Lockfile trick: Package an npm project with Nix in 20 lines

December 18, 2022

I recently needed to package a JavaScript npm project with Nix. This is very simple to do and does not require any ugly workarounds like importing from derivation or generating Nix files and works just fine with sandboxed builds. Here's the general idea and an example.

The idea is to use the project lockfile (`package-lock.json`) to tell Nix how to fetch the dependencies. Then we use the project's own tools (e.g. `npm`) to actually perform the build.

The idea is not new and I've talked about it before in my 2019 NixCon talk: An Overview of Language Support in Nix. It's the idea powering naersk and napalm.

> NOTE: When I first wrote napalm, the `npm cache` command wasn't very stable and hence the solution I describe here is much simpler, more lightweight (i.e. does not require an npm registry written in Haskell …) and more reliable. Also with time I've come to realize that mileage does vary significantly from project to project, so instead of releasing yet another opinionatted library like napalm or naersk I'll just explain the idea and suggest you just copy/paste it to your projects. :)

The files

Let's start with some basics. Most `npm` projects contain at least two files:

- `package.json`: this describes your project, by giving it a name and specifying some dependencies with loose versioning.
- `package-lock.json`: this is the npm lockfile that specifies exactly which versions of dependencies are used.

Here's a simple `package.json` (the project description) that simply specifies two dependencies, namely the `typescript` compiler, and the corresponding LSP server:

```
{
  "name": "ts-env",
  "author": "",
  "license": "ISC",
  "dependencies": {
    "typescript": "^4.9.4",
    "typescript-language-server": "^2.2.0"
  }
}
```

Nothing fancy going on. What's of real interest to us though is the lockfile:

```
{
  "name": "ts-env",
  "version": "1.0.0",
  "lockfileVersion": 2,
  "requires": true,
  "packages": {
    "": {
      "name": "ts-env",
      "dependencies": { ... }
    },
    "node_modules/commander": {
      "version": "9.4.1",
      "resolved": "https://registry.npmjs.org/commander/-/commander-9.4.1.tgz",
      "integrity": "sha512-
5EEkTNyHNGFPD2H+c/dXXfQZYa/scCKasxWcXJaWnNJ99pnQN9Vnmqow+p+PlFPE63Q6mThaZws1T+HxfpgtPw==",
      },
    ...
  }
}
```

This is a stripped down version of the actual lockfile, but as you can see each dependency specifies its `npmjs.org` URL (the registry where packages are uploaded to

and downloaded from) and an `integrity` field, which is the hash of the tarball (that can be found at the URL `resolved`). We'll use those two fields (`resolved` and `integrity`) to tell Nix exactly how to download the dependencies.

Note that the lockfile (`package-lock.json`) can be generated from the `package.json` by running `npm install`.

Fetching the dependencies as derivations

As mentioned above, each dependency in the lockfile has a URL (`resolved`) and a hash (`integrity`). These can be used as arguments of nixpkgs' `fetchurl` to turn the URL into a derivation.

However in order to be able to generate the `fetchurl` derivations we need Nix to know about those values. For that we use `builtins.readFile` and `builtins.fromJSON` to read `package-lock.json` as a Nix attribute set. Then we can read the interesting fields of the lockfile (`packages` and `dependencies`) and clean it up a bit (in particular we drop the `""` package, which is basically the local package, which of course does not have a URL or hash since… it's local).

Here's what this looks like:

```
let
  pkgs = import <nixpkgs> { };

  # The path to the npm project
  src = ./.;

  # Read the package-lock.json as a Nix attrset
  packageLock = builtins.fromJSON (builtins.readFile (src + "/package-lock.json"));

  # Create an array of all (meaningful) dependencies
  deps = builtins.attrValues (removeAttrs packageLock.packages [ "" ])
    ++ builtins.attrValues (removeAttrs packageLock.dependencies [ "" ])
  ;

  # Turn each dependency into a fetchurl call
  tarballs = map (p: pkgs.fetchurl { url = p.resolved; hash = p.integrity; }) deps;

  # Write a file with the list of tarballs
  tarballsFile = pkgs.writeTextFile {
    name = "tarballs";
```

```
    text = builtins.concatStringsSep "\n" tarballs;
  };
in
```

Note that as the last step we wrote the list of `tarballs` to a file (`tarballsFile`). The file really just contains a list of derivations/store paths which we'll need to read from during the actual build:

```
/nix/store/pp5fx4grxk686dwdsrp7i39pbc2lsznx-commander-9.4.1.tgz
/nix/store/cqhbvmj5ny6nxgn40lgd1ma2r8lnd6p0-crypto-random-string-4.0.0.tgz
/nix/store/6cs6lnz1x96y68nwm7fqc5k9j5a7f2lv-type-fest-1.4.0.tgz
/nix/store/38g318wbnnqlpfmg66pdbvkhv9883v58-deepmerge-4.2.2.tgz
/nix/store/v8dwmkyar0p5g12ia0ikmyb8dhcim5jg-find-up-6.3.0.tgz
...
```

Speaking of "actual build", what would that look like? Well first, we populate an npm cache with all the dependencies. We don't do this ourselves but instead use the `npm cache` command, which takes a tarball, computes its checksum (hash), opens it up to figure some details like its project name etc (each tarball has a `package.json`) and then stores it in a cache somewhere (we don't have to care about the details). Then we run `npm ci` which reads the lockfile and installs the necessary dependencies; in this case however the dependencies are not downloaded but fetched directly from the cache.

```
pkgs.stdenv.mkDerivation {
  inherit (packageLock) name version;
  inherit src;
  buildInputs = [ pkgs.nodejs ];
  buildPhase = ''
    export HOME=$PWD/.home
    export npm_config_cache=$PWD/.npm
    mkdir -p $out/js
    cd $out/js
    cp -r $src/. .

    while read package
    do
      echo "caching $package"
      npm cache add "$package"
    done <${tarballsFile}

    npm ci
```

```
    '';

    installPhase = ''
      ln -s $out/js/node_modules/.bin $out/bin
    '';
  }
```

And that's pretty much it. A quick `nix-build` later and you're able to access all those sweet packages:

```
$ nix-build
this derivation will be built:
  /nix/store/58ad9kzwsqr7gkmjhwf40ahfjvzxphwa-ts-env.drv
building '/nix/store/58ad9kzwsqr7gkmjhwf40ahfjvzxphwa-ts-env.drv'...
... nixpkgs stuff
building
caching /nix/store/pp5fx4grxk686dwdsrp7i39pbc2lsznx-commander-9.4.1.tgz
caching /nix/store/cqhbvmj5ny6nxgn40lgd1ma2r8lnd6p0-crypto-random-string-4.0.0.tgz
... more dependencies being cached

[##################] reify:typescript-language-server: ... npm output
added 34 packages in 430ms

12 packages are looking for funding
  run `npm fund` for details

$ ./result/bin/tsc --help
tsc: The TypeScript Compiler - Version 4.9.4

COMMON COMMANDS

  tsc
  Compiles the current project (tsconfig.json in the working directory.)
...
```

Voilà! By reading information from the lockfile from Nix we're able to set up an npm environment with the sandbox enabled and without any import from derivation or need to generate Nix files.

This can be tweaked to instead e.g. build an npm project with `npm run build` or do whatever you might need to do. Again, I discuss the more general idea in this talk, and something based on this is implemented in napalm but really I recommend just copy pasting the code here and adapting it to your needs.

Like what I do? Let me know and support me!
Thank you ❤️
