FLOX

Using Nix to build pretty small images

Bryan Honof

Why talk about Nix at a Cloud Native track?

FLOX


ECHO
CHAMBER
David A
Jonny opened the door to the one
place he always heard the truth.

FLOX

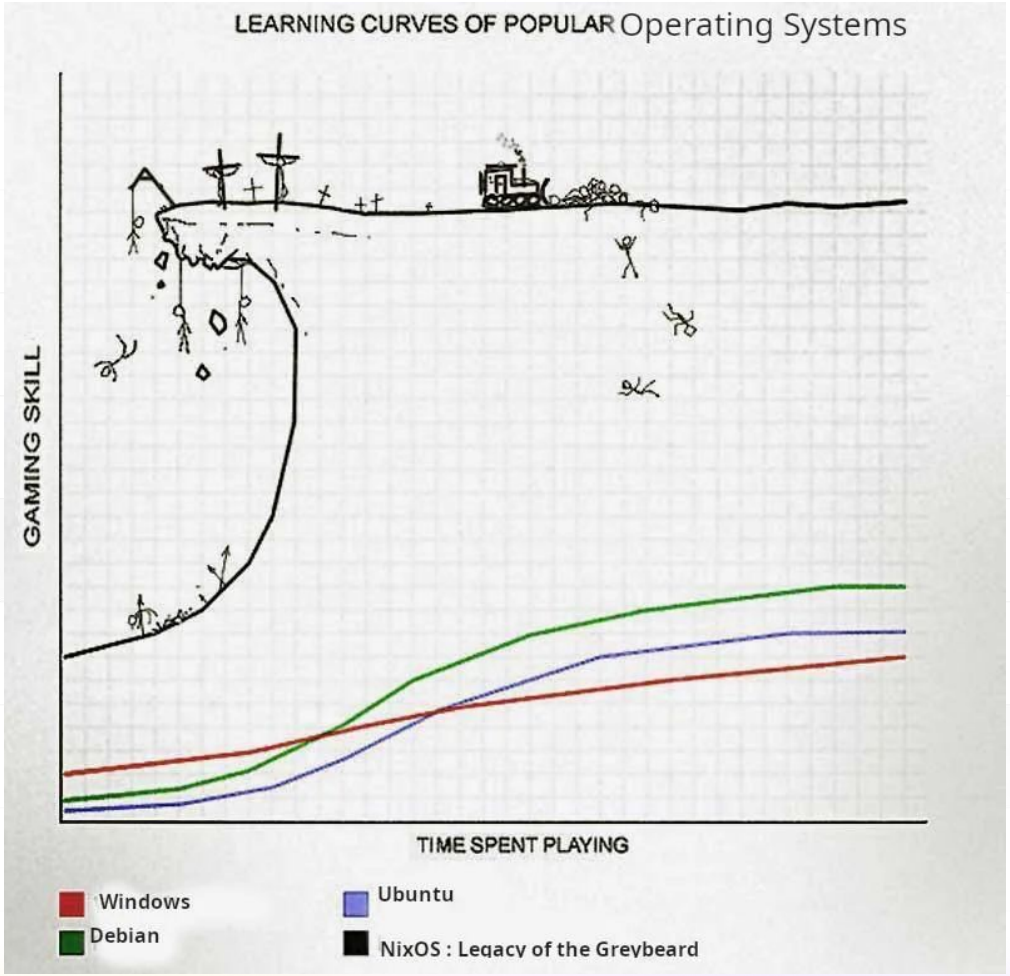

```
$ whoami
name:     Bryan HONOF
role:     Software Engineer @ Flox
email:    bryan@flox.dev
socials:
  - 🐘 @bryanhonof@mastodon.social
  - 🦋 @bjth.xyz
  - 🧳 in/bryanhonof
```

FLOX

So, what is this Nix thing?

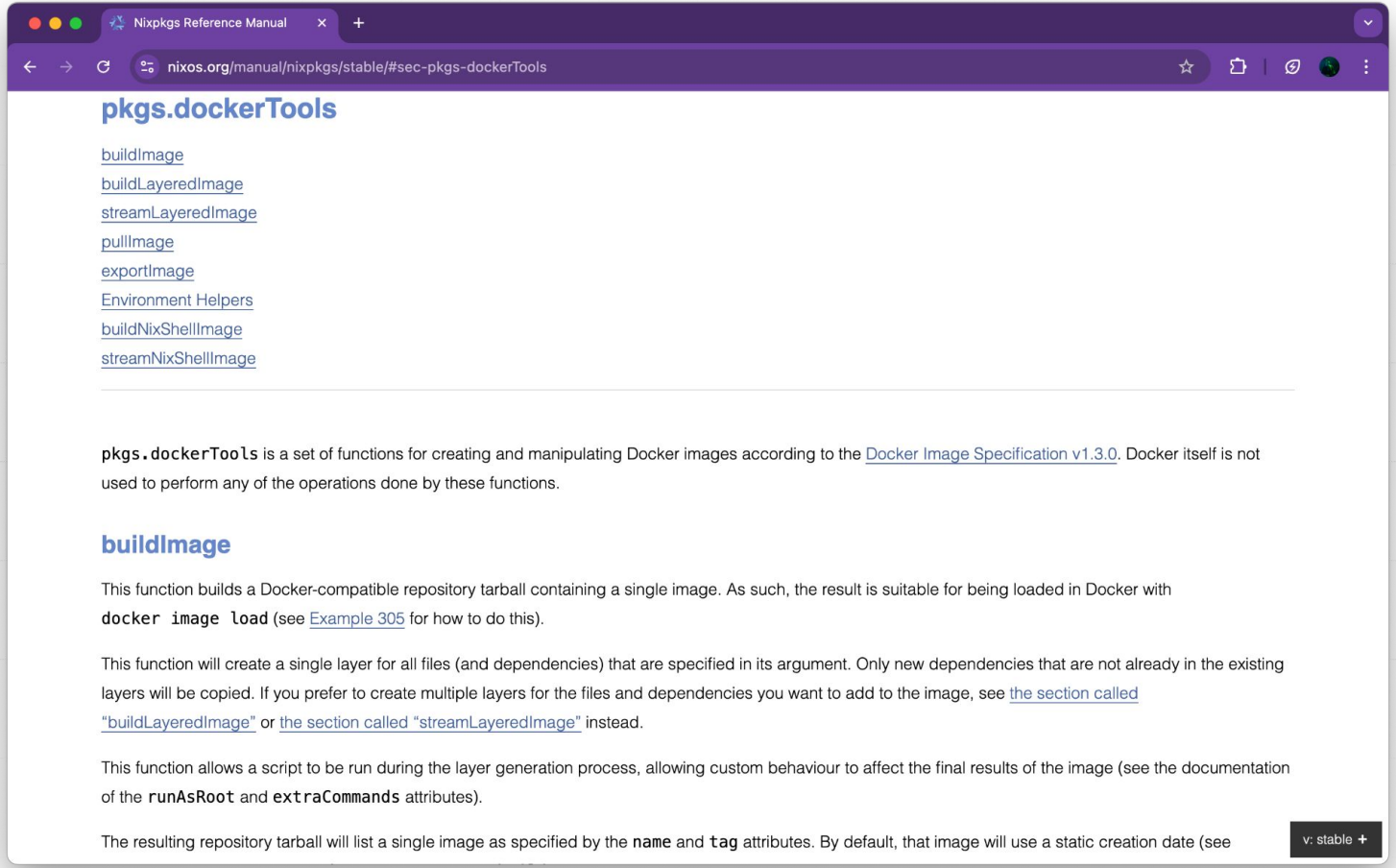
- Nix, as in the CLI tool you interface with
- Nix, as in the language you write into *.nix files
- Nix, as in nixpkgs the package repository
- Nix, as in NixOS the Linux distribution

And a bunch of community projects having Nix somewhere in their name


LEARNING CURVES OF POPULAR Operating Systems
GAMING SKILL
TIME SPENT PLAYING
Windows
Debian
Ubuntu
NixOS : Legacy of the Greybeard

FLOX

Nix as a way to build container images


Nixpkgs Reference Manual
nixos.org/manual/nixpkgs/stable/#sec-pkgs-dockerTools
pkgs.dockerTools
buildImage
buildLayeredImage
streamLayeredImage
pullImage
exportImage
Environment Helpers
buildNixShellImage
streamNixShellImage
pkgs.dockerTools is a set of functions for creating and manipulating Docker images according to the Docker Image Specification v1.3.0. Docker itself is not used to perform any of the operations done by these functions.
buildImage
This function builds a Docker-compatible repository tarball containing a single image. As such, the result is suitable for being loaded in Docker with docker image load (see Example 305 for how to do this).
This function will create a single layer for all files (and dependencies) that are specified in its argument. Only new dependencies that are not already in the existing layers will be copied. If you prefer to create multiple layers for the files and dependencies you want to add to the image, see the section called "buildLayeredImage" or the section called "streamLayeredImage" instead.
This function allows a script to be run during the layer generation process, allowing custom behaviour to affect the final results of the image (see the documentation of the runAsRoot and extraCommands attributes).
The resulting repository tarball will list a single image as specified by the name and tag attributes. By default, that image will use a static creation date (see
v: stable

FLOX

But first, Dockerfile!

```
FROM alpine:3.14

# Sure, cowsay is in a different repository...
RUN apk add \
      --no-cache \
      --repository=http://dl-cdn.alpinelinux.org/alpine/edge/testing/ \
      cowsay

# No hello package in upstream, so we'll cheat :(
ADD --chmod=755 <<EOF /usr/local/bin/hello
#!/bin/sh
echo 'Hello, world!'
EOF

ADD --chmod=755 <<EOF /usr/local/bin/app
#!/bin/sh
hello | cowsay
EOF

ENTRYPOINT [ "/usr/local/bin/app" ]
```

```
FROM alpine:3.14

# Sure, cowsay is in a different repository...
RUN apk add \
      --no-cache \
      --repository=http://dl-cdn.alpinelinux.org/alpine/edge/testing/ \
      cowsay

# No hello package in upstream, so we'll cheat :(
ADD --chmod=755 <<EOF /usr/local/bin/hello
#!/bin/sh
echo 'Hello, world!'
EOF

ADD --chmod=755 <<EOF /usr/local/bin/app
#!/bin/sh
hello | cowsay
EOF

ENTRYPOINT [ "/usr/local/bin/app" ]
```

```
FROM alpine:3.14

# Sure, cowsay is in a different repository...
RUN apk add \
      --no-cache \
      --repository=http://dl-cdn.alpinelinux.org/alpine/edge/testing/ \
      cowsay

# No hello package in upstream, so we'll cheat :(
ADD --chmod=755 <<EOF /usr/local/bin/hello
#!/bin/sh
echo 'Hello, world!'
EOF

ADD --chmod=755 <<EOF /usr/local/bin/app
#!/bin/sh
hello | cowsay
EOF

ENTRYPOINT [ "/usr/local/bin/app" ]
```

```
FROM alpine:3.14

# Sure, cowsay is in a different repository...
RUN apk add \
      --no-cache \
      --repository=http://dl-cdn.alpinelinux.org/alpine/edge/testing/ \
      cowsay

# No hello package in upstream, so we'll cheat :(
ADD --chmod=755 <<EOF /usr/local/bin/hello
#!/bin/sh
echo 'Hello, world!'
EOF

ADD --chmod=755 <<EOF /usr/local/bin/app
#!/bin/sh
hello | cowsay
EOF

ENTRYPOINT [ "/usr/local/bin/app" ]
```

```
$ docker build --tag 'hellocowsay:alpine' -f Dockerfile.alpine .
…
$ docker run --rm hellocowsay:alpine
 ---------------
< Hello, world! >
 ---------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

FLOX

```
$ docker image inspect hellocowsay:alpine \
  | jq -r '.[0].Size' \
  | numfmt --to=iec-i
38Mi
```

Not bad, but what about other distros as a base?

```
# What does stable mean here? When does it change?
# Should I be pinning it better?
# Is this debian:stable the same one that I use in all my other containers?
FROM debian:stable

# Will apt-get update only change when I change the tag above?
# What will happen if packages dissapear from the upstream registry?
RUN apt-get update && apt-get install -y hello cowsay

# Apparently cowsay lives under /usr/games/cowsay these days, which isn't added
# to PATH?
# Where do I put the binary? /bin? /usr/bin? /usr/local/bin? All of the above?
# Is /bin/bash good practice here? Why not /usr/bin/env bash, or /bin/sh?
ADD --chmod=755 <<EOF /usr/local/bin/app
#!/bin/bash
hello | /usr/games/cowsay
EOF

# No questions here, this is the only part that I can't think of any assumptions
# that are made when typing it.
ENTRYPOINT [ "/usr/local/bin/app" ]
```

```
FROM debian:stable

RUN apt-get update && apt-get install -y hello cowsay

ADD --chmod=755 <<EOF /usr/local/bin/app
#!/bin/bash
hello | /usr/games/cowsay
EOF

ENTRYPOINT [ "/usr/local/bin/app" ]
```

```
$ docker build --tag 'hellocowsay:debian' -f Dockerfile.debian .
…
$ docker run --rm hellocowsay:debian
 ----------------
< Hello, world! >
 ----------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

FLOX

```
$ docker image inspect hellocowsay:debian \
  | jq -r '.[0].Size' \
  | numfmt --to=iec-i
200Mi
```

FLOX

```
{ dockerTools, hello, cowsay, writeTextFile, lib, shell }:
dockerTools.buildLayeredImage {
  name = "hello";
  config.Cmd = let
      app = writeTextFile {
        name = "app.sh";
        executable = true;
        text = ''
          #!${shell}/bin/sh
          ${lib.getExe hello} | ${lib.getExe cowsay}
        '';
      };
    in
    [ "${app}" ];
}
```

FLOX

```
{ dockerTools, hello, cowsay, writeTextFile, lib, shell }:
dockerTools.buildLayeredImage {
  name = "hello";
  config.Cmd = let
      app = writeTextFile {
        name = "app.sh";
        executable = true;
        text = ''
          #!${shell}/bin/sh
          ${lib.getExe hello} | ${lib.getExe cowsay}
        '';
      };
    in
    [ "${app}" ];
}
```

FLOX

```
{ dockerTools, hello, cowsay, writeTextFile, lib, shell }:
dockerTools.buildLayeredImage {
  name = "hello";
  config.Cmd = let
      app = writeTextFile {
        name = "app.sh";
        executable = true;
        text = ''
          #!${shell}/bin/sh
          ${lib.getExe hello} | ${lib.getExe cowsay}
        '';
      };
    in
    [ "${app}" ];
}
```

FLOX

```
{ dockerTools, hello, cowsay, writeTextFile, lib, shell }:
dockerTools.buildLayeredImage {
  name = "hello";
  config.Cmd = let
      app = writeTextFile {
        name = "app.sh";
        executable = true;
        text = ''
          #!${shell}/bin/sh
          ${lib.getExe hello} | ${lib.getExe cowsay}
        '';
      };
    in
    [ "${app}" ];
}
```

FLOX

```
{ dockerTools, hello, cowsay, writeTextFile, lib, shell }:
dockerTools.buildLayeredImage {
  name = "hello";
  config.Cmd = let
      app = writeTextFile {
        name = "app.sh";
        executable = true;
        text = ''
          #!${shell}/bin/sh
          ${lib.getExe hello} | ${lib.getExe cowsay}
        '';
      };
    in
    [ "${app}" ];
}
```

FLOX

```
{ dockerTools, hello, cowsay, writeTextFile, lib, shell }:
dockerTools.buildLayeredImage {
  name = "hello";
  config.Cmd = let
      app = writeTextFile {
        name = "app.sh";
        executable = true;
        text = ''
          #!${shell}/bin/sh
          ${lib.getExe hello} | ${lib.getExe cowsay}
        '';
      };
    in
    [ "${app}" ];
}
```

FLOX

```
let
  pkgs = import <nixpkgs> { };
in
pkgs.callPackage ./hello-container.nix {
  hello = pkgs.hello;
  cowsay = pkgs.cowsay;
  shell = pkgs.bash;
}
```

Cool, ugly syntax!

```
$ nix build -f default.nix && docker load -i ./result
…
$ docker run --rm hello
 _____________
< Hello, world! >
 -------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

```
$ docker image inspect hello \
    | jq -r '.[0].Size' \
    | numfmt --to=iec-i
116Mi
```

FLOX

```
let
  pkgs = import <nixpkgs> { };
in
pkgs.callPackage ./hello-container.nix {
  hello = pkgs.pkgsMusl.hello;
  cowsay = pkgs.pkgsMusl.cowsay;
  shell = pkgs.pkgsMusl.busybox;
}
```

```
let
  pkgs = import <nixpkgs> { };
in
pkgs.callPackage ./hello-container.nix {
  hello = pkgs.pkgsMusl.hello;
  cowsay = pkgs.pkgsMusl.cowsay;
  shell = pkgs.pkgsMusl.busybox;
}
```

```
$ nix build -f default.nix && docker load -i ./result
…
$ docker run --rm hello
 _____________
< Hello, world! >
 -------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

```
$ docker image inspect hello \
    | jq -r '.[0].Size' \
    | numfmt --to=iec-i
75Mi
```

FLOX

Let's look at a more realistic/simple example

```
$ docker image pull memcached:alpine
…
$ docker image inspect memcached:alpine \
  | jq -r '.[0].Size' \
  | numfmt --to=iec-i
11Mi
```

```
{ dockerTools, lib, memcached }:
dockerTools.buildLayeredImage {
  name = "memcached";
  uid = 10000;
  gid = 10000;
  uname = "memcached";
  gname = "memcached";
  contents = [
    (dockerTools.fakeNss.override {
      extraPasswdLines = [ "memcached:x:10000:10000:memcached service
user:/var/empty:/bin/sh" ];
      extraGroupLines = [ "memcached:x:10000" ];
    })
  ];
  config.Cmd = [ "${lib.getExe memcached}" "-l" "0.0.0.0" "-p" "11211"];
  config.ExposedPorts = {
    "11211/tcp" = { };
  };
  config.User = "memcached:memcached";
}
```

FLOX

```
{ dockerTools, lib, memcached }:
dockerTools.buildLayeredImage {
  name = "memcached";
  uid = 10000;
  gid = 10000;
  uname = "memcached";
  gname = "memcached";
  contents = [
    (dockerTools.fakeNss.override { … })
  ];
  config.Cmd = [ "${lib.getExe memcached}" "-l" "0.0.0.0" "-p" "11211"];
  config.ExposedPorts = {
    "11211/tcp" = { };
  };
  config.User = "memcached:memcached";
}
```

FLOX

```
{ dockerTools, lib, memcached }:
dockerTools.buildLayeredImage {
  name = "memcached";
  uid = 10000;
  gid = 10000;
  uname = "memcached";
  gname = "memcached";
  contents = [
    (dockerTools.fakeNss.override { … })
  ];
  config.Cmd = [ "${lib.getExe memcached}" "-l" "0.0.0.0" "-p" "11211"];
  config.ExposedPorts = {
    "11211/tcp" = { };
  };
  config.User = "memcached:memcached";
}
```

FLOX

```
let
  pkgs = import <nixpkgs> { };
in
pkgs.callPackage ./memcached-container.nix {
  memcached = pkgs.pkgsMusl.memcached;
}
```

```
$ docker image inspect memcached \
  | jq -r '.[0].Size' \
  | numfmt --to=iec-i
5.4Mi
```

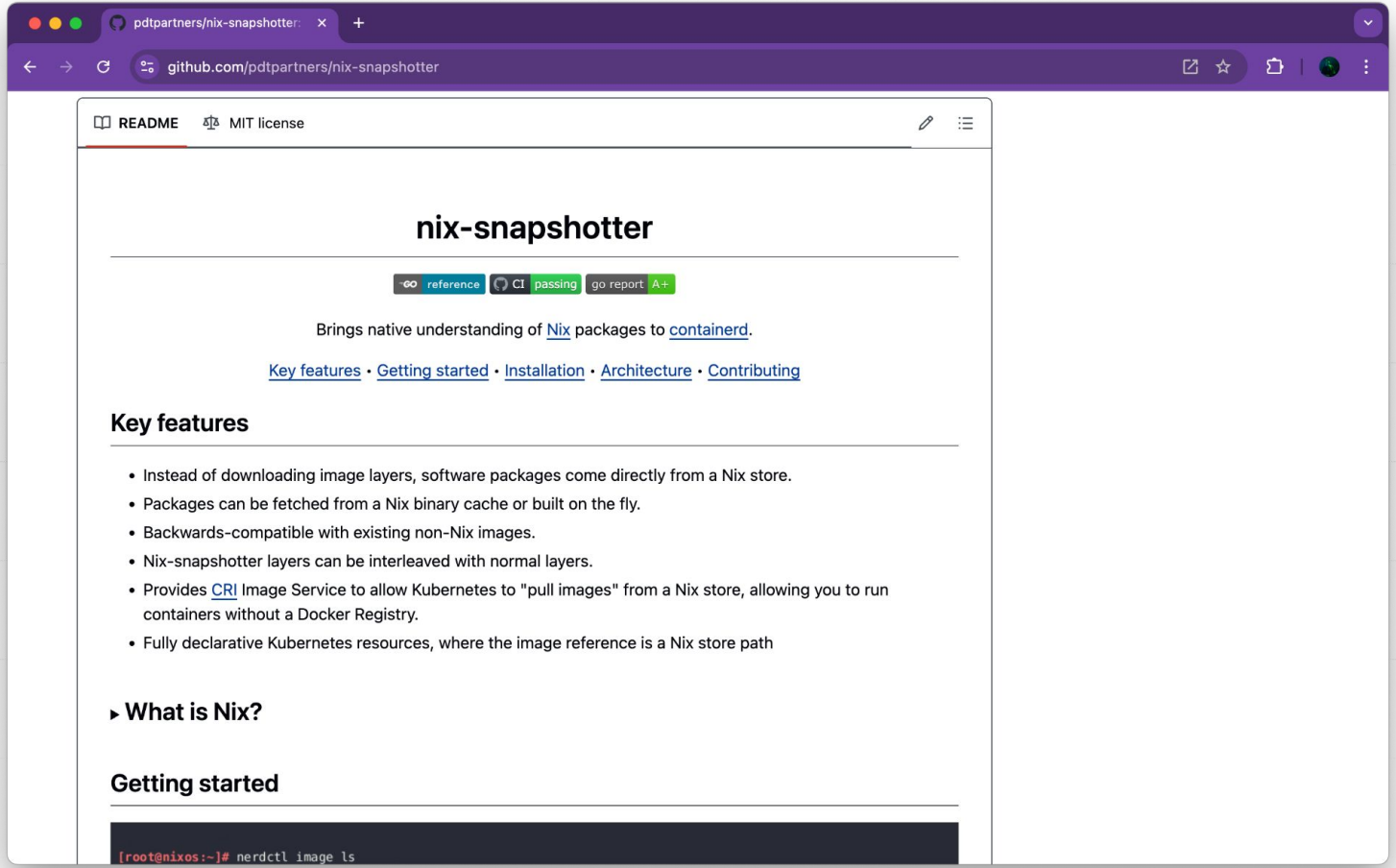
FLOX

5.4Mi < 11Mi

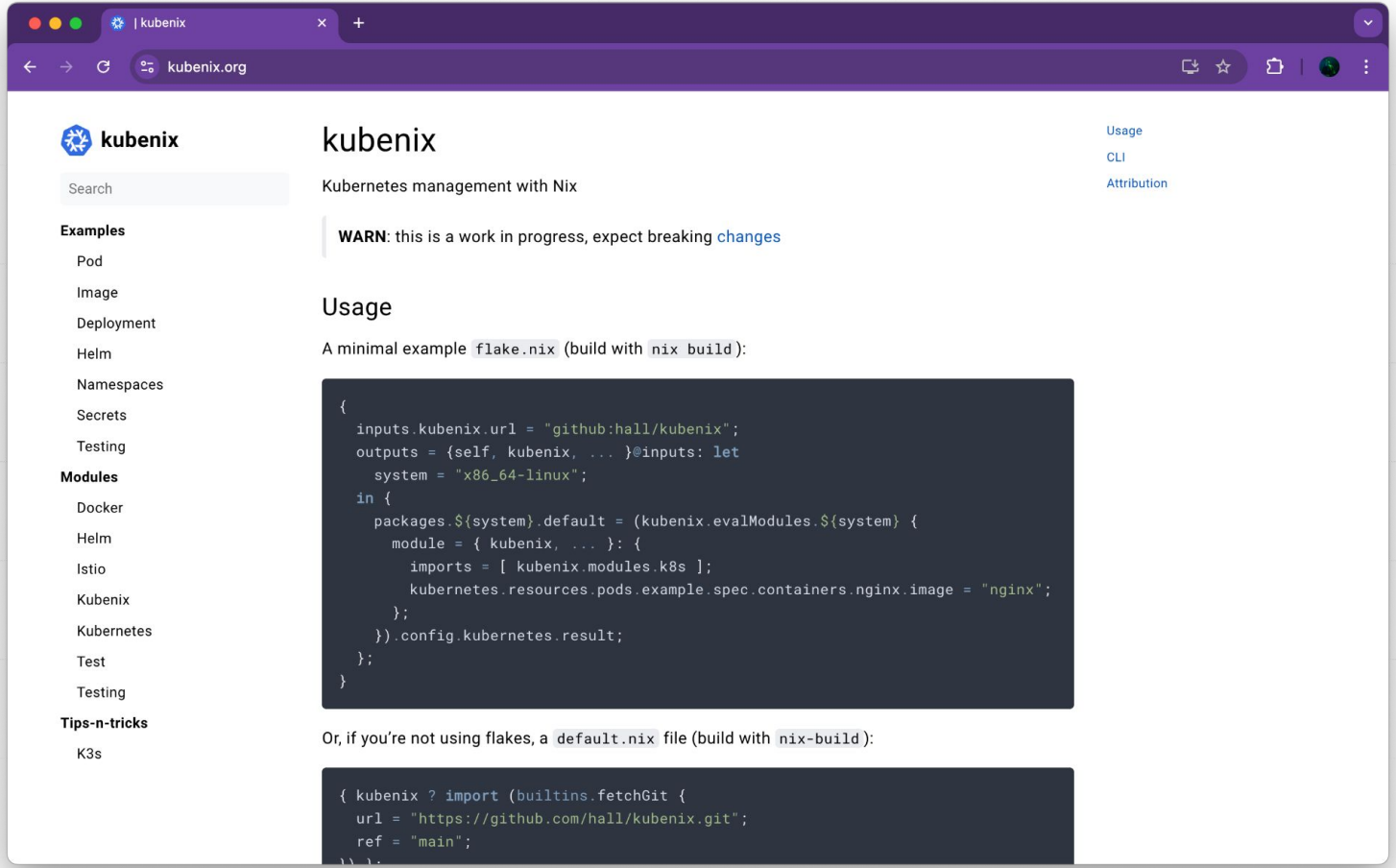
FLOX

FLOX

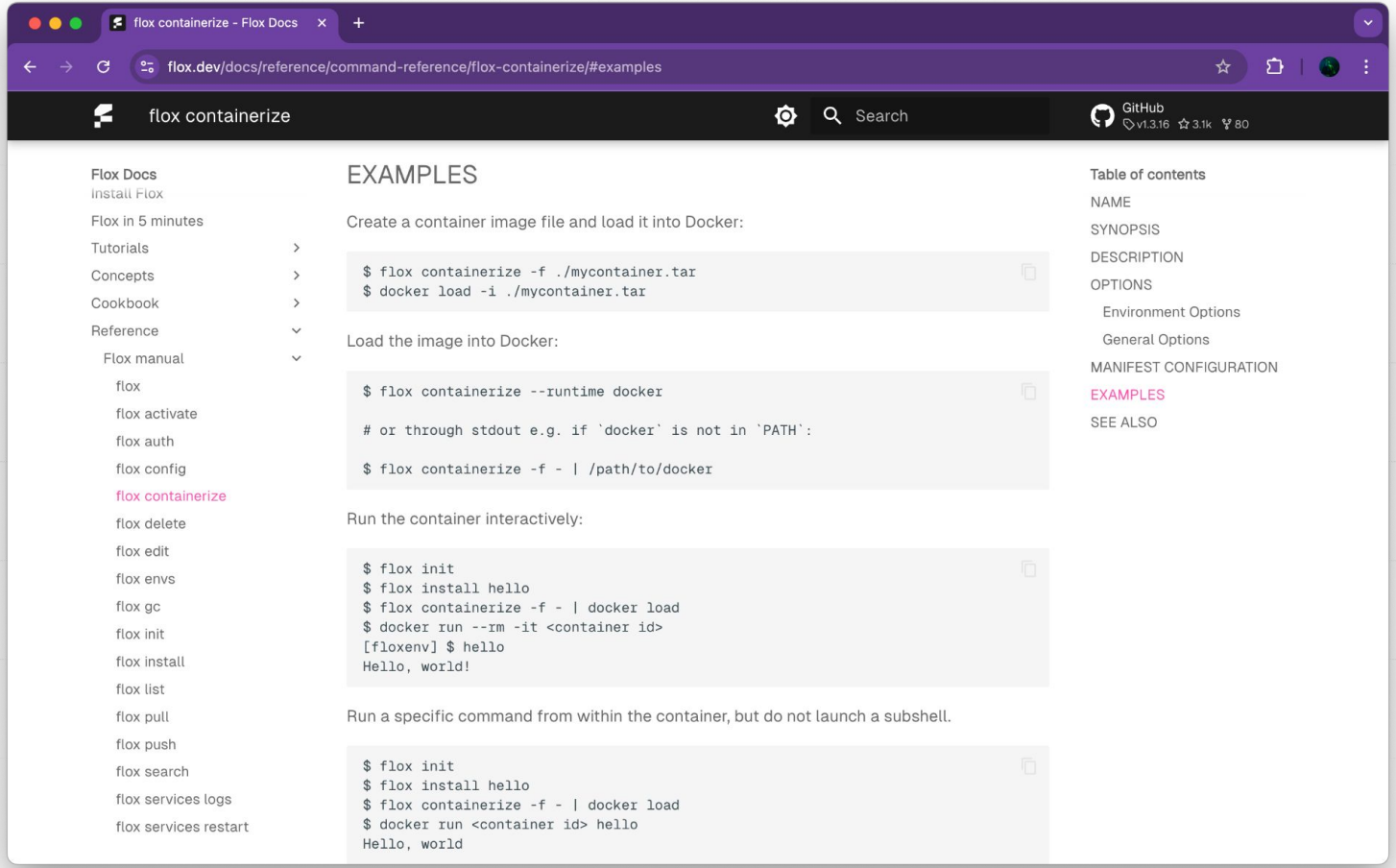
What does the future hold?


pdtpartners/nix-snapshotter
github.com/pdtpartners/nix-snapshotter
README
MIT license
nix-snapshotter
reference
CI passing
go report A+
Brings native understanding of Nix packages to containerd.
Key features • Getting started • Installation • Architecture • Contributing
Key features
Instead of downloading image layers, software packages come directly from a Nix store.
Packages can be fetched from a Nix binary cache or built on the fly.
Backwards-compatible with existing non-Nix images.
Nix-snapshotter layers can be interleaved with normal layers.
Provides CRI Image Service to allow Kubernetes to "pull images" from a Nix store, allowing you to run containers without a Docker Registry.
Fully declarative Kubernetes resources, where the image reference is a Nix store path
What is Nix?
Getting started
[root@nixos:~]# nerdctl image ls

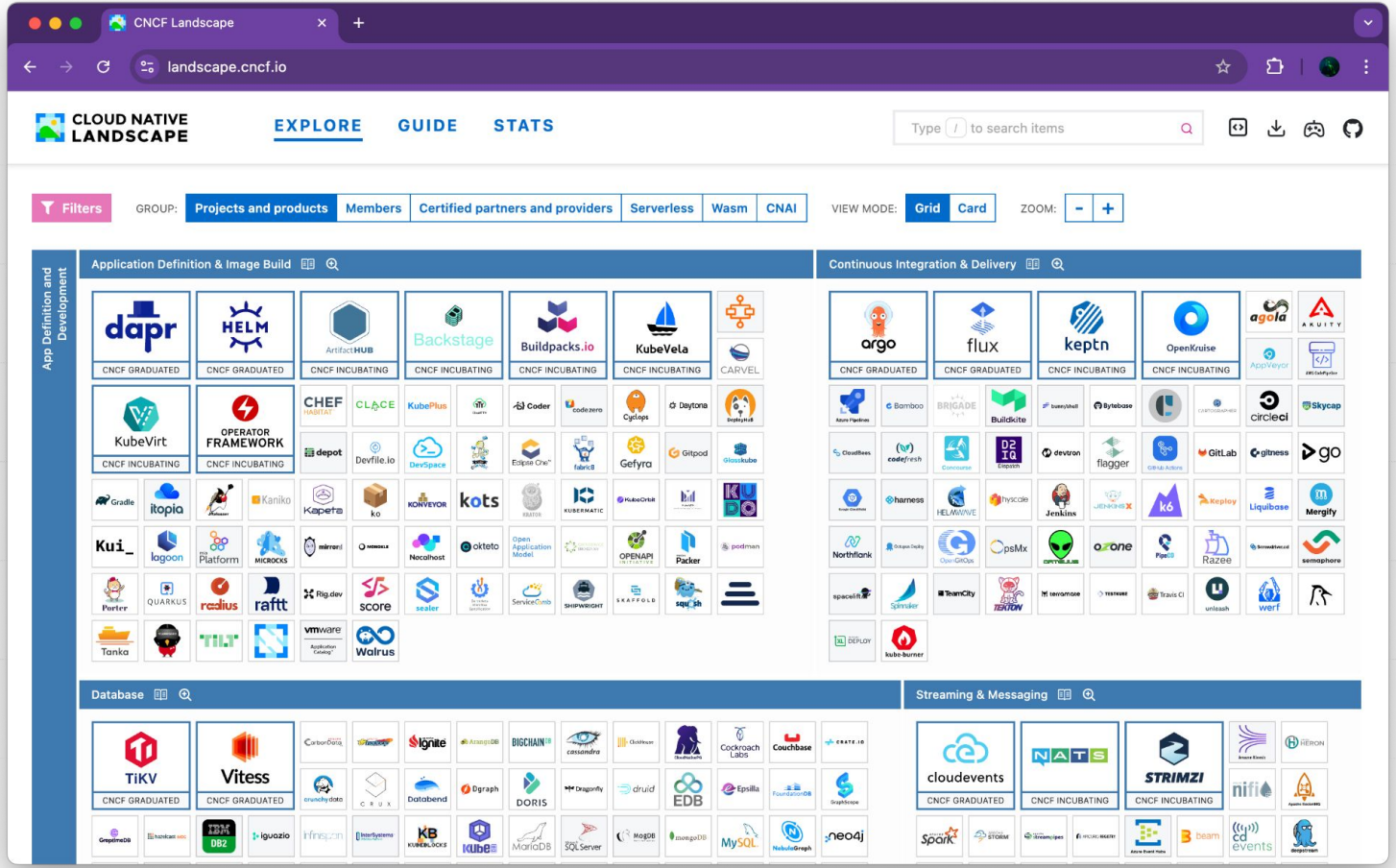
FLOX


| kubenix
kubenix.org
kubenix
Search
Examples
Pod
Image
Deployment
Helm
Namespaces
Secrets
Testing
Modules
Docker
Helm
Istio
Kubenix
Kubernetes
Test
Testing
Tips-n-tricks
K3s
kubenix
Kubernetes management with Nix
WARN: this is a work in progress, expect breaking changes
Usage
A minimal example flake.nix (build with nix build):
{
inputs.kubenix.url = "github:hall/kubenix";
outputs = {self, kubenix, ... }@inputs: let
system = "x86_64-linux";
in {
packages.${system}.default = (kubenix.evalModules.${system} {
module = { kubenix, ... }: {
imports = [kubenix.modules.k8s];
kubernetes.resources.pods.example.spec.containers.nginx.image = "nginx";
};
}).config.kubernetes.result;
};
}
Or, if you're not using flakes, a default.nix file (build with nix-build):
{ kubenix ? import (builtins.fetchGit {
url = "https://github.com/hall/kubenix.git";
ref = "main";
Usage
CLI
Attribution

FLOX


flox containerize - Flox Docs
flox.dev/docs/reference/command-reference/flox-containerize/#examples
flox containerize
Search
GitHub
v1.3.16 3.1k 80
Flox Docs
Install Flox
Flox in 5 minutes
Tutorials
Concepts
Cookbook
Reference
Flox manual
flox
flox activate
flox auth
flox config
flox containerize
flox delete
flox edit
flox envs
flox gc
flox init
flox install
flox list
flox pull
flox push
flox search
flox services logs
flox services restart
EXAMPLES
Create a container image file and load it into Docker:
$ flox containerize -f ./mycontainer.tar
$ docker load -i ./mycontainer.tar
Load the image into Docker:
$ flox containerize --runtime docker
# or through stdout e.g. if `docker` is not in `PATH`:
$ flox containerize -f - | /path/to/docker
Run the container interactively:
$ flox init
$ flox install hello
$ flox containerize -f - | docker load
$ docker run --rm -it <container id>
[floxenv] $ hello
Hello, world!
Run a specific command from within the container, but do not launch a subshell.
$ flox init
$ flox install hello
$ flox containerize -f - | docker load
$ docker run <container id> hello
Hello, world
Table of contents
NAME
SYNOPSIS
DESCRIPTION
OPTIONS
Environment Options
General Options
MANIFEST CONFIGURATION
EXAMPLES
SEE ALSO

FLOX

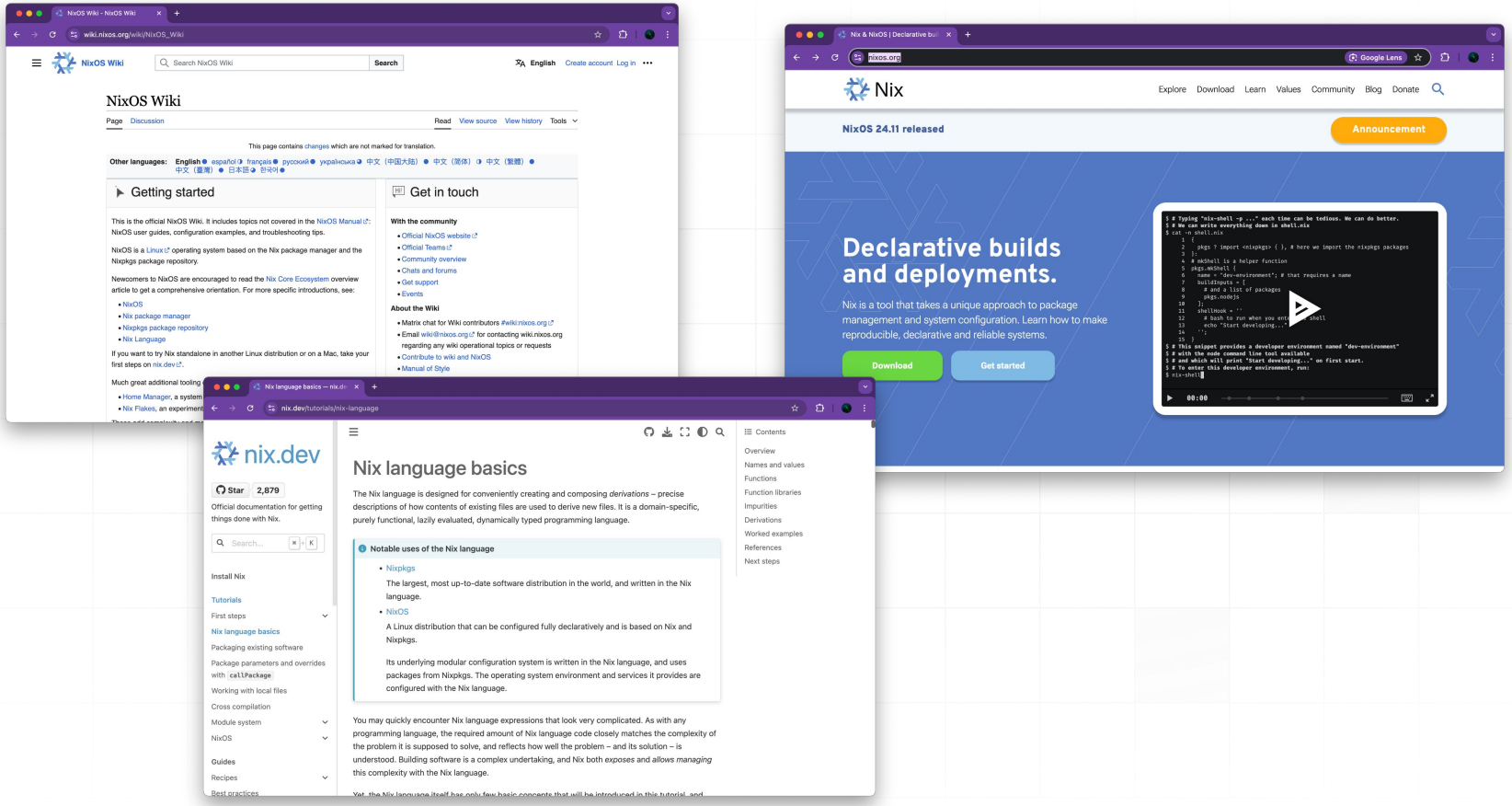

CNCF Landscape
landscape.cncf.io
CLOUD NATIVE LANDSCAPE
EXPLORE
GUIDE
STATS
Type / to search items
Filters
GROUP:
Projects and products
Members
Certified partners and providers
Serverless
Wasm
CNAI
VIEW MODE:
Grid
Card
ZOOM:
App Definition and Development
Application Definition & Image Build
Continuous Integration & Delivery
Database
Streaming & Messaging
dapr
HELM
Backstage
Buildpacks.io
KubeVela
argo
flux
keptn
OpenKruise
KubeVirt
OPERATOR FRAMEWORK
TiKV
Vitess
cloudevents
NATS
STRIMZI
CNCF GRADUATED
CNCF INCUBATING

FLOX

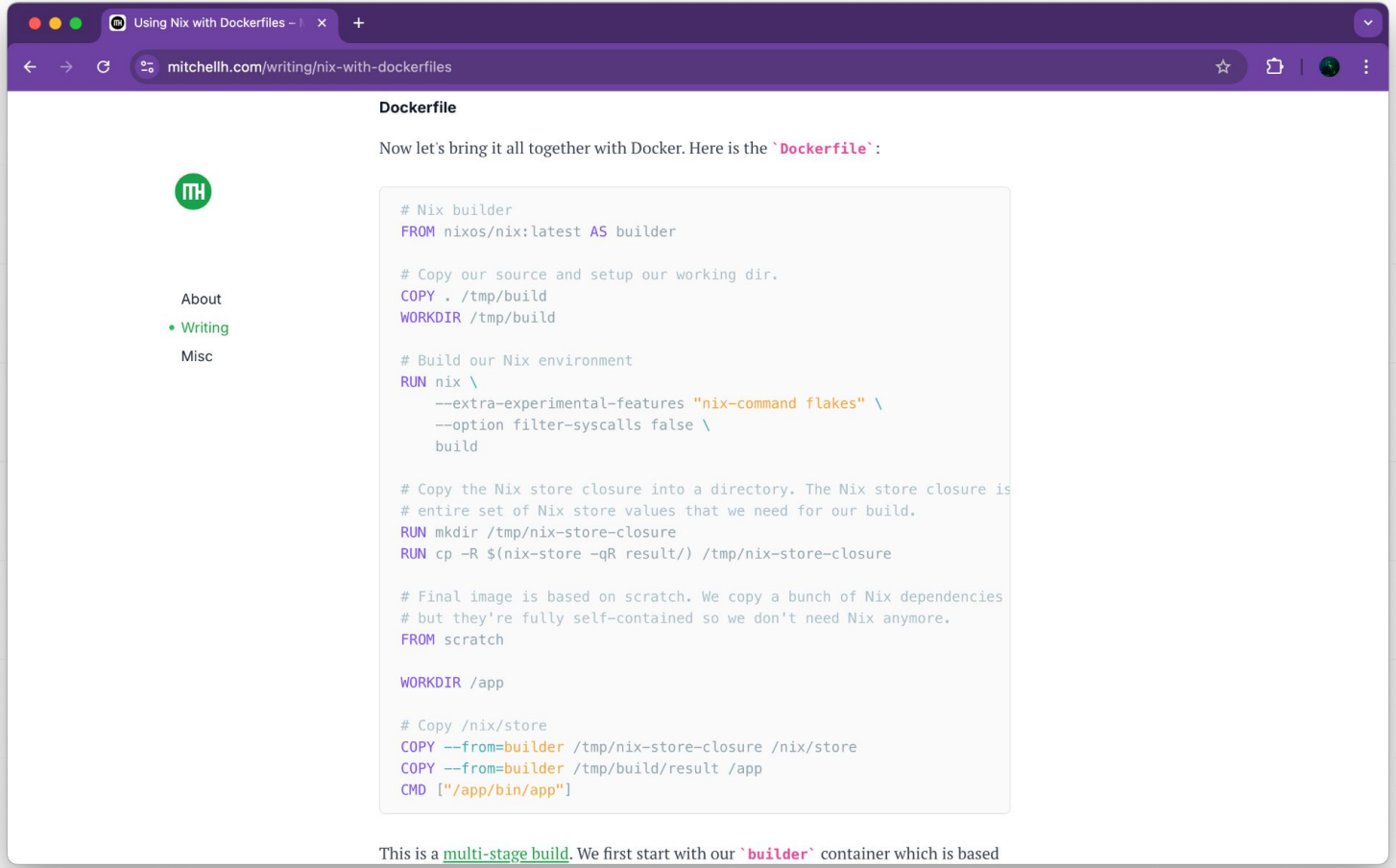
Some downsides

But the advantages outweigh the downsides

Resources to check out!


NixOS Wiki - NixOS Wiki
wiki.nixos.org/wiki/NixOS_Wiki
NixOS Wiki
Search NixOS Wiki
Search
English
Create account
Log in
NixOS Wiki
Page
Discussion
Read
View source
View history
Tools
This page contains changes which are not marked for translation.
Other languages:
English
español
français
русский
українська
中文（中国大陆）
中文（简体）
中文（繁體）
中文（臺灣）
日本語
한국어
Getting started
This is the official NixOS Wiki. It includes topics not covered in the NixOS Manual: NixOS user guides, configuration examples, and troubleshooting tips.
NixOS is a Linux operating system based on the Nix package manager and the Nixpkgs package repository.
Newcomers to NixOS are encouraged to read the Nix Core Ecosystem overview article to get a comprehensive orientation. For more specific introductions, see:
NixOS
Nix package manager
Nixpkgs package repository
Nix Language
If you want to try Nix standalone in another Linux distribution or on a Mac, take your first steps on nix.dev.
Much great additional tooling
Home Manager, a system
Nix Flakes, an experiment
Get in touch
With the community
Official NixOS website
Official Teams
Community overview
Chats and forums
Get support
Events
About the Wiki
Matrix chat for Wiki contributors #wiki:nixos.org
Email wiki@nixos.org for contacting wiki.nixos.org regarding any wiki operational topics or requests
Contribute to wiki and NixOS
Manual of Style
Nix & NixOS | Declarative bui
nixos.org
Google Lens
Nix
Explore
Download
Learn
Values
Community
Blog
Donate
NixOS 24.11 released
Announcement
Declarative builds and deployments.
Nix is a tool that takes a unique approach to package management and system configuration. Learn how to make reproducible, declarative and reliable systems.
Download
Get started
$ # Typing "nix-shell -p ..." each time can be tedious. We can do better.
$ # We can write everything down in shell.nix
$ cat -n shell.nix
1 {
2 pkgs ? import <nixpkgs> { },
3 }:
4 # mkShell is a helper function
5 pkgs.mkShell {
6 name = "dev-environment"; # that requires a name
7 buildInputs = [
8 # and a list of packages
9 pkgs.nodejs
10];
11 shellHook = ''
12 # bash to run when you ente shell
13 echo "Start developing..."
14 '';
15 }
$ # This snippet provides a developer environment named "dev-environment"
$ # with the node command line tool available
$ # and which will print "Start developing..." on first start.
$ # To enter this developer environment, run:
$ nix-shell
00:00
Nix language basics — nix.dev
nix.dev/tutorials/nix-language
nix.dev
Star
2,879
Official documentation for getting things done with Nix.
Search...
Install Nix
Tutorials
First steps
Nix language basics
Packaging existing software
Package parameters and overrides with callPackage
Working with local files
Cross compilation
Module system
NixOS
Guides
Recipes
Best practices
Nix language basics
The Nix language is designed for conveniently creating and composing derivations – precise descriptions of how contents of existing files are used to derive new files. It is a domain-specific, purely functional, lazily evaluated, dynamically typed programming language.
Notable uses of the Nix language
Nixpkgs
The largest, most up-to-date software distribution in the world, and written in the Nix language.
NixOS
A Linux distribution that can be configured fully declaratively and is based on Nix and Nixpkgs.
Its underlying modular configuration system is written in the Nix language, and uses packages from Nixpkgs. The operating system environment and services it provides are configured with the Nix language.
You may quickly encounter Nix language expressions that look very complicated. As with any programming language, the required amount of Nix language code closely matches the complexity of the problem it is supposed to solve, and reflects how well the problem – and its solution – is understood. Building software is a complex undertaking, and Nix both exposes and allows managing this complexity with the Nix language.
Contents
Overview
Names and values
Functions
Function libraries
Impurities
Derivations
Worked examples
References
Next steps

FLOX


Using Nix with Dockerfiles –
mitchellh.com/writing/nix-with-dockerfiles
About
Writing
Misc
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 Nix store closure is
entire set of Nix store values that we need for our build.
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 of Nix dependencies
but they're fully self-contained so we don't need Nix anymore.
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

FLOX


Docker: How To Debug Distro
iximiuz.com/en/posts/docker-debug-slim-containers/
Home Series Articles Newsletter Labs New! Projects About
Docker: How To Debug Distroless And Slim Containers
October 8, 2022 (Updated: April 25, 2024) • Containers
Ivan Velichko
Software Engineer at day. Tech Storyteller at night. Helping people master Containers.
✓ Hard topics clearly explained.
Containers
Kubernetes
Linux / Unix
Networking
Programming
Debugging Containers Like a Pro
Learning Series
Docker: How To Debug Distroless And Slim Containers
Kubernetes Ephemeral Containers and kubectl debug Command
Containers 101: attach vs. exec - what's the difference?
Why and How to Use containerd From Command Line
Docker: How To Extract Image Filesystem Without Running Any Containers
KiND - How I Wasted a Day Loading Local Docker Images
Don't miss new posts in the series! Subscribe to the blog updates and get deep technical write-

FLOX

Using Nix to build pretty small images

Bryan Honof