



Using Nix with Dockerfiles

[Nix](#) is a powerful cross-platform package management tool. The benefits of Nix are far reaching, but one big benefit is that once you adopt Nix, you can get a consistent environment across development (on both Linux and Mac), CI, and production.

I've been [using Nix for many years](#) and recently started building Docker images using a Dockerfile paired with Nix. This post will explain the benefits of this approach along with a basic example to show how it looks and feels.


Sorry, this is not a Nix introduction post. You don't need to know how to use Nix to read this post, but I am also not going to explain basic Nix concepts or introduce the Nix language. If you do not know Nix, you can still read this post and use it to determine if Nix is interesting for you to learn.

Dockerfiles Are Easy, Why Nix?

Dockerfiles are quite easy -- you just chain together shell commands -- so understandably there is some hesitation to introduce a tool like Nix into the mix. The practical reason is that if you use Nix, you'll get always-working environments on local machines, CI, Docker, and more *for free*¹; you have little to no duplicated effort.

A typical, non-Nix approach is to have a separate effort for local development, CI, and Docker (we'll call Docker our "production" environment since this blog post is about building Docker images):

- For local development, you may have a giant README, may use Docker compose, may use [Vagrant](#), etc.
- Then, when you need to run or test your code in CI, you're probably writing YAML definitions to setup the environment again. A lot of times, the CI environment is different enough that you have slightly different shell requirements.
- Finally, for production, you have a Dockerfile with its own set of shell commands to run to build your final image.

Individually, this is *fine*, although I'd argue still a bit annoying. But when you put it all together, it is very annoying and very brittle. I hope I'm not the only one that has updated the local development environment and Dockerfile for a feature only to realize I broke CI. Or got my development environment working and a green check mark  on a PR only to realize that the production runtime is now broken.

These problems all go away with Nix. Nix is your single source of truth of what you need to *build and run* your software. You update this single source of truth and it generally just works everywhere. You don't need to maintain multiple descriptions of how to build and run your software anymore.

I'm serious, I haven't had a "works on my machine [but not on others]" issue in *years*. Not a single one. I'm actively trying to work through my [Nix God complex](#). It has been so long that when I see non-Nix users complain about issues getting software to run in various environments, I'm truly confused. It's like someone looking at a river and lamenting about having to wade through to cross it while I'm riding a bicycle across a bridge.

This is just one benefit, but I think it is the most practical one. Nix purists² will tout other things such as purism, reproducibility, powerful language, etc. This is all true, but I think the real pain point is **you want your environments to Just Work**.

In isolation, I submit to you that using Nix with Docker is mostly just harder than using Docker by itself. But by realizing the compounding benefits of using Nix to then be able to enhance your CI and development environments with the same configuration, using Nix with Docker becomes *easier* than using Docker by itself and also has numerous other practical benefits.

I'm going to focus on Docker images in this blog post, so I won't be showing how to use the same configuration for CI, development, etc. in any sort of detail. There are numerous blog posts on these topics. For local development environments, see [Nix and Direnv](#). For CI, take a look [at my own GitHub Actions workflows](#).

I want to explain the big idea at a high level, then I'll show a real example with code and shell commands. The idea is:

1. Write Nix code to describe how to build and run your application.
2. Use a Dockerfile and the [official Nix image](#) to build your application using Nix using roughly one shell command.
3. Use a multi-stage build ``FROM scratch`` to copy your built application into the smallest possible image. This final image doesn't have Nix installed at all -- we just used Nix to build.

Step 1 is the reusable bit. This same code can also be used to build a development or CI environment. As noted earlier, I won't go into detail about that in this blog post.

There are other "Nix and Docker" blog posts out there that use *only Nix* to build Docker images and don't use ``docker`` or Dockerfiles at all. That is possible and completely fine, but I wanted to write about using a Dockerfile because it is likely more familiar and less intimidating to people and because so many tools in the ecosystem often ingest and use Dockerfiles.

An Example: Python and Flask

As a real world example, I will use Nix to build a Docker image for running a [Flask](#) web application. The complete code is available [on GitHub](#).

The Python App

We aren't here to learn to learn Flask, so you can view the Flask application code in ``src/app.py``. It looks roughly like the below code block. It is just the Flask quickstart code to output "Hello World" on the root page.

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

Nix Flake

Next, we need to create a Nix flake to describe how to build your application. A Nix flake is a kind of Nix environment that describes how to create development environments, build packages, etc. It is similar to a `pyproject.toml` or `Cargo.toml` or `go.mod` or `package.json` and so on, but for Nix.

The Nix flake is in `flake.nix` and looks like this:

```
{
  description = "flask-example";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/release-22.05";
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let pkgs = import nixpkgs { inherit system;
      in with pkgs; rec {
        # Development environment
        devShell = mkShell {
          name = "flask-example";
          nativeBuildInputs = [ python3 poetry ];
        };

        # Runtime package
        packages.app = poetry2nix.mkPoetryApplication {
          projectDir = ./.;
        };

        # The default package when a specific package is not specified
        defaultPackage = packages.app;
      }
    )
}
```

```
);  
}
```

Did you just tell me to go f*ck myself? ([Comic](#))

Before I learned Nix, this is usually how I felt whenever someone dropped a block of Nix code. Seeing any unfamiliar code for the first time is usually intimidating and on top of that Nix isn't the most aesthetically pleasing, but please hang in there since this is the last time you'll see Nix code in this blog post and its mostly inconsequential to the remainder of this post.

Most of this is boilerplate. The key bits are the lines with ``devShell`` and ``packages.app``. ``devShell`` creates our development environment with Python and [Poetry](#) installed. And ``packages.app`` describes how to build our final package. Nix has first-class knowledge of Poetry, so we can just ask it to build our Poetry application. Most mainstream languages have a similar high-level helper to make using Nix easier.

[Install Nix on your system](#) and you can verify everything works by running ``nix build``. This should build the package and you can run the application at ``result/bin/app``.

```
$ nix build  
...  
  
$ result/bin/app  
* Serving Flask app 'src.app'  
* Debug mode: off  
* Running on http://127.0.0.1:5000  
Press CTRL+C to quit
```

Pause! This is low-key amazing. If you're unfamiliar with Nix, the impressiveness of what just happened may have gone unnoticed. Take a look at what `result/bin/app` is (cat` it) and follow that rabbit hole. It is a script to run your Python app that depends only on Nix-installed software. It does not depend on or conflict with your local system at all. If you have other versions of Python, Flask, etc. installed, it won't matter at all; your app is perfectly packaged.`

Dockerfile

Now let's bring it all together with Docker. Here is the

`Dockerfile`:`

```
# Nix builder
FROM nixos/nix:latest AS builder

# Copy our source and setup our working dir.
COPY . /tmp/build
WORKDIR /tmp/build

# Build our Nix environment
RUN nix \
  --extra-experimental-features "nix-command flakes" \
  --option filter-syscalls false \
  build

# Copy the Nix store closure into a directory. The
# entire set of Nix store values that we need for
RUN mkdir /tmp/nix-store-closure
RUN cp -R $(nix-store -qR result/) /tmp/nix-store-closure

# Final image is based on scratch. We copy a bunch
# but they're fully self-contained so we don't need
FROM scratch

WORKDIR /app

# Copy /nix/store
COPY --from=builder /tmp/nix-store-closure /nix/store
COPY --from=builder /tmp/build/result /app
CMD ["/app/bin/app"]
```

This is a [multi-stage build](#). We first start with our `builder` container which is based on `nixos/nix`. This is the Nix official base image that just has `nix` installed.

In this builder, we first run `nix build`:

```
RUN nix \
  --extra-experimental-features "nix-command flakes" \
  --option filter-syscalls false \
  build
```

This is just what we did earlier. The extra flags are to ensure the Nix command has `flakes` available (they're still marked as experimental) and the `filter-syscalls` option lets Apple Silicon cross-compile to Intel if you want.

The next step is:

```
RUN mkdir /tmp/nix-store-closure
RUN cp -R $(nix-store -qR result/) /tmp/nix-store-closure
```

This is the critical step. `nix store -qR result` outputs the full list of Nix directories that our application needs. Specifically, it is the *closure* of dependencies that only our application needs. I know this can be confusing, so to put it one final way: it is the smallest possible set of dependencies (files and folders) that our application needs to run and literally nothing else.

Finally, we use a `from scratch` container to build our final image:

```
FROM scratch

WORKDIR /app

COPY --from=builder /tmp/nix-store-closure /nix/store
COPY --from=builder /tmp/build/result /app
CMD ["/app/bin/app"]
```


We copy our closure into `~/nix/store`, which ensures that we have all the dependencies our application needs. Then we copy the `result` symlink to `/app`. Then, we set `/app/bin/app` as the entrypoint. This is similar to how we ran `result/bin/app` above when testing our Nix file.

Try it!

Build and run the Docker image:

```
$ docker build -t flask-example:dev .  
...  
  
$ docker run --rm flask-example:dev  
* Serving Flask app 'src.app'  
* Debug mode: off  
* Running on http://127.0.0.1:5000  
Press CTRL+C to quit
```

Downsides

There aren't many downsides to building Docker images this way, but in the interest in intellectual honesty, I'll note as many as I can think of.

The most obvious downside is that this requires Nix knowledge. Nix has a reputation for not being particularly easy to learn. In recent years, Nix documentation has improved greatly and there are additional resources like [Zero to Nix](#) which help a lot. Additionally the [Nix Installer](#) landscape has gotten much, much better.

Given the non-zero time cost to learning Nix, I think this downside is only worth it if you plan on using Nix for additional functionality such as CI or development environments as well. In my opinion, if you end up learning Nix this is inevitable because it's just *so dang nice*.

Another downside is that the layers produced in the Docker image are not optimal. The single `run nix build` command produces a giant layer with all the dependencies in it. From a build-time perspective, this is really fast because almost all of your dependencies will be downloaded from a binary cache. But, it is not optimal for caching and basically every redeploy will require your runtime environment to redownload the largest layer of your image.

Nix is able to make more optimal Docker image layers by using the native Nix `dockerTools` to build an image instead of a Dockerfile, but the whole point of this blog post is to show you the Dockerfile approach.

Next

I think that was pretty easy. The Dockerfile is less than 15 lines (without comments) and by using Nix its using the exact same code you'd use to build your development and CI environments, so its all shared logic. The Dockerfile never has to change.

This uses a plain-old Dockerfile so it integrates really nicely with tools you're likely familiar with already such as `docker`, various CI/CD tools, PaaS offerings, etc.

Finally, you get all the extra benefits of Nix: your Docker image has the smallest possible set of files to run your application and nothing more, the contents of your Docker image are reproducible (metadata may change the hash of the image itself), etc. These may or may not be important to you, but they have no downside and you get them for free.

As I said earlier, the benefits of this approach compound *greatly* when you start reusing your Nix configuration for other environments such as local development and CI. Therefore, I recommend using this approach to enhance

your current Nix usage or as a gateway to more expanded Nix usage.

1. With the arguably high up front cost of learning Nix, yes, but I think the payoff is way higher than the cost of admission. [↩](#)
2. ❤️ you folks, but trying to reach the masses! [↩](#)

© 2024 Mitchell Hashimoto.

