the "Works on my machine" problem
  - However, this is the standard interface for package building
    (where the tool sets up the "host environment" in a well-defined way)
    $\rightsquigarrow$ every OSS project has to support this mode of building
    ... just not as the default for delopment!
- Just use a mono-repo
  - Copy in source code (`git subtree` if you're fancy)
  - write build description ("only until everyone uses our build system ...")
    $\rightsquigarrow$ later rules for foreign build systems supported
  - Set up a team to maintain that (de-facto) fork
    Task: separate between upstream code and custom patches

Explore (Bazel ≈2018)

- Iterpret `WORKSPACE` as a `starlark` file
  - Imperative language, so latest assignment/definition wins
  - At `load` statements need to acces dependencies, so freeze there

Explore (Bazel ≈2018)

- Iterpret `WORKSPACE` as a `starlark` file
  - Imperative language, so latest assignment/definition wins
  - At `load` statements need to acces dependencies, so freeze there
- Transitive dependencies handled via the `deps`-pattern

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

http_archive(
  name = "com_example_foo",
  urls = ["https://example.com/foo/foo-1.2.3.tar.gz"],
  strip_prefix = "foo-1.2.3",
  sha256 = "0a6717e765818538f153ac27ca6cf97c4290cb25dd374017e1dc2bd0d6f6bf5c",
)

load("@com_example_foo//:deps.bzl", "foo_deps")
foo_deps()
```

with implementation

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

def foo_deps():
  if "com_example_bar" not in native.existing_rules():
    http_archive(name="com_example_bar", ...)
  ...
```

⇝ Depth-first traversal, with each project only declaring its direct dependencies

Explore (Bazel ≈2018), cont'd

- Could generate a transcript with resolved arguments and hash of repo tree

```
resolved = [
    ...
    {
        "original_rule_class": "@bazel_tools//tools/build_defs/repo:git.bzl%git_repository",
        "original_attributes": {
            "name": "com_google_protobuf",
            "remote": "https://github.com/google/protobuf",
            "branch": "master"
        },
        "repositories": [
            {
                "rule_class": "@bazel_tools//tools/build_defs/repo:git.bzl%git_repository",
                "output_tree_hash": "a776ce4f591327c6b23d88d367d6208a88af6ad889e08f7b86a0edfc76fcfd96",
                "attributes": {
                    "remote": "https://github.com/google/protobuf",
                    "commit": "a6e1cc7e328c45a0cb9856c530c8f6cd23314163",
                    "shallow_since": "2018-09-17",
                    "init_submodules": False,
                    "verbose": False,
                    "strip_prefix": "",
                    "patches": [],
                    "patch_tool": "patch",
                    "patch_args": [
                        "-p0"
                    ],
                    "patch_cmds": [],
                    "name": "com_google_protobuf"
                }
            }
        ]
    }

]
```

- Could use that instead of the `WORKSPACE` file

Explore (Bazel ≈2018), summary

- Imperatively discover dependencies in a depth-first way
- Clear operational semantics (freeze at `load` statements), but still caused confusion
  $\rightsquigarrow$ maybe operational semantics is not the right approach?
- Arguments resolved (e. g., branch to commit) to generate a lock file,
  hashes to verify
- Used in a global, unstructured, collection of repositories
  $\rightsquigarrow$ agreement on naming?

Declare (Bazel ≈2024)

- Still only care about direct dependencies, but only declare (no more manual coding of depth-first search)

```
module(
    name = "fmt",
    version = "11.2.0",
    bazel_compatibility = [">=7.2.1"],
    compatibility_level = 10,
)

bazel_dep(name = "rules_cc", version = "0.1.1")
bazel_dep(name = "rules_license", version = "1.0.0")
```

Declare (Bazel ≈2024)

- Still only care about direct dependencies, but only declare (no more manual coding of depth-first search)

```
module(
    name = "fmt",
    version = "11.2.0",
    bazel_compatibility = [">=7.2.1"],
    compatibility_level = 10,
)

bazel_dep(name = "rules_cc", version = "0.1.1")
bazel_dep(name = "rules_license", version = "1.0.0")
```

- Resolved via (one or more) registries, that also contain the source description

```
{
    "integrity": "sha256-cS13hosxUt1hjE1k+q3e/MWWX5D13m5tOdXdzQvoLUI=",
    "strip_prefix": "rules_cc-0.1.1",
    "url": "https://github.com/bazelbuild/rules_cc/releases/download/0.1.1/rules_cc-0.1.1.tar.gz",
    "patches": {
        "module_dot_bazel_version.patch": "sha256-2WyM/DEDM/Dj9MQU0Jl79eQ0Ju3QHGsgIuLO1OIwdt4="
    },
    "patch_strip": 1
}
```

Declare (Bazel ≈2024), cont'd

- Global resolution of dependencies to one version per module
  - solves the problem of linking different versions of a library
  - opens the problem of building a library against a different version of its dependency
- Modules use local names for their dependencies ("apparent" vs "canoncial" repository name)
  - Can have simple local names, no problems with naming conflicts
  - Enforcing restriction to the declared dependencies
- Lock files `MODULE.bazel.lock` with hashes
- Target evaluation as if it were a mono-repo

Another Declarative Approach (Justbuild ≈2022)

Meanwhile, at a different build system . . .

- Repositories use local names, bound in a global repository configuration

```
{ "repositories":
  { "":
    { "repository": {"type": "file", "path": "."}
    , "bindings": {"rules": "rules-cc"}
    }
  , "rules-cc":
    { "repository":
      { "type": "git"
      , "repository": "https://github.com/just-buildsystem/rules-cc.git"
      , "branch": "master"
      , "commit": "2f549d26cf8def0a9b3a74790137b0b75b0a21b8"
      , "subdir": "rules"
      }
    }
  }
}
```

Another Declarative Approach (Justbuild ≈2022)

Meanwhile, at a different build system . . .

- Repositories use local names, bound in a global repository configuration
- Can be generated from declaration of direct inputs

```
{ "repositories":
  { "":
    { "repository": {"type": "file", "path": "."}
    , "bindings": {"rules": "rules-cc"}
    }
  , "rules-cc":
    { "repository":
      { "type": "git"
      , "repository": "https://github.com/just-buildsystem/rules-cc.git"
      , "branch": "master"
      , "commit": "2f549d26cf8def0a9b3a74790137b0b75b0a21b8"
      , "subdir": "rules"
      }
    }
  }
}
```

```
{ "repositories":
  { "":
    { "repository": {"type": "file", "path": "."}
    , "bindings": {"rules": "rules-cc"}
    }
  }
, "imports":
  [ { "source": "git"
    , "url": "https://github.com/just-buildsystem/rules-cc.git"
    , "branch": "master"
    , "repos": [{"repo": "rules", "alias": "rules-cc"}]
    }
  ]
}
```

  - libraries are built precisely against the declared version of their dependencies
  - opens the problem of linking different version so the same library

⇝ slightly different design choices for dependency import, but also declarative

Another Declarative Approach (Justbuild ≈2022)

Meanwhile, at a different build system . . .

- Repositories use local names, bound in a global repository configuration
- Can be generated from declaration of direct inputs

```
{ "repositories":
  { "":
    { "repository": {"type": "file", "path": "."}
    , "bindings": {"rules": "rules-cc"}
    }
  , "rules-cc":
    { "repository":
      { "type": "git"
      , "repository": "https://github.com/just-buildsystem/rules-cc.git"
      , "branch": "master"
      , "commit": "2f549d26cf8def0a9b3a74790137b0b75b0a21b8"
      , "subdir": "rules"
      }
    }
  }
}
```

```
{ "repositories":
  { "":
    { "repository": {"type": "file", "path": "."}
    , "bindings": {"rules": "rules-cc"}
    }
  }
, "imports":
  [ { "source": "git"
    , "url": "https://github.com/just-buildsystem/rules-cc.git"
    , "branch": "master"
    , "repos": [{"repo": "rules", "alias": "rules-cc"}]
    }
  ]
}
```

  - libraries are built precisely against the declared version of their dependencies
  - opens the problem of linking different version so the same library

⇝ slightly different design choices for dependency import, but also declarative

- Separation of dependency import, fetch, and build (even different tools)

Using Repository Stucture in Evaluation (Justbuild ≈2022)

Evaluation, however, uses the repository structure . . .

- Have a graph of repositories, know the transitive dependencies
- Repositories often have pinned content (given by commit hash, archive hash, . . .)

⇝ If nothing have changed, can take target value from cache

Using Repository Stucture in Evaluation (Justbuild ≈2022)

Evaluation, however, uses the repository structure ...

- Have a graph of repositories, know the transitive dependencies
- Repositories often have pinned content (given by commit hash, archive hash, ...)

⇝ If nothing have changed, can take target value from cache

- We even have this as a service and avoid fetching dependencies
  - Repository content queries (commit hash, archive hash, ... → tree hash)
  - Target queries (repo graph (hash), configuration, target), answer hash, artifacts directly pushed to remote execution (built on the fly, if not cached already)

Using Repository Stucture in Evaluation (Justbuild ≈2022)

Evaluation, however, uses the repository structure . . .

- Have a graph of repositories, know the transitive dependencies
- Repositories often have pinned content (given by commit hash, archive hash, . . .)

⇝ If nothing have changed, can take target value from cache

- We even have this as a service and avoid fetching dependencies
  - Repository content queries (commit hash, archive hash, . . . → tree hash)
  - Target queries (repo graph (hash), configuration, target), answer hash, artifacts directly pushed to remote execution (built on the fly, if not cached already)
- Especially useful when bootstrapping
  (instead of more and more build images have one with `/bin/sh` and `tcc`, and build tools from there)

Using Repository Stucture in Evaluation (Justbuild ≈2022)

Evaluation, however, uses the repository structure . . .

- Have a graph of repositories, know the transitive dependencies
- Repositories often have pinned content (given by commit hash, archive hash, . . .)

⇝ If nothing have changed, can take target value from cache

- We even have this as a service and avoid fetching dependencies
  - Repository content queries (commit hash, archive hash, . . . → tree hash)
  - Target queries (repo graph (hash), configuration, target), answer hash, artifacts directly pushed to remote execution (built on the fly, if not cached already)
- Especially useful when bootstrapping (instead of more and more build images have one with `/bin/sh` and `tcc`, and build tools from there)
- Logical repositories can reside in the same `git` repository! ⇝ organize the project structure (think “reverse visibility”)

Logical Repositories in a Monorepo

- Even in a monorepo, there is an internal structure
  - differnt projects/teams
  - often formalized: code ownership, approval requirements, ...
- The dependency structure between those subprojects changes rarely (therefore, litte effort to maintain explicit description thereof)
  - compilers, tool chains
  - base libraries
  - frameworks
  - applications

Justbuild Target Value Example

```
$ just-mr analyse libbar
INFO: Performing repositories setup
INFO: Found 2 repositories involved
INFO: Setup finished, exec ["just","analyse","-C","/example/.home/.cache/...
INFO: Requested target is [["@","","","libbar"],{}]
INFO: Analysed target [["@","","","libbar"],{}]
INFO: Export targets found: 1 cached, 0 uncached, 0 not eligible for caching
INFO: Result of target [["@","","","libbar"],{}]: {
        "artifacts": {
          "libbar.a": {"data":{"file_type":"f","id":"d4b3d0780802611407dc...
        },
        "provides": {
          "compile-args": [
          ],
          "compile-deps": {
            "foo.hpp": {"data":{"file_type":"f","id":"c2f3fff7ff446f92f2a...
          },
          "debug-hdrs": {
          },
          "debug-srcs": {
          },
          "dwarf-pkg": {
          },
          "link-args": [
            "libbar.a",
            "libfoo.a"
          ],
          "link-deps": {
            "libfoo.a": {"data":{"file_type":"f","id":"e8d8b9899fbf552ef2...
          },
          "lint": [
          ],
          "package": {
            "cflags-files": {},
            "ldflags-files": {},
            "name": "bar"
          },
          "run-libs": {
          },
          "run-libs-args": [
          ]
        },
        "runfiles": {
          "bar.hpp": {"data":{"file_type":"f","id":"2bd3ee3212fd330137391...
        }
      }
$
```

Using Repository Stucture in Evaluation (Justbuild ≈2022), Prerequisites

- Concept of what a target looks like to others (indendent of its origin!)
  - artifacts (e. g., `libbar.a`)
  - `dev`-artifacts (e. g., `bar.hpp`)
  - additional usage information
    - link order
    - transitive artifacts needed for building, linking, ...
    - abstract graph nodes for reflection (e. g., `proto` library)
    - ...

  $\rightsquigarrow$ Notion of equality (Targets coinciding in this data must be indistinguishable)

Using Repository Stucture in Evaluation (Justbuild ≈2022), Prerequisites

- Concept of what a target looks like to others (indendent of its origin!)
  - artifacts (e. g., `libbar.a`)
  - dev-artifacts (e. g., `bar.hpp`)
  - additional usage information
    - link order
    - transitive artifacts needed for building, linking, ...
    - abstract graph nodes for reflection (e. g., `proto` library)
    - ...

  $\rightsquigarrow$ Notion of equality (Targets coinciding in this data must be indistinguishable)
- Target must only depend on its repo and the transitive dependencies thereof
  $\rightsquigarrow$ Notion of equality for repos
  - no comparison on target-references
  - would have to include global name if we did Bazel-style path mangling

Output paths

- Many build systems (including Bazel, ...) do path mangling
  - Command-line interpolation with `$(location ...)`
  - Leaking canonical repository name $\rightsquigarrow$ different actions if main/dep repo
  - Full branching on config transtions, even if only tiny part really depends

Output paths

- Many build systems (including Bazel, ...) do path mangling
  - Command-line interpolation with `$(location ...)`
  - Leaking canonical repository name $\rightsquigarrow$ different actions if main/dep repo
  - Full branching on config transtions, even if only tiny part really depends
- Why? To get unique outputpath (relative to the build root)
  - overlapping outputs check, confinement of outputs to package

Output paths

- Many build systems (including Bazel, ...) do path mangling
  - Command-line interpolation with `$(location ...)`
  - Leaking canonical repository name $\rightsquigarrow$ different actions if main/dep repo
  - Full branching on config transtions, even if only tiny part really depends
- Why? To get unique outputpath (relative to the build root)
  - overlapping outputs check, confinement of outputs to package
- Why? Becaues build is seen as imperative: Want a side-effect on the file system!
  - Synchronisation of processes; no two builds with same output root simultaneously

Output paths

- Many build systems (including Bazel, ...) do path mangling
  - Command-line interpolation with `$(location ...)`
  - Leaking canonical repository name $\rightsquigarrow$ different actions if main/dep repo
  - Full branching on config transtions, even if only tiny part really depends
- Why? To get unique outputpath (relative to the build root)
  - overlapping outputs check, confinement of outputs to package
- Why? Becaues build is seen as imperative: Want a side-effect on the file system!
  - Synchronisation of processes; no two builds with same output root simultaneously
- However our main model of computation, remote execution, is functional
  - Cachable input/output relation
  - actions independent, inputs can be placed anywhere in the logical space

Output paths

- Many build systems (including Bazel, ...) do path mangling
  - Command-line interpolation with `$(location ...)`
  - Leaking canonical repository name $\rightsquigarrow$ different actions if main/dep repo
  - Full branching on config transtions, even if only tiny part really depends
- Why? To get unique outputpath (relative to the build root)
  - overlapping outputs check, confinement of outputs to package
- Why? Becaues build is seen as imperative: Want a side-effect on the file system!
  - Synchronisation of processes; no two builds with same output root simultaneously
- However our main model of computation, remote execution, is functional
  - Cachable input/output relation
  - actions independent, inputs can be placed anywhere in the logical space
- Easy to make local builds model remote execution:
  fresh action directory, hard-link in, action, hard-link out; artifacts in CAS

Output paths

- Many build systems (including Bazel, ...) do path mangling
  - Command-line interpolation with `$(location ...)`
  - Leaking canonical repository name $\rightsquigarrow$ different actions if main/dep repo
  - Full branching on config transtions, even if only tiny part really depends
- Why? To get unique outputpath (relative to the build root)
  - overlapping outputs check, confinement of outputs to package
- Why? Becaues build is seen as imperative: Want a side-effect on the file system!
  - Synchronisation of processes; no two builds with same output root simultaneously
- However our main model of computation, remote execution, is functional
  - Cachable input/output relation
  - actions independent, inputs can be placed anywhere in the logical space
- Easy to make local builds model remote execution:
  fresh action directory, hard-link in, action, hard-link out; artifacts in CAS
- Break with `make` and use functional approach?! (e. g., `justbuild`)
  $\rightsquigarrow$ compute result, ask for specific output (or tell where to install)

Summary

- External dependency handling: went from operational to denotational semantics
- Splitting in (logical) repos can document code structure
- This additional structure can be used to keep target and action graph small
- This works best, when using a more functional understanding of build
  $\rightsquigarrow$ time to fully break compatibility with `make`?