BUILDING NIX FLAKES FROM RUST WORKSPACES

22 September 2022 — by Tor Hovland

nix rust webassembly

Did you know that with Nix you can easily define and load development environments with all the tools that you need, without having to install anything (except Nix) on your local machine? It may be as simple as this `shell.nix` file:

```
{ pkgs ? import <nixpkgs> {} }:
pkgs.mkShell {
  buildInputs = with pkgs; [ rustc cargo cargo-flamegraph ];
}
```

Now, if you run `nix-shell`, you'll be given a shell with Rust, Cargo, and Cargo Flamegraph available to you. That's pretty neat, but what if you want to take it a step further, and use Nix to package your Rust code? There are many options available, with different trade offs, and it can be quite overwhelming to choose between them, although there is some information on the NixOS Wiki. In this post we are going to try the different options on a simple but not entirely trivial Rust code sample.

But why Nix? Cargo uses lock files and does a good job of keeping track of Rust dependencies. But Nix goes further, also taking into account both system dependencies and the Rust compiler itself. Also, in a polyglot environment, Nix can simplify the build process by not requiring a concoction of compilers and tools to be installed.

Our Rust code includes:

- an app that we want to compile into a native executable
- a WebAssembly library
- a common package used by both of the above

We also want to be able to build Nix flakes, for the *hermetic* packaging they provide.

The complete sample code for each Nix packaging variant is available on GitHub.

THE RUST CODE SAMPLE

The common package fetches cat images, like this:

```rust
use serde::Deserialize;

#[derive(Deserialize)]
pub struct Cat {
    pub url: String
}

pub async fn fetch_cats() -> Result<Vec<Cat>, reqwest::Error> {
    Ok(reqwest::get("https://api.thecatapi.com/v1/images/search")
        .await?
        .json::<Vec<Cat>>()
        .await?)
}
```

As you can see, this code depends on `serde` as well as `reqwest`.

The native app calls this code and prints the URL of the first cat found, simply assuming that at least one cat was retrieved.

```rust
use std::error::Error;
use cats;

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let cats = cats::fetch_cats().await?;
    println!("There's a cat at {}", cats[0].url);
    Ok(())
}
```

The WebAssembly code does something very similar, but with explicit error handling in order to simplify the interface provided to any Javascript client code.

```rust
#[wasm_bindgen]
```

```
pub async fn cat_url() -> String {
    let cats = cats::fetch_cats().await.expect("cat response");
    cats[0].url.to_string()
}
```

When these packages are not connected to a single Cargo workspace, the native app can be built by running `cargo build` in the `app` package directory. The WebAssembly package can also be built like that, but it will get compiled into native code, which is not what we want. Instead, we build it by running `cargo build --target wasm32-unknown-unknown` in the `wasm` package directory.

If we are using a Cargo workspace, however, things are a little different. Now we can build the entire workspace by running `cargo build` in the root directory, but as you may imagine, this will compile everything into native code. Of course, running `cargo build --target wasm32-unknown-unknown` isn't going to help, because it will try to compile the native app into WebAssembly, which doesn't work and is not what we want either.

There are two ways to fix this. We can either go into each package directory and run the appropriate commands as before, or we can specify individual workspace members like this:

```
cargo build -p app
cargo build -p wasm --target wasm32-unknown-unknown
```

Now let's see how we can achieve the same from within a Nix flake.

THE FLAKE

We'll define our Nix flake like this:

```
{
  description = "A flake for building a Rust workspace using buildRu

  inputs = {
    rust-overlay.url = "github:oxalica/rust-overlay";
    flake-utils.follows = "rust-overlay/flake-utils";
    nixpkgs.follows = "rust-overlay/nixpkgs";
  };

  outputs = inputs: with inputs;
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = nixpkgs.legacyPackages.${system};
        code = pkgs.callPackage ./. { inherit nixpkgs system rust-ove
      in rec {
        packages = {
          app = code.app;
          wasm = code.wasm;
          all = pkgs.symlinkJoin {
            name = "all";
            paths = with code; [ app wasm ];
          };
        default = packages.all;
        };
      }
    );
}
```

What's nice about this flake is that we can essentially reuse it for any of the variants we will be trying later. The most interesting part is where we make it call `./.`, which makes it look for a `default.nix` file. This is where we put everything that is specific to the tool we are using. The output of `default.nix` is expected to be a derivation called `app` and another one called `wasm`. As you can see, we define both of these as flake output packages, but we also define an output called `all` which contains both. We set this as the default package, so that when we run `nix build`, we actually get everything.

BUILDRUSTPACKAGE

The first possibility for setting up `default.nix` is to use `buildRustPackage`, which is built-in into `nixpkgs`. We can build the native app like this:

```
app = pkgs.rustPlatform.buildRustPackage {
  pname = "app";
  version = "0.0.1";
  src = ./.;
  cargoBuildFlags = "-p app";

  cargoLock = {
    lockFile = ./Cargo.lock;
  };

  nativeBuildInputs = [ pkgs.pkg-config ];
  PKG_CONFIG_PATH = "${pkgs.openssl.dev}/lib/pkgconfig";
};
```

The WebAssembly code is not as straightforward, though, because `buildRustPackage` insists on setting the `--target` flag to either the (native) host system, or to whatever we're cross-compiling against. The cross-compilation should actually work, but when configuring it, Nix ends up building a Rust compiler from scratch where the target is set to WebAssembly everywhere. This eventually fails.

Instead, we have to override the `cargo build` step like this:

```
wasm = rustPlatformWasm.buildRustPackage {
  pname = "wasm";
  ...

  buildPhase = ''
    cargo build --release -p wasm --target=wasm32-unknown-unknown
  '';
  installPhase = ''
    mkdir -p $out/lib
    cp target/wasm32-unknown-unknown/release/*.wasm $out/lib/
    '';
  };
```

Please note `rustPlatformWasm` here, which uses the Rust overlay to get a toolchain with support for the `wasm32-unknown-unknown` target. See the code repository for details.

While buildRustPackage works, it is quite basic. Since each app corresponds to a single Nix derivation, if anything at all changes, such as source code, dependencies, or Nix config, the app needs to be rebuilt entirely.

Interestingly, it also only seems to work when the Rust code is in a workspace. When it's just three separate packages, the path dependency to `../cats` leads to a build error. This is the case for the WebAssembly app, where we override `buildPhase`, as well as for the native app, where we don't.

Now let's see if any of the other tools can do a better job, starting with naersk.

NAERSK

In order to use naersk we add it to our flake inputs, and pass it to our `default.nix` file:

```
naersk.url = "github:nix-community/naersk";
...
code = pkgs.callPackage ./. { inherit nixpkgs system naersk rust-ove
```

The real changes are in `default.nix`, but even here it doesn't look dramatically different from before:

```
let
  ...
```

```
    naerskLib = pkgs.callPackage naersk {};

    naerskLibWasm = pkgs.callPackage naersk {
      rustc = rustWithWasmTarget;
    };
  in {
    app = naerskLib.buildPackage {
      name = "app";
      src = ./.;
      cargoBuildOptions = x: x ++ [ "-p" "app" ];
      nativeBuildInputs = [ pkgs.pkg-config ];
      PKG_CONFIG_PATH = "${pkgs.openssl.dev}/lib/pkgconfig";
    };
    wasm = naerskLibWasm.buildPackage {
      name = "wasm";
      src = ./.;
      cargoBuildOptions = x: x ++ [ "-p" "wasm" ];
      copyLibs = true;
      CARGO_BUILD_TARGET = wasmTarget;
    };
  }
```

It’s nice that naersk lets us use `CARGO_BUILD_TARGET`, so we don’t have to override the build and install phase, like we had to for buildRustPackage. Even nicer is that naersk splits the app code and the third-party dependencies into separate derivations, so as long as we don’t update any dependencies, builds are fast. We can even modify our local `cats` dependency without triggering a full build.

However, like before, we can not get the build to work unless our packages are structured inside a Cargo workspace. This is a known issue.

CRANE

We can add crane to our flake inputs like this:

```
crane.url = "github:ipetkov/crane";
```

As crane is heavily inspired by naersk, it is no surprise that it works very similarly:

```
let
  ...

  craneLib = crane.mkLib pkgs;
  craneLibWasm = craneLib.overrideToolchain rustWithWasmTarget;
in
{
  app = craneLib.buildPackage {
    src = ./.;
    cargoExtraArgs = "-p app";
    nativeBuildInputs = [ pkgs.pkg-config ];
    PKG_CONFIG_PATH = "${pkgs.openssl.dev}/lib/pkgconfig";
  };
  wasm = craneLibWasm.buildPackage {
    src = ./.;
    cargoExtraArgs = "-p wasm --target ${wasmTarget}";

    # Override crane's use of --workspace, which tries to build ever
    cargoCheckCommand = "cargo check --release";
    cargoBuildCommand = "cargo build --release";
  };
}
```

Out of the box, the WebAssembly build doesn't work because crane runs Cargo with the `--workspace` flag, which means that it tries to also build the native app to WebAssembly. Luckily, we can override this using `cargoCheckCommand` and `cargoBuildCommand`.

Like naersk, crane splits the app code and the third-party dependencies into separate derivations, allowing for fast builds as long as dependencies aren't updated. And, once again, trying to build separate packages outside of a workspace fails.

Where crane is trying to improve on naersk is to make it easier to compose different Cargo invocations as completely separate derivations. For example, you can have one derivation that builds all your dependencies, and additional derivations for running Clippy, building the code, running tests with code coverage, etc. These can of course depend on each other, and Nix will make sure that you don't have to wait for output that has already been built.

CARGO2NIX

Let's now take a look at cargo2nix. Like before, we need to add a flake input:

```
cargo2nix.url = "github:cargo2nix/cargo2nix/release-0.11.0";
```

However, due to a fix we'll come back to in a moment, we're using our own fork for now.

While the other tools parsed `Cargo.lock` implicitly, with cargo2nix we need to explicitly generate a `Cargo.nix` file like this:

```
nix run github:cargo2nix/cargo2nix
git add Cargo.nix
```

Building the native app is quite straightforward:

```
let
  pkgs = import nixpkgs {
    inherit system;
    overlays = [cargo2nix.overlays.default];
  };
  rustPkgs = pkgs.rustBuilder.makePackageSet {
    rustVersion = "1.61.0";
    packageFun = import ./Cargo.nix;
  };
in {
  app = (rustPkgs.workspace.app {}).bin;
}
```

Building the WebAssembly should also have been as easy as this:

```
rustWithWasmTarget = pkgs.rust-bin.stable.${rustVersion}.default.ove
    targets = [ wasmTarget ];
};

rustPkgsWasm = pkgs.rustBuilder.makePackageSet {
  rustVersion = "1.61.0";
  packageFun = import ./Cargo.nix;
  rustToolchain = rustWithWasmTarget;
  target = wasmTarget;
};
```

Unfortunately, this exposes a bug in the way cargo2nix handles target-specific dependencies. It skips native-only dependencies if the *host* platform is `wasm32`, but it should really check the *target* platform. We can work around this by specifying a cross-system that is `wasm32`, and the one supported by Nix is `wasm32-wasi`:

```
pkgsWasm = import nixpkgs {
  inherit system;
  crossSystem = {
    system = "wasm32-wasi";
    useLLVM = true;
  };
  overlays = [cargo2nix.overlays.default];
};
```

There is another problem you may run into, because cargo2nix now thinks you're building for `wasm32-unknown-wasi`, and not `wasm32-unknown-unknown`. Your `Cargo.nix` file may contain dependencies like this:

```
${ if hostPlatform.parsed.kernel.name == "wasi" then "getrandom" els
```

This means we are now getting dependencies we are not supposed to get, and which fail to build. We need to guide cargo2nix to the correct kernel name here:

```
packageFun = attrs: import ./Cargo.nix (attrs // {
  hostPlatform = attrs.hostPlatform // {
    parsed = attrs.hostPlatform.parsed // {
      kernel.name = "unknown";
    };
  };
});
```

You may wonder if we cannot similarly work around the problem mentioned above, and altogether skip the `crossSystem` config, by simply setting `cpu.name = "wasm32"`. Unfortunately, I have not had success with that. Anyway, we can finally get our WebAssembly:

```
wasm = (rustPkgsWasm.workspace.wasm {}).out;
```

This wouldn't actually produce any WebAssembly output, but with the help from my colleagues Alexei Drake and Yorick van Pelt we were able to submit a fix.

And for the first time, we managed to also build the code as separate crates, not within a common workspace. All we needed to do was to delete the top-level Cargo files, generate `Cargo.lock` and `Cargo.nix` for both `/app` and `/wasm`, and update the references to `Cargo.nix` in `default.nix`.

Like naersk and crane, our own code and the dependencies are split into separate derivations, but cargo2nix doesn't stop there. All crates get their own derivation, which means that if we update only one dependency (in `Cargo.lock` and subsequently in `Cargo.nix`), we may still enjoy a quick build. This is helpful if just one or two of your dependencies change frequently. Or if you need to somehow break a very long CI build into stages, although you would have to be a little creative, because cargo2nix doesn't have any support for building just some dependencies.

OTHER OPTIONS

There are even more tools for building Rust code with Nix, but they were not up to the challenge, for various reasons.

- carnix is an old tool that is no longer maintained, and superseded by crate2nix.
- crate2nix also seems to not be maintained much any more, and it doesn't support building WebAssembly anyway.
- dream2nix is a very exciting project that aims to unify the many "2nix" converters into a common framework. However, it doesn't seem to be mature enough for our purposes, with little documentation and apparently no way to specify a WebAssembly build target.
- nocargo is another option under development, that, like cargo2nix, will build one derivation per crate. But, as its name suggests, it will not depend on Cargo at all, only Rustc. Unfortunately, it wasn't able to build our sample code. While it seemingly built the Cats library and the native app without issue, the resulting app output was empty. And the

WebAssembly build failed with error messages. Also, having path dependencies outside of a workspace wasn't supported.

CONCLUSION

Which tool should you use, then? As always, it depends. If you have a simple app and just want to package it with Nix, `buildRustPackage` may be all you need. But I think you'll soon appreciate the faster builds that the other tools provide by splitting your code and dependencies into separate derivations. Crane does seem to be an improvement over naersk, and because it delegates all the hard parts to Cargo, it can stay clear of all the problems that cargo2nix will need to handle.

If, for any reason, you need to split your dependencies into one derivation per crate, then cargo2nix seems to be your only option. The downsides are that you need to manage a `Cargo.nix` file, and that you may run into bugs if you have complicated builds.

Nocargo looks promising, and may well become the preferred choice when it has matured.

BEHIND THE SCENES

Tor is a Rust developer at Tweag who lives in Trondheim, Norway with his wife and two sons.

TOR HOVLAND

TECH GROUP

Sustainable software development with Nix

NIX

Learn more!

TECH GROUP

PROGRAMMING LANGUAGES AND COMPILERS

Research, create, improve and maintain programming languages and their tooling to enhance developer productivity and to deliver reliable, maintainable, correct and performant software with minimum effort.

Learn more!

If you enjoyed this article, you might be interested in joining the Tweag team.

← Optimizing Nickel's Array Contracts

Four months into The Nix Book →

COMPANY

About
Open Source
Careers
Contact Us

WHAT WE DO

Strategy
Product Development
Platform Modernization
Digital Operations
Work

INSIGHTS

Modus Blog
Ospo Blog
Research
Innovation podcast

CONNECT WITH US