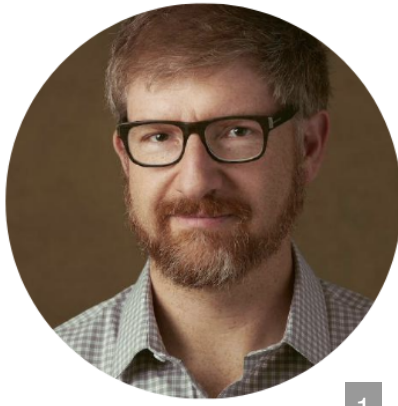
Aspect

DEVELOPMENT

rules_js

Build and test JavaScript programs with Bazel

Alex Eagle
Co-founder, Aspect.dev
alex@aspect.dev



Slides: https://hackmd.io/@aspect/rules_js

Bazel: most scalable polyglot Build System.

1️⃣ Introductions

2️⃣ Fetch and install npm packages

3️⃣ Runtime module resolutions

4️⃣ How to use rules_js

INTRODUCTIONS

- Alex Eagle
- Aspect Development
- Bazel
- NodeJS
- pnpm
- GH/bazelbuild/**rules_nodejs**

WHO IS ALEX EAGLE

- Worked at Google on DevInfra 2008-2020
- Bazel most of that time: TL for Google's CI, build/test results UI, Angular CLI
- twitter.com/jakeherringbone

WHAT IS ASPECT

I Co-founded Aspect Development to make Bazel the industry-standard full-stack build system

- https://aspect.dev - Support and consulting to help you adopt Bazel
- https://aspect.build - Products making Bazel easier to use
- https://github.com/aspect-build - rules_js is part of our Bazel rules ecosystem

WHAT IS BAZEL

- Build system for “every” langugage
- Incremental: re-build proportional to what you changed
- Cached/parallel: distribute over server farm
- Scalable: works for Google’s 2 billion line repo
- Unix Philosophy: just spawns subprocesses, which can be any tool

More: https://www.aspect.dev/resources

WHEN TO CONSIDER BAZEL FOR FRONTEND

- **Large-scale**: 1M SLOC / 100 devs
- **Monorepo**: same use cases as Nx/Rush/Lerna
- **Polyglot/full-stack**: parachute anywhere
- **Integration testing**: fast test against backend
- **Have a DevInfra team**: economy of scale

More: https://www.aspect.dev/resources

NONE OF THOSE APPLY?

Small, disconnected JS apps shouldn’t use Bazel.

The build system recommended by your framework is well supported for small-to-medium scale.

WHAT IS NODEJS

JavaScript engine that runs outside the browser.

Typically used for running dev tools to
build and test JavaScript programs.

WHAT IS `pnpm`

- “Fast, disk space efficient package manager”: https://pnpm.io/
- Works with nearly the whole ecosystem
- Used by the https://rushjs.io/ monorepo JS-only build tool
- Happens to fit perfectly with Bazel semantics!

WHAT IS `rules_nodejs`

Bazel rules forked from Google-internal

- toolchain to run hermetic NodeJS interpreter
- shared Bazel interfaces (“Providers”) like TypeScript `DeclarationInfo`

rules_js is a layer on `rules_nodejs`

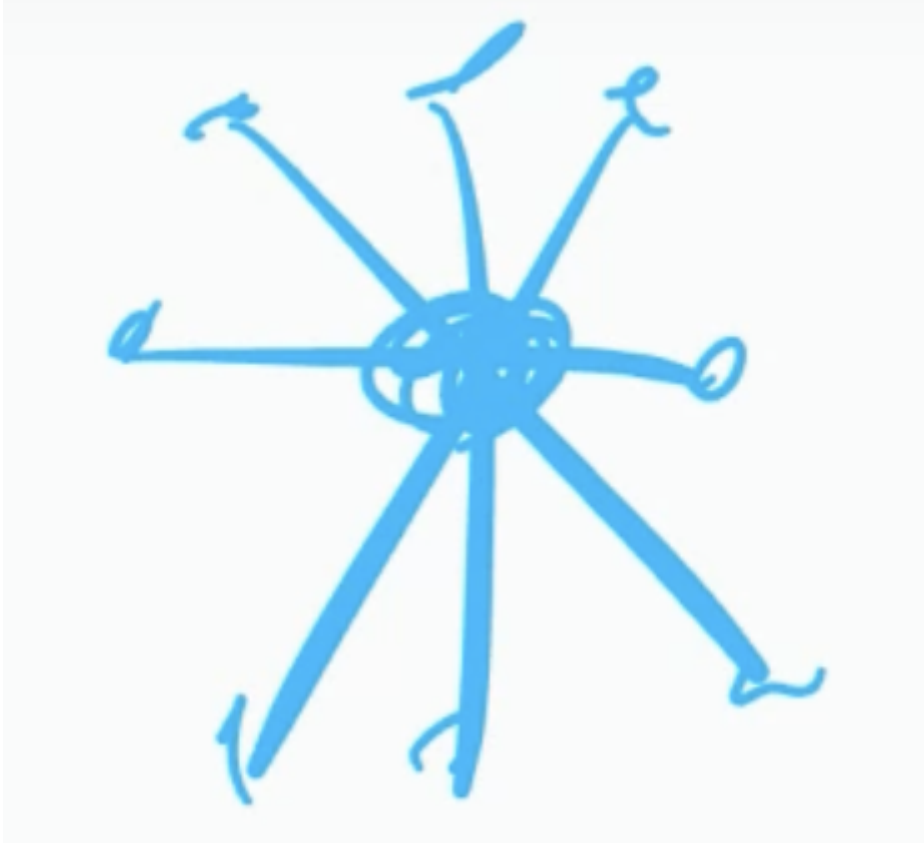
`build_bazel_rules_nodejs` is replaced

BUILD SYSTEMS:

MATRIX / HUB-AND-SPOKE

 The JS ecosystem took a wrong turn

- Grunt and Gulp fell out of favor
- Instead, each tool became a Build System
- Now each tool needs a plugin for each language

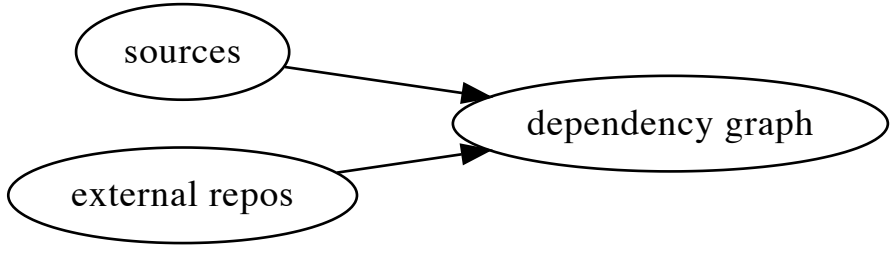


HOW BAZEL WORKS

In five minutes 🤔

BAZEL: LOADING PHASE

Load and evaluate all extensions, `BUILD` files and macros that are needed for the build.

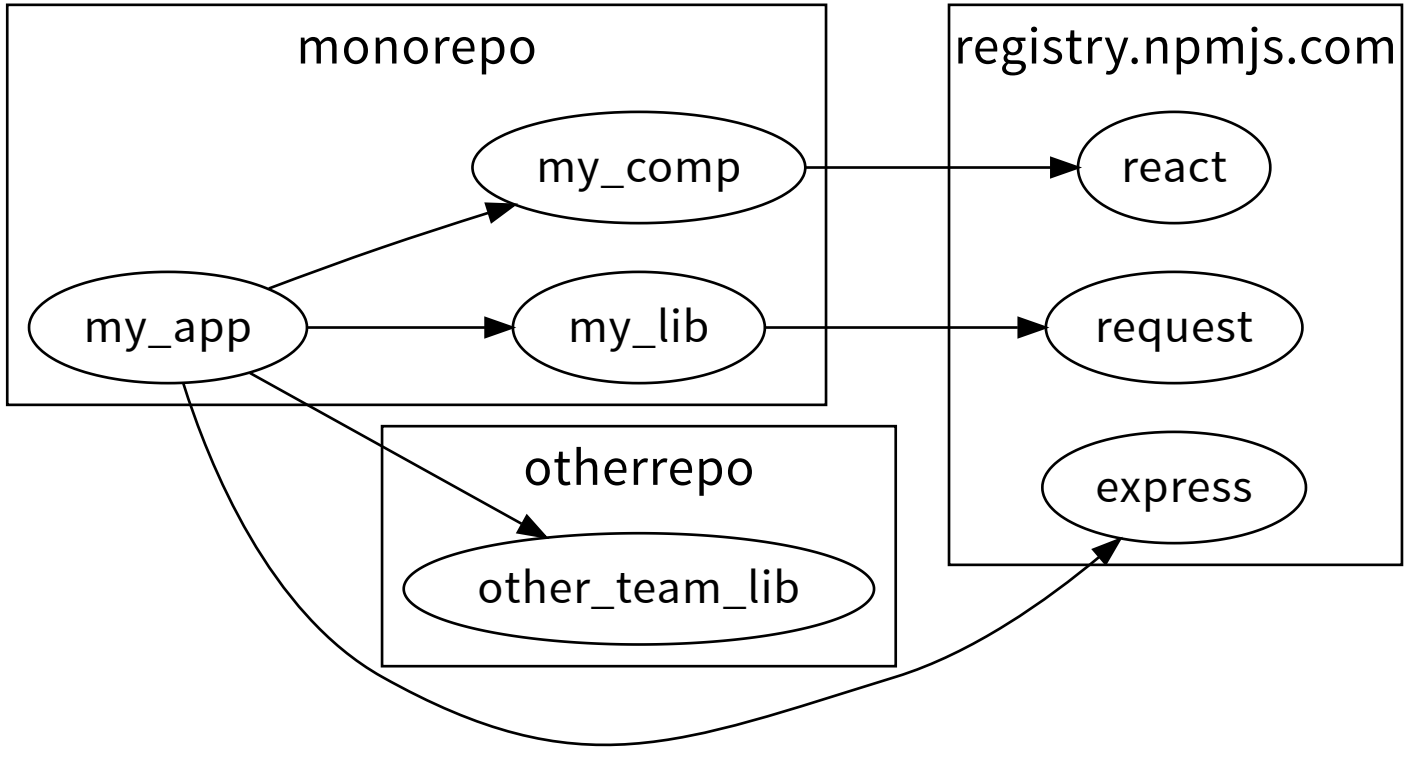


```
bazel fetch [targets]
```

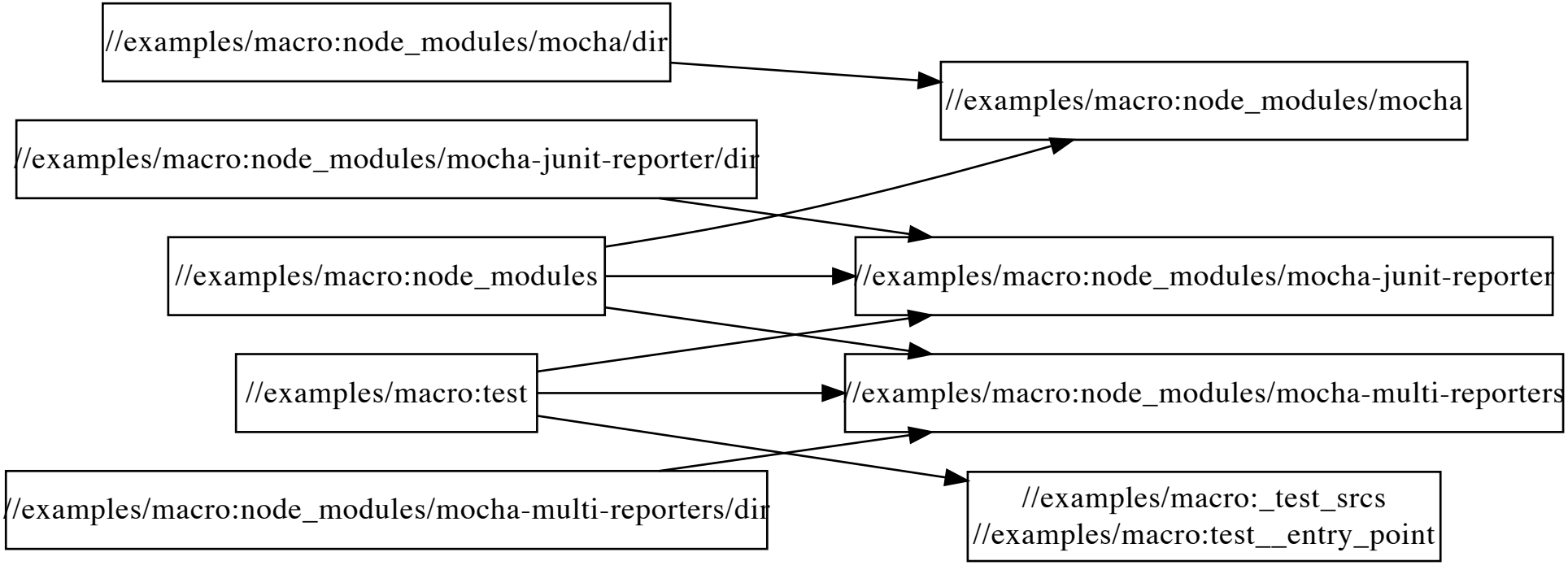
https://blog.aspect.dev/configuring-bazels-downloader

- Bazel is full-featured for fetching external deps
- Can air-gap, security scan, artifactory, etc
- Supply-chain secure, Trust-on-first-use model
- Cache based on integrity hashes

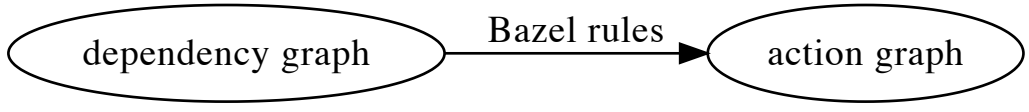
BAZEL: DEPENDENCY GRAPH



bazel query --output=graph [targets]



BAZEL: ANALYSIS PHASE

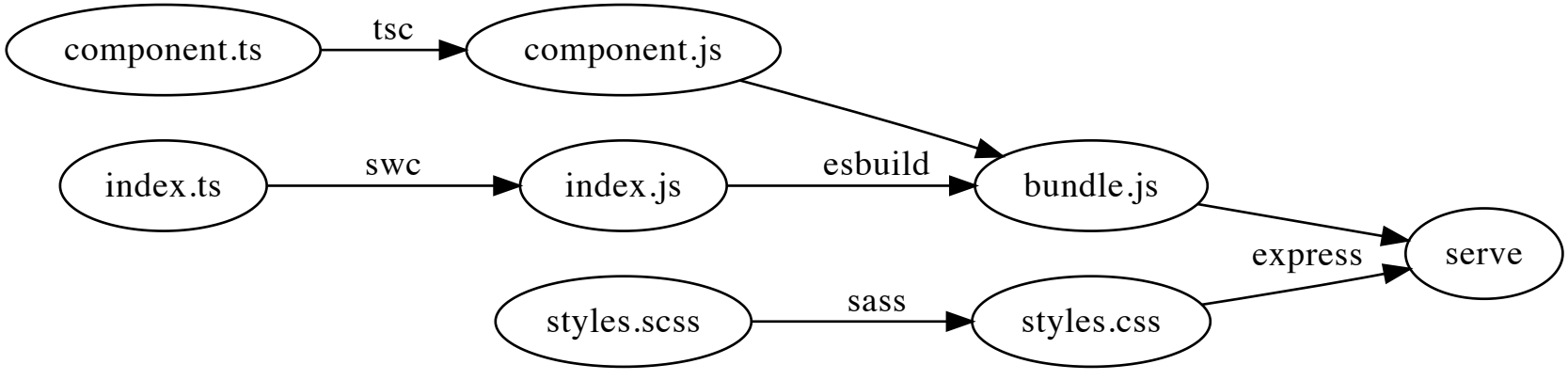


Action: for a requested output, how to generate it from some inputs and tools

e.g. “if you need `hello.js`, run `swc` on `hello.ts`”.

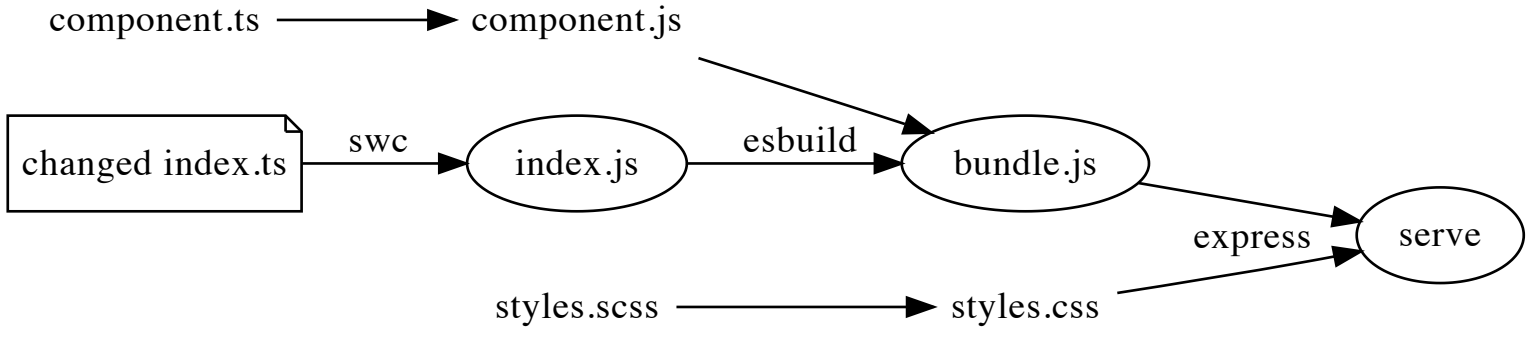
Requires predicting the outputs!

bazel aquery [targets]



BAZEL: EXECUTION PHASE

Execute a subset of the action graph by spawning subprocesses (e.g. `node`)



User requested certain targets be built.

Bazel is lazy and will only:

- fetch precise dependencies needed
- run actions required by the transitive dependency closure of those targets
- run actions that are a “cache miss”

2️⃣ FETCH AND INSTALL NPM PACKAGES

HOW NPM/YARN SOLVE IT

`npm install`

Install everything needed for the whole package/workspace

Any build/test script can depend on all npm packages 🙄

HOW GOOGLE SOLVES IT

Vendor the world: copy npm ecosystem sources into VCS

- Never fetches from the internet
- Never runs any package installation

You *could* do it this way too. 😉

HOW RULES_NODEJS SOLVED IT

Just wrap `[npm|yarn] install` - install the world

👎 Guaranteed slow when repo rule invalidates

👎 Extra bad when “eager fetching” npm deps

RULES_JS: IDEAL SOLUTION

👉 Port pnpm to Starlark 👈

- re-use pnpm's resolver (via lockfile)
- fetch with Bazel's downloader
- unpack tarballs with Bazel
- re-use `@pnpm/lifecycle` to run hooks
  - these are actions - can be remote cached
- link `node_modules`

https://blog.aspect.dev/rulesjs-npm-benchmarks

Best case:

- BUILD file declares fine-grained deps
- build only depends on one library
- we only fetch/install one library!

WORKSPACES

Mix of third-party and first-party deps in a tree of package.json files.

Google: single version policy
rules_nodejs: independent top-level dep installs
rules_js: supports pnpm workspaces!

3️⃣ RESOLVING NPM DEPENDENCIES AT RUNTIME

HOW IT WORKS IN NPM

NodeJS programs rely on a `node_modules` folder

“Was a big mistake” says NodeJS creator, and Deno fixes it (but here we are 🙄)

The location of `node_modules` is expected to be relative to the location of the importing script.

HOW GOOGLE SOLVES IT: PATCH `require`

Same strategy as “PnP”, e.g. Yarn PnP.

👎 not compatible. Many npm packages wrote their own `require` implementation.

HOW RULES_NODEJS SOLVES IT: RUNTIME “LINKER”

Similar to `npm link`: use symlinks to make monorepo libraries appear in the node_modules tree

👎 Slow beginning of every NodeJS spawn

👎 Links appear in source tree w/o sandbox

👎 Bins don’t work with `genrule/ctx.actions.run`

👎 Not compatible with “persistent workers”

HOW RULES_JS SOLVES IT

👉 Linker is now just a standard Bazel target 👈

Node.js tools assume the working dir is a single tree of src/gen/node_modules: we can do that!

- “link” to `bazel-bin/node_modules/...`
- copy sources to `bazel-bin`
- actions first `cd bazel-out/[arch]/bin`

4️⃣ HOW TO USE RULES_JS

Documentation and migration guide:
https://docs.aspect.build/rules_js

INSTALL

Copy the `WORKSPACE` snippet from latest release.

https://github.com/aspect-build/rules_js/releases

ADOPT PNPM

Just run `pnpm install` and check that your workflows work.

A few npm packages still have "hoisting bugs" where they don't declare correct dependencies and accidentally rely on npm or yarn-specific layout.

IMPORT `pnpm-lock.yaml`

`npm_translate_lock` converts to Bazel's format (Starlark).

`WORKSPACE`

```
load("@aspect_rules_js//npm:npm_import.bzl",
     "npm_translate_lock")
npm_translate_lock(
    name = "npm",
    pnpm_lock = "//:pnpm-lock.yaml",
)

# Load the starlark version of the lockfile
load("@npm//:repositories.bzl", "npm_repositories")
npm_repositories()
```

LINK THE NPM PACKAGES

`BUILD` (next to `package.json`)

```
load("@npm//:defs.bzl", "js_link_all_packages")

js_link_all_packages()
```

Result of `bazel build :all` is now

```
# the virtual store
bazel-bin/node_modules/.aspect_rules_js
 # symlink into the virtual store
bazel-bin/node_modules/some_pkg
# If you used pnpm-workspace.yaml:
bazel-bin/packages/some_pkg/node_modules/some_dep
```

```
bazel build examples/...
```

LINK FIRST-PARTY PACKAGES

First declare the package...

`my-lib/BUILD`

```
load("@aspect_rules_js//npm:defs.bzl", "npm_package")

npm_package(
    name = "lib",
    srcs = [
        "index.js",
        "package.json",
    ],
)
```

LINK FIRST-PARTY PACKAGES

... then link to `bazel-bin/node_modules` tree...

`app/BUILD`

```
load("@aspect_rules_js//npm:defs.bzl", "npm_link_package")

npm_link_package(
    name = "node_modules/@mycorp/mylib",
    src = "//examples/lib"
)
```

...then depend on it just like it came from npm!

app/BUILD

```
1 js_binary(
2     name = "my_app",
3     data = [
4         "//:node_modules/react-dom",
5         "//:node_modules/@mycorp/mylib",
6     ],
7     entry_point = "index.js",
8 )
```

RUNNING NPM TOOLS

1. Just call the `bin` entries from package.json
2. Write a macro wrapping a `bin` entry
3. Write a custom rule
4. Use an existing custom rule (e.g. rules_ts vs `tsc`)

There are also more advanced ways, see rules_js/examples

`bin` ENTRIES ARE PROVIDED FOR ALL PACKAGES

BUILD

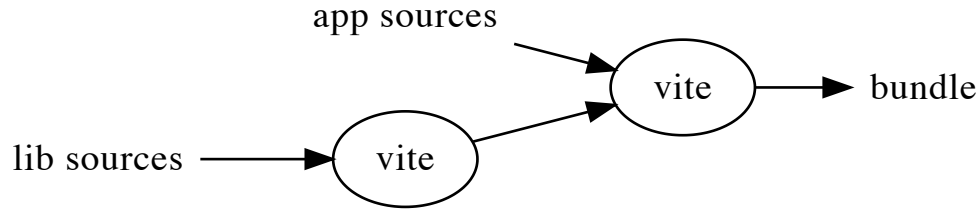
```
load("@npm//typescript:package_json.bzl", typescript_bin =

typescript_bin.tsc(
    name = "compile",
    srcs = [
        "fs.ts",
        "tsconfig.json",
        "//:node_modules/@types/node",
    ],
    outs = ["fs.js"],
    chdir = package_name(),
    args = ["-p", "tsconfig.json"],
)
```

Each bin exposes three rules:

Use	With	To
`foo`	`bazel build`	produce outputs
`foo_binary`	`bazel run`	side-effects
`foo_test`	`bazel test`	assert exit `0`

WRAP EXISTING BUILD SYSTEM

Use “component libraries” to get coarse granularity



Pretty fast developer loop in https://github.com/aspect-build/bazel-examples/tree/main/vue

```
ibazel run :vite
```

WRITE A MACRO

Bazel macros are like preprocessor definitions.

Good way to give “syntax sugar”,
compose a few rules, set defaults.

Indistinguishable from custom rules at use site

Example: `mocha` test

```
def mocha_test(name, srcs, args = [], data = [], env = {}
    bin.mocha_test(
        name = name,
        args = [
            "--reporter",
            "mocha-multi-reporters",
            "--reporter-options",
            "configFile=$(location //examples/macro:mocha_
            native.package_name() + "/*test.js",
        ] + args,
        data = data + srcs + [
            "//examples/macro:mocha_reporters.json",
            "//examples/macro:node_modules/mocha-multi-re
            "//examples/macro:node_modules/mocha-junit-re
```

https://github.com/aspect-build/rules_js/blob/main/examples/macro/mocha.bzl

WRITE A CUSTOM RULE

Harder and not recommended for most users.

Start from
https://bazel.build/rules/rules-tutorial
and use
https://github.com/bazel-contrib/rules-template

USE AN EXISTING CUSTOM RULE

From https://github.com/aspect-build:

- rules_esbuild - Bazel rules for https://esbuild.github.io/ JS bundler
- rules_terser - Bazel rules for https://terser.org/ - a JavaScript minifier
- rules_swc - Bazel rules for the swc toolchain https://swc.rs/
- rules_ts - Bazel rules for the tsc compiler from http://typescriptlang.org

- rules_webpack - Bazel rules for webpack bundler https://webpack.js.org/
- rules_rollup - Bazel rules for https://rollupjs.org/ - a JavaScript bundler
- rules_jest - Bazel rules to run tests using https://jestjs.io
- rules_deno - Bazel rules for Deno http://deno.land

... and many more by other vendors

http://docs.aspect.build

Catalog coming soon at https://bazel-contrib.github.io/SIG-rules-authors/

EXAMPLE CUSTOM RULE: `ts_project`

No more `rootDirs` in `tsconfig.json` 😁

BUILD

```
load("@bazel_skylib//rules:write_file.bzl", "write_file")

# Create a test fixture that is a non-trivial sized TypeS
write_file(
    name = "gen_ts",
    out = "big.ts",
    content = [
        "export const a{0}: number = {0}".format(x)
        for x in range(100000)
    ],
)
```

BUILD

```
load("@aspect_rules_ts//ts:defs.bzl", "ts_project")

ts_project(
    name = "tsc",
    srcs = ["big.ts"],
    declaration = True,
    source_map = True,
)
```

ts_project with custom transpiler

BUILD

```
load("@aspect_rules_swc//swc:defs.bzl", "swc_transpiler")

ts_project(
    name = "swc",
    srcs = ["big.ts"],
    out_dir = "build-swc",
    transpiler = partial.make(
        swc_transpiler,
        args = ["--env-name=test"],
        swcrc = ".swcrc",
    ),
)
```

Benchmarks: ts_project w/ SWC

https://blog.aspect.dev/rules-ts-benchmarks

Transpile-only use case on large project

```
bazel build :devserver
```

PUTTING IT ALL TOGETHER

Sophisticated teams can assemble their own toolchain.

Create an entire JS build system just by composing existing tools in a macro!

Example: an entire custom build system called “differential loading”:

ROADMAP

rules_js 1.0.0 is available now

Coming soon ™

- Gazelle extension to generate `BUILD` files from srcs
- Bazel 6.0 package manager: bzlmod instead of `WORKSPACE`
  https://blog.aspect.dev/bzlmod

THANK YOU!

These slides: https://hackmd.io/@aspect/rules_js

Thanks conference organizers and everyone who helped launch rules_js.

Come work with us on OSS!
http://aspect.dev/careers
Paid support and consulting: http://aspect.dev
Our projects: github.com/aspect-build