

The screenshot shows the FlakeHub interface for the 'Above-Os/tailscale' flake. At the top, there's a navigation bar with the FlakeHub logo, a user icon, 'Log in', and a menu icon. Below the header, there's a small thumbnail image of a 4x4 grid of red squares. The main title is 'FLAKE' in blue, followed by the name 'Above-Os / tailscale'. Below the title, there are several metadata items: a version icon with '1.50.0', a commit hash 'a30a7198bee21c331ab9e561fcc4e3ceb88f7044', a circular icon with 'Above-Os/tailscale', a BSD-3-Clause license icon, a 'Releases' section, and a creation timestamp 'Created Wed, 27 Sep 2023 04:44:42 GMT'. There are also sharing links for Twitter, GitHub, RSS, and FlakeHub.

USAGE

Show

```
nix flake show "https://flakehub.com/f/Above-Os/tailscale/1.50.0.t"
```

Add tailscale to your flake

```
fh add "Above-Os/tailscale/1.50.0"
```

Flakes are easier with our official CLI. [Learn more about the CLI.](#)

OUTPUTS

Systems x86_64-
packages

Publish your Nix flakes!
Set up GitHub Actions to automatically push your flakes to FlakeHub.

The packages fl...

shell.

x86_64-linux

Linux on a 64 bit x86 processor, like Intel or AMD.

`packages.x86_64-`
`linux.tailscale`

Show 2 more systems

README

Tailscale

<https://tailscale.com>

Private WireGuard® networks made easy

Overview

This repository contains the majority of Tailscale's open source code. Notably, it includes the `tailscaled` daemon and the `tailscale` CLI tool. The `tailscaled` daemon runs on Linux, Windows, macOS, and to varying degrees on FreeBSD and OpenBSD. The Tailscale iOS and Android apps use this repo's code, but this repo doesn't contain the mobile GUI code.

Other Tailscale repos of note:

the Android app is at <https://github.com/tailscale/tailscale-android>

the Synology package is at <https://github.com/tailscale/tailscale-synology>

the QNAP package is at <https://github.com/tailscale/tailscale-qpkg>

the Chocolatey packaging is at <https://github.com/tailscale/tailscale-chocolatey>

For background on which parts of Tailscale are open source and why, see

<https://tailscale.com/opensource/>.

Using

We serve packages for a variety of distros and platforms at

<https://pkgs.tailscale.com>.

Other clients

The macOS, iOS, and Windows clients use the code in this repository but additionally include small GUI wrappers. The GUI wrappers on non-open source platforms are themselves not open source.

Building

We always require releases with one or more `go install` commands.

Publish your Nix flakes!



Set up GitHub Actions to automatically push your flakes to FlakeHub.



If you're packaging Tailscale for distribution, use `build.nix` instead, to

burn commit IDs and version info into the binaries:

```
./build_dist.sh tailscale.com/cmd/tailscale
```

```
./build_dist.sh tailscale.com/cmd/tailscaled
```

If your distro has conventions that preclude the use of `build_dist.sh`, please do the equivalent of what it does in your distro's way, so that bug reports contain useful version information.

Bugs

Please file any issues about this code or the hosted service on the issue tracker.

Contributing

PRs welcome! But please file bugs. Commit messages should reference bugs.

We require Developer Certificate of Origin Signed-off-by lines in commits.

See `git log` for our commit message style. It's basically the same as Go's style.

About Us

Tailscale is primarily developed by the people at <https://github.com/orgs/tailscale/people>. For other contributors, see:

<https://github.com/tailscale/tailscale/graphs/contributors>

<https://github.com/tailscale/tailscale-android/graphs/contributors>

Legal

WireGuard is a registered trademark of Jason A. Donenfeld.



© 2023 Determinate Systems, Inc. All rights reserved.

[Terms of service](#) | [Code of conduct](#) | [DMCA policy](#) | [Usage policy](#) | [Security](#) | [Privacy](#) | [Status](#)

 Publish your Nix flakes!

Set up GitHub Actions to automatically push your flakes to FlakeHub.



[Xe](#)[Blog](#)[Contact](#)[Resume](#)[Talks](#)[VODs](#)[Signalboost](#)

Automagically assimilating NixOS machines into your Tailnet with Terraform

Published on 12/07/2022, 3629 words, 14 minutes to read



a girl, Phoenix girl, fluffy hair, pixie cut, red hair, red eyes, chuunibyou, war, a hell on earth, Beautiful and detailed explosion, Cold machine, Fire in eyes, burning, Metal texture, Exquisite cloth, Metal carving, volume, best quality, Metal details, Metal scratch, Metal defects, masterpiece, best quality, best quality, illustration, highres, masterpiece, contour deepening, illustration, (beautiful detailed girl), beautiful detailed glow, green necklace, green earrings, kimono, fan, grin - Eimis Anime Diffusion v1.0

For the sake of argument, let's say that you want to create all of your cloud infrastructure using [Terraform](#), but you also want to use [NixOS](#) and [Nix flakes](#). One of the main problems you will run into is the fact that Nix flakes and Terraform are both declarative and there's no easy way to shim Terraform states and Nix flake attributes. I think I've found a way to do this and today you're going to learn how to glue these two otherwise conflicting worlds together.



Requirements

In order to proceed with this tutorial as written, you will need to have the following things already set up:

- A [Tailscale](#) account.
- A [Scaleway](#) account.
- An amd64 Linux machine with Nix installed or an aarch64 Linux VM with either Apple Rosetta set up or `qemu-user` configured to let you run amd64 builds on an aarch64 host.
- An AWS [Route 53](#) domain set up.
- A GitHub account.

<Mara> Pedantically, Scaleway can be replaced with any other server host.



You can also remove all of the Tailscale-specific configuration. You can also use a different DNS provider. You may want to check [the Terraform registry](#) for your provider of choice. Most common and uncommon clouds *should* have a Terraform provider, but facts and circumstances may vary. GitHub can be replaced with any other git host.

I am also making the following assumptions when writing this tutorial:

- You have a Tailscale [ACL tag](#) named `tag:prod` that you can use [Tailscale SSH](#) to access.
- You have [Nix flakes enabled](#).
- The device running all this is on your tailnet.



Making a new GitHub repo

One of the first things you will need to do is [create a new GitHub repository](#). You can give it any name you like, but I named mine [automagic-terraform-nixos](#).

Once you have created your repo, clone it locally:

```
git clone git@github.com:Xe/automagic-terraform-nixos.git
```

Create a ` `.gitignore` file with the following entries in it:

```
result  
.direnv  
.env  
.terraform
```

Fetch credentials

Now that you have a new GitHub repository to store files in, you need to collect the various credentials that Terraform will use to control your infrastructure providers. For ease of use you will store them in a file called ` `.env` and use a shell command to load those values into your shell.

Variable	How to get it
`TAILSCALE_TAILNET`	Copy organization name from the admin panel .
`TAILSCALE_API_KEY`	Create an API key in the admin panel .
`SCW_ACCESS_KEY`	Create credentials in the console and copy the access key.
`SCW_SECRET_KEY`	Create credentials in the console and copy the secret key.

Next you will need to configure the AWS CLI, and by extension the default AWS API client.  AWS has [an excellent guide](#) on doing this that I will not repeat here.



<Mara> If you don't have the AWS CLI installed, use `nix run nixpkgs#awscli2` in place of the `aws` command in that documentation.

Finally, set all of those variables into your environment with this command:

```
export $(cat .env |xargs -L 1)
```



<Mara> If you do this often, you may want to alias this command to `loaddotenv` in your shell profile.

Configuring Terraform

In your git repo, create a new file called `main.tf`. This is where your Terraform configuration is going to live. You can use any name you like, but the convention is to use `main.tf` for the "main" resources and any supplemental resources can live in their own files.

One of the best practices with Terraform is to store its view of the world in a non-local store such as [Amazon S3](#). My state bucket is named `within-tf-state`, but your state bucket name will differ. Please see the upstream Terraform documentation for more information on how to establish such a state bucket.



<Mara> If you don't set up a state bucket, Terraform will default to storing its state in the current working directory. This state file will include generated secrets such as a Tailscale authkey. It is best to store this in S3 to avoid leaking secrets in your GitHub repository on accident.

```
# main.tf
terraform {
  backend "s3" {
    bucket = "within-tf-state"
    key    = "prod"
    region = "us-east-1"
  }
}
```



Now that you have the state backend set up, you need to declare the providers that this Terraform configuration will use. This will help ensure that Terraform is fetching the right providers from the right owners. Add this block of Terraform configuration right below the `backend "s3"` block you just declared:

```
# main.tf
terraform {
  # below the backend "s3" config

  required_providers {
    aws = {
      source = "hashicorp/aws"
    }

    cloudinit = {
      source = "hashicorp/cloudinit"
    }

    tailscale = {
      source = "tailscale/tailscale"
    }

    scaleway = {
      source = "scaleway/scaleway"
    }
  }
}
```

This configuration needs a few variables for things that are managed in the outside world. Scaleway requires that every resource is part of a "project", and you will need to put that project ID into your configuration. The Scaleway provider also allows us to have a default project ID, so you're going to put your project ID in a variable.

The Route 53 (AWS DNS) zone will also be put in its own variable.



```
# main.tf
variable "project_id" {
  type      = string
  description = "Your Scaleway project ID."
}
```

```
variable "route53_zone" {
    type      = string
    description = "DNS name of your route53 zone."
}
```

You can load your defaults into `terraform.tfvars`

```
# terraform.tfvars
project_id = "2ce6d960-f3ad-44bf-a761-28725662068a"
route53_zone = "xeserv.us"
```

Change your project ID and Route 53 zone name accordingly.

Once that is done, you can configure the Scaleway provider. If you want to have all resources default to being provisioned in Scaleway's Paris datacentre, you could use a configuration that looks like this:

```
# main.tf
provider "scaleway" {
    zone      = "fr-par-1"
    region    = "fr-par"
    project_id = var.project_id
}
```

Now that you have all of the boilerplate declared, you can get Terraform ready with the command `terraform init`. This will automatically download all the needed Terraform providers and set up the state file in S3.

`terraform init`



<Mara> If you don't already have terraform installed, you can run it without installing it by replacing `terraform` with `nix run nixpkgs#terraform` in any of these commands



Now that Terraform is initialized, you can import your Route 53 zone into your configuration by creating a `data` resource pointing to it:

```
# main.tf
data "aws_route53_zone" "dns" {
  name = var.route53_zone
}
```

To confirm that everything is working correctly, run `terraform plan` and see if it reports that it needs to create 0 resources:

```
terraform plan
```

If it reports that your DNS zone does not exist, please verify the configuration in `terraform.tfvars` and try again.

Create the Tailscale authkey for your new NixOS server using the `tailscale_tailnet_key` resource:

```
# main.tf
resource "tailscale_tailnet_key" "prod" {
  reusable      = true
  ephemeral     = false
  preauthorized = true
  tags          = ["tag:prod"]
}
```

Next you will need to create the cloud-init configuration for this virtual machine. Cloud-init is not exactly the best tool out there to manage this kind of assimilation, but it is widely adopted enough because it does the job well enough that you can rely on it.



There's many ways to create a cloud-init configuration in Terraform, but I feel that it's best to use the cloudinit provider for this. It will let you assemble a cloud-init configuration from multiple "parts", but this example will only use one part.

```
data "cloudinit_config" "prod" {
```

```
gzip          = false
base64_encode = false

part {
    content_type = "text/cloud-config"
    filename     = "nixos-infect.yaml"
    content = sensitive(<<EOT
#cloud-config
write_files:
- path: /etc/NIXOS_LUSTRE
  permissions: '0600'
  content: |
    etc/tailscale/authkey
- path: /etc/tailscale/authkey
  permissions: '0600'
  content: "${tailscale_tailnet_key.prod.key}"
- path: /etc/nixos/tailscale.nix
  permissions: '0644'
  content: |
    { pkgs, ... }:
    {
      services.tailscale.enable = true;

      systemd.services.tailscale-autoconnect = {
        description = "Automatic connection to Tailscale";
        after = [ "network-pre.target" "tailscale.service" ];
        wants = [ "network-pre.target" "tailscale.service" ];
        wantedBy = [ "multi-user.target" ];
        serviceConfig.Type = "oneshot";
        path = with pkgs; [ jq tailscale ];
        script = ''
          sleep 2
          status=$(tailscale status -json | jq -r .BackendState)
          if [ $status = "Running" ]; then # if so, then do nothing
            exit 0
          fi
          tailscale up --authkey $(cat /etc/tailscale/authkey) --ssh
        '';
      };
    };
  runcmd:
    - sed -i 's:#.*$::g' /root/.ssh/authorized_keys
    - curl https://raw.githubusercontent.com/elitak/nixos-infect/master/nixos-infect | NIXOS_IMP
EOT
)
```



```
}
```

At the time of writing, Scaleway doesn't have a prebaked NixOS image for creating new servers. One route you could take would be to make your own prebaked image and then customize it as you want, but I think it's more exciting to use [nixos-infect](#) to convert an Ubuntu install into a NixOS install. The `runcmd` block at the end of the cloud-config file tells cloud-init to run nixos-infect to rebuild the VPS into NixOS unstable, but you can change this to any other version of NixOS.



<Cadey> I personally use NixOS unstable on my servers because I value things being up to date and rolling release.

This sounds a bit arcane (and at some level it is), but at a high level it relies on the `/etc/NIXOS_LUSTRATE` file as described in [the NixOS manual section on installing NixOS from another Linux distribution](#). You will use cloud-init in the Ubuntu side to plop down the tailscale authkey into `/etc/tailscale/authkey` on the target machine and then making sure it gets copied to the NixOS install by putting the path `/etc/tailscale/authkey` into the `NIXOS_LUSTRATE` file.

One of the other things you *could* do here is install Tailscale and authenticate to its control plane in the Ubuntu side and then add `var/lib/tailscale` to the `NIXOS_LUSTRATE` file, but I feel that could take a bit longer than it already takes to infect the cloud instance with NixOS.

One of the features that nixos-infect has is the ability to customize the target NixOS install with arbitrary Nix expressions. This configuration puts a NixOS module into `/etc/nixos/tailscale.nix` that does the following:



- Enables Tailscale's node agent tailscaled
- Creates a systemd oneshot job (something that runs as a one-time script rather than a persistent service) that will authenticate the machine to Tailscale and set up [Tailscale SSH](#)

The oneshot will read the relevant authkey from `/etc/tailscale/authkey`, which is why it is moved over from Ubuntu.



<Cadey> Strictly speaking, you don't *have to* create a floating IP address to attach to the server, but it is the best practice to do this. If you replace your production host in the future it may be a good idea to have its IPv4 address remain the same. DNS propagation takes *forever*.

```
resource "scaleway_instance_server" "prod" {  
  type = "DEV1-S"  
  image = "ubuntu_jammy"  
  ip_id = scaleway_instance_ip.prod.id  
  enable_ipv6 = true  
  cloud_init = data.cloudinit_config.prod.rendered  
  tags = ["nixos", "http", "https"]  
}
```

Finally you can create `prod.your.domain` DNS entries with this configuration:

```
resource "aws_route53_record" "prod_A" {  
  zone_id = data.aws_route53_zone.dns.zone_id  
  name    = "prod"  
  type    = "A"  
  records = [scaleway_instance_ip.prod.address]  
  ttl     = 300  
}  
  
resource "aws_route53_record" "prod_AAAA" {  
  zone_id = data.aws_route53_zone.dns.zone_id  
  name    = "prod"  
  type    = "AAAA"  
  records = [scaleway_instance_server.prod.ipv6_address]  
  ttl     = 300  
}
```



<Mara> The reason behind creating two separate DNS entries is an exercise



 for the reader.

```
{  
  inputs = {  
    nixpkgs.url = "nixpkgs/nixos-unstable";  
    flake-utils.url = "github:numtide/flake-utils";  
  };  
  
  outputs = { self, nixpkgs, flake-utils }:  
    let  
      mkSystem = extraModules:  
        nixpkgs.lib.nixosSystem rec {  
          system = "x86_64-linux";  
          modules = [  
            # bake the git revision of the repo into the system  
            ({ ... }: { system.configurationRevision = self.sourceInfo.rev; })  
            ] ++ extraModules;  
        };  
        in flake-utils.lib.eachSystem [ "x86_64-linux" "aarch64-linux" ] (system:  
          let pkgs = import nixpkgs { inherit system; };  
          in rec {  
            devShells.default =  
              pkgs.mkShell { buildInputs = with pkgs; [ terraform awscli2 ]; };  
          } // { # TODO: put nixosConfigurations here later  
        );  
    }  
}
```

The outputs function may look a bit weird here, but we're doing two things with it:

- Creating a development environment (devShell) with terraform and the AWS cli installed for both amd64 and aarch64 linux systems
- Setting up for defining `nixosConfigurations` dynamically

It's also worth noting that the `mkSystem` function defined at the top of the outputs function will bake in the git commit of the custom configuration into the resulting NixOS configuration. This will make it impossible to deploy changes that are not committed to git.



Gluing the two worlds together

Now you can do the exciting bit: glue the two worlds of Nix flakes and Terraform together using the `\`local-exec` provisioner` and a shell script like this:

```
#!/usr/bin/env bash

set -e
[ ! -z "$DEBUG" ] && set -x

USAGE(){
    echo "Usage: `basename $0` <server_name>"
    exit 2
}

if [ -z "$1" ]; then
    USAGE
fi

server_name="$1"
public_ip="$2"

ssh_ignore(){
    ssh -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no $*
}

ssh_victim(){
    ssh_ignore root@"${public_ip}" $*
}

mkdir -p "./hosts/${server_name}"
echo "${public_ip}" >> ./hosts/"${server_name}"/public-ip

until ssh_ignore "root@${server_name}" uname -av
do
    sleep 30
done

scp -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no "root@${server_name}:/etc/nixos/flakes/default.nix" ./hosts/"${server_name}"/default.nix
cat <<EOC >> ./hosts/"${server_name}"/default.nix
{ ... }: {
    imports = [ ./hardware-configuration.nix ];
    boot.cleanTmpDir = true;
```



```

zramSwap.enable = true;
networking.hostName = "${server_name}";
services.openssh.enable = true;
services.tailscale.enable = true;
networking.firewall.checkReversePath = "loose";
users.users.root.openssh.authorizedKeys.keys = [
  "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAIM6NPbPIcCTzeEsjyx0goWyj6fr2qzcfKCCd0Uqg0N/v" # alres
];
system.stateVersion = "23.05";
}

EOC

git add .
git commit -sm "add machine ${server_name}: ${public_ip}"
nix build .#nixosConfigurations."${server_name}".config.system.build.toplevel

export NIX_SSHOPTS='-o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no'
nix-copy-closure -s root@"${public_ip}" $(readlink ./result)
ssh_victim nix-env --profile /nix/var/nix/profiles/system --set $(readlink ./result)
ssh_victim $(readlink ./result)/bin/switch-to-configuration switch

git push

```

Add the provisioner script to your `scaleway_instance_server` by adding this block of configuration right at the end of its definition:

```

# main.tf
resource "scaleway_instance_server" "prod" {
  # ...

  provisioner "local-exec" {
    command = "${path.module}/assimilate.sh ${self.name} ${self.public_ip}"
  }

  provisioner "local-exec" {
    when      = destroy
    command = "rm -rf ${path.module}/hosts/${self.name}"
  }
}

```



This will trigger the `assimilate.sh` script to run every time a new instance is created and delete host-specific configuration when an instance is destroyed.

Then you can hook up the `nixosConfigurations` output to the folder structure that script creates by adding the following configuration to your `flake.nix` file:

```
) // {  
    nixosConfigurations = let hosts = builtins.readDir ./hosts;  
    in builtins.mapAttrs (name: _: mkSystem [ ./hosts/${name} ]) hosts;  
};
```

This works because I am making hard assumptions about the directory structure of the `hosts` folder in your git repository. When I wrote this configuration, I assumed that the `hosts` folder would look something like this:

```
hosts  
└─ tf-srv-naughty-perlman  
    ├─ default.nix  
    ├─ hardware-configuration.nix  
    └─ public-ip
```

Each host will have its own folder named after itself with configuration in `default.nix` and that will point to any other relevant configuration (such as `hardware-configuration.nix`). Because this directory hierarchy is predictable, you can get a listing of all the folders in the `hosts` directory using the `builtins.readDir` function:

```
nix-repl> builtins.readDir ./hosts  
{ tf-srv-naughty-perlman = "directory"; }
```

Then you can use `builtins.mapAttrs` to loop over every key->value pair in the attribute set that `builtins.readDir` returns and convert the hostnames into NixOS system definitions:

```
nix-repl> hosts = builtins.readDir ./hosts  
nix-repl> builtins.mapAttrs (name: _: ./hosts/${name}) hosts  
{ tf-srv-naughty-perlman = /home/cadey/code/Xe/automagic-terraform-nixos/hosts/tf-srv-naughty-
```





<Mara> The rest of this is an exercise for the reader.

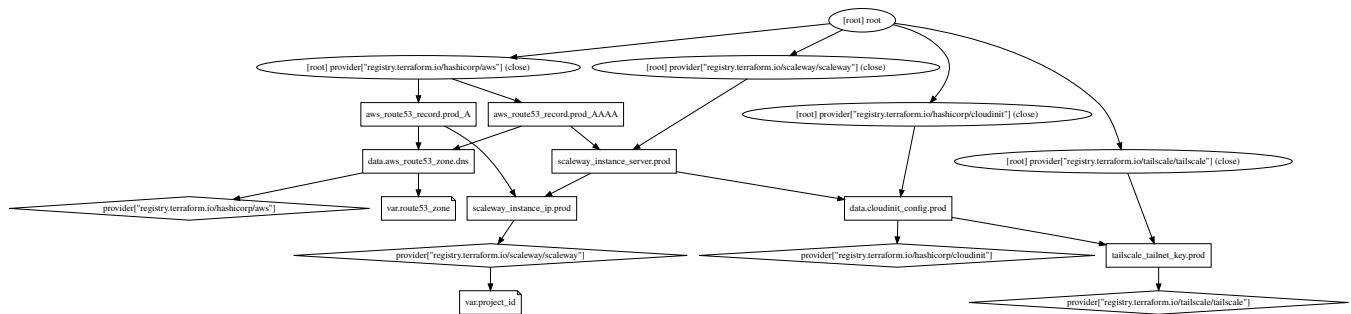
Creating your server

Finally, now that everything is put into place you can create your server using `terraform apply`:

```
terraform apply
```

Terraform will print off a list of things that it thinks it needs to do. Please read this over and be sure that it's proposing a plan that makes sense to you. When you are satisfied that Terraform is going to do the correct thing, follow the instructions it gives you. If you are not satisfied it's going to do the correct thing, press control-c.

Let it run and it will automatically create all of the infrastructure you declared in `main.tf`. The entire graph of infrastructure should look something like this:



<Mara> If that is too small for you, click [here](#). There is a lot going on in the graph because Terraform lists everything and its ultimate dependents.



You can SSH into the server using this command:

```
ssh root@generated-server-name
```

Manually pushing configuration changes

There are many NixOS tools that you can use to push configuration changes like [deploy-rs](#), but you can also manually push configuration changes by following these three steps:

- Build the new system configuration for the target machine
- Copy the system configuration to the target machine
- Activate that new configuration

You can automate these steps using a script like the following:

```
#!/usr/bin/env bash
# pushify.sh

set -e
[ ! -z "$DEBUG" ] && set -x

# validate arguments
USAGE(){
    echo "Usage: `basename $0` <server_name>"
    exit 2
}

if [ -z "$1" ]; then
    USAGE
fi

server_name="$1"
public_ip=$(cat ./hosts/${server_name}/public-ip)

ssh_ignore(){
    ssh -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no $*
}

ssh_victim(){
    ssh_ignore root@${public_ip} $*
}

# build the system configuration
nix build .#nixosConfigurations."${server_name}".config.system.build.toplevel
```



```
# copy the configuration to the target machine
export NIX_SSHOPTS=' -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no'
nix-copy-closure -s root@"${public_ip}" $(readlink ./result)

# register it to the system profile
ssh_victim nix-env --profile /nix/var/nix/profiles/system --set $(readlink ./result)

# activate the new configuration
ssh_victim $(readlink ./result)/bin/switch-to-configuration switch
```

You can use it like this:

```
./pushify.sh generated-server-name
```

Rollbacks

To roll back a configuration, SSH into the server and run `nixos-rebuild --rollback switch`.

Setting up automatic updates

One of the neat and chronically underdocumented features of NixOS is the [system.autoUpgrade](#) module. This allows a NixOS system to periodically poll for changes in its configuration or updates to NixOS itself and apply them automatically. It will even reboot if the kernel was upgraded.

In order to set it up, create a folder named `common` and put the following file in it:

```
# common/default.nix
{ ... }: {
    system.autoUpgrade = {
        enable = true;
        # replace this with your GitHub repo
        flake = "github:Xe/automagic-terraform-nixos";
    };
}
```



Then add `./common` to the list of modules in the `mkSystem` function like this:

```
mkSystem = extraModules:  
  nixpkgs.lib.nixosSystem rec {  
    system = "x86_64-linux";  
    modules = [  
      ./common  
      ({ ... }: { system.configurationRevision = self.sourceInfo.rev; })  
    ] ++ extraModules;  
  };
```

Commit these changes to git and deploy the configuration to your server:

```
git add .  
git commit -sm "set up autoUpgrade"  
git push  
./pushify.sh generated-server-name
```

Your NixOS machines will automatically pull changes to your GitHub repository once per day somewhere around `04:40` in the morning, local time. You can manually trigger this by running the following command:

```
ssh root@generated-server-name  
systemctl start nixos-upgrade.service  
journalctl -fu nixos-upgrade.service
```

Exercises for the reader

This tutorial has told you everything you need to know about setting up new NixOS servers with Terraform. Here are some exercises that you can do to help you learn new and interesting things about configuring your new NixOS machines:



- Set up backups to borgbase.
- Set up encrypted secret management with agenix.
- Create an AWS IAM user for your machine and copy the secret files to it. How would you do that programmatically with a new machine? Hint: `NIXOS_LUSTRATE` can help! Use that

for Let's Encrypt.

- Try some of the services listed in the [NixOS manual](#). How would you expose one of them over Tailscale?
- How would you make an instance on [Vultr](#) using this Terraform manifest? How about [Digital Ocean](#)?
- How would you attach a [VPC](#) to your server and expose it to your other machines as a [subnet router](#) with Tailscale?
- Set up a [Pleroma server](#). Be sure to use [Let's Encrypt](#) to get an HTTPS certificate!

I hope this was enlightening! Enjoy your new servers and have fun exploring things in NixOS!



Facts and circumstances may have changed since publication. Please contact me before jumping to conclusions if something seems wrong or unclear.

Tags: Terraform, NixOS, Scaleway

Copyright 2012-2024 Xe Iaso (Christine Dodrill). Any and all opinions listed here are my own and not representative of any of my employers, past, future, and/or present.

Like what you see? Donate on [Patreon](#) like [these awesome people](#)!

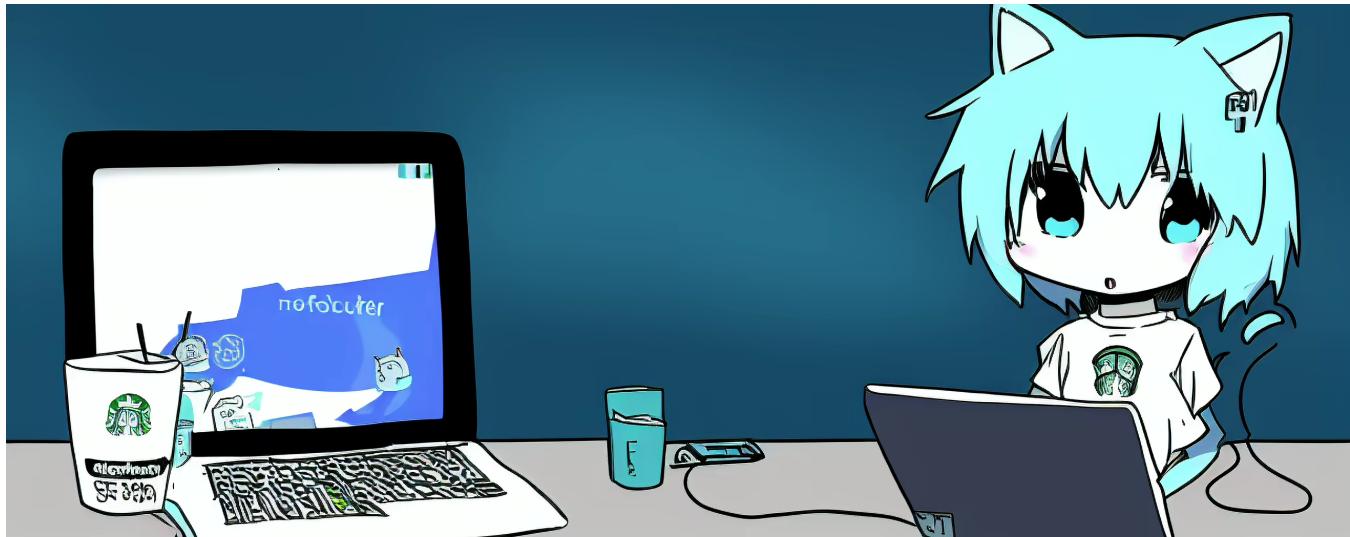
Served by xesite v4 (/nix/store/9rxy3y9y9ffwx6xdvb56pgciar4b5138-xesite_v4-20240128 /bin/xesite) with site version [a1841325](#), source code available [here](#).



[Xe](#)[Blog](#)[Contact](#)[Resume](#)[Talks](#)[VODs](#)[Signalboost](#)

Building Go programs with Nix Flakes

Published on 12/14/2022, 823 words, 3 minutes to read



Baby blue gopher, laptop computer, starbucks, 1girl, hacker vibes, manga, thick outlines, evangelion, angel attack, chibi, cat ears - Waifu Diffusion v1.3 (float16)

Sometimes you wake up and realize that reality has chosen violence against you. The consequences of this violence mean that it's hard to cope with the choices that other people have made for you and then you just have to make things work. This is the situation that I face when compiling things written in [Go](#) in my NixOS configurations.

However, I have figured out a way past this wicked fate and have forged a new path. I have found [`gomod2nix`](#) to help me out of this pit of philosophical doom and despair. To help you understand the solution, I want to take a moment to help you understand the problem and why it is such a huge pain in practice.

The problem

Most package management ecosystems strive to be deterministic. This means that the package managers want to make sure that the same end state is achieved if the same inputs and commands are given. For a long time, the Go community just didn't have a story for making package management deterministic at all. This lead to a cottage industry of a billion version management tools that were all mutually incompatible and lead people to use overly complicated dependency resolution strategies.

At some point people at Google had had enough of this chaos (even though they aren't affected by it due to all of their projects not using the Go build system like everyone else) and the [vgo proposal](#) was unleashed upon us all. One of the things that Go modules (then vgo) offered was the idea of versioned dependencies for projects. This works decently enough for the Go ecosystem and gives people easy ways to create deterministic builds even though their projects rely on random GitHub repositories.

The main problem from the NixOS standpoint is that the Go team uses a hash method that is not compatible with Nix. They also decided to invent their own configuration file parsers for some reason, these don't have any battle-tested parsers in Nix. So we need a bridge between these two worlds.



<Mara> There are many ways to do this in NixOS, however `gomod2nix` is the only way we are aware of that uses a tool to code-generate a data file full of hashes that Nix *can* understand. In upstream nixpkgs you'd use something like [`buildGoModule`](#), however you have a lot more freedom with your own projects.

Getting started with new projects

One of the easiest ways to set this up for a new Go project is to use their Nix template. To do this, [enable flakes](#) and run these commands in an empty folder:



```
nix flake init -t github:nix-community/gomod2nix#app  
git init
```

Then add everything (including the generated `gomod2nix.toml`) to git with `git add`:

```
git add .
```



<Mara> This is needed because Nix flakes respects gitignores. If you don't add things to the git staging area, git doesn't know about the files at all, and Nix flakes can't know if it should ignore them.

Then you can enter a development environment with `nix develop` and build your program with `nix build`. When you add or remove dependencies from your project, you need to run `gomod2nix` to fix the `gomod2nix.toml`.

```
gomod2nix
```

Grafting it into existing projects

If you already have an existing Go program managed with Nix flakes, you will need to add `gomod2nix` to your flake inputs, nixpkgs overlays, and then use it in your `packages` output. Add this to your `inputs`:

```
{
  inputs = {
    nixpkgs.url = "nixpkgs/nixos-unstable";
    utils.url = "github:numtide/flake-utils";

    gomod2nix = {
      url = "github:tweag/gomod2nix";
      inputs.nixpkgs.follows = "nixpkgs";
      inputs.utils.follows = "utils";
    };
  };
}
```



Then you will need to add it to the arguments in your `outputs` function:

```
outputs = { self, nixpkgs, utils, gomod2nix }:
```

And finally apply its overlay to your `nixpkgs` import. This may differ with how your flake works, but in general you should look for something that imports the `nixpkgs` argument and add the `gomod2nix` overlay to it something like this:

```
let pkgs = import nixpkgs {
  inherit system;
  overlays = [ gomod2nix.overlays.default ];
};
```

You can then use `pkgs.buildGoApplication` as [the upstream documentation](#) suggests. If you want a more complicated example of using `buildGoApplication`, check [my experimental repo](#).



<Mara> If you want to expose the `gomod2nix` tool in a devShell, add `gomod2nix.packages.\${system}.default` to the `buildInputs` list. The total list of tools could look like this:

```
devShells.default = pkgs.mkShell {
  buildInputs = with pkgs; [
    go
    gopls
    gotools
    go-tools
    gomod2nix.packages.${system}.default
    sqlite-interactive
  ];
};
```

Then everything will work as normal.



[Share](#)

Facts and circumstances may have changed since publication. Please contact me before jumping to conclusions if something seems wrong or unclear.

Tags: golang, nix, nixos

Copyright 2012-2024 Xe Iaso (Christine Dodrill). Any and all opinions listed here are my own and not representative of any of my employers, past, future, and/or present.

Like what you see? Donate on [Patreon](#) like [these awesome people!](#)

Served by xesite v4 (/nix/store/9rxy3y9y9ffwx6xdvb56pgciar4b5138-xesite_v4-20240128/bin/xesite) with site version [a1841325](#) , source code available [here](#).



Circumventing Network Bans with WireGuard

2024-01-09 :: LGUG2Z

#nixos #plex #hetzner #self-hosted #networking #mullvad #wireguard
#vpn

Before this week, it had been a long time since I visited the [Plex subreddit](#).

I shared my [last article](#) there, which was a technical write-up of moving my Plex instance from a Hetzner auction server to a virtual machine running on hardware in my home network, and the considerations that influenced the migration.

It didn't take long for me to realize that a culture of hostility towards even the *mention* of Hetzner or other cloud hosting providers has strongly taken root since Plex announced its blanket network ban on IP ranges associated with Hetzner data centers.

I saw many posts and comments of users asking about issues with their Plex instances that had for years been working without issue on Hetzner servers until this past October when Plex enacted their very poorly communicated network ban, which hit a significant number of customers like myself who had paid for a lifetime Plex Pass.

Although I myself am not pursuing this option for reasons outlined in my last article, I wanted to share a clear and detailed example of how to circumvent Plex's ban on IPs originating from Hetzner data centers (because gatekeeping is for losers).



WireGuard VPN Connection Details

This can work with any WireGuard VPN provider (even with your own WireGuard server on another machine!) but for the sake of simplicity I have chosen to

use [Mullvad](#) as the reference in this tutorial.

- Go to [mullvad.net](#) and open an account
 - This is actually a very cool process; no email, no details, they just provide you a secret account ID
- Once you have an account, navigate to “[Add time to your account](#)”
 - You can just add 1 month of time for 5 EUR if you want to try out the quality of their service
- Head over to the [WireGuard Configuration](#) page
 - Select “Linux” and then hit “Generate key”
 - Select a Country, City and Server for your exit location (bottom of the page)
 - Scroll down a little more to hit “Download file” and get your authentication details
 - You’ll only be able to do this once! Make sure you do it before you navigate away

At this point you’ll have a `.conf` file containing the fields `Interface.PrivateKey` and `Interface.Address` which you’ll need later.

Server Configuration

Below is a fully annotated NixOS server configuration which sets some sane server defaults, configures SSH access, firewall rules, and brings up a Plex container which sends all outgoing requests through a WireGuard VPN using your new connection details.

This server configuration does not include hardware configuration, which is naturally prone to variation, especially on auction servers, however it should not be too difficult to adapt my [nixos-hetzner-robot-starter](#) template ([video walkthrough](#)) to work with your server’s hardware.



```
{
  config,
}: let
  # These are helper functions to look up uids and gids
  # which take a single string argument
  #
```

```

# eg. uid "samira" → returns the uid for the "samira" user
uid = username:
  if config.users.users.${username}.uid == null
  then "1000"
  else toString config.users.users.${username}.uid;

# eg. gid "users" → returns the gid for the "users" group
gid = group:
  if config.users.groups.${group}.gid == null
  then "100"
  else toString config.users.groups.${group}.gid;

# FIXME: Set your username for this server
username = "<YOUR PREFERRED USERNAME>";

# FIXME: Set this or you won't be able to SSH
publicKeys = [
  "<YOUR PUBLIC KEY>"
  "<OPTIONALLY ANY OTHER PUBLIC KEYS OF YOURS>"
];

in {
  # This stops Docker interference with dhcp
  networking.dhcpcd.denyInterfaces = ["veth*"];

  # This allows incoming connections on port 32400 for Plex
  networking.firewall.allowedTCPPorts = [32400];

  # This sets the hostname of the server, this can be anything you like
  networking.hostName = "plex-on-hetz";

  # This is so that users in the "wheel" group don't need to
  # enter their password for sudo commands
  security.sudo.wheelNeedsPassword = false;

  # This enables SSH access to the server and only allows
  # root SSH connections with SSH keys, never with passwords
  services.openssh = {
    enable = true;
    settings.PermitRootLogin = "prohibit-password";
  };
}

```



```
# This sets the SSH public key(s) you can use to connect
# to the server with the "root" user
users.users.root.openssh.authorizedKeys.keys = publicKeys;

# This creates your user on the server
users.users.${username} = {
    # This specifies that you are a user that gets home directory
    isNormalUser = true;
    # This adds your user to the "wheel" and "docker" groups
    # In the "wheel" group, you don't need to use your password for sudo
    # In the "docker" group, you don't need to use sudo for docker commands
    extraGroups = ["wheel" "docker"];

    # This sets the SSH public key(s) you can use to connect
    # to the server with your user
    openssh.authorizedKeys.keys = publicKeys;
};

# This is a service that bans hosts and IPs that produce authentication
# errors when trying to SSH multiple times in quick succession
services.fail2ban = {
    enable = true;
    # FIXME: Add this so you don't get locked out by mistake
    ignoreIP = ["<YOUR HOME IP ADDRESS>"];
};

# This enables Docker, makes sure it runs when the server starts
# and automatically prunes dangling resources to keep space free
virtualisation.docker = {
    enable = true;
    enableOnBoot = true;
    autoPrune.enable = true;
};

# This is where you can add Docker containers, you can think of this
# block as being conceptually similar to docker-compose in some ways
virtualisation.oci-containers = {
    backend = "docker";

    # This is the gluetun container: https://github.com/qdm12/gluetun
    # gluetun is a thin docker container for multiple VPN providers that
    # supports WireGuard
```



```

containers.gluetun = {
    # This is so that the container starts automatically after server reboot
    autoStart = true;
    image = "qmcgaw/gluetun:v3.37.0";
    # This is so that we can access Plex on http://localhost:32400
    # from the server, ie. outside of the container
    ports = [
        "32400:32400"
    ];
    # This is where you enter authentication information you get from Mullvad
    environment = {
        VPN_SERVICE_PROVIDER = "mullvad";
        VPN_TYPE = "wireguard";
        # FIXME: Don't forget to add your connection details here!
        WIREGUARD_PRIVATE_KEY = "<Interface.PrivateKey in your downloaded config file";
        WIREGUARD_ADDRESSES = "<Interface.Address in your downloaded config file";
    };
    # This capability is required by gluetun
    extraOptions = [
        "--cap-add=NET_ADMIN"
    ];
};

# This is the Plex container
containers.plex = {
    # This is so that the container starts automatically after server reboot
    autoStart = true;
    image = "plexinc/pms-docker:1.32.8.7639-fb6452ebf";
    # This is so that the `plex` user inside of the container has the same
    # UID and GID as your user to avoid permissions issues with any directories
    # that are mounted
    environment = {
        PLEX_UID = uid username;
        PLEX_GID = gid "users";
    };
    volumes = [
        # This is so that the configuration can be persisted outside of the container
        # in the ${HOME}/plex directory on the server
        #
        # NOTE: If you are migrating an instance started with `services.plex` then
        # will need to set this as
        # "/path/to/your/plex-nixos/config/dir:/config/Library/Application\ Support\Plex\ Media\ Server\ Configuration"
    ];
};

```



```

"/home/${username}/plex:/config"
# This is where you share your media files that are under your user a
# on the server with the Plex container so they can be seen, indexed .
#
# NOTE: If you are migrating an instance started with `services.plex.
# will need to make sure the paths inside the container (on the right
# match the paths that your Plex instance running as a NixOS service
#
# TODO: Whatever makes sense for you
"/home/${username}/path/to/tv:/data/tv"
"/home/${username}/path/to/movies:/data/movies"
"/home/${username}/path/to/music:/data/music"
];
# This is to make sure that this container won't start until the `gluet
# has started and is healthy
dependsOn = ["gluetun"];
# This is to make sure that network requests from this container go thr
# WireGuard VPN running in the `gluetun` container
#
# This is the key part that allows us to circumvent Plex's Hetzner netw
# it ensures that requests to https://*.plex.tv endpoints look like the
# from whichever WireGuard server we are connected to in the `gluetun` -
#
# This setup also ensures that other processes running on the server ca
# sending outgoing HTTP requests normally without going through the VPN
extraOptions = [
    "--network=container:gluetun"
];
};

};

}

```

If you have any questions or comments you can reach out to me on [Twitter](#) ·  [Mastodon](#).

If you're interested in what I read to come up with solutions like this one, you can subscribe to my [Software Development RSS feed](#).

If you'd like to watch me writing code while explaining what I'm doing, you can also [subscribe to my YouTube channel](#).

If you found this content valuable, or if you are a happy user of [komorebi](#) or my [NixOS starter templates](#), please consider sponsoring me on [GitHub](#) or tipping me on [Ko-fi](#).

Mullvad and **Hetzner**: [Please feel free](#) to give me some free VPN time / compute power for all this free positive PR ;)

© 2024 Powered by [Hugo](#) :: [Theme](#) made by [panr](#)



Cloudflare and NixOS Tips When Deploying a Personal Mastodon Server

2024-01-14 :: LGUG2Z

#nixos #mastodon #cloudflare #fediverse #self-hosted

For the most part I feel very much at home on the Hachyderm Mastodon server; it's probably the best social media experience that I can remember having and I have had the pleasure of interacting with so many cool and impassioned people there.

Hachyderm implements the default 500 character post limit which is hard-coded into the Mastodon codebase and as of writing these, seems unlikely to ever be made configurable.

Every now and then, especially when adding summaries to long (1hr+) live programming videos that I share across the Fediverse, I come up against that limit.

At the end of last year, I had an idea: Why don't I just self-host my own Mastodon instance that allows for posts that are longer than 500 chars, make longer posts on that account, and then boost them from my main account on Hachyderm?

Sounds easy, right? Well...

Using my domain



I wanted to be able to use my current domain so that I could be looked up as `@jeeezy@lgug2z.com`. This is actually quite well documented and can be done by setting the `WEB_DOMAIN` environment variable.

> `WEB_DOMAIN` is an optional environment variable allowing the installation of Mastodon on one domain, while having the users' handles on a different domain, e.g. addressing users as `@alice@example.com` but accessing Mastodon on `mastodon.example.com`. This may be useful if your domain name is already used for a different website but you still want to use it as a Mastodon identifier because it looks better or shorter.

To install Mastodon on `mastodon.example.com` in such a way it can serve `@alice@example.com`, set `LOCAL_DOMAIN` to `example.com` and `WEB_DOMAIN` to `mastodon.example.com`. This also requires additional configuration on the server hosting `example.com` to redirect requests from `https://example.com/.well-known/webfinger` to `https://mastodon.example.com/.well-known/webfinger`.

In my case, I set `WEB_DOMAIN` to `social.lgug2z.com` and `LOCAL_DOMAIN` to `lgug2z.com`.

However, `lgug2z.com` currently hosts a Hugo website that is deployed to Cloudflare Pages.

“No problem”, I thought to myself, “I’ll just set up a redirect rule.”

This was in fact, a big problem, as after hours of trying to debug federation issues between my new instance and other Mastodon servers, I realized that **Cloudflare’s Redirect Rules do nothing on a URI path where a Cloudflare Pages site is deployed!**

It would have been nice if this edge case was **clearly** documented somewhere by Cloudflare.

My solution for this was to just add a `.well-known/webfinger` file to the `static` folder of my Hugo site and populate it with the JSON payload returned from `https://social.lgug2z.com/.well-known/webfinger?resource=acct:jeozy@lgug2z.com`. While this is not a particularly elegant solution, it does the job for a single-user Mastodon instance.



Deploying a custom build of Mastodon on NixOS

As previously mentioned, the 500 character post limit is hard-coded in the Mastodon codebase.

There is a [very detailed post](#) about how to change the source code to set a higher character limit that is kept up to date by @chris@fosstodon.org, which helpfully includes git patch files.

I initially tried overriding `pkgs.mastodon` both directly in the service definition and via a NixOS overlay to apply this patch, to no avail.

When I tried forking Mastodon and applying the latest patch to the v4.2.3 release, the patch application failed, so I just cut myself a `release/v4.2.3` branch on my fork and [made the changes there](#).

I once again set an override, this time for `src`, to build Mastodon from this revision on my fork, and managed to get a little bit further. Now, querying <https://social.lgug2z.com/api/v2/instance> showed that `configuration.statuses.max_characters` was indeed set to my new value of `5000`, but this was not reflected in the UI.

Digging around in my generated Caddyfile at `/etc/caddy/caddy_config`, showed that although I was referencing `cfg.package` (instead of `pkgs.mastodon`) from my NixOS module when configuring Caddy, I was still being routed to the frontend files of `pkgs.mastodon` without my `src` override.

After a lot of trial, error and GitHub code searches prefixed with `lang:nix`, I found [some code](#) by @sandro@c3d2.social which suggested that in addition to overriding `src`, I should also override `mastodonModules` in order to ensure that the Mastodon UI would be served not from `pkgs.mastodon`, but from my override.

This was indeed the missing piece, and once I also added an override for `mastodonModules.src`, I was able to see the updated UI with the new 5000 maximum character post limit!



My `mastodon.nix` module

If you are reading this article because you're trying to achieve something similar [ap:w](#), here is my `mastodon.nix` module which uses Caddy as a reverse proxy to serve a custom build of Mastodon on a subdomain (in my case, `social.lgug2z.com`) while allowing your server to show up on the Fediverse as the root domain (in my case, `jeezy@lgug2z.com`).

```
{  
  config,  
  pkgs,  
  domain,  
  ...  
}: let  
  cfg = config.services.mastodon;  
in {  
  services.mastodon = {  
    enable = true;  
    package = pkgs.mastodon.overrideAttrs (_: let  
      pname = "mastodon-lgug2z";  
      src = pkgs.fetchFromGitHub {  
        owner = "LGUG2Z";  
        repo = "mastodon";  
        # forked from v4.2.3 with max chars set to 5000  
        rev = "b24adb69fa41a580fa2781a44661b7b707e3f765";  
        hash = "sha256-BvcvUAcIW5lYT1gTrKsIVIbmDQpAE3KxOiLLWoUtYhw=";  
      };  
      in {  
        inherit src pname;  
        mastodonModules = pkgs.mastodon.mastodonModules.overrideAttrs (_: {  
          inherit src;  
          pname = "${pname}-modules";  
        });  
      );  
    );  
  
    localDomain = "${domain}";  
    extraConfig = {  
      WEB_DOMAIN = "social.${domain}";  
      SINGLE_USER_MODE = "true";  
    };  
    configureNginx = false;  
    smtp.fromAddress = "";  
  };  
}
```



```
streamingProcesses = 1;

};

networking.firewall.allowedTCPPorts = [80 443];

users.users.caddy.extraGroups = ["mastodon"];
systemd.services.caddy.serviceConfig.ReadWriteDirectories = pkgs.lib.mkForc

services.caddy = {
  enable = true;
  virtualHosts."${cfg.extraConfig.WEB_DOMAIN}".extraConfig = ''
    handle_path /system/* {
      file_server * {
        root /var/lib/mastodon/public-system
      }
    }

    handle /api/v1/streaming/* {
      reverse_proxy unix//run/mastodon-streaming/streaming.socket
    }

    route * {
      file_server * {
        root ${cfg.package}/public
        pass_thru
      }
      reverse_proxy * unix//run/mastodon-web/web.socket
    }

    handle_errors {
      root * ${cfg.package}/public
      rewrite 500.html
      file_server
    }

    encode gzip

    header /* {
      Strict-Transport-Security "max-age=31536000;"
    }

    header /emoji/* Cache-Control "public, max-age=31536000, immutable"
}
```



```
header /packs/* Cache-Control "public, max-age=31536000, immutable"
header /system/accounts/avatars/* Cache-Control "public, max-age=31536000, immutable"
header /system/media_attachments/files/* Cache-Control "public, max-age=31536000, immutable"
";
};

}
```

If you have any questions or comments you can reach out to me on [Twitter](#) and [Mastodon](#).

If you're interested in what I read to come up with solutions like this one, you can subscribe to my [Software Development RSS feed](#).

If you'd like to watch me writing code while explaining what I'm doing, you can also [subscribe to my YouTube channel](#).

If you found this content valuable, or if you are a happy user of [komorebi](#) or my [NixOS starter templates](#), please consider sponsoring me on [GitHub](#) or tipping me on [Ko-fi](#).

© 2024 Powered by [Hugo](#) :: [Theme](#) made by [panr](#)



Deploying a Cloudflare R2-Backed Nix Binary Cache (Attic!) on Fly.io

2024-01-16 :: LGUG2Z

#nixos #cloudflare #fly.io #attic

I have tried running the [Attic Nix Binary Cache](#) on my Hetzner dedicated server in Germany a few times in the past, but the peering issues and the latency to Xfinity in Seattle have always made me throw my hands up in frustration.

This morning I noticed [a comment](#) by Zhaofeng on the repo issue tracker.

> As a NixOS aficionado myself, I begrudgingly admit that I've been running my instance on fly.io 😊

I'm not sure if this comment is still current, but hey, if Zhaofeng is/was running his binary cache on [fly.io](#), there's no reason why we can't too, right?

Storage

While Attic does support local storage, I figured I'd use Cloudflare's R2 a storage backend, both as an excuse to try out the [free tier](#) (10GB), and because it'll be easy enough to lift-and-shift if I ever decide to move the server from fly.io.

It's easy enough to create a new R2 bucket and grab a read-write API token scoped to the bucket on the [Cloudflare Dashboard](#).



Server Configuration

Once we have our credentials, we can put together a configuration file for the Attic server based on the [example config](#).

```
listen = "[::]:8080"
token-hs256-secret-base64 = "<generate this with openssl rand 64 | base64 -w0

[database]
url = "sqlite:///data/attic.db?mode=rwc"

[storage]
bucket = "<your bucket name>"
type = "s3"
region = "auto"
endpoint = "https://<your cloudflare account id>.r2.cloudflarestorage.com"

[storage.credentials]
access_key_id = "<your access key id>"
secret_access_key = "<your secret access key>

[chunking]
nar-size-threshold = 65536
min-size = 16384
avg-size = 65536
max-size = 262144

[compression]
type = "zstd"

[garbage-collection]
interval = "12 hours"
```



One thing to note is that we are storing our SQLite database file at `/data/attic.db` – this is going to be a fly volume that we'll create soon.

You should probably `git-crypt` this file (or encrypt it with `sops` or `age`) in your configuration repo since it has sensitive credentials.

Not sure about how to handle secrets in NixOS configuration repos? I have a big old article all about handling secrets in NixOS!

Dockerfile

Luckily, there is an automatically published Docker image available for us to extend. There isn't much to do here except bring in our configuration file, reference it when we start `atticd` and ensure it's running in `monolithic` mode.

```
FROM ghcr.io/zhaofengli/attic:latest
COPY ./server.toml /attic/server.toml
EXPOSE 8080
CMD ["-f", "/attic/server.toml", "--mode", "monolithic"]
```

Fly Config

The last piece of the deployment puzzle is to put together a `fly.toml` file for our `atticd` instance.

Let's start by ensuring we have a volume created in our desired region.

```
fly volume create atticdata -r sea -n 1
```

The volume can be called whatever we want, we just need to make sure to update the `source` in the `[mounts]` section in our `fly.toml` file.



```
app = "<pick your own fly app name>"
primary_region = "sea"

[mounts]
source = "atticdata"
```

```
destination = "/data"

[http_service]
internal_port = 8080
force_https = true
auto_stop_machines = true
auto_start_machines = true
min_machines_running = 0
processes = ["app"]

[services.concurrency]
type = "connections"
hard_limit = 1000
soft_limit = 1000
```

That's pretty much it, time to `fly deploy`! (I do this with `--ha=false` since I don't want to create more than one machine)

Generating an Attic token

We'll need to execute a command on the newly deployed fly.io machine to generate a "godmode" token for ourselves. Let's start by getting the machine ID.

```
fly machine list --json | jq --raw-output '.[].id'
```

Then we can call the `atticadm` command on the fly.io machine via `fly machine exec` to generate our token.



```
fly machine exec <your machine id> 'atticadm make-token --sub "<your preferre"
```

We need to keep this token somewhere safe and preferably encrypted. I keep

mine encrypted with `sops-nix` in my NixOS configuration monorepo and have it decrypted and mounted to `/run/secrets/attic/token` on machines that have been given access to decrypt it.

Logging into Attic

This part is pretty simple! We can call the server whatever we like in our configuration; here “fly” will do. The final two arguments are the URL generated for our fly.io app and our access token.

```
# if pkgs.attic is installed in environment.systemPackages  
attic login fly https://<your fly app name>.fly.dev $(cat /run/secrets/attic/  
  
# if pkgs.attic is not installed, we can always run it directly from the flak  
nix run github:zhaofengli/attic#default login fly https://<your fly app name>
```

Pushing to our Attic cache

Now that we have the `atticd` cache server running on fly.io and our computer authenticated with a token that allows us to push, let’s set up a cache and push our entire(!) system configuration.

Note: In my case 256MB of RAM was not enough to keep up with everything I wanted to push! I suggest scaling the RAM to at least 512MB, if not 1GB with `fly scale memory 512`. The snippet below assumes that you have bumped the RAM to 512MB which *should* be able to handle three concurrent jobs (`-j 3`) without the machine giving “out of memory” errors and restarting the `atticd` process.



```
# if pkgs.attic is installed in environment.systemPackages
attic cache create system -j 3
attic push system /run/current-system -j 3

# if pkgs.attic is not installed, we can always run it directly from the flak
nix run github:zhaofengli/attic#default cache create system -j 3
nix run github:zhaofengli/attic#default push system /run/current-system -j 3
```

Configuring NixOS to use our Attic binary cache

There are two things that we need to do before we can configure NixOS to use our new binary cache.

First, we need to get the public key of our `system` cache.

```
attic cache info system
```

Next, we need to create a `netrc` file which contains our `attic` token. The format looks like this:

```
machine <your fly app name>.fly.dev
password <your attic token>
```

Since this file once again contains sensitive information, if you want to store this in your configuration repo, I recommend encrypting it. In the example below, I have added the `netrc` file contents via `sops-nix`, which mounts the decrypted contents to `/run/secrets/attic/netrc` for machines that have been given decryption access.

```
{
```

```
nix = {
    settings = {
        substituters = [
            "https://nix-community.cachix.org?priority=41" # this is a useful pub
            "https://numtide.cachix.org?priority=42" # this is also a useful publ
            "https://<your fly app name>.fly.dev/system?priority=43"
        ];
        trusted-public-keys = [
            "nix-community.cachix.org-1:mB9FSh9qf2dCimDSUo8Zy7bkq5CX+/rkCWyvRCYg3
            "numtide.cachix.org-1:2ps1kLBUWjxIneOy1Ik6cQjb41X0iXVXeHigGmycPPE="
            "<your cache public key>"
        ];
    };
};

netrc-file = config.sops.secrets."attic/netrc".path;
};
```

```
}
```

Now, when our NixOS machines are rebuilt:

- First the official NixOS cache (with a priority of 40) will be checked
- Next, the `nix-community` public cache (with a priority of 41) will be checked
- Next, the `numtide` public cache (with a priority of 42) will be checked
- Finally, our private cache (with a priority of 43) will be checked
- If there are no cache hits at all, the package will be built from source

What we have at this point is pretty damn cool.

In the next article we'll make this even cooler still, by setting up GitHub Actions jobs to build each of our NixOS system configurations whenever we push a new commit, and push the outputs of each those builds to our private system cache!



If you have any questions or comments you can reach out to me on [Twitter](#) and [Mastodon](#).

If you're interested in what I read to come up with solutions like this one, you can subscribe to my [Software Development RSS feed](#).

If you'd like to watch me writing code while explaining what I'm doing, you can also [subscribe to my YouTube channel](#).

If you found this content valuable, or if you are a happy user of [komorebi](#) or my [NixOS starter templates](#), please consider sponsoring me on [GitHub](#) or tipping me on [Ko-fi](#).

© 2024 Powered by [Hugo](#) :: [Theme](#) made by [panr](#)



Handling Secrets in NixOS: An Overview

2023-11-14 :: LGUG2Z

#nixos #devops #git-crypt #agenix #sops-nix

There are a number of different approaches available for NixOS users to handle secrets. The most popular tend to be [git-crypt](#), [agenix](#) and [sops-nix](#). But which one should you use?

To hopefully help you in answering this question for yourself, here is an overview of a few common use cases and what I think is most appropriate for each.

Managing Your Own Physical Machines

Maybe you have a desktop, a Macbook and a Raspberry Pi which you are managing from a single NixOS flake repo. Maybe you even have a NixOS dedicated server somewhere running in a datacenter which functions as your media server running Plex or Jellyfin.

If you are primarily using NixOS on your own physical machines, used exclusively by you, *and* you want to be able to publish your flake repo publicly, I think you can get pretty far with [git-crypt](#).

I have been a happy user of [git-crypt](#) for a long time, even before I started using NixOS, and naturally it was my first instinct to use when setting up my own configuration flake repo.

Using [git-crypt](#) will ensure that your secret files are encrypted when you push your repo to a remote like GitHub, however, using this approach means that your secrets will end up in [/nix/store](#) unencrypted, which is readable to all users on a machine. If you're exclusively managing your own physical machines, this isn't really an issue for you to worry about.



Here is how I'd suggest getting started with `git-crypt` for a personal flake repo:

Initialize the repo with `git-crypt init`, make a directory dedicated to secrets, and use a `.gitattributes` file to ensure that every file that you create in that `secrets` subdirectory will always be encrypted.

```
secrets/** filter=git-crypt diff=git-crypt
```

Since the only user of this personal flake repository with access to decrypt secrets will be you, it's more convenient to export a symmetric secret key and base64 encode it so that you can throw it in 1Password or something similar.

```
git-crypt export-key ./secret-key  
cat ./secret-key | base64 --encode > ./secret-key-base64
```

You can store the value of `./secret-key-base64` in your password manager, and if you ever need to decrypt the files in this repo on another machine, you can just decode the key before using it.

```
pbpaste | base64 --decode > ./secret-key  
git-crypt unlock ./secret-key
```



Up until this point, this is all pretty basic stuff. Next let's take a look at how we can ergonomically handle secrets within a flake.

I like to have a single `secrets.json` file that is structured with top-level keys describing what the secret(s) relate to:

```
{
  "github": {
    "oauth_token": "ghp_..."
  },
  "gitlab": {
    "oauth_token": "glpat-..."
  },
  "tailscale": {
    "authkey": "tskey-auth-..."
  }
}
```

Then, at the top-most level of my flake I declare a variable called “secrets” which reads the values from this file and deserializes them into a Nix object.

This `secrets` variable can then be passed as a member of `specialArgs` to make it available to the various NixOS system configurations.

```
{
  description = "My NixOS configurations";

  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-23.05";
  inputs.nixpkgs-unstable.url = "github:nixos/nixpkgs/nixos-unstable";

  outputs = inputs:
    with inputs: let
      secrets = builtins.fromJSON (builtins.readFile "${self}/secrets/secrets"
    in {
      # make sure to "inherit secrets;" in the nixpkgs.lib.nixosSystem.spec
    }
}
```



In particular, I find this very useful and easy to use inside of `home-manager` to do things like set URL overrides in my `gitconfig` so HTTPS clones

of private repositories automatically use an oauth token.

```
{  
    secrets,  
    ...  
}: {  
    programs.git = {  
        enable = true;  
        extraConfig = {  
            url = {  
                "https://oauth2:${secrets.github.oauth_token}@github.com" = {  
                    insteadOf = "https://github.com";  
                };  
                "https://oauth2:${secrets.gitlab.oauth_token}@gitlab.com" = {  
                    insteadOf = "https://gitlab.com";  
                };  
            };  
        };  
    };  
}
```

You can also reference other encrypted files in the `secrets` dir and use `home-manager` to move them into place for you. For example, with an `.npmrc` file:

```
{  
    secrets,  
    ...  
}: {  
    home.username.LGUG2Z.file.".npmrc".source = ./secrets/.npmrc;  
}
```



Providing Runtime Secrets to Remote Machines and VMs

So `git-crypt` works well for personal secrets like GitHub tokens, secrets that live in configuration files under your `$HOME` folder, but what about when you need to provide a runtime secret to a service running on a remote server? Usually these services run on their own user accounts and groups, and `home-manager` is not a good fit for provisioning secrets to these kinds of non-user system accounts.

In this case, there are two options available: `agenix` and `sops-nix`.

Agenix

`agenix` is, in my opinion, the simpler of the two to set up, and that's probably why it has a bigger mindshare in the NixOS community right now.

You get started by creating a dedicated subdirectory and creating a `secrets.nix` file inside of it.

```
let
  personal_key = "ssh-rsa AAAA....";
  remote_server_key = "ssh-rsa AAAA....";
  keys = [personal_key remote_server_key];
in {
  "guest_accounts.json.age".publicKeys = keys;
}
```

In this file you declare variables for the public keys of the machines that will need access to the secrets (you can grab these by running `ssh-keyscan user@remote-ip`), and then create references to files that you will later create, linking each of them to one or more public keys.

At this point, we have a file, which says “we’re gonna make these encrypted files” and “the private keys for the linked public keys will be able to decrypt them”.

The next step is to create the encrypted file by running `agenix -e service_account.json.age` in the same subdirectory. This opens up your



\$EDITOR for you to type/paste your secret into. When you save and close the text editor, the file will be encrypted, and this file can be added to git .

If you ever need to add more public keys for other machines that you want to be able to decrypt these secrets, you can just run agenix -r inside the subdirectory and commit the changes.

Next, in order to get these encrypted secrets into your flake, follow the [Install via Flakes](#) steps on the project README (expand the content section) to ensure that you have the agenix.nixosModules.default module loaded, and then start mapping references to your encrypted files.

```
{  
  inputs.agenix.url = "github:ryantm/agenix";  
  # ... other inputs  
  
  outputs = { self, nixpkgs, agenix }: {  
    nixosConfigurations.yourhostname = nixpkgs.lib.nixosSystem {  
      system = "x86_64-linux";  
      modules = [  
        # ... your config  
        agenix.nixosModules.default  
        {  
          age.secrets."guest_accounts.json".file = ./secrets/guest_accounts.j  
        }  
      ];  
    };  
  };  
}
```

Unlike git-crypt , this approach means that your secrets will end up in /nix/store but they will be encrypted there, so even if another user or process can find the file, they won't be able to make any sense of it.



What about making use of these secrets? Well, there is where you'll need to change your approach a little if you are coming from git-crypt ; you can't really refer to the contents of these secrets in the /nix/store in your

NixOS configuration (ie. to do string interpolation) because they are encrypted, so you have to instruct your services to read from the decrypted files in `/run/agenix` when they start.

Typically this is done by using `systemd`'s `LoadCredential` option, which will make a copy of the decrypted secret in `/run/agenix` available to a service in `/run/credentials/your-service.service/filename`, and then instructing the service via environment variables to use the file in that location.

Below is an example from my previous agenix article where we loaded a `guest_accounts.json` secret file to be read by `nitter.service`.

```
{  
    systemd.services.nitter.serviceConfig.LoadCredential = [  
        "guest_account.json:${config.age.secrets."guest_accounts.json".path}"  
    ];  
  
    systemd.services.nitter.serviceConfig.Environment = [  
        "NITTER_CONF_FILE=/var/lib/private/nitter/nitter.conf"  
        "NITTER_ACCOUNTS_FILE=%d/guest_account.json"  
    ];  
}
```

As you can see, this process is geared very much towards files, so even if you wanted to encrypt a GitHub OAuth token, you'd have to create a unique file with `agenix -e github_token` and then find a way to `cat` the value of that into your `systemd` script or service.

Here is an example from a `systemd` timer that I use to periodically update this website.



```
{  
    systemd.services."update-lgug2z-com" = {  
        startAt = "hourly";  
        serviceConfig = {  
            Type = "oneshot";
```

```

ExecStart = ''
    ${pkgs.bash}/bin/bash -c "${pkgs.httpie}/bin/http POST \
        https://api.cloudflare.com/client/v4/accounts/'$(cat ${config.ag-
-A bearer -a '$(cat ${config.age.secrets."cloudflare_pages_api_t-
';
    };
};

}

```

Note that you have to wrap this in a `bash -c` call to enable the use of `cat`.

The final generated `systemd` timer definition which gets symlinked from `/nix/store` looks like this; no secrets exposed!

```
# /etc/systemd/system/update-lgug2z-com.service
```

```
[Unit]
```

```
[Service]
```

```
Environment="LOCALE_ARCHIVE=/nix/store/fzm1flvb7zmz3ij4sscn521shz2f76jh-glibc
Environment="PATH=/nix/store/w8vm09hri2zz7yacryzzxvsapik4ps4-coreutils-9.1/b
Environment="TZDIR=/nix/store/951696yxqlphz378fx126wijnrih08mz3-tzdata-2023c/s
```

```
ExecStart=/nix/store/0rwyq0j954a7143p0wzd4rhycny8i967-bash-5.2-p15/bin/bash -
    https://api.cloudflare.com/client/v4/accounts/$(cat /run/secrets/cloudfla
    -A bearer -a $(cat /run/secrets/cloudflare/pages_api_token))"
```

```
Type=oneshot
```

```
# /etc/systemd/system/update-lgug2z-com.timer
```

```
[Unit]
```

```
[Timer]
```

```
OnCalendar=hourly
```



Sops-Nix

So `agenix` lets us get provision secret files for services which remain encrypted in the `/nix/store`, but are available decrypted for the specific services to which we allow access. However the tradeoff here is that we lose some of the convenience of being able to have smaller secrets such as tokens all collected together in a single file in our repo.

`sops-nix` is a little trickier to get set up, but offers a way for you to retain your single encrypted file of secrets as a source of truth, both for individual secret values and larger secrets files, all while providing clean `git diff`s to make it easier to see what has changed.

Getting started requires you to create a `.sops.yaml` file at the root of your flake repo.

keys:

- `&remote` age1 ...
- `&personal` age1 ...

creation_rules:

- `path_regex`: `secrets/[^\n]+\\.(yaml|json|env|ini|sops)$`

key_groups:

- `age`:
 - `*remote`
 - `*personal`

Let's walk through this. At the top we define some keys, just like we did in `secrets.nix` for `agenix`. However, the keys here are a little different. You will need to convert your SSH public keys into `age` public keys, and it's simpler to do this using `ed25519` keys instead of `rsa` keys.



You can generate these `age` public keys for remote servers by running `ssh-keygen user@remote-ip | ssh-to-age` (you might need to `nix-shell -p ssh-to-age` first if you don't have the package on your system), and for your local machine by running `ssh-to-age -i ~/.ssh/id_ed25519.pub`.

Next is the `creation_rules` section, where instead of providing explicit

names for each encrypted file like we did in `secrets.nix` with `agenix`, we just define a regex of encryptable files, and the public keys corresponding to the private keys that we want to allow to decrypt them.

Before we start creating rules, it's important to make sure that we have `ssh-to-age` in our system packages and create an `.envrc` for our flake repo, otherwise we won't be able to decrypt any of our secrets locally!

```
# make this point to wherever your own es25519 ssh key is
export SOPS_AGE_KEY=$(ssh-to-age -i ~/.ssh/id_ed25519 -private-key)
```

With the `.sops.yaml` and `.envrc` files created, we can create an encrypted file in our dedicated subdirectory by running `sops secrets/secrets.yaml` and fill it with some values.

```
github:
  oauth_token: ghp_...
gitlab:
  oauth_token: glpat-...
tailscale:
  authkey: tskey-auth-...
guest_accounts.json: |
  [
    {
      "id": "some-id",
      "token": "some-token",
      "grants": ["some" "grants"],
    }
  ]
npmrc: |
  //registry.npmjs.org/_authToken=npm_...
  //some.other.registry.org/_authToken=npm_...
```



Notice that we can store both file contents and individual secrets like

tokens here; this is kind of the best of both worlds from the `git-crypt` and `agenix` approaches, especially if like me you find it convenient to only have to encrypt a single file.

Again, in order to get these encrypted secrets into your flake, follow the [Install sops-nix \(Flakes\)](#) steps on the project README (expand the content section) to ensure that you have the `sops.nixosModules.default` module loaded, and then start mapping references to your encrypted secrets.



```
{
  inputs.sops-nix.url = "github:Mic92/sops-nix";

  outputs = { self, nixpkgs, sops-nix }: {
    # change `yourhostname` to your actual hostname
    nixosConfigurations.yourhostname = nixpkgs.lib.nixosSystem {
      # customize to your system
      system = "x86_64-linux";
      modules = [
        # ... your config
        sops-nix.nixosModules.sops
        {
          sops = {
            defaultSopsFile = ./secrets/secrets.yaml;
            age.sshKeyPaths = ["/etc/ssh/ssh_host_ed25519_key"];
            secrets = {
              "github/oauth_token" = {};
              "gitlab/oauth_token" = {};
              "tailscale/authkey" = {};
              "guest_accounts.json" = {};
              "npmrc" = {
                owner = "youruser";
                path = "/home/youruser/.npmrc"
              };
            };
          };
        ];
      };
    };
  };
}
```



There is a little more code here, but it's not as bad as it looks. Let's walk through it. Since we can have multiple files encrypted by `sops` , `sops-nix` helpfully allows us to define a `defaultSopsFile` , which is useful if you're like me and you like to just keep a single encrypted file.

Then we have to set the paths to the private keys on the remote machine that will be used to decrypt the encrypted `defaultSopsFile` that will end up in

the `/nix/store` .

Finally, we can declare our secrets. Each secret is output into `/run/secrets` into a separate file, with a path that is taken from the object structure. For example, `github.oauth_token`'s value will be output to `/run/secrets/github/oauth_token` .

We can also use this to our advantage by storing stringified verisons of entire files in the way that we have for `guest_accounts.json` and `npmrc`, to ensure that these will be output to `/run/secrets/guest_accounts.json` and `/run/secrets/.npmrc` respectively.

Notice that in the case of `npmrc`, we can actually set the owner for this file and a location for it to be symlinked to, which is a useful way to replicate the placing of files in the `$HOME` directory used in the `git-crypt` approach (you can also do this with `agenix`).

To make these secrets in their decrypted forms available to `systemd` services, just follow the same steps outlined for `agenix`, but reference the path from `sops` instead of `age` .

Summary

Hopefully this overview of when you might want to use these different approaches will help you to decide what is right for your use case. As a little tl;dr:

- Just managing your own machines? `git-crypt` is fine
- Managing your own machines and remote servers? You'll need `agenix` or `sops-nix` to provide secrets to services
- Wanna keep as much as possible in a single encrypted file? `sops-nix` is probably the way to go

However, keep in mind that you don't have to pick only one of these! For example, you can use a `git-crypt` ed `secrets.json` to populate values on machines that you use for secret values in your configuration files and also use `agenix` or `sops-nix` to provide secrets to services on remote servers and VMs!



If you have any questions you can reach out to me on [Twitter](#) and [Mastodon](#).

If you're interested in what I read to come up with solutions like this one, you can subscribe to my [Software Development RSS feed](#).

If you'd like to watch me writing code while explaining what I'm doing, you can also [subscribe to my YouTube channel](#).

On 11/14/2023 I was impacted by large scale layoffs at my previous employer. I am currently looking for work. I am an experienced SRE with a strong passion for developer enablement. Please reach out if you are hiring for a role that you think I'd be a good fit for.

If you found this content valuable, please consider sponsoring me on [GitHub](#) or tipping me on [Ko-fi](#) to help me through this uncertain period.

© 2024 Powered by [Hugo](#) :: [Theme](#) made by [panr](#)



[Xe](#)[Blog](#)[Contact](#)[Resume](#)[Talks](#)[VODs](#)[Signalboost](#)

How to fix terraform and nix flakes

Published on 10/27/2023, 346 words, 2 minutes to read



A pink haired avali wearing a sweater and sweatpants drinking coffee indoors. - Furryrock

Recently Terraform [changed licenses](#) to the Business Source License. This is a non-free license in the eyes of Nix, so now whenever you update your project flakes, you get greeted by this lovely error:

```
error: Package 'terraform-1.6.2' in /nix/store/z1nvpjx9vd4151vx2krxzmx2p1a36pf9-source/pkgs/ap
```

- To temporarily allow unfree packages, you can use an environment variable for a single invocation of the nix tools.

```
$ export NIXPKGS_ALLOW_UNFREE=1
```

Note: For `nix shell`, `nix build`, `nix develop` or any other Nix 2.4+ (Flake) command, `--impure` must be passed in order to read this

Monitor K8s Applications Effortlessly & Gain 100% Production Visibility | Start Free

Ads by
EthicalAds

[Close Ad](#)

b) For `nixos-rebuild` you can set

```
{ nixpkgs.config.allowUnfree = true; }  
in configuration.nix to override this.
```

Alternatively you can configure a predicate to allow specific packages:

```
{  
  nixpkgs.config.allowUnfreePredicate = pkg: builtins.elem (lib.getName pkg) [  
    "terraform"  
  ];  
}
```

c) For `nix-env`, `nix-build`, `nix-shell` or any other Nix command you can add

```
{ allowUnfree = true; }  
to ~/.config/nixpkgs/config.nix.
```

The extra fun part is that when you're using a flake with a per-project version of nixpkgs, none of those workarounds work. Here's what you have to do instead:

In your flake you'll usually have an import of nixpkgs like this:

```
let  
  pkgs = import nixpkgs { inherit system; };  
in  
  crimes_etc
```

Or like this:

```
let  
  pkgs = nixpkgs.legacyPackages.${system};  
in  
  different_crimes_etc
```

You'll want to change that to this:

```
let
pkgs = import nixpkgs { inherit system; config.allowUnfree = true; };
in
working_crimes_etc
```

This allows you to bypass the license check for all packages in nixpkgs so that things Just Work™. If you want to only do this for terraform, you can make a separate instance of nixpkgs to pull out only terraform, but I think that overall it's probably easier to just eliminate the problem entirely.

I hope this helps you out!



<Cadey> Don't you love the intersection of computers and capitalism? It's the best.



<Aoi> Tell me about it.

Share



Facts and circumstances may have changed since publication. Please contact me before jumping to conclusions if something seems wrong or unclear.

Tags: nix, terraform, enshittification

Copyright 2012-2024 Xe Iaso (Christine Dodrill). Any and all opinions listed here are my own and not representative of any of my employers, past, future, and/or present.

/bin/xesite) with site version [d58e5083](#) , source code available [here](#).

Monitor K8s Applications Effortlessly & Gain 100% Production Visibility | Start Free

Ads by
EthicalAds

[Close Ad](#)

[Xe](#)[Blog](#)[Contact](#)[Resume](#)[Talks](#)[VODs](#)[Signalboost](#)

My NixOS Desktop Flow

Published on 04/25/2020, 2858 words, 11 minutes to read

Before I built my current desktop, I had been using a [2013 Mac Pro](#) for at least 7 years. This machine has seen me through living in a few cities (Bellevue, Mountain View and Montreal), but it was starting to show its age. Its 12 core Xeon is really no slouch (scoring about 5 minutes in my "compile the linux kernel" test), but with Intel security patches it was starting to get slower and slower as time went on.

So in March (just before the situation started) I ordered the parts for my new tower and built my current desktop machine. From the start, I wanted it to run Linux and have 64 GB of ram, mostly so I could write and test programs without having to worry about ram exhaustion.

When the parts were almost in, I had decided to really start digging into [NixOS](#). Friends on IRC and Discord had been trying to get me to use it for years, and I was really impressed with a simple setup that I had in a virtual machine. So I decided to jump head-first down that rabbit hole, and I'm honestly really glad I did.

NixOS is built on a more functional approach to package management called [Nix](#). Parts of the configuration can be easily broken off into modules that can be reused across machines in a deployment. If [Ansible](#) or other tools like it let you customize an existing Linux distribution to meet your needs, NixOS allows you to craft your own Linux distribution around your needs.

Unfortunately, the Nix and NixOS documentation is a bit more dense than most other Linux programs/distributions are, and it's a bit easy to get lost in it. I'm going to attempt to

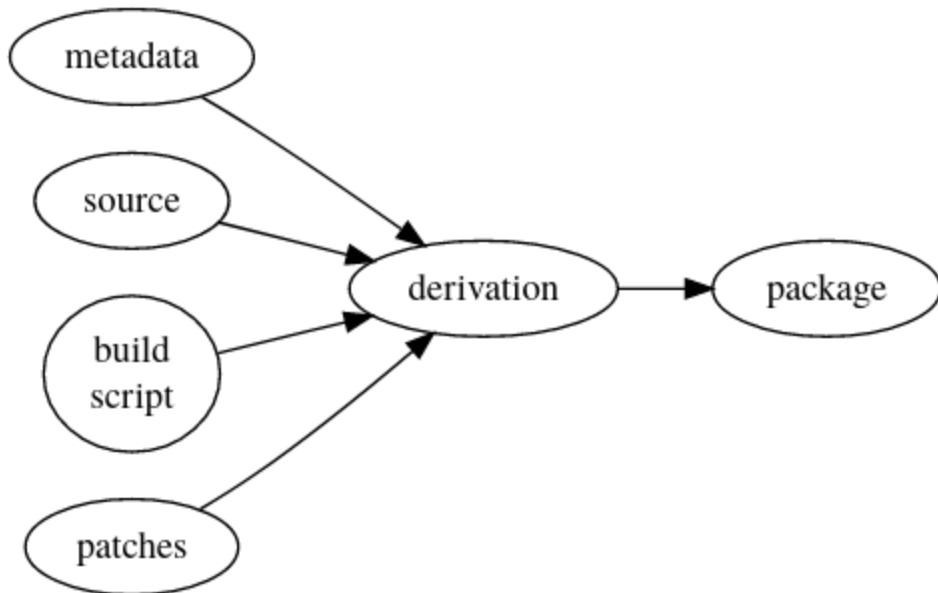
[Codeium: Free AI For Devs](#) Code Completions, Chat, and Search [Get Codeium today. Yes, it's really free.](#)

Ads by
EthicalAds

[Close Ad](#)

What is a Package?

Earlier, I mentioned that Nix is a *functional* package manager. This means that Nix views packages as a combination of inputs to get an output:



This is how most package managers work (even things like Windows installer files), but Nix goes a step further by disallowing package builds to access the internet. This allows Nix packages to be a lot more reproducible; meaning if you have the same inputs (source code, build script and patches) you should *always* get the same output byte-for-byte every time you build the same package at the same version.

A Simple Package

Let's consider a simple example, my [gruvbox-inspired CSS file](#)'s `'default.nix'` file:

```
{ pkgs ? import <nixpkgs> { } }: 
```

```
pkgs.stdenv.mkDerivation {  
  pname = "gruvbox-css";  
  version = "latest";  
  src = / .
```

[Codeium: Free AI For Devs](#) Code Completions, Chat, and Search [Get Codeium today. Yes, it's really free.](#)

Ads by
EthicalAds

[Close Ad](#)

```
mkdir -p $out
cp -rf $src/gruvbox.css $out/gruvbox.css
";
}
```

This creates a package named `gruvbox-css` with the version `latest`. Let's break this down its `default.nix` line by line:

```
{ pkgs ? import <nixpkgs> { } }:
```

This creates a function that either takes in the `pkgs` object or tells Nix to import the standard package library [nixpkgs](#) as `pkgs`. nixpkgs includes a lot of utilities like a standard packaging environment, special builders for things like snaps and Docker images as well as one of the largest package sets out there.

```
pkgs.stdenv.mkDerivation {
  # ...
}
```

This runs the [`stdenv.mkDerivation`](#) function with some arguments in an object. The "standard environment" comes with tools like GCC, bash, coreutils, find, sed, grep, awk, tar, make, patch and all of the major compression tools. This means that our package builds can build C/C++ programs, copy files to the output, and extract downloaded source files by default. You can add other inputs to this environment if you need to, but for now it works as-is.

Let's specify the name and version of this package:

```
pname = "gruvbox-css";
version = "latest";
```

`pname` stands for "package name". It is combined with the version to create the resulting [Codeium: Free AI For Devs](#) Code Completions, Chat, and Search [Get Codeium today. Yes, it's really free.](#)

Ads by
EthicalAds

[Close Ad](#)

Let's tell Nix how to build this package:

```
src = ./;
phases = "installPhase";
installPhase = ''
  mkdir -p $out
  cp -rf $src/gruvbox.css $out/gruvbox.css
'';
```

The `src` attribute tells Nix where the source code of the package is stored. Sometimes this can be a URL to a compressed archive on the internet, sometimes it can be a git repo, but for now it's the current working directory `./`.

This is a CSS file, it doesn't make sense to have to build these, so we skip the build phase and tell Nix to directly install the package to its output folder:

```
mkdir -p $out
cp -rf $src/gruvbox.css $out/gruvbox.css
```

This two-liner shell script creates the output directory (usually exposed as `'\$out'`) and then copies `gruvbox.css` into it. When we run this through Nix with `nix-build`, we get output that looks something like this:

```
$ nix-build ./default.nix
these derivations will be built:
  /nix/store/c99n4ixraigf4jb0jfjxbkzicd79scpj-gruvbox-css.drv
building '/nix/store/c99n4ixraigf4jb0jfjxbkzicd79scpj-gruvbox-css.drv' ...
installing
/nix/store/ng5qnhwyrk9zaidjv00arhx787r0412s-gruvbox-css
```

And `/nix/store/ng5qnhwyrk9zaidjv00arhx787r0412s-gruvbox-css` is the output package. Looking at its contents with `ls`, we see this:

gruvbox.css

A More Complicated Package

For a more complicated package, let's look at the [build directions of the website you are reading right now](#):

```
{ pkgs ? import (import ./nix/sources.nix).nixpkgs }:
with pkgs;

assert lib.versionAtLeast go.version "1.13";

buildGoPackage rec {
  pname = "christinewebsite";
  version = "latest";

  goPackagePath = "xeiaso.net";
  src = ./.;
  goDeps = ./nixdeps.nix;
  allowGoReference = false;

  preBuild = ''
    export CGO_ENABLED=0
    buildFlagsArray+=(-pkgdir "$TMPDIR")
  '';

  postInstall = ''
    cp -rf $src/blog $bin/blog
    cp -rf $src/css $bin/css
    cp -rf $src/gallery $bin/gallery
    cp -rf $src/signalboost.dhall $bin/signalboost.dhall
    cp -rf $src/static $bin/static
    cp -rf $src/talks $bin/talks
    cp -rf $src/templates $bin/templates
  '';;
}
```

Codeium: Free AI For Devs Code Completions, Chat, and Search [Get Codeium today. Yes, it's really free.](#)

Ads by
EthicalAds

[Close Ad](#)

Breaking it down, we see some similarities to the gruvbox-css package from above, but there's a few more interesting lines I want to point out:

```
{ pkgs ? import (import ./nix/sources.nix).nixpkgs }:
```

My website uses a pinned or fixed version of nixpkgs. This allows my website's deployment to be stable even if nixpkgs changes something that could cause it to break.

```
with pkgs;
```

With expressions are one of the more interesting parts of Nix. Essentially, they let you say "everything in this object should be put into scope". So if you have an expression that does this:

```
let
  foo = {
    ponies = "awesome";
  };
in with foo; "ponies are ${ponies}!"
```

You get the result `ponies are awesome!`. I use `with pkgs` here to use things directly from nixpkgs without having to say `pkgs.` in front of a lot of things.

```
assert lib.versionAtLeast go.version "1.13";
```

This line will make the build fail if Nix is using any Go version less than 1.13. I'm pretty sure my website's code could function on older versions of Go, but the runtime improvements are important to it, so let's fail loudly just in case.

```
buildGoPackage {
  # ...
```

Codeium: Free AI For Devs Code Completions, Chat, and Search [Get Codeium today. Yes, it's really free.](#)

Ads by
EthicalAds

[Close Ad](#)

``buildGoPackage`` builds a Go package into a Nix package. It takes in the [Go package path](#), list of dependencies and if the resulting package is allowed to depend on the Go compiler or not.

It will then compile the Go program (and all of its dependencies) into a binary and put that in the resulting package. This website is more than just the source code, it's also got assets like CSS files and the image earlier in the post. Those files are copied in the `postInstall` phase:

```
postInstall = ''
  cp -rf $src/blog $bin/blog
  cp -rf $src/css $bin/css
  cp -rf $src/gallery $bin/gallery
  cp -rf $src/signalboost.dhall $bin/signalboost.dhall
  cp -rf $src/static $bin/static
  cp -rf $src/talks $bin/talks
  cp -rf $src/templates $bin/templates
'';
```

This results in all of the files that my website needs to run existing in the right places.

Other Packages

For more kinds of packages that you can build, see the [Languages and Frameworks](#) chapter of the nixpkgs documentation.

If your favorite language isn't shown there, you can make your own build script and do it more manually. See [here](#) for more information on how to do that.

`nix-env` And Friends

Building your own packages is nice and all, but what about using packages defined in nixpkgs? Nix includes a few tools that help you find, install, upgrade and remove packages

[Codeium: Free AI For Devs](#) Code Completions, Chat, and Search [Get Codeium today. Yes, it's really free.](#)

Ads by
EthicalAds

[Close Ad](#)

`nix search`

When looking for a package to install, use `'\$ nix search name` to see if it's already packaged. For example, let's look for **graphviz**, a popular diagramming software:

```
$ nix search graphviz
```

```
* nixos.graphviz (graphviz)
```

Graph visualization tools

```
* nixos.graphviz-nox (graphviz)
```

Graph visualization tools

```
* nixos.graphviz_2_32 (graphviz)
```

Graph visualization tools

There are several results here! These are different because sometimes you may want some features of graphviz, but not all of them. For example, a server installation of graphviz wouldn't need X windows support.

The first line of the output is the attribute. This is the attribute that the package is imported to inside nixpkgs. This allows multiple packages in different contexts to exist in nixpkgs at the same time, for example with python 2 and python 3 versions of a library.

The second line is a description of the package from its metadata section.

The `nix` tool allows you to do a lot more than just this, but for now this is the most important thing.

`nix-env -i`

`nix-env` is a rather big tool that does a lot of things (similar to pacman in Arch Linux), so I'm going to break things down into separate sections.

Let's pick an instance **graphviz** from before and install it using `nix-env`:

Codeium: Free AI For Devs Code Completions, Chat, and Search [Get Codeium today. Yes, it's really free.](#)

Ads by
EthicalAds

[Close Ad](#)

```
$ nix-env -iA nixos.graphviz
installing 'graphviz-2.42.2'
these paths will be fetched (5.00 MiB download, 13.74 MiB unpacked):
  /nix/store/980jk7qbcfrlnx8jsmdx92q96wsai8mx-gts-0.7.6
  /nix/store/fij1p8f0yjpv35n342ii9p wfahj8rlbb-graphviz-2.42.2
  /nix/store/jy35xihlnb3az0vdksyg9rd2f38q2c01-libdevil-1.7.8
  /nix/store/s895dnwlprwpfp75pzq70qzfdn8mwfzc-lcms-1.19
copying path '/nix/store/980jk7qbcfrlnx8jsmdx92q96wsai8mx-gts-0.7.6' from 'https://cache.nixos
copying path '/nix/store/s895dnwlprwpfp75pzq70qzfdn8mwfzc-lcms-1.19' from 'https://cache.nixos
copying path '/nix/store/jy35xihlnb3az0vdksyg9rd2f38q2c01-libdevil-1.7.8' from 'https://cache.
copying path '/nix/store/fij1p8f0yjpv35n342ii9p wfahj8rlbb-graphviz-2.42.2' from 'https://cache
building '/nix/store/r4fqdwpicqjpa97biis1jlxzb4ywi92b-user-environment.drv'...
created 664 symlinks in user environment
```

And now let's see where the `dot` tool from graphviz is installed to:

```
$ which dot
/home/cadey/.nix-profile/bin/dot

$ readlink /home/cadey/.nix-profile/bin/dot
/nix/store/fij1p8f0yjpv35n342ii9p wfahj8rlbb-graphviz-2.42.2/bin/dot
```

This lets you install tools into the system-level Nix store without affecting other user's environments, even if they depend on a different version of graphviz.

`nix-env -e`

`nix-env -e` lets you uninstall packages installed with `nix-env -i`. Let's uninstall graphviz:

```
$ nix-env -e graphviz
```

Now the `dot` tool will be gone from your shell:

[Codeium: Free AI For Devs](#) Code Completions, Chat, and Search [Get Codeium today. Yes, it's really free.](#)

Ads by
EthicalAds

[Close Ad](#)

which: no dot in (/run/wrappers/bin:/home/cadey/.nix-profile/bin:/etc/profiles/per-user/cadey/

And it's like graphviz was never installed.

Notice that these package management commands are done at the *user* level because they are only affecting the currently logged-in user. This allows users to install their own editors or other tools without having to get admins involved.

Adding up to NixOS

NixOS builds on top of Nix and its command line tools to make an entire Linux distribution that can be perfectly crafted to your needs. NixOS machines are configured using a configuration.nix file that contains the following kinds of settings:

- packages installed to the system
- user accounts on the system
- allowed SSH public keys for users on the system
- services activated on the system
- configuration for services on the system
- magic unix flags like the number of allowed file descriptors per process
- what drives to mount where
- network configuration
- ACME certificates

and so much more

At a high level, machines are configured by setting options like this:

```
# basic-lxc-image.nix
{ config, pkgs, ... }:
```

Codeium: Free AI For Devs Code Completions, Chat, and Search [Get Codeium today. Yes, it's really free.](#)

Ads by
EthicalAds

[Close Ad](#)

```
networking.hostName = "example-for-blog";
environment.systemPackages = with pkgs; [ wget vim ];
}
```

This would specify a simple NixOS machine with the hostname `example-for-blog` and with wget and vim installed. This is nowhere near enough to boot an entire system, but is good enough for describing the base layout of a basic [LXC](#) image.

For a more complete example of NixOS configurations, see [here](#) or repositories on [this handy NixOS wiki page](#).

The main configuration.nix file (usually at `/etc/nixos/configuration.nix`) can also import other NixOS modules using the `imports` attribute:

```
# better-vm.nix
{ config, pkgs, ... }:
{
  imports = [
    ./basic-lxc-image.nix
  ];
}

networking.hostName = "better-vm";
services.nginx.enable = true;
}
```

And the `better-vm.nix` file would describe a machine with the hostname `better-vm` that has wget and vim installed, but is also running nginx with its default configuration.

Internally, every one of these options will be fed into auto-generated Nix packages that will describe the system configuration bit by bit.

`nixos-rebuild`

roll back to them if you need to. `nixos-rebuild` is the tool that helps you commit configuration changes to the system as well as roll them back.

If you want to upgrade your entire system:

```
$ sudo nixos-rebuild switch --upgrade
```

This tells nixos-rebuild to upgrade the package channels, use those to create a new base system description, switch the running system to it and start/restart/stop any services that were added/upgraded/removed during the upgrade. Every time you rebuild the configuration, you create a new "generation" of configuration that you can roll back to just as easily:

```
$ sudo nixos-rebuild switch --rollback
```

Garbage Collection

As upgrades happen and old generations pile up, this may end up taking up a lot of unwanted disk (and boot menu) space. To free up this space, you can use `nix-collect-garbage`:

```
$ sudo nix-collect-garbage  
< cleans up packages not referenced by anything >
```

```
$ sudo nix-collect-garbage -d  
< deletes old generations and then cleans up packages not referenced by anything >
```

The latter is a fairly powerful command and can wipe out older system states. Only run this if you are sure you don't want to go back to an older setup.

How I Use It

Codeium: Free AI For Devs Code Completions, Chat, and Search [Get Codeium today. Yes, it's really free.](#)

Ads by
EthicalAds

[Close Ad](#)

and just about every program I use on a regular basis defined in their own NixOS modules so I can pick and choose things for new machines.

When I want to change part of my config, I edit the files responsible for that part of the config and then rebuild the system to test it. If things work properly, I commit those changes and then continue using the system like normal.

This is a little bit more work in the short term, but as a result I get a setup that is easier to recreate on more machines in the future. It took me a half hour or so to get the configuration for zathura right, but now I have a zathura module that lets me get exactly the setup I want every time.

TL;DR

Nix and NixOS ruined me. It's hard to go back.



Facts and circumstances may have changed since publication. Please contact me before jumping to conclusions if something seems wrong or unclear.

Tags:

Copyright 2012-2024 Xe Iaso (Christine Dodrill). Any and all opinions listed here are my own and not representative of any of my employers, past, future, and/or present.

Like what you see? Donate on [Patreon](#) like [these awesome people!](#)

Served by xesite v4 (/nix/store/9rxy3y9y9ffwx6xdvb56pgciar4b5138-xesite_v4-20240128/bin/xesite) with site version [d58e5083](#), source code available [here](#).

[Codeium: Free AI For Devs](#) Code Completions, Chat, and Search [Get Codeium](#)
today. Yes, it's really free.

Ads by
EthicalAds

[Close Ad](#)



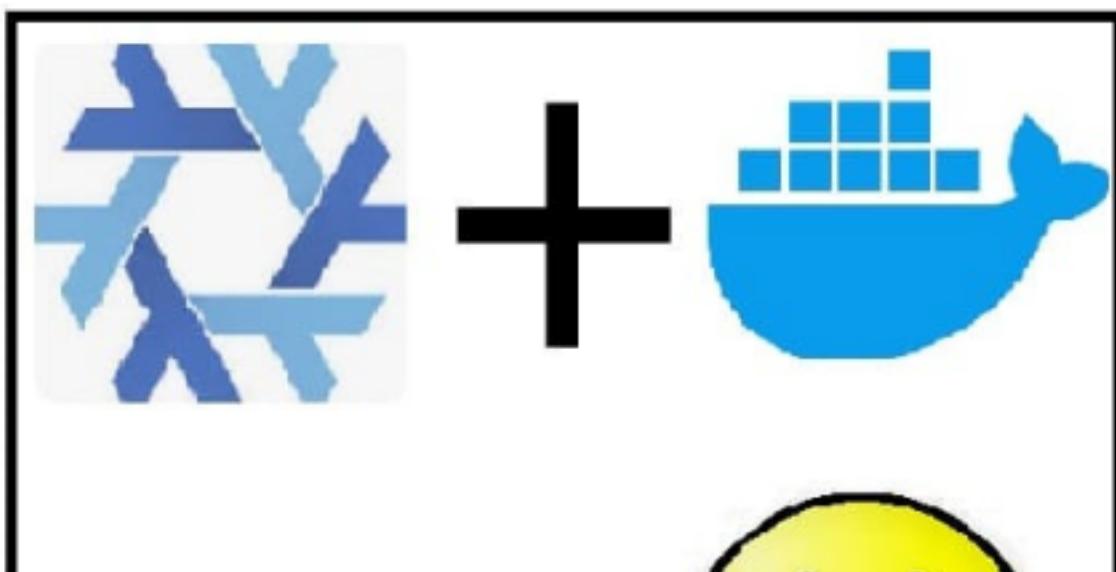
GET IN TOUCH

Nix, Docker - or both?

"It worked on my machine!" Most of us who've worked in IT for any length of time have heard this complaint. You develop your software, you test it exhaustively. It looks great. Then you deploy it into production and it crashes. And the blame game starts.



Dec 24, 2023 - 8 min read



"It worked on my machine!"

Most of us who've worked in IT for any length of time have heard this complaint. You develop your software, you test it exhaustively. It looks great. Then you deploy it into production and it crashes. And the blame game starts.

Why does this happen? And how can you avoid it?

The Problem

The main cause of the "it worked on my machine" syndrome is that almost no software stands alone - it depends not only on the host's operating system, but on other software that may be installed on it. These dependency-related problems can be a nightmare to track down and fix.

It's probably worth mentioning another common syndrome: "It worked before I made changes - and it wasn't my changes that broke it!" This usually happens because something in the development environment has changed, and just rebuilding the software caused it to break. Your system may have updated some of the software your package depends on without you even being aware of it - and your package isn't compatible with the new versions.

Suggested Solutions

How can you avoid this? There are many proposed solutions, but you'll often be told: "Use Docker". Others say, "Use Nix." So which should you use?

Docker or Nix?

That's a bit like asking whether you should have a washing machine or a vacuum cleaner. Although they both save time in the house, they don't do the same job. If you can afford them, you have both.



Likewise, Nix and Docker are designed for different purposes:

- Docker is a deployment tool, whereas Nix is a package management tool
- Docker offers a reproducible run-time environment, whereas Nix offers a reproducible build.

You'd use the right tool for the right job. And since they are both free and work well together, there's no reason why you shouldn't use both.

In this article, I'll look at both tools: what they do best, how they make your workflow easier and more reliable, and how they complement each other to get the best of both technologies.

What is Docker?

Let's look at Docker first.

Docker allows you to package your software as images containing all the dependencies it needs to run successfully. At run time, your image is loaded into a container, which is isolated from the host machine. If the host machine is running a completely different operating system, it doesn't matter - the software in the container runs exactly the same as it did on your development machine.

What's the difference between a container and a virtual machine?

A virtual machine emulates an entire computer, including the hardware and the operating system. Virtual machines are resource-hungry - they require large amounts of RAM and hard disk space, since they must run an entire operating system.

A container is a more lightweight solution. Containers ideally contain a software application complete with the libraries it needs to perform its task. However, it takes skill and discipline to build a container that simple, and many containers are built on the entire root filesystem of the underlying operating system, as well as including build-time dependencies. This can make them quite large.

Docker containers are generally at least 30% more resource-efficient compared to virtual machines, with [one benchmark](#) finding the container to be 26 times more efficient in a particular case. If you're interested in actual benchmarks, [this article](#) goes into performance comparisons in some detail.



This performance gain is only true when running on Linux: if it's running on MacOS, the container has to run in Docker's environment which is effectively a virtual machine. For Windows, Docker must run on WSL (Windows System for Linux), which comes with its own overheads and issues. It uses Windows Hyper-V, so again, effectively, it's using a virtual machine.

How does Docker work?

Docker uses various features of Linux that make containerisation possible. These include:

- **Namespaces.** These allow various resources, such as networking and file systems, to be isolated.
- **Control groups.** These allow a group of processes to be managed and limited in their access to resources.

So under Linux, a container can run in a sandbox, where it's isolated from the host machine and any other containers. Under other operating systems, containers run in a sandbox within Docker's virtual machine.

When is Docker useful?

Docker is useful wherever you need to quickly and reliably distribute software on a variety of different platforms. Here are some of the scenarios where Docker has been used to advantage.

- **Microservices.** This is a popular trend in system architecture, which gained momentum with the advent of Docker. Instead of developing monolithic applications to cater to the entire needs of an organization, the software is designed as a collection of microservices that can be developed independently and can interact with each other. This means updates can be developed and deployed quickly and efficiently. Docker is perfect for this, as each service works in its own container and is not affected by dependencies that may have changed when other services are rebuilt.
- **Developing for the cloud.** With Docker, your applications can be run anywhere, and you're no longer subject to vendor lock-in.
- **Creating reproducible environments** for testing, staging, and production. Applications may need various services to be running to test them, and this can easily be achieved with Docker containers. You can be sure that the test and production environments are identical.

The bottom line is: Docker is great for any application that changes often and is a pain to deploy!

Developing for Docker



Let's have a quick look at how an application is packaged and deployed using Docker. You build an image, which can be used anywhere to create a running Docker container

- List all your dependencies, starting with the operating system and version your application needs to run
- Dockerhub, the Docker repository, contains useful base images that you can use when building your own image. Examples include various flavours and versions of Linux, and language-related images such as Python libraries. You can also build your own base images for future use, or use images offered by third parties. These images can be stored in any repo, such as Github, or stored on Dockerhub.
- You can layer these images to create the exact environment you need
- Create a configuration file, known as a Dockerfile, that defines exactly what images and local files you'll use to build your image
- The image can now be built, tested, and pushed to Dockerhub or your own repository. From there, it can be pulled to any Linux, MacOS, or Windows machine, and used to bring up a container running your software
- Since many applications may consist of more than one container, each running a different service, Docker has provided the Docker Compose utility to allow you to manage a group of containers with a single command.

As you can see, Docker eliminates the 'it worked on my machine' syndrome easily and efficiently. What it doesn't address is the 'it worked until I rebuilt it' syndrome.

This is because the Docker build process doesn't have a mechanism to restrict the build to a specific version of packages the application depends on. Typically, Docker builds use commands such as apt-get to load the relevant software, and this simply loads the latest version. Sometimes you'll find your application doesn't work with the latest version, and occasionally the software may not even be available anymore.

And that's where Nix comes in.



What is Nix?

Nix is a package management system with a difference. Nix aims to solve both the 'it worked on my machine' and the 'it worked until I rebuilt it' syndromes. With Nix, every dependency is locked into a specific version. It has a unique approach, where instead of software being installed into a common bin directory, all software resides in the Nix

store, and is identified uniquely by a hash of its inputs. When your software is deployed, Nix will ensure that the exact versions of libraries and other dependencies are available on the host machine, and will install them if they're not.

If you need to have different versions of the same software, or the libraries it depends on - no problem. They can co-exist, and you're able to specify the version you need for a given task.

How does Nix work?

Nix includes its own functional language, which is used to define exact instructions for building a package. When the package is built, Nix establishes a dependency graph, and calculates a hash from these dependencies. This hash is used to create a unique name for the build, and the package is then built and placed in the Nix store.

Except in very rare cases, builds will from a practical point of view be entirely reproducible, although there may be a slight bit-level difference due to compiler idiosyncrasies. When the application is deployed, it will always pick up the correct versions of all libraries and other dependencies. If the application works on the test machine, it will work in production. If you rebuild the software, it will refer to the correct versions of any dependencies, and will not break.

It also has the advantage that it's very easy to roll back to a previous version of the software without disturbing other applications. You can do this with a single command

When is Nix useful?

Nix can be used for building and deploying any type of software, but it is particularly useful for:

- Software that changes often, especially when continuous integration is in place
- Critical applications where you can't afford downtime while a new version is installed. There's no need to bring down the system during installation since the old and new versions can co-exist, and switching versions after the installation takes only a few moments.
- Where a development environment and a production environment are held on the same machine
- When you need to be able to set up the exact development environment quickly for new team members



- When software dependencies need to be auditable. Nix has the facility to generate a full and accurate dependency graph, making it simple to audit the SBOM (Software Bill of Materials.)

Developing with Nix

Setting up the development environment

Nix allows you to easily create a reproducible development and testing environment. The `Nixpkgs` repository contains a huge collection of useful software, including language compilers, IDEs, libraries, and utilities for all the major programming languages, and lots of useful development tools.

You can also include your own repositories and files.

You would define the software you need in a Nix configuration, and then install everything with a single command.

Building and deploying your software

You specify all the dependencies and steps needed to build the software in a Nix configuration. Your software is then built with a single command, and placed in the Nix store.

Nix allows various deployment methods, and there are useful utilities such as `cachix`, `deploy-rs` and `Colmena`, which simplify distribution.

Why would you use Nix and Docker together?

It's simple to use Nix to build your Docker images, and there are several reasons why you might want to do this.

- To ensure reproducible builds, and eliminate the 'it worked until I rebuilt it' syndrome. As we've seen, Docker doesn't address this problem. If you use Nix to deliver packages required by the build, Nix ensures that every build uses exactly the same versions of all software as the original.
- `Nixpkgs` includes some really useful tools for building images.
 - In a complex system, much thought and effort needs to go into how your image



should be layered for best results. With Nix, you don't need to do this: you just specify what you want, and Nix works out the best way of achieving it.

- When built with Nix, the image is generally very much smaller and more efficient because:
 - Nix includes only the dependencies that you need.
 - Nix makes it easy to start with an empty image, instead of starting with the root file system of an operating system distribution, and thus including a lot of unneeded files.
 - Nix includes only the runtime environment, not the build environment
- The build process is usually much faster.
- Since Nix builds software inside a sandbox, you will only be able to include the dependencies you've specified. You can't accidentally include untrusted sources or software that may infringe license agreements. This also prevents accidental dependencies being downloaded from the Internet.
- You don't have to clean up files in your image that were only needed for the build.
- You can use Nix tools to deploy your system easily.

Final thoughts

Should you use Nix or Docker? The answer is: use each of them for what they excel at. But the benefits of using both together are huge. My bottom line is: use both!

SHARE



Donating SrvOS to nix-community

[PREVIOUS POST](#)

Sign up to our website

Subscribe now to get notified about new articles!

EMAIL ADDRESS

SUBSCRIBE

[Contact](#)

[About](#)

[Impressum](#)



[GET IN TOUCH](#)



Guekka's blog

About Posts Tags

Search

NixOS as a server, part 1: Impermanence

Guekka February 20, 2023 [Projects] #nix #self-hosting

A few months ago, I woke up with the idea of hosting my own services. I went through a lot of tries. LXC, Debian, Alpine, (rootless or not) Docker, podman, portainer...

But no solution felt perfect. Until I decided to have a try at hosting using NixOS.

I'm going to assume you know about NixOS and have some prior experience. However, for a small summary: NixOS is a Linux distribution revolving around the Nix package manager. Its main advantage is having a reproducible environment through a declarative configuration. This means that you can copy an entire computer configuration easily: if it works somewhere, it will work anywhere.

My main focus point is reproducibility, so that's why we'll start with configuring **impermanence**.

What's impermanence?

Originally, a philosophic concept. But in our case, impermanence means erasing the / drive at each reboot. You read that right, erasing **almost** everything at each reboot. This part stands on the shoulders of those who did it before me:

- [Erase your darlings: immutable infrastructure for mutable systems - Graham Christensen](#)
- [NixOS *: tmpfs as root](#)
- [Encrypted Btrfs Root with Opt-in State on NixOS](#)
- [Paranoid NixOS Setup - Xe Iaso](#)
- [nix-community/impermanence: NixOS module](#)

The goal is the following: over years, configuration files accumulate. Sometimes editing /etc is required, because of a bug or an obscure configuration. NixOS

allows us to avoid this manual file editing, but it does not **force** us to do so. We can still have a lot of important state, breaking the reproducibility promise.

So what can we do instead? Erase everything, at each reboot. This way, we'll be sure the only source of truth is our configuration.

Installing the system

I'm currently using a **quickemu** VM. This is not a recommended setup and is only done for testing. Configuration file:

CONF

```
#!/usr/bin/quickemu --vm
guest_os="linux"
disk_img="nixos-22.11-minimal/disk.qcow2"
iso="nixos-22.11-minimal/latest-nixos-minimal-x86_64-linux.iso"
disk_size="50G"
ram="4G"
```

Let's first format it:

SH

```
DISK=/dev/vda

parted "$DISK" -- mklabel gpt
parted "$DISK" -- mkpart ESP fat32 1MiB 1GiB
parted "$DISK" -- set 1 boot on
mkfs.vfat "$DISK"1
```

SH

```
parted "$DISK" -- mkpart Swap linux-swap 1GiB 9GiB
mkswap -L Swap "$DISK"2
swapon "$DISK"2
```

Using swap in 2023!?

Yes.

SH

```
parted "$DISK" -- mkpart primary 9GiB 100%
mkfs.btrfs -L Butter "$DISK"3
```

While the `impermanence` module recommends using `tmpfs` for `/`, I chose to use `btrfs`: I do not have RAM to waste. Furthermore, this will allow us to use a nice script we'll see later on.

Let's create `btrfs` subvolumes:

SH

```
mount "$DISK"3 /mnt
btrfs subvolume create /mnt/root
btrfs subvolume create /mnt/home
btrfs subvolume create /mnt/nix
btrfs subvolume create /mnt/persist
btrfs subvolume create /mnt/log
```

And now, the crucial part:

SH

```
btrfs subvolume snapshot -r /mnt/root /mnt/root-blank
```

We just took a snapshot of that empty volume. We will restore it at each reboot. We can now mount the subvolumes and let nixos-generate-config do its job

SH

```
mount -o subvol=root,compress=zstd,noatime "$DISK"3 /mnt
mkdir /mnt/home
mount -o subvol=home,compress=zstd,noatime "$DISK"3 /mnt/home

mkdir /mnt/nix
mount -o subvol=nix,compress=zstd,noatime "$DISK"3 /mnt/nix

mkdir /mnt/persist
mount -o subvol=persist,compress=zstd,noatime "$DISK"3 /mnt/persist

mkdir -p /mnt/var/log
mount -o subvol=log,compress=zstd,noatime "$DISK"3 /mnt/var/log

mkdir /mnt/boot
mount "$DISK"1 /mnt/boot

nixos-generate-config --root /mnt
```

Lastly, we only have to edit the generated configuration files at `/mnt/etc/nixos`.

My final configuration is available [here](#). You can follow all the steps by looking at the [commits](#).

Configuring the system

- Checking that we have the correct mount options in `/mnt/etc/nixos/hardware-configuration.nix`.

I've added "compress=zstd" "noatime" to all filesystems. We also need to add `neededForBoot` to `/var/log` and `/persist`.

- Replacing default values in `configuration.nix`

I've enabled `networkmanager`, removed most suggested options and enabled `system.copySystemConfiguration`.

This last option copies the current configuration to `/run/current_system/configuration.nix`. You should not rely on it: keep your configuration in a git repository. But it can serve as some kind of last chance.

- Declaring a user, including ssh

NIX

```
users.mutableUsers = false;
users.users.user = {
  isNormalUser = true;
  extraGroups = [ "wheel" ];

  openssh.authorizedKeys.keys = [ "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAICWVNch9BcjMqS/Xwe
    # passwordFile needs to be in a volume marked with `neededForBoot = true`
    passwordFile = "/persist/passwords/user";
};
```

Here we have completely disabled imperative user modification. This does not matter much, as imperative changes would be erased anyway at start. We thus need to provide a password. We're using `passwordFile` for that: a path to a file containing the hashed password.

Here's how to generate that file: `sudo mkpasswd -m sha-512 "hunter2" > /mnt/persist/passwords/user`.

The SSH key was generated using `'ssh-keygen -t ed25519 -C "nixos"`.

- Enabling openSSH We're going to use Xe's configuration:

NIX

```
services.openssh = {
  enable = true;
  passwordAuthentication = false;
  allowSFTP = false; # Don't set this if you need sftp
  challengeResponseAuthentication = false;
  extraConfig = ''
    AllowTcpForwarding yes
    X11Forwarding no
    AllowAgentForwarding no
    AllowStreamLocalForwarding no
    AuthenticationMethods publickey
  '';
};
```

This reduces attack surface, for example by disabling stream-local forwarding and disabling password authentication.

This will be enough for now. Let's install the system before going to the next step: `sudo nixos-install --root /mnt && sudo reboot`. You should be able to connect by SSH using the previously defined key, or login using the password you defined in `/persist/passwords/user`.

Configuring impermanence

We've created our volumes, we've configured the system... But I promised we would reset our system at each reboot. Let's do that now! We're going to use the following script, credit of mt-caret. Do not forget to replace `vda3` with your data partition.

16/07/23 update: it was brought to my attention that `postDeviceCommands` can cause data loss. While I did not experience any issue, I have updated the script to use a safer alternative.

NIX

```
boot.initrd = {
    enable = true;
    supportedFilesystems = [ "btrfs" ];

    systemd.services.restore-root = {
        description = "Rollback btrfs rootfs";
        wantedBy = [ "initrd.target" ];
        requires = [
            "dev-vda3"
        ];
        after = [
            "dev-vda3"
            # for luks
            "systemd-cryptsetup@${{config.networking.hostName}.service}"
        ];
        before = [ "sysroot.mount" ];
        unitConfig.DefaultDependencies = "no";
        serviceConfig.Type = "oneshot";
        script = ''
            mkdir -p /mnt

            # We first mount the btrfs root to /mnt
            # so we can manipulate btrfs subvolumes.
            mount -o subvol=/ /dev/vda3 /mnt

            # While we're tempted to just delete /root and create
            # a new snapshot from /root-blank, /root is already
            # populated at this point with a number of subvolumes,
            # which makes `btrfs subvolume delete` fail.
            # So, we remove them first.
            #
            # /root contains subvolumes:
            # - /root/var/lib/portables
            # - /root/var/lib/machines
            #
            # I suspect these are related to systemd-nspawn, but
```

```
# since I don't use it I'm not 100% sure.  
# Anyhow, deleting these subvolumes hasn't resulted  
# in any issues so far, except for fairly  
# benign-looking errors from systemd-tmpfiles.  
btrfs subvolume list -o /mnt/root |  
cut -f9 -d' ' |  
while read subvolume; do  
    echo "deleting /$subvolume subvolume..."  
    btrfs subvolume delete "/mnt/$subvolume"  
done &&  
echo "deleting /root subvolume..." &&  
btrfs subvolume delete /mnt/root  
  
echo "restoring blank /root subvolume..."  
btrfs subvolume snapshot /mnt/root-blank /mnt/root  
  
# Once we're done rolling back to a blank snapshot,  
# we can unmount /mnt and continue on the boot process.  
umount /mnt  
';  
};
```

We can then specify the files we want to keep.

But which files do we want to keep? Let's find out. Thanks to another useful script of mt-caret, we can list the differences between our current / and the blank state:

SH

```
#!/usr/bin/env bash
# fs-diff.sh
set -euo pipefail

OLD_TRANSID=$(sudo btrfs subvolume find-new /mnt/root-blank 9999999)
OLD_TRANSID=${OLD_TRANSID##transid marker was }

sudo btrfs subvolume find-new "/mnt/root" "$OLD_TRANSID" |
sed '$d' |
cut -f17- -d' ' |
sort |
uniq |
while read path; do
    path="/$path"
    if [ -L "$path" ]; then
        : # The path is a symbolic link, so is probably handled by NixOS already
    elif [ -d "$path" ]; then
        : # The path is a directory, ignore
    else
        echo "$path"
    fi
done
```

Used like this:

SH

```
sudo mkdir /mnt ; sudo mount -o subvol=/ /dev/vda3 /mnt ; ./fs-diff.sh
```

Here's the result of mine:

```
/etc/.clean
/etc/group
/etc/machine-id
/etc/nixos/configuration.nix
/etc/nixos/hardware-configuration.nix
/etc/passwd
/etc/resolv.conf
/etc/shadow
/etc/ssh/authorized_keys.d/user
/etc/ssh/ssh_host_ed25519_key
/etc/ssh/ssh_host_ed25519_key.pub
/etc/ssh/ssh_host_rsa_key
/etc/ssh/ssh_host_rsa_key.pub
/etc/subgid
/etc/subuid
/etc/sudoers
/etc/.updated
/root/.nix-channels
/root/.nix-defexpr/channels
/var/lib/NetworkManager/internal-84e273c2-b91a-3a96-b341-8234a339bdc7-enp0s8.lease
/var/lib/NetworkManager/internal-84e273c2-b91a-3a96-b341-8234a339bdc7-enp0s9.lease
/var/lib/NetworkManager/NetworkManager-intern.conf
/var/lib/NetworkManager/secret_key
/var/lib/NetworkManager/timestamps
/var/lib/nixos/auto-subuid-map
/var/lib/nixos/declarative-groups
/var/lib/nixos/declarative-users
/var/lib/nixos/gid-map
/var/lib/nixos/uid-map
/var/lib/systemd/catalog/database
/var/lib/systemd/random-seed
/var/.updated
```

That's not too bad!

Out of these, there's almost nothing I want to preserve.

Let's make use of the `impermanence` module. We need to download it:

NIX

```
let
  impermanence = builtins.fetchTarball "https://github.com/nix-community/impermanence/archive/1.0.0.tar.gz"
in
{
  imports = [ "${impermanence}/nixos.nix" ./hardware-configuration.nix ]
  // the whole configuration
}
```

And now, we can just tell it the files and directories that we want:

NIX

```
# configure impermanence
environment.persistence."/persist" = {
  directories = [
    "/etc/nixos"
  ];
  files = [
    "/etc/machine-id"
    "/etc/ssh/ssh_host_ed25519_key"
    "/etc/ssh/ssh_host_ed25519_key.pub"
    "/etc/ssh/ssh_host_rsa_key"
    "/etc/ssh/ssh_host_rsa_key.pub"
  ];
  security.sudo.extraConfig = ''
    # rollback results in sudo lectures after each reboot
    Defaults lecture = never
  '';
};
```

What an ergonomic interface.

Wait, did you just say Nix was ergonomic?

Well, yes. Sometimes.

I have not saved my network manager configuration, but you may need to.

When new files are set to be preserved, it is necessary to copy them manually to `/persist`:

SH

```
sudo nixos-rebuild boot

sudo mkdir /persist/etc

sudo cp -r {,/persist}/etc/nixos
sudo cp {,/persist}/etc/machine-id

sudo mkdir /persist/etc/ssh

sudo cp {,/persist}/etc/ssh/ssh_host_ed25519_key
sudo cp {,/persist}/etc/ssh/ssh_host_ed25519_key.pub
sudo cp {,/persist}/etc/ssh/ssh_host_rsa_key
sudo cp {,/persist}/etc/ssh/ssh_host_rsa_key.pub
```

Now, if we reboot and list files again:

```
/etc/.clean
```

```
/etc/group  
/etc/passwd  
/etc/resolv.conf  
/etc/shadow  
/etc/ssh/authorized_keys.d/user  
/etc/subgid  
/etc/subuid  
/etc/sudoers  
/etc/.updated  
/root/.nix-channels  
/var/lib/NetworkManager/internal-84e273c2-b91a-3a96-b341-8234a339bdc7-enp0s9.lease  
/var/lib/NetworkManager/NetworkManager-intern.conf  
/var/lib/NetworkManager/secret_key  
/var/lib/NetworkManager/timestamps  
/var/lib/nixos/auto-subuid-map  
/var/lib/nixos/declarative-groups  
/var/lib/nixos/declarative-users  
/var/lib/nixos/gid-map  
/var/lib/nixos/uid-map  
/var/lib/systemd/catalog/database  
/var/lib/systemd/random-seed  
/var/.updated
```

Success! The files we persisted are no longer showing up.

What about our home directory?

It is possible to setup the impermanence module for our home directory. However, I did not want to go through `home-manager` installation. Furthermore, a home directory is meant to be stateful.

In our case, we are creating a server, so it would still make sense to configure it. If you are interested, have a look at [tmpfs at home](#).

Next steps

In the next part, we will make ~~the entire nix store~~ by making it only available through Tailscale. We will also ~~setup a root service~~.

© 2024 Guekka's blog

I hope you've enjoyed this article! Thanks for reading to the end!

Powered by [Zola](#) & [Abridge](#)

← NixOS as a server, part 2: Flake, tailscale



RESEARCH

PROGRAMMING

ARTIFICIAL INTELLIGENCE

INTERVIEWS

OTHER

Our New Nix Deployment Tool: deploy-rs

Article by Mika Logan
November 24th, 2020

6 min read

191



At Serokell, we use a lot of Nix. While it helps us in so many ways, we felt that the options for Nix+NixOS deployments were actually quite lacking, and no existing tool fit the requirements for our infrastructure.

We tried out [morph](#), [NixOps](#), some custom per-project deployment scripts, even plain old [nixos-rebuild](#), and some non-Nix solutions, but none of them solved every problem.

Therefore, we decided to look into creating our own solution: [deploy-rs](#).

We found that it must be:

- **Remote.** We don't (always) want to perform the builds on the target server.
- **Stateless.** Synchronizing state data between each of our developers and automatic deployments is not worth any benefit.
- **sudo -compatible.** We don't want to need a root login for deployments, even if it is a full NixOS system closure.
- **Safe.** We don't want to waste Nix's amazing ability to roll back if activation fails.
- **Re-usable.** The program must fit every one of our Nix deployments.

There are a lot of NixOS deployment tools that already fit these needs, though we would still have to give root access to our CI/CD machines, or use a non-Nix solution for all app deployments, both of which are undesirable.

The solution we came up with is to *not* make or use a NixOS deployment tool at all, but instead create a Nix deployment tool that supports NixOS and any other type of Nix profile, system or otherwise.

The result was [this](#), a wonderfully simple bash script. It could not only deploy a NixOS system, but also any profile you want (as outlined in [this](#) example).

It makes great use of Nix's new [flakes](#) feature and treats NixOS deployments the same way it treats a deployment of any other profile, giving us the ability to deploy any application's root-less profiles alongside the

NixOS profile.



It was an amazing first prototype and showed that Nix deployment could still be improved upon, though it still had a number of issues. It was in Bash (which is fine, though risky to continue to develop given it is critical infrastructure), had theoretical rollback issues if certain properties changed, and had some since-abandoned ideas in it (such as `bootstrap`) from previous experimentation.

`deploy-rs` is a full re-implementation of the ideas behind this initial tool. We had already struck a good idea but just needed to polish it.

This tool follows the same pattern but expands upon it, and is written entirely in Rust so we can enjoy additional safety, expressiveness, and speed. We kept the same design, the flakes, the multi-profile implementation, but improved the interface and added more functionality.

Features >

We've implemented a lot of the basic features present in a lot of the alternative NixOS deployment tools, though our own tool has some distinct features that solve issues in both NixOS and non-NixOS deployment profiles.

For a more complete list, you should read the official [README](#) of the project.

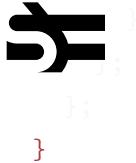
Multi-profile >

A standard NixOS deployment tool will let you deploy a NixOS configuration to a machine. However `deploy-rs` is a *Nix* deployment tool with no explicit dependency on NixOS at all.

For each of the servers (called nodes), you may have any number of profiles, which get deployed independently of each other, you can have one be a NixOS profile installing as root, another be some other type of profile deploying as any user you want.

```
{\n    deploy.nodes.example = {\n        hostname = "localhost";\n        profiles = {\n            system = {\n                user = "root";\n                path = deploy-rs.lib.x86_64-linux.activate.nixos self.nixosConfigurations.example;\n            };{\n                hello = {\n                    user = "hello";\n                };\n            };\n        };\n    };\n}
```





```
path = deploy-rs.lib.x86_64-linux.activate.custom self.defaultPackage.x86_64-1
```



Magic rollback



As mentioned before, one of the things that make Nix/NixOS great is the ability to roll back. This is something many of the other tools support, but only when activation fails. We decided to take this to the next level. `deploy-rs` (if `magic-rollback` is enabled) will create a “canary” file after profile activation, which will get deleted by the deploying end, otherwise the profile will roll-back and re-activate after 30 seconds (though the timeout is configurable).

This means we don’t have to worry about causing disasters as we modernize our infrastructure to use our new tooling. If we remove our own keys, mess up the networking, or otherwise prevent ourselves from accessing the server, the mistake will only last for 30 seconds at most.

Static activation paths



If you have read through the code of `deploy` above, you might have noticed this comment:

```
# Assuming that activation command didn't change
eval "$SUDO $activate"
```

This is actually important, as it is making the assumption that the activation command to re-activate the previous generation is the same as the one used to activate the current generation. This means that if you change your activation command and it results in a failure, the rollback will fail too.

In `deploy-rs` we solved this by having the activation path be static and standardized: `deploy-rs` will always run the file in `${yourProfile}/deploy-rs-activate`. To bring back some of the convenience of the `activate` field in our bash ancestor, we include a utility called `lib.activate` in our flake. To activate with a custom command, you can use `deploy-rs.lib.x86_64-linux.activate.custom pkgs.hello ./bin/hello`, to activate a NixOS system you can use `deploy-rs.lib.x86_64-linux.activate.nixos self.nixosConfigurations.some-random-system`, and if you just want to write the profile without activating anything, you can use `deploy-rs.lib.x86_64-linux.activate.noop`.



Interactive mode



If you provide the `-i` flag, `deploy-rs` will print everything that will be activated in toml and prompt you

continuing.



```
[examplecomputer.system]
user = "root"
ssh_user = "exampleuser"
path = "/nix/store/xxrgxdg8x9zy8p0ky3d2hff4wf5igjpk-activatable-nixos-system-exampleco
hostname = "localhost"
ssh_opts = []

[examplecomputer.exampleuser-home-manager]
user = "exampleuser"
ssh_user = "exampleuser"
path = "/nix/store/k32k88vbyx9yqk7fk80bk0vcybzbkhra-activatable-home-manager-generatio
hostname = "localhost"
ssh_opts = []

INFO deploy > Are you sure you want to deploy these profiles?
>
```

Non-flake support



While the implementation and mere concept of this tool stem from the new unstable-only Nix flakes command, we've made sure that this tool will work with any stable version of Nix too.

Once you have created your flake and `flake.lock` with a newer version of Nix, you can use `flake-compat` to make your flake accessible on older Nix versions through a `default.nix`. `deploy-rs` will automatically detect the lack of flakes support, read from the `default.nix`, and perform certain workarounds (such as manually building the `.checks` attribute in absence of `nix flakes check`).

Automatic checks



`deploy-rs`'s flake ships with some utility checks to put in your flake checks and will automatically check all of them before deployment (unless `--skip-checks` is specified). This helps you make sure there are no issues in your infrastructure before each deployment.



You can use our included checks by simply adding this to your `flake.nix`:



Passing Nix arguments



Any trailing argument given to `deploy-rs` will be passed on to Nix itself, so you can do things like `deploy . -- --override-input nixpkgs ./nixpkgs`.

How we use it



There are examples already included in the GitHub repository, though if you want to see some real-world usage, check out our [web app cluster definition](#), where we exclusively utilize `deploy-rs` for NixOS deployments, and [one of the apps we run on it](#) which utilizes `deploy-rs` for deploying the service without root.

It's worth noting that although in these examples we have the systemd service in the NixOS part of the deployment, and just a simple `systemctl restart` on the app end, you can just as easily use systemd user services to accomplish the same thing without sudo rules. We just chose not to.

You can even use it to deploy a [home-manager](#) configuration without root.

The future



It has reached a feature-complete stage and passed review by our own team, though our work is far from done. We want to grow `deploy-rs` into the ultimate deployment tool, with well-designed tooling for all of our use cases.

Our next immediate goal (alongside bugfixes and code cleanup) is to support systemd portable services, to avoid the issue in the above section where we are using system-level services for user applications.

We're also considering the idea of a module system to add features such as a module for automatic `kexec` illustration of servers allowing you to convert systems to NixOS during deployment, a module for integrating with [Vault](#), and more.



Conclusion



We hope this tool is as useful to others as it has been to us, allowing you to deploy with Nix without needing root and a full system configuration to deploy a full NixOS profile.

Feel free to check out [deploy-rs](#) on GitHub and leave us a star if you like our project.



TAGGED: nix rust library

Share:



191 upvotes

Get new articles via email

No spam – you'll only receive stuff we'd like to read ourselves.

Enter your e-mail

Accept [Privacy notice](#)

Subscribe





Monads. Homegrown & Sustainable.

[Serokell Shop →](#)



[More from Serokell](#)



HACKAGE SEARCH



Hackage Search: Regex-Based Online Code Search

Have you ever wanted to find something on Hackage? If yes, we have a tool for you.

December 14th, 2020 | 3 min read

Biotech

Blockchain

Fintech

Managed IT Services

ML Consulting

Smart Contract Audit

Elixir Development

Haskell Development

Nix Development

Python Development

Rust Development

TypeScript Development

How We Work

Privacy Policy

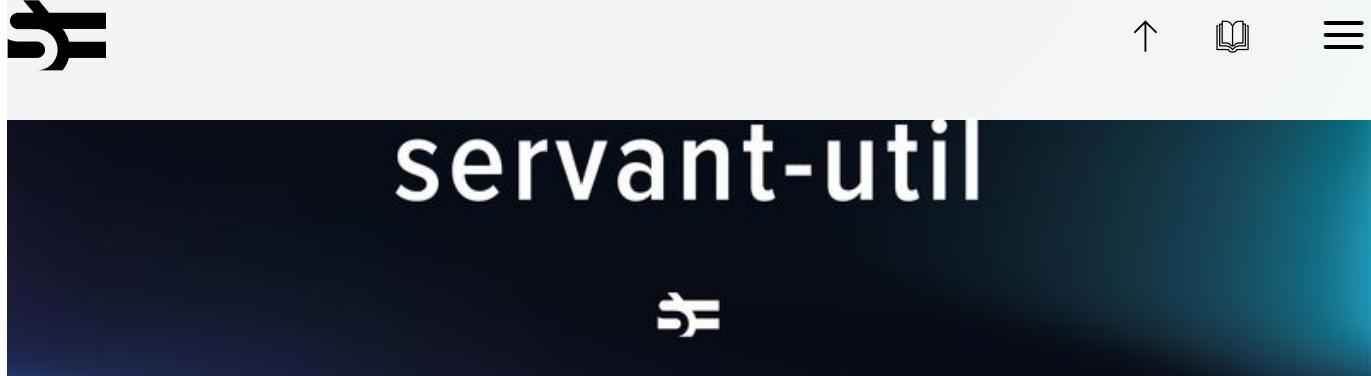
(+372) 699-1531

hi@serokell.io



Pille tn 7/5-13, Keskkonna linnaosa, Tallinn, Harju maakond, 10135, Estonia

Serokell: Rate 5.0 based on 9 Google Business reviews



servant-util: Extending servant with Database Integration

Learn more about servant-util, our new library that helps servant users with webserver-database integration.

August 16th, 2021 | 14 min read



[Xe](#)[Blog](#)[Contact](#)[Resume](#)[Talks](#)[VODs](#)[Signalboost](#)

Paranoid NixOS Setup

Published on 07/18/2021, 4278 words, 16 minutes to read

Most of the time you can get away with a fairly simple security posture on NixOS. Don't run services as root, separate each service into its own systemd units, don't run packages you don't trust the heritage of and most importantly don't give random people shell access with passwordless sudo.

Sometimes however, you have good reasons to want to lock everything down as much as humanly possible. This could happen when you want to create production servers for something security-critical such as a bastion host. In this post I'm going to show you a defense-in-depth model for making a NixOS server that is a bit more paranoid than usual, as well as explanations of all the moving parts.

High-level Ideas

At a high-level I'm assuming the following things about this setup:

- It should be very difficult to get in as a passive attacker
- But the defense doesn't stop at "just hope they don't get in"
- It should be annoying for attackers to get a user-level shell
- But ensure they'll be able to anyways if they're dedicated enough
- It should be difficult for attackers to run their own code on the system
- But assume that it could happen and make evidence of that very loud
- It should be aggravating for attackers to access the package manager on the system

- But ensure that they can't do anything very easily even if they can access the package manager itself

Some additional goals:

- Make the system only manageable by a central management system such as morph or nixops
- Only make SSH visible over a VPN of some kind, such as Tailscale or another WireGuard setup
- Mount the root filesystem on a tmpfs
- Have explicitly defined persistent folders
- Mark everything as `noexec` except for the mount that `/nix/store` is on
- Don't make the system too difficult to use in the process



<Cadey> Disclaimer: I am a Tailscale employee. Tailscale did not review this post for accuracy or content, though this setup is based on conversations I've had with a coworker at Tailscale.

Along the way we'll be making a system that I'm naming `meeka`. We'll put its configuration in a folder named `meeka`:

```
# hosts/meeka/configuration.nix
{ ... }:

{
    networking.hostName = "meeka";
    services.openssh.enable = true;
}
```

Low-hanging Fruit

There are some easy things we can get out of the way. One of the biggest ways that people get in is to make services visible to attack in the first place.

The Firewall

Let's get one of the lowest-hanging fruits out of the way: the firewall. Most of the background radiation of the internet is in the form of automated probes to development ports and SSH traffic. NixOS actually includes a firewall by default! You can see more information on how to configure it [here](#), but here's a good collection of values to use by default:

```
# hosts/meeka/firewall.nix
{ ... }:

{
  networking.firewall.enable = true;
}
```

VPN for Access

Generally, it's probably okay to use SSH over the unprotected internet for accessing your machines. However, this is all about maximum paranoia, so we're going to use a VPN to get into the machine. [Tailscale](#) is a fairly direct thing to set up in NixOS:

```
# hosts/meeka/tailscale.nix
{ ... }:

{
  services.tailscale.enable = true;

  # Tell the firewall to implicitly trust packets routed over Tailscale:
  networking.firewall.trustedInterfaces = [ "tailscale0" ];
}
```

When you boot into the server, you can log in like normal using the `tailscale up` command. You can probably isolate down the server using [ACLs](#) if you want to make sure things are a bit more paranoid.

It may be good to set up a second way to get into the machine, just in case. I personally try to leave at least 3 ways into my servers, but the super paranoid production-facing

servers should probably only be able to be connected to over a VPN of some kind.

If you want to see more about how to set up WireGuard on NixOS, see [here](#) for more information.

Locking Down the Hatches

Now that we're getting out of the easy stuff, let's go to the more defense in depth stuff. Here we're going to talk about separation of concerns and all those other fun things.

Each Service Gets its own User Account

I am going to use the word "service" annoyingly vague here. In this world, a "service" is a human-oriented view of "computer does the thing I want it to do". This website you're reading this post on could be one service, and it should have a separate account from other services. See [here](#) for more information on how to set this up.

Lock Down Services Within Systemd

`systemd` is a suite of tools that NixOS uses to manage a huge chunk of the system. It is kinda complicated and very large in scope, however this also means that you get access to a lot of convenient security management features. One of them is the `Protect*` unit options in `systemd.exec(5)`, which can be used to lock down permissions to the resource and system call level. Let's cover some of my favorites that you can slipstream into services:

Also take a look at `systemd-analyze security yourservicename.service`, that will give you a lot more things to search through the systemd documentation for.

``ProtectHome` / `ProtectSystem``

These options allow you to change how systemd presents critical system files and ``/home`` to a given process. You can use this to remove the ability for a service to modify system files or peek into user's home directories, even as root. This allows you to put a lot more limits on a service's power.

`NoNewPrivileges`

If this is set, child processes of this service cannot gain more privileges period. Even if the child process is a suid binary.



<**Mara**> A suid binary is a binary that has the suid flag set. This makes the Linux kernel change the active user field of that binary to the owner of the binary when you run it. This is a huge part of how the magic behind sudo and ping works.

`ProtectKernel{Logs, Modules, Tuneables}`

These ones are fairly simple so I'm gonna use some bullet trees for them:

- `ProtectKernelLogs`: If set to true, the service cannot access the kernel message buffer that you get by running `dmesg` or reading from `/proc/kmsg`.
- `ProtectKernelModules`: If set to true, the service cannot load or unload kernel modules.
- `ProtectKernelTunables`: If set to true, various twiddly bits in `/proc` and `/sys` that let you control tunable values in the kernel will be made read-only. Most of the time these values are set early in the system boot process and never twiddled with again, so it's reasonable to deny a service (and its child processes) access to these



<**Mara**> Why should I bother making all of these changes to my services though? Isn't it overkill to have a webapp running as a service user get denied access to even look at the kernel log?



<**Cadey**> To be honest, it can look like paranoid overkill, but this isn't just for the service itself. This is for defense in *depth*, which means that you want to make sure that things are reasonably secure even if an attacker manages to get code execution on one of your services. These settings prevent the service's view of the system from having too much detail, which can make the attacking process more annoying. Remember that the goal here isn't to make the system attack-proof, nothing is. The goal is to annoy the

attacker enough that they give up. This is not perfect and probably will fall apart if your enemy is the Mossad, but it's at least an attempt to lock things down just in case the attackers aren't sending their "A" game. You may also want to look into `InaccessiblePaths` to block away other folders that you deem "forbidden" as facts and circumstances demand.

Lock Down Nix Access

Nix is the package manager for NixOS. Nix can be invoked by users. Nix lets users access things like compilers and scripting languages. These can be used to run exploit tools. This can be understandably problematic from a security standpoint.

NixOS has an option called `nix.allowedUsers` that lets you specify which users or groups are allowed to do anything with the Nix daemon, and by extension the Nix package manager. For a fairly standard setup, you can probably get away with the following which allows everyone that can `sudo` to access the Nix daemon:

```
# configuration/meeka/nix.nix
{ ... }:

{
  nix.allowedUsers = [ "@wheel" ];
}
```

However if you want to prevent everyone but root, you can use a configuration like this:

```
# configuration/meeka/nix.nix
{ ... }:

{
  nix.allowedUsers = [ "root" ];
}
```

You may also want to block access to the NixOS cache CDN with an external firewall rule if you really don't trust things. You can block it by blocking the fastly IP range `151.101.0.0/16`.



<**Mara**> I'd suggest doing this firewall change on the level above the NixOS machine itself, just in case the machine gets owned and then they ditch your firewall rules in an effort to aid in exfiltration.

Making the System Amnesiac

Most of these steps go way deep down the security rabbit hole. A lot of these are focused on limiting access to persistent storage so that persistence is opted into, not opted out of. These steps will essentially mount the root filesystem on a tmpfs that is cleared out on every reboot, with persistent data written to a subfolder in `/nix` that a symlink/bindmount farm is linked to. Most of these steps will require you to reprovision your NixOS machines and may require you to build your own custom images for cloud providers. Your experience and mileage may vary.

These steps will be based on the excellent work done in these posts/projects:

- [impermanence](#)
- [NixOS *: tmpfs as root](#)
- [Erase your darlings](#)

Partitioning/Setup

Normally the NixOS partition setup looks a bit like this:

- `/boot` for either BIOS boot or EFI files
- 2x ram for swap (swap is not a panacea, however it can sometimes give you valuable time to debug a problem)
- `/` for everything else



<Mara> It's worth noting that technically NixOS works fine if you make only one big filesystem and put `/boot` on there directly, but this may only pan out for BIOS booting systems.

Given that `/` is going to become an in-memory tmpfs, we can instead move the partitioning to look like this:

- `/boot` for either BIOS boot or EFI files
- 2x ram for swap
- `/nix` for everything else

Assuming you are installing NixOS from scratch in a VM to test this part out, the partitioning setup commands could look something like this:

```
dev=/dev/vda # replace me with the actual device
parted ${dev} -- mklabel msdos
parted ${dev} -- mkpart primary ext4 1M 512M
parted ${dev} -- set 1 boot on
parted ${dev} -- mkpart primary ext4 512MiB 100%
mkfs.ext4 -L boot ${dev}1
mkfs.ext4 -L nix ${dev}2
```



<Mara> Wait, ext4? I thought you were a zfs stan?

<Mara> I normally am, however in this case it's probably better to keep Mara is enby the scary production servers as boring and vanilla as possible, especially when doing a more weird setup like this.

The exact size of your `/boot` partition may vary based on facts and circumstances, however in practice I've found 512 MB to be a not-terrible default.

Make your "root mount" with a tmpfs:

```
mount -t tmpfs none /mnt
```

Then you need to create the persistent folders on `/nix/persist`. I've found these defaults to be not-horrible:

```
mkdir -p /mnt/{boot,nix,etc/{nixos,ssh},var/{lib,log},srv}
```



<Mara> We use `/srv` as the home for our services. Adjust this as your facts and circumstances demand.

Then mount those two partitions to your tmpfs:

```
mount ${dev}1 /mnt/boot  
mount ${dev}2 /mnt/nix
```

And create matching folders in `/mnt/nix/persist`:

```
mkdir -p /mnt/nix/persist/{etc/{nixos,ssh},var/{lib,log},srv}
```

Then finally create some bind mounts to tie everything together for the meantime. These bindmounts will be handled by impermanence in the future, however for now the quick and dirty method will suffice:

```
mount -o bind /mnt/nix/persist/etc/nixos /mnt/etc/nixos  
mount -o bind /mnt/nix/persist/var/log /mnt/var/log
```

Then generate a base config with `nixos-generate-config`:

```
nixos-generate-config --root /mnt
```

And open `/etc/nixos/hardware-configuration.nix` to edit the settings for the tmpfs mount on `/`. At a high level you'll need to change this:

```
fileSystems."/ = {
  device = "none";
  fsType = "tmpfs";
  options = [ "defaults" "mode=755" ];
};
```

to this:

```
fileSystems."/ = {
  device = "none";
  fsType = "tmpfs";
  options = [ "defaults" "size=2G" "mode=755" ];
};
```

This will limit `/` to taking up 2 GB of storage at most. This will mostly contain temporary files and the like, but you should adjust this as makes sense given the amount of ram your systems have. I personally think that 512 MB could make sense depending on what you are doing.

Using Impermanence

Now we get to add impermanence to the mix to handle making all of those pesky bind mounts for us on boot. One of the easiest ways you can add its module to the nix search path is to set the `NIX_PATH` environment variable like this:

```
export NIX_PATH=nixpkgs=channel:nixos-21.05:impermanence=https://github.com/nix-community/impe
```

This will set the import path `<impermanence>` to point to the git repository for impermanence. Depending on your security needs you may want to mirror the impermanence git repo, but keep in mind it needs to point to a tarball for Nix to understand what to do with it.

Once you have that added, you can add the impermanence configuration to your `/etc/nixos/configuration.nix`:

```
environment.persistence."/nix/persist" = {
```

```
directories = [
    "/etc/nixos" # nixos system config files, can be considered optional
    "/srv"        # service data
    "/var/lib"    # system service persistent data
    "/var/log"    # the place that journald dumps its logs to
];
};
```

Finally you'll want to set these configuration lines for files in `/etc/sshd`. I've tried doing it directly in `environment.persistence.<name>.directories` directly but it seems to make `sshd.service` unable to generate its host keys, which is slightly important for sshd to work at all. These lines will point the files to the right places:

```
environment.etc."ssh/ssh_host_rsa_key".source
= "/nix/persist/etc/ssh/ssh_host_rsa_key";
environment.etc."ssh/ssh_host_rsa_key.pub".source
= "/nix/persist/etc/ssh/ssh_host_rsa_key.pub";
environment.etc."ssh/ssh_host_ed25519_key".source
= "/nix/persist/etc/ssh/ssh_host_ed25519_key";
environment.etc."ssh/ssh_host_ed25519_key.pub".source
= "/nix/persist/etc/ssh/ssh_host_ed25519_key.pub";
```

The machine ID may be important too if you want to read logs locally after you reboot, or if you have any services that expect the machine ID to not change.

```
environment.etc."machine-id".source
= "/nix/persist/etc/machine-id";
```

From here you can continue with `nixos-install` like normal (though you may want to add `--no-root-passwd` if you added a default root password to your config for bootstrap reasons only). However if you want to be lazy you can read below where I show you how to automatically create an ISO that does all this for you.

Repeatable Base Image with an ISO

Using the setup I mentioned [in a past post](#), you can create an automatic install ISO that will take a blank disk to a state where you can SSH into it and configure it further using a

tool like [morph](#). Take a look at [this folder](#) in my nixos-configs repo for more information. Most of the magic is done with the `build` script. It's basically the last few sections of this article turned into nix files. If you build it yourself you'll want to take care with the line that looks like this:

```
users.users.root.initialPassword = "hunter2";  
users.users.root.openssh.authorizedKeys.keyFiles = [ (fetchKeys "Xe") ];
```

This sets the root password to `hunter2` (a reasonably secure default for bootstrapping systems only, holy crap do not use this in production) so you can log in with the console and the list of SSH keys from [here](#). Replace `Xe` with your GitHub username. This is not the most deterministic, but if GitHub is down you probably have bigger problems. It's also a decent crutch to help you bootstrap things. If this bothers you you can set authorized keys as normal:

```
users.users.root.openssh.authorizedKeys.keys = [  
    "ssh-yolo swag420blazeit"  
];
```

You can turn this into an EC2 image with something like [packer](#).

Audit Tracing

The Linux kernel has some fancy auditing powers that are criminally under-used.



<[Mara](#)> Isn't that because the audit subsystem has the ergonomics of driving a submarine down a road?

Well, yes but until I learn how to summon the right kinds of daemons, I can start with this audit rule to log every single time a program is attempted to be run:

```
# hosts/meeka/auditd.nix  
{ ... }:
```

```
{  
    security.auditd.enable = true;  
    security.audit.enable = true;  
    security.audit.rules = [  
        "-a exit,always -F arch=b64 -S execve"  
    ];  
}
```

You can monitor these logs with `journalctl -f`. If you don't see any audit logs show up, ssh in from another window and run some commands like `ls`. You should see a flurry of them show up.

Send All Logs Off-Machine

You should really treat all system-local logs as radioactive. They are liabilities and in some cases can present problematic situations when faced with questionable interpretations of things like the GDPR. Not to mention attackers will be tempted to wipe all record of their attacks from them. I don't really have a suggestion for the best practice here, but I'm sure that people smarter than me have come up with good suggestions in my place. Either way, get them off the system as fast as possible.

You should probably have some process scraping the audit logs to check for programs outside of `/nix/store` being executed. That can sometimes point to signs of a break-in.

Optional Steps

Normally a lot of these suggestions are aimed at not totally interfering with normal usability so that in case you need to debug things you can do so with surgical precision. However, depending on your level of paranoia you may want to go a step further and disable some things that most may consider to be a "core part of basic usability". Just be aware that these things may make debugging an errant system difficult.

Rip Out `sudo`

sudo is a commonly used tool that allows users to assume superuser powers for a short

amount of time. The things they do with `sudo` are logged to the system, but this project has been known to occasionally have security issues.



<**Mara**> Isn't that because it's written in C and C is inherently unsafe even though hordes of "experts" decry otherwise?



<**Cadey**> Don't say that, you'll incite the horde.

NixOS lets us rip that out if we want to:

```
# hosts/meeka/sudo.nix
{ ... }:

{
    security.sudo.enable = false;
}
```

If you want to keep it around but instead limit its use to users that are in the `wheel` group, you can instead opt for something like this:

```
# hosts/meeka/sudo.nix
{ ... }:

{
    security.sudo.execWheelOnly = true;
}
```

Rip Out Default Packages

By default NixOS comes with a few packages like nano, perl and rsync to help you get started using it. These are great and all, but can be slightly incredibly problematic from a security standpoint. Rip them out like this:

```
# hosts/meeka/no-defaults.nix
```

```
{ lib, ... }:

{
```

```
    environment.defaultPackages = lib.mkForce [];
}
```



<Mara> The `lib.mkForce` function forcibly overrides the contents of that value to what you give as an argument. This is useful for saying "no, heck you, I want it to be set to this no matter what anyone else says". This can be a useful hammer when correcting the security model of NixOS services when you have a good reason to.

Disable sshd Features

sshd is great. You can use it to log into systems, proxy traffic and more. sshd is also horrible because you can proxy traffic and more, turning a machine into an unexpected jumpbox for attackers. This is not ideal for machines that you don't expect to be jumpboxes. Disable this feature and some more (such as X11 forwarding, SSH agent forwarding and stream-local forwarding) like this:

```
# configuration/meeka/sshd.nix
{ ... }:

{

services.openssh = {
    passwordAuthentication = false;
    allowSFTP = false; # Don't set this if you need sftp
    challengeResponseAuthentication = false;
    extraConfig = ''
        AllowTcpForwarding yes
        X11Forwarding no
        AllowAgentForwarding no
        AllowStreamLocalForwarding no
        AuthenticationMethods publickey
    '';
};

}
```

Mark All Partitions but `/nix/store` as `noexec`

This is the most paranoid of the ideas in this post. The idea is that if you lock down the package manager so random services can't install software and you also make it impossible for them to write and run executable files outside of `/nix/store`, it becomes very difficult to exploit kernel bugs to get root. Add this with the other systemd isolation features that disable access to device nodes and twiddly system flags and you have a defense in depth setup that will make an attacker's life hard. They will have to get code execution in your services to do any damage.

Keep in mind that doing this will likely break the heck out of Nix when it needs to build things. In my testing it's been fine, however I am not an expert in these things. Something else to keep in mind is that you should configure your services to be denied access to `/nix/persist` and instead only allow them access to individual paths in the bind mounts on `/`, just in case they do that to try and sneak an executable through. This will not stop them from making a shell script and running it with `bash ./foo.sh`, but it will make it annoying to run things like C executables, which is much more important in this case.

For this you can set the following NixOS options:

```
# hosts/meeka/noexec.nix
{ ... }:

{
  fileSystems."/".options = [ "noexec" ];
  fileSystems."/etc/nixos".options = [ "noexec" ];
  fileSystems."/srv".options = [ "noexec" ];
  fileSystems."/var/log".options = [ "noexec" ];
}
```

This will make `/nix/store` (or symlinks to files in `/nix/store`) the only binaries that are allowed to be executed. This is a rather extreme step, but it should fairly sufficiently prevent any attacker from getting very far with exploits written in languages like C (which also means that it prevents bitcoin miner bots from running).

PCI Compliance Tip

PCI Compliance requires you to have an antivirus program installed on every server. It doesn't say anything about the program *running*, but just it being installed is enough. Get one step closer to PCI compliance with this one neat trick:

```
# hosts/meeka/pci-compliance-pass.nix
{ pkgs, ... }:

{
  environment.systemPackages = with pkgs; [ clamav ];
}
```



<Mara> ...doesn't that defeat the spirit of the thing?



<Cadey> To be honest, if you get to the end of those post and have an "all yes config" of this setup, installing an antivirus program to satisfy requirements that were primarily written for windows servers is probably one of the easiest steps you can take.

All in all, this entire setup will let you get a rather paranoid configuration that will reject everything outside of the golden path of what you told the machines to do. It will take some work to get to here (as well as being willing to experiment with a few virtual machines to test this process a few times before feeling safe enough to put this into production), but the end result should be a decently secure setup.

Obligatory warning: don't put this directly into production unless you know what you are doing, or at least can claim you know what you are doing with enough certainty to make servers difficult to debug. Have a way to "break the glass" and go back to a less noexec setup if you need to, it will save your ass.



<Mara> Oh, also be sure to import all of those random `*.nix` files if you



want to use it in one cohesive system config. That may be a slight bit entirely essential. ^_^

Share

Facts and circumstances may have changed since publication. Please contact me before jumping to conclusions if something seems wrong or unclear.

Tags: paranoid, noexec

Copyright 2012-2024 Xe Iaso (Christine Dodrill). Any and all opinions listed here are my own and not representative of any of my employers, past, future, and/or present.

Like what you see? Donate on [Patreon](#) like [these awesome people!](#)

Served by xesite v4 (/nix/store/9rxy3y9y9ffwx6xdvb56pgciar4b5138-xesite_v4-20240128 /bin/xesite) with site version [d58e5083](#), source code available [here](#).

Rapid Introduction to Nix

The goal of this mini-tutorial is to introduce you to Nix the language, including flakes, as quickly as possible while also preparing the motivated learner to dive deeper into the whole Nix ecosystem[↗]. At the end of this introduction, you will be able to create a `flake.nix` that builds a package and provides a developer environment shell.



🔥 Purely functional

If you are already experienced in [purely functional programming](#), it is highly recommended to read [Nix - taming Unix with functional programming](#) to gain a foundational perspective into Nix being purely functional but in the context of *file system* (as opposed to values stored in memory).

[..] we can treat the file system in an operating system like memory in a running program, and equate package management to memory management

Pre-requisites

- **Install Nix:** Nix can be [installed on Linux and macOS](#). If you are using [NixOS](#), it already comes with Nix pre-installed.
- **Play with Nix:** Before writing Nix expressions, it is useful to get a feel for working with the `nix` command. See [First steps with Nix](#)

Attrset

To learn more

- [Official manual](#)
- [nix.dev on attrsets](#)

The [Nix programming language](#) provides a lot of general constructs. But at its most basic use, it makes heavy use of *nested hash maps* otherwise called an “attrset”. They are equivalent to [Map Text a](#) in Haskell. The following is a simple example of an attrset:

```
{  
  foo = {  
    bar = 1;  
  };  
}
```

We have an outer attrset with a single key `foo`, whose value is another attrset with a single key `bar` and a value of `1`.

repl

Nix expressions can be readily evaluated in the [Nix repl](#). To start the repl, run `nix repl`.

```
$ nix repl
Welcome to Nix 2.12.0. Type :? for help.

nix-repl>
```

You can then evaluate expressions:

```
nix-repl> 2+3
5

nix-repl> x = { foo = { bar = 1; }; }

nix-repl> x
{ foo = { ... }; }

nix-repl> x.foo
{ bar = 1; }

nix-repl> x.foo.bar
1

nix-repl>
```

Flakes

To learn more

- [Serokell Blog: Basic flake structure](#)

A Nix **flake** is defined in the `flake.nix` file, which denotes an attrset containing two keys `inputs` and `outputs`. *Outputs* can reference *inputs*. Thus, changing an *input* can change the *outputs*. The following is a simple example of a flake:

```
{  
    inputs = {};  
  
    outputs = inputs: {  
        foo = 42;  
    };  
}
```

This flake has zero `inputs`. `outputs` is a [function](#) that takes the (realised) inputs as an argument and returns the final output attrset. This output attrset, in our example, has a single key `foo` with a value of 42.

We can use the [`nix flake show`](#) command to see the output structure of a flake:

```
$ nix flake show  
path:/Users/srid/code/nixplay?lastModified=1675373998&narHash=sha256-if  
└─foo: unknown  
$
```

We can use [`nix eval`](#) to evaluate any output. For example,

```
$ nix eval .#foo  
42
```

Graph

A flake can refer to other flakes in its inputs. Phrased differently, a flake's outputs can be used as inputs in other flakes. The most common example is the `nixpkgs` flake which gets used as an input in most flakes. Intuitively, you may visualize a flake to be a node in a larger graph, with inputs being the incoming arrows and outputs being the outgoing arrows.

Inputs

To learn more

- [URL-like syntax](#) used by the `url` attribute

Let's do something more interesting with our `flake.nix` by adding the `nixpkgs` input:

```
{  
  inputs = {  
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";  
  };  
  
  outputs = inputs: {  
    # Note: If you are macOS, substitute `x86_64-linux` with `aarch64-d  
    foo = inputs.nixpkgs.legacyPackages.x86_64-linux.cowsay;  
  };  
}
```

About `nixpkgs-unstable`

The `nixpkgs-unstable` branch is frequently updated, hence its name, but this doesn't imply instability or unsuitability for use.

The `nixpkgs` flake has an output called `legacyPackages`, which is indexed by the platform (called “system” in Nix-speak), further containing all the packages for that system. We assign that package to our flake output key `foo`.

You can use `nix repl` to explore the outputs of any flake, using TAB completion:

```
$ nix repl --extra-experimental-features 'flakes repl-flake' github  
Welcome to Nix 2.12.0. Type :? for help.  
  
Loading installable 'github:nixos/nixpkgs/nixpkgs-unstable#'...  
Added 5 variables.  
nix-repl> legacyPackages.aarch64-darwin.cowsay  
«derivation /nix/store/0s2vdpkpdiljmh8y06xgdw5vg2cqfs0m-cowsay-3.7.  
  
nix-repl>
```

Predefined outputs

Nix commands treat `certain outputs as special`. These are:

Output	Nix command	Description
packages	<code>nix build</code>	Derivation output
devShells	<code>nix develop</code>	Development shells
apps	<code>nix run</code>	Runnable applications
checks	<code>nix flake check</code>	Tests and checks

All of these predefined outputs are further indexed by the “system” value.

Packages

To learn more

- `pkgs.stdenv.mkDerivation` ↗ can be used to build a custom package from scratch

`packages` is the most often used output. Let us extend our previous `flake.nix` to use it:

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
  };

  outputs = inputs: {
    foo = 42;
    packages.x86_64-linux = {
      cowsay = inputs.nixpkgs.legacyPackages.x86_64-linux.cowsay;
    };
  };
}
```

Here, we are producing an output named `packages` that is an attrset of systems (currently, only `x86_64-linux`) to attrsets of packages. We are defining exactly one package, `cowsay`, for the `x86_64-linux` system.

```
$ nix flake show
path:/Users/srid/code/nixplay?lastModified=1675374260&narHash=sha256-FF
```

```

├── foo: unknown
└── packages
    └── aarch64-darwin
        └── cowsay: package 'cowsay-3.7.0'

```

Notice that `nix flake show` recognizes the type of packages. With `foo`, it couldn't (hence type is `unknown`) but with `packages`, it can (hence type is "package").

The `packages` output is recognized by `nix build`.

```
$ nix build .#cowsay
```

The `nix build` command takes as argument a value of the form `<flake-url>#<package-name>`. By default, `.` (which is a [flake URL](#)) refers to the current flake. Thus, `nix build .#cowsay` will build the `cowsay` package from the current flake under the current system. `nix build` produces a `./result` symlink that points to the Nix store path containing the package:

```
$ ./result/bin/cowsay hello
```

```

<-- hello -->
-----
 \  ^__^
  \  (oo)\_____
    (__)\       )\/\
        ||----w |
        ||     ||
```

If you run `nix build` without arguments, it will default to `.#default`.

Apps

A flake app is similar to a flake package except it always refers to a runnable program. You can expose the `cowsay` executable from the `cowsay` package as the default flake app:

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
  };
}
```

```
outputs = inputs: {
  apps.x86_64-linux = {
    default = {
      type= "app";
      program = "${inputs.nixpkgs.legacyPackages.x86_64-linux.cowsay}"
    } ;
  };
};

}
```

Now, you can run `nix run` to run the cowsay app, which is equivalent to doing `nix build .#cowsay && ./result/bin/cowsay` in the previous flake.

Interlude: demo

DevShells

To learn more

- [Official Nix manual](#)
- [NixOS Wiki](#)

Like `packages`, another predefined flake output is `devShells` - which is used to provide a `development` shell aka. a nix `shell` or devshell. A devshell is a sandboxed environment containing the packages and other shell environment you specify. `nixpkgs` provides a function called `mkShell` that can be used to create devshells.

As an example, we will update our `flake.nix` to provide a devshell that contains the `jq` tool.

```
{
  inputs = {
    nixpkgs = {
      url = "github:NixOS/nixpkgs/nixos-unstable";
    };
  };
  outputs = inputs: {
    foo = 42;
    devShells = { # nix develop
      aarch64-darwin = {
        default =
          let
            pkgs = inputs.nixpkgs.legacyPackages.aarch64-darwin;
          in pkgs.mkShell {
            packages = [
              pkgs.jq
            ];
          };
      };
    };
  };
}
```

`nix flake show` will recognize this output as a “development environment”:

```
$ nix flake show
path:/Users/srid/code/nixplay?lastModified=1675448105&narHash=sha256-di
├─devShells
│ └─aarch64-darwin
│    └─default: development environment 'nix-shell'
└─foo: unknown
```

Just as `packages` can be built using `nix build`, you can enter the devshell using `nix develop` ↗:

```
$ nix develop
> which jq
/nix/store/33n0kx526i5dnv2gf39qv1p3a046p9yd-jq-1.6-bin/bin/jq
> echo '{"foo": 42}' | jq .foo
42
>
```

Typing `Ctrl+D` or `exit` will exit the devshell.

Conclusion

This mini tutorial provided a rapid introduction to Nix flakes, enabling you to get started with writing simple flake for your projects. Consult the links above for more information. There is a lot more to Nix than the concepts presented here! You can also read [Zero to Nix](#) ↗ for a highlevel introduction to all things Nix and flakes.

See also

- A (more or less) one page introduction to Nix, the language ↗
- [Nix - taming Unix with functional programming](#) ↗



Links to this page

[Nixifying a Haskell project using nixpkgs](#)

A basic understanding of the Nix and Flakes is assumed. See Rapid Introduction to



tazjin / nix-1p

Code Issues 7 Pull requests Actions Projects Security Insights

A (more or less) one page introduction to Nix, the language.

[code.tvl.fyi/about/nix/nix-1p](#)

732 stars 22 forks 18 watching 1 Branch 0 Tags Activity

Public repository

master 1 Branch 0 Tags Go to file Go to file Add file Code ...

tazjin and clbot docs(nix-1p): clarify wording for what is an expression ... yesterday

README.md docs(nix-1p): clarify wording for what i... yesterday

default.nix chore(nixery): use nix-1p from within t... 2 years ago

README

Nix - A One Pager

[Nix](#), the package manager, is built on and with Nix, the language. This page serves as a fast intro to most of the (small) language.

Unless otherwise specified, the word "Nix" refers only to the language below.

Please file an issue if something in here confuses you or you think something important is missing.

If you have Nix installed, you can try the examples below by running `nix repl` and entering code snippets there.

Table of Contents

- [Overview](#)
- [Language constructs](#)
 - [Primitives / literals](#)
 - [Operators](#)
 - [// \(merge\) operator](#)
 - [Variable bindings](#)
 - [Functions](#)
 - [Multiple arguments \(currying\)](#)
 - [Multiple arguments \(attribute sets\)](#)
 - [if ... then ... else ...](#)
 - [inherit keyword](#)
 - [with statements](#)
 - [import / NIX_PATH / <entry>](#)
 - [or expressions](#)
- [Standard libraries](#)
 - [builtins](#)
 - [pkgs.lib](#)
 - [pkgs itself](#)
- [Derivations](#)
- [Nix Idioms](#)
 - [File lambdas](#)
 - [callPackage](#)
 - [Overrides / Overlays](#)

Overview

View view

Nix is:

- **purely functional.** It has no concept of sequential steps being executed, any dependency between operations is established by depending on *data* from previous operations.

Any valid piece of Nix code is an *expression* that returns a value.

Evaluating a Nix expression *yields a single data structure*, it does not execute a sequence of operations.

Every Nix file evaluates to a *single expression*.

- **lazy.** It will only evaluate expressions when their result is actually requested.

For example, the builtin function `throw` causes evaluation to stop. Entering the following expression works fine however, because we never actually ask for the part of the structure that causes the `throw`.

```
let attrs = { a = 15; b = builtins.throw "Oh no!"; };
in "The value of 'a' is ${toString attrs.a}"
```



- **purpose-built.** Nix only exists to be the language for Nix, the package manager. While people have occasionally used it for other use-cases, it is explicitly not a general-purpose language.

Language constructs

This section describes the language constructs in Nix. It is a small language and most of these should be self-explanatory.

Primitives / literals

Nix has a handful of data types which can be represented literally in source code, similar to many other languages.

```
# numbers
42
1.72394

# strings & paths
"hello"
./some-file.json

# strings support interpolation
"Hello ${name}"

# multi-line strings (common prefix whitespace is dropped)
```
first line
second line
```

# lists (note: no commas!)
[ 1 2 3 ]

# attribute sets (field access with dot syntax)
{ a = 15; b = "something else"; }

# recursive attribute sets (fields can reference each other)
rec { a = 15; b = a * 2; }
```



Operators

Nix has several operators, most of which are unsurprising:

Syntax	Description
<code>+ , - , * , /</code>	Numerical operations
<code>+</code>	String concatenation
<code>++</code>	List concatenation
<code>==</code>	Equality
<code>> , >= , < , <=</code>	Ordering comparators
<code>&&</code>	Logical AND
<code> </code>	Logical OR
<code>e1 -> e2</code>	Logical implication (i.e. <code>!e1 e2</code>)
<code>!</code>	Boolean negation
<code>set.attr</code>	Access attribute <code>attr</code> in attribute set <code>set</code>
<code>set ? attribute</code>	Test whether attribute set contains an attribute
<code>left // right</code>	Merge <code>left</code> & <code>right</code> attribute sets, with the right set taking precedence

// (merge) operator

The `//`-operator is used pervasively in Nix code. You should familiarise yourself with it, as it is likely also the least familiar one.

It merges the left and right attribute sets given to it:

```
{ a = 1; } // { b = 2; }

# yields { a = 1; b = 2; }
```



Values from the right side take precedence:

```
{ a = "left"; } // { a = "right"; }

# yields { a = "right"; }
```



The merge operator does *not* recursively merge attribute sets;

```
{ a = { b = 1; }; } // { a = { c = 2; }; }

# yields { a = { c = 2; }; }
```



Helper functions for recursive merging exist in the [lib library](#).

Variable bindings

Bindings in Nix are introduced locally via `let` expressions, which make some variables available within a given scope.

For example:

```
let
  a = 15;
  b = 2;
in a * b

# yields 30
```



Variables are immutable. This means that after defining what `a` or `b` are, you can not *modify* their value in the scope in which they are available.

You can nest `let`-expressions to shadow variables.

Variables are *not* available outside of the scope of the `let` expression. There are no global variables.

Functions

All functions in Nix are anonymous lambdas. This means that they are treated just like data. Giving them names is accomplished by assigning them to variables, or setting them as values in an attribute set (more on that below).

```
# simple function
# declaration is simply the argument followed by a colon
name: "Hello ${name}"
```



Multiple arguments (currying)

Technically any Nix function can only accept **one argument**. Sometimes however, a function needs multiple arguments. This is achieved in Nix via [currying](#), which means to create a function with one argument, that returns a function with another argument, that returns ... and so on.

For example:

```
name: age: "${name} is ${toString age} years old"
```



An additional benefit of this approach is that you can pass one parameter to a curried function, and receive back a function that you can re-use (similar to partial application):

```
let
  multiply = a: b: a * b;
  doubleIt = multiply 2; # at this point we have passed in the value for 'a' and
                        # receive back another function that still expects 'b'
in
  doubleIt 15

# yields 30
```



Multiple arguments (attribute sets)

Another way of specifying multiple arguments to a function in Nix is to make it accept an attribute set, which enables multiple other features.

enables multiple other features.

```
{ name, age }: "${name} is ${toString age} years old"
```



Using this method, we gain the ability to specify default arguments (so that callers can omit them):

```
{ name, age ? 42 }: "${name} is ${toString age} years old"
```



Or in practice:

```
let greeter = { name, age ? 42 }: "${name} is ${toString age} years old";
in greeter { name = "Slartibartfast"; }

# yields "Slartibartfast is 42 years old"
# (note: Slartibartfast is actually /significantly/ older)
```



Additionally we can introduce an ellipsis using `...`, meaning that we can accept an attribute set as our input that contains more variables than are needed for the function.

```
let greeter = { name, age, ... }: "${name} is ${toString age} years old";
person = {
  name = "Slartibartfast";
  age = 42;
  # the 'email' attribute is not expected by the 'greeter' function ...
  email = "slartibartfast@magrath.ea";
};
in greeter person # ... but the call works due to the ellipsis.
```



Nix also supports binding the whole set of passed in attributes to a parameter using the `@` syntax:

```
let func = { name, age, ... }@args: builtins.attrNames args;
in func {
  name = "Slartibartfast";
  age = 42;
  email = "slartibartfast@magrath.ea";
}

# yields: [ "age" "email" "name" ]
```



Warning: Combining the `@` syntax with default arguments can lead to surprising behaviour, as the passed attributes are bound verbatim. This means that defaulted arguments are not included in the bound attribute set:

```
({ a ? 1, b }@args: args.a) { b = 1; }
# throws: error: attribute 'a' missing

({ a ? 1, b }@args: args.a) { b = 1; a = 2; }
# => 2
```



if ... then ... else ...

Nix has simple conditional support. Note that `if` is an **expression** in Nix, which means that both branches must be specified.

```
if someCondition
```



```
then "it was true"  
else "it was false"
```

inherit keyword

The `inherit` keyword is used in attribute sets or `let` bindings to "inherit" variables from the parent scope.

In short, a statement like `inherit foo;` expands to `foo = foo; .`

Consider this example:

```
let  
  name = "Slartibartfast";  
  # ... other variables  
in {  
  name = name; # set the attribute set key 'name' to the value of the 'name' var  
  # ... other attributes  
}
```



The `name = name;` line can be replaced with `inherit name; :`

```
let  
  name = "Slartibartfast";  
  # ... other variables  
in {  
  inherit name;  
  # ... other attributes  
}
```



This is often convenient, especially because `inherit` supports multiple variables at the same time as well as "inheritance" from other attribute sets:

```
{  
  inherit name age; # equivalent to `name = name; age = age;`  
  inherit (otherAttrs) email; # equivalent to `email = otherAttrs.email`;  
}
```



with statements

The `with` statement "imports" all attributes from an attribute set into variables of the same name:

```
let attrs = { a = 15; b = 2; };  
in with attrs; a + b # 'a' and 'b' become variables in the scope following 'with'
```



The scope of a `with` -"block" is the expression immediately following the semicolon, i.e.:

```
let attrs = { /* some attributes */ };  
in with attrs; /* this is the scope of the `with` */
```



import / NIX_PATH / <entry>

Nix files can import each other by using the builtin `import` function and a literal path:

```
# assuming there is a file lib.nix with some useful functions
let myLib = import ./lib.nix;
in myLib.usefulFunction 42
```



The `import` function will read and evaluate the file, and return its Nix value.

Nix files often begin with a function header to pass parameters into the rest of the file, so you will often see imports of the form `import ./some-file { ... } .`

Nix has a concept of a `NIX_PATH` (similar to the standard `PATH` environment variable) which contains named aliases for file paths containing Nix expressions.

In a standard Nix installation, several [channels](#) will be present (for example `nixpkgs` or `nixos-unstable`) on the `NIX_PATH`.

`NIX_PATH` entries can be accessed using the `<entry>` syntax, which simply evaluates to their file path:

```
<nixpkgs>
# might yield something like `/home/tazjin/.nix-defexpr/channels/nixpkgs`
```



This is commonly used to import from channels:

```
let pkgs = import <nixpkgs> {};
in pkgs.something
```



or expressions

Nix has a keyword called `or` which can be used to access a value from an attribute set while providing a fallback to a default value.

The syntax is simple:

```
# Access an existing attribute
let set = { a = 42; };
in set.a or 23
```



Since the attribute `a` exists, this will return `42`.

```
# ... or fall back to a default if there is no such key
let set = { };
in set.a or 23
```



Since the attribute `a` does not exist, this will fall back to returning the default value `23`.

Note that `or` expressions also work for nested attribute set access.

Standard libraries

Yes, libraries, plural.

Nix has three major things that could be considered its standard library and while there's a lot of debate to be had about this point, you still need to know all three

about this point, you still need to know all three.

builtins

Nix comes with several functions that are baked into the language. These work regardless of which other Nix code you may or may not have imported.

Most of these functions are implemented in the Nix interpreter itself, which means that they are rather fast when compared to some of the equivalents which are implemented in Nix itself.

The Nix manual has [a section listing all builtins](#) and their usage.

Examples of builtins that you will commonly encounter include, but are not limited to:

- `derivation` (see [Derivations](#))
- `toJSON` / `fromJSON`
- `toString`
- `toPath` / `fromPath`

The builtins also include several functions that have the (spooky) ability to break Nix' evaluation purity. No functions written in Nix itself can do this.

Examples of those include:

- `fetchGit` which can fetch a git-repository using the environment's default git/ssh configuration
- `fetchTarball` which can fetch & extract archives without having to specify hashes

Read through the manual linked above to get the full overview.

pkgs.lib

The Nix package set, commonly referred to by Nixers simply as [nixpkgs](#), contains a child attribute set called `lib` which provides a large number of useful functions.

The canonical definition of these functions is [their source code](#). I wrote a tool ([nixdoc](#)) in 2018 which generates manual entries for these functions, however not all of the files are included as of July 2019.

See the [Nixpkgs manual entry on lib](#) for the documentation.

These functions include various utilities for dealing with the data types in Nix (lists, attribute sets, strings etc.) and it is useful to at least skim through them to familiarise yourself with what is available.

```
{ pkgs ? import <nixpkgs> {} }:

  with pkgs.lib; # bring contents pkgs.lib into scope

  strings.toUpperCase "hello"

  # yields "HELLO"
```



pkgs itself

The Nix package set itself does not just contain packages, but also many useful functions which you might run into while creating new Nix packages.

One particular subset of these that stands out are the [trivial builders](#), which provide utilities for writing text files or

shell scripts, running shell commands and capturing their output and so on.

```
{ pkgs ? import <nixpkgs> {} }:  
  
pkgs.writeText "hello.txt" "Hello dear reader!"  
  
# yields a derivation which creates a text file with the above content
```



Derivations

When a Nix expression is evaluated it may yield one or more *derivations*. Derivations describe a single build action that, when run, places one or more outputs (whether they be files or folders) in the Nix store.

The builtin function `derivation` is responsible for creating derivations at a lower level. Usually when Nix users create derivations they will use the higher-level functions such as [stdenv.mkDerivation](#).

Please see the manual [on derivations](#) for more information, as the general build logic is out of scope for this document.

Nix Idioms

There are several idioms in Nix which are not technically part of the language specification, but will commonly be encountered in the wild.

This section is an (incomplete) list of them.

File lambdas

It is customary to start every file with a function header that receives the files dependencies, instead of importing them directly in the file.

Sticking to this pattern lets users of your code easily change out, for example, the specific version of `nixpkgs` that is used.

A common file header pattern would look like this:

```
{ pkgs ? import <nixpkgs> {} }:  
  
# ... 'pkgs' is then used in the code
```



In some sense, you might consider the function header of a file to be its "API".

callPackage

Building on the previous pattern, there is a custom in `nixpkgs` of specifying the dependencies of your file explicitly instead of accepting the entire package set.

For example, a file containing build instructions for a tool that needs the standard build environment and `libsvg` might start like this:

```
# my-funky-program.nix  
{ stdenv, libsvg }:
```



```
stdenv.mkDerivation { ... }
```

Any time a file follows this header pattern it is probably meant to be imported using a special function called `callPackage` which is part of the top-level package set (as well as certain subsets, such as `haskellPackages`).

```
{ pkgs ? import <nixpkgs> {} }:  
  
let my-funky-program = pkgs.callPackage ./my-funky-program.nix {};  
in # ... something happens with my-funky-program
```



The `callPackage` function looks at the expected arguments (via `builtins.functionArgs`) and passes the appropriate keys from the set in which it is defined as the values for each corresponding argument.

Overrides / Overlays

One of the most powerful features of Nix is that the representation of all build instructions as data means that they can easily be *overridden* to get a different result.

For example, assuming there is a package `someProgram` which is built without our favourite configuration flag (`--mimic-threaten-tag`) we might override it like this:

```
someProgram.overrideAttrs(old: {  
    configureFlags = old.configureFlags or [] ++ ["--mimic-threaten-tag"];  
})
```



This pattern has a variety of applications of varying complexity. The top-level package set itself can have an `overlays` argument passed to it which may add new packages to the imported set.

Note the use of the `or` operator to default to an empty list if the original flags do not include `configureFlags`. This is required in case a package does not set any flags by itself.

Since this can change in a package over time, it is useful to guard against it using `or`.

For a slightly more advanced example, assume that we want to import `<nixpkgs>` but have the modification above be reflected in the imported package set:

```
let  
  overlay = (final: prev: {  
    someProgram = prev.someProgram.overrideAttrs(old: {  
      configureFlags = old.configureFlags or [] ++ ["--mimic-threaten-tag"];  
    });  
  });  
in import <nixpkgs> { overlays = [ overlay ]; }
```



The `overlay` function receives two arguments, `final` and `prev`. `final` is the [fixed point](#) of the overlay's evaluation, i.e. the package set *including* the new packages and `prev` is the "original" package set.

See the Nix manual sections [on overrides](#) and [on overlays](#) for more details (note: the convention has moved away from using `self` in favor of `final`, and `prev` instead of `super`, but the documentation has not been updated to reflect this).

No packages published

[Releases](#)

No releases published

[Contributors](#) 5



Languages

- [Nix](#) 100.0%



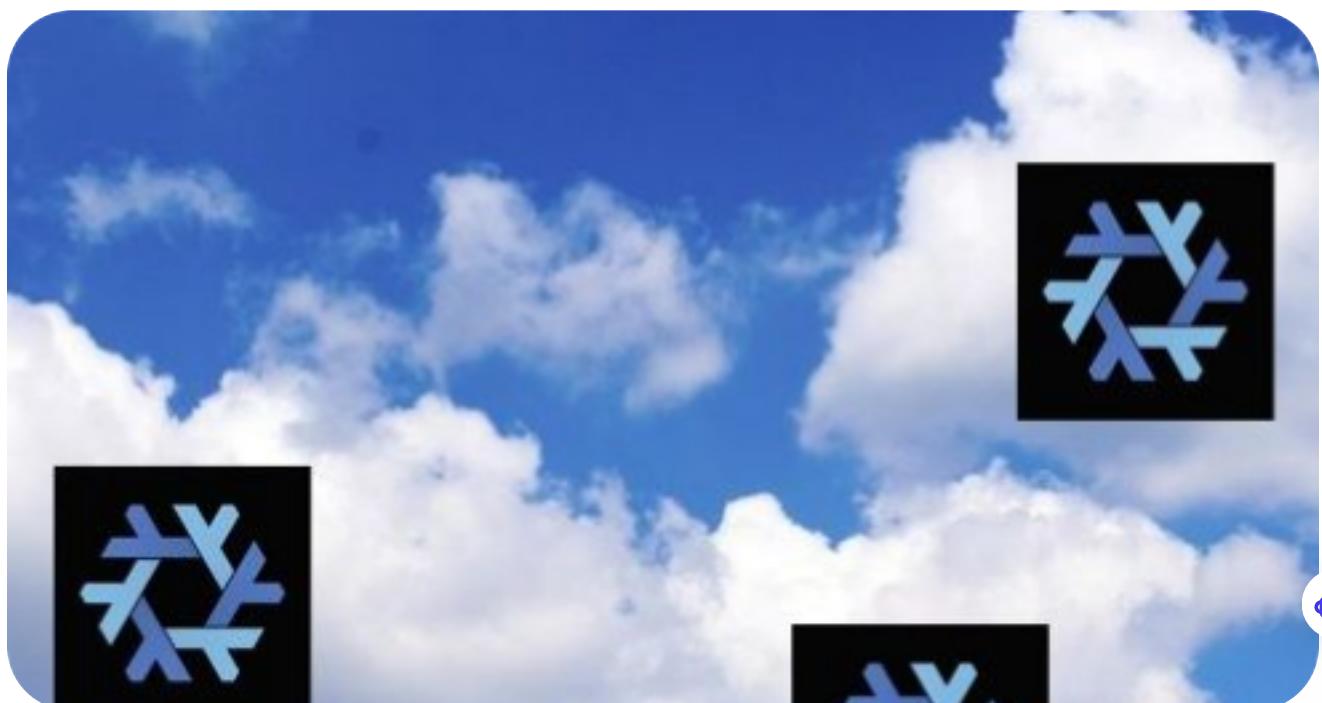
GET IN TOUCH

We don't need NixOS cloud images anymore

A simple free tool lets you install NixOS remotely in the cloud - or anywhere else.



Jul 6, 2023 - 7 min read



We don't need NixOS cloud images? There's an easy workaround? Your reaction to this may be: Yay! Tell me more - now! Or it may be more like this: Huh? Why do we need NixOS cloud images anyway?

Let's deal with the second reaction first.

What are cloud images and why do we need them?

When you create a server in the cloud, the cloud provider lets you select from their available images to get the operating system you want. Most of them have a list of the most common operating systems, and you can choose. Google Cloud, for example, includes options like Debian, Centos, Ubuntu and Windows Server.

Very few providers include NixOS. For NixOS lovers, this is a monumental disaster.

The good news

Ok, now we're all on the same page. Let me share the good news: we no longer need NixOS cloud images, because with a simple free tool, you can install NixOS painlessly anywhere you like! The tool is, appropriately, named `nixos-anywhere`, and you can use it directly from [Numtide's Github repo](#).

In this article, I'll take a look at this tool, and share a worked example showing how I used it to install NixOS on Vultr.

Nixos-anywhere is the brainchild of Numtide members [Jörg](#) and [Lassulus](#), with a bit of help from other collaborators.

To use it, all you have to do is:

- Create your server in the cloud with a standard Linux image, for example Ubuntu
- Set up an SSH connection
- Create your configurations for the new server on your local machine
- Issue a single CLI command on your local computer
- Sit back, relax with a cup of coffee, and everything gets done for you.





Even better news

That's the good news. Now let me share the excellent news. With `nixos-anywhere`, you can preconfigure your new server, and create as many identical servers as you like in the cloud or anywhere else. If you need a mixture of local servers and servers on various different cloud providers - no problem. Just tweak the configurations a bit, then create each new server with a single CLI command.

And there's no need to babysit your installations. Everything gets done automatically. If you get it right once, it will work in exactly the same way every time. Just like Nix, in fact.

So how does it work? First, `nixos-anywhere` connects to the remote machine via SSH. Next, it uses `kexec` to boot into a NixOS installer image. It then uses another tool, `disko`, to partition your disks, create and format file systems, and mount them. Then it uses the configurations you defined to build and install the NixOS system before rebooting. And that's it. You now have a working NixOS system in the cloud.

Let's look at how I used `nixos-anywhere` to install NixOS on Vultr, a popular cloud provider. This same process will also work with most other cloud providers. With a bit of tweaking, it works on most bare metal servers as well.

I'll be working from my local machine, which runs Ubuntu and has Nix installed.

Installation Example

Step 1: Create the server.

I created a small cloud server on Vultr. Because `nixos-anywhere` needs to `kexec` into a NixOS image, the new server needs a minimum of 2 GB of RAM to be able to store the image. This is the server I chose for the test.





I picked the latest version of Ubuntu as my operating system. Any 64-bit Linux with kexec support would have worked just as well. I included my SSH key when creating the server. This is important, as nixos-anywhere needs to access the new server this way.

SSH Keys ([Manage](#))



Step 2: Check the existing disk set-up

I logged into the new Vultr server, and had a look at what it had in the way of disks using lsblk.

```
root@jill:~# lsblk
NAME   MAJ:MIN RM    SIZE RO TYPE MOUNTPOINTS
loop0    7:0      0    73M  1 loop /snap/core22/607
loop1    7:1      0  73.8M  1 loop /snap/core22/750
loop2    7:2      0 163M  1 loop /snap/lxd/24643
loop3    7:3      0  53.3M  1 loop /snap/snapd/19457
loop4    7:4      0 170.1M  1 loop /snap/lxd/24918
loop5    7:5      0  53.3M  1 loop /snap/snapd/19361
sr0     11:0     1 1024M  0 rom
vda     252:0     0    80G  0 disk
```



```
└vda1 252:1      0     80G  0 part /
```

Ok, so it has one disk named vda with a single partition.

Step 3: Configure disko

Since nixos-anywhere uses the [disko](#) tool to partition and format the disks, the next task is to configure disko. Luckily, disko has lots of sample configurations catering for most of the common disk partitioning and formatting options. For this exercise, I chose [this one](#). You may like to take a moment to have a look at this file.

It's nice and simple. It will create a standard GPT partition that works with both EFI and BIOS systems. The only thing that I need to check is that the name of the disk matches the disk name I noted in Step 2. The disk on Vultr is named /dev/vda, and yes, it does match, so I don't need to change anything.

On my local machine, I created a directory called VultrTest and saved this file in there as disk-config.nix.

Step 4: Create a flake

In its simplest form, nixos-anywhere uses flakes. If you're not familiar with flakes, this is a great time to try them out. I'd recommend reading [the flake documentation](#). If you haven't used flakes before, you'll need to enable them. The document in the previous link tells you how to do this.

So in this step, in the directory VultrTest, I created a flake using `nix flake init`, and edited it to suit my requirements. You can see the edited flake [here](#).

Explaining the syntax of the flake is beyond the scope of this article, but I'll just mention a few points of interest.

- The output configuration is given a name in the following entry. This names the configuration as `vultr`.

```
nixosConfigurations.vultr = nixpkgs.lib.nixosSystem
```

- Under `modules`, the following entry specifies that a configuration named `vconf.nix`



under the `configs` directory should be included.

```
./configs/vconf.nix
```

Step 5: Create NixOS configurations for the new server

In a new directory named `configs` under `VultrTest`, I created the file named `vconf.nix`, as referred to in the flake. In here, I put some configurations that will be applied to the new server when it's built. I enabled flakes and added a few packages from `nixpkgs`. Most importantly, I included my SSH key so I can still log into the new server after the installation. You can include any NixOS configurations in this file. You can see my configurations [here](#).

Step 6: Create `flake.lock`

Almost ready to go. The last thing that needed doing before I could do the installation was to create the `flake.lock` file. To do this, I ran the following Nix command in the `VultrTest` directory:

```
nix flake lock
```

I now had everything I needed for the installation. At this point I copied `VultrTest` and its contents to Github for safekeeping.

Step 7: Do the installation

I could now run `nixos-anywhere`. From the `VultrTest` directory, I used the following Nix command to do the installation:

```
nix run github:numtide/nixos-anywhere --flake ./#vultr root@78.141.223.17
```



Note that `#vultr` matches the name I gave to the configuration in Step 4.

The command ran for perhaps 15 minutes, displaying information about each step as it progressed. Finally, it terminated with no error messages, and the notation:

Done!

And that's it! NixOS had been successfully installed on the new server. To check this, I needed to remotely log in to it. Since the old operating system had been overridden, I first had to delete any entries for it in `.ssh/known_hosts`, because the keys would have changed. I then used SSH to log in to the new server to check that NixOS was running, and the packages I'd specified in `vconf.nix` had been installed.

Maintaining the configuration on the new server

It's worth noting that `nixos-anywhere` doesn't create a `configuration.nix` on the new server. This is because the configurations are held in the flake and any files that it references. If I needed to make changes to the configuration of this server later, I would make the necessary changes to the files I saved on GitHub, and reference this when running `nixos-rebuild`. My rebuild command would then be:

```
nixos-rebuild --flake github:JillThornhill/VultrTest#vultr switch
```

Storing the configuration remotely makes it really easy to rebuild the server if it ever crashes. It also means it's easy to create more identical servers. And lastly, storing it on GitHub gives you all the advantages of version control: you always know exactly what's been changed if anything's not working correctly.

Summary

In summary, `nixos-anywhere` provides a simple way to install NixOS where no cloud image is available. It also lets you create any number of identical servers, and get on with other important stuff (e.g. having a coffee break) while all the hard work is done for you. In fact, it's a system administrator's dream come true!





SHARE



Why Choose Nix for Package Management?

[PREVIOUS POST](#)

We are deep-stack developers.

[NEXT POST](#)

Sign up to our website

Subscribe now to get notified about new articles!

EMAIL ADDRESS

SUBSCRIBE

[Contact](#)

[About](#)

[Impressum](#)



[GET IN TOUCH](#)





≡ Q

GET IN TOUCH

Why Choose Nix for Package Management?

Time saved with Nix > time spent learning Nix: T or F?



Jun 27, 2023 - 8 min read



Why Nix?

I'm a Nix newbie. You should learn Nix, they told me. It's a really, really good option for package management under Linux and Mac. I'm not that easy to convince. Why should it be better than just using apt or dpkg? Or setting up a Docker environment? And as for using NixOS as an operating system, why would you want to do that?

But yes, I'm now convinced. Why?

In this article, I'll take you through some of the points that sold me on Nix and NixOS. Bear in mind that I'm not (yet) a Nix guru, so this won't be a highly technical article. You won't learn much about the innards of Nix — this is, essentially, an outsider's point of view.

But first, for those who don't already know - what exactly is Nix? And what is NixOS?

Nix and NixOS

What is Nix? And NixOS?

Nix is a package manager for Linux and Mac, but it uses a different approach to conventional tools like apt and dpkg. More about that later. NixOS, on the other hand is a Linux distribution, with features especially attractive to Nix users. At this point, let's just say that if you're sold on Nix, you'll want to go the whole way and try NixOS.

Why is Nix Different?

In Linux, packages are installed in a common `bin` directory. If you have more than one version of a piece of software — for example a live version and a test version — it becomes complicated. You have to be very careful when specifying dependencies and setting environment variables.

With Nix, each package is defined declaratively, complete with its dependencies, and is held in the Nix store (`/nix/store`). Each package definition, including a description of its dependencies, is described using the Nix configuration language. According to the official website, this makes all packages



- Reproducible
- Declarative

- Reliable

Other characteristics of Nix include:

- Packages are built and stored in isolation. Because of this, it's possible to use different versions of any software, without the danger of one version overwriting another.
- Nix retains information to allow you to easily roll back or roll forward between generations of a package. In NixOS, this also applies to the system as a whole.
- NixOS configurations are modular.

Sounds great – but does this really add up to enough advantages to make it worthwhile learning Nix? Nix has a notoriously steep learning curve (although a revision of the documentation is currently underway, and this should help solve the problem!) So, since time is money, Nix had better be good if it's worth my while to spend time learning it. But is it?

To find out, I spoke to some of the Numtide team to ask why they use it, and why they would recommend it to others. Let's have a look at the features I've mentioned, and what experienced Nix users have to say about them.

From the Numtide Team

Nix = reproducibility

“To me, the main advantage is reproducibility. I describe the system I would like to have: these technologies and these programs. I can then easily recreate the identical system anywhere, any time. If I have a system crash and need to reinstall, it's all there. I don't have to remember what I had and reconstruct it. I can also share this environment with another team member, knowing it will work in exactly the same way” — Aldo

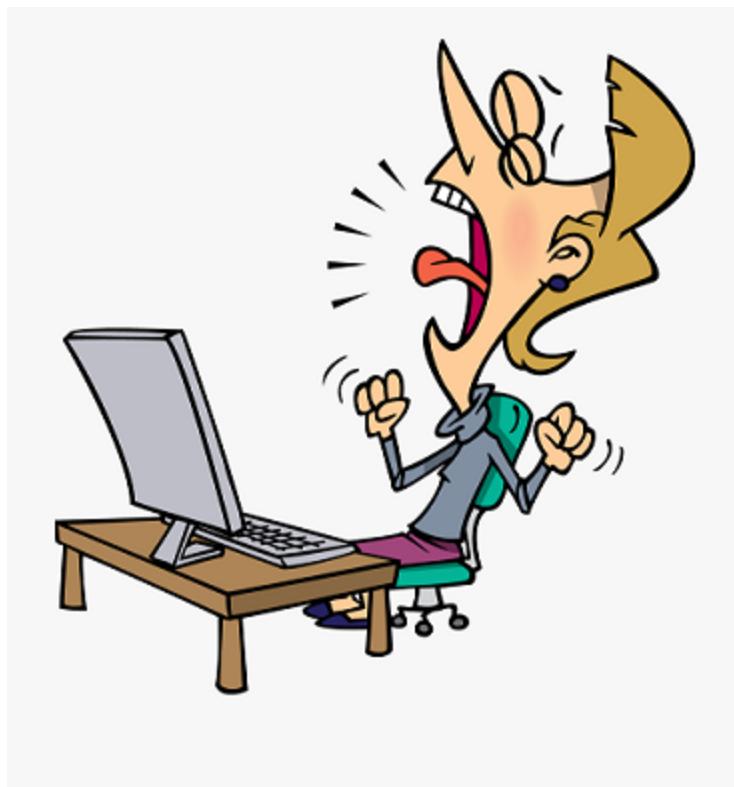
I think this is really, really cool. If you've spent as many hours as I have over the years trying to make a new machine work the same way as an old one, you'll love this!



“If I build a pipeline for CI/CD (Continuous Integration/Continuous Deployment) I know that if it works on my machine, it will also work on the remote. The final builds will be identical to the local build” — Antony

So no more scratching my head when my software doesn't work on the client's machine?

No more trying to track down an elusive dependency? With Nix, if I get it right at the beginning, it will work anywhere? I like that! I like it a lot.



“Outside of Nix, the current state-of-the-art tools used to manage infrastructure have limitations. I've always been afraid of pressing the button to activate a new deployment. With Nix, if it worked before, it will work again. Reproducibility is guaranteed” — David

“It's very easy to replicate an exact set-up. Reverting or upgrading then becomes very low-cost” — Dragan

Yeah, this could save me a lot of time and stress.

“You can describe the particular tool or developer environment once and have it available everywhere. You can be confident that if you come back even years later, it will work in the same way.



This also applies to machine configurations. You can be sure everyone is working on the same thing.

And if you're having problems figuring out why your build fails, it's easy for someone else to reproduce the issue and investigate it” — Florian

“For me, the main advantage is immutability. If you follow the same process, you will always have the same state. If you get it correct once, you can reproduce it anywhere. If you have different architecture, different conditions, different operating system, it doesn't matter. It will just work”. — Jeff

“When you're working with someone else, you can easily reproduce the exact environment. This often happens in the research field, where you take someone's idea and develop it further.” — Jorg

“I first got involved with Nix when my team grew, and we had issues with everyone having a different environment. With Nix, everyone's working on an identical environment.

When it works on test, I know it will work in production.

Recreating a system is easy. It's like magic. It's exactly the same system for all intents and purposes as what you had running before.” — Samuel

Nix is declarative

In Nix, everything is specified in the declarative Nix language. You configure exactly what's needed for each package. And with NixOS, this applies to the system as a whole. Why is this an advantage? Again, let's go to the Nix community for answers.

“With NixOS, I'm able to have a record of the configuration I used to build my machine. I don't have to remember each step if I need to rebuild, and I won't end up with a different state. I can also re-use most of the configuration when I need to build something different.

I can easily automate the deployment of both software and infrastructure.

In NixOS, you know you have exactly what you specified, and nothing else. You can easily investigate it, prune it and tune it until you're happy.” — Antony

“As an example: I bought a new laptop. It was blank and needed to be set up. The normal scenario would be:

- *Manually install an operating system*
- *Copy in a backup of the configuration files*
- *Spend several hours tweaking.*



With NixOS, I spent half an hour on my desktop tweaking existing Nix configurations, and pulling in standard configurations for the hardware. I applied this to the new machine, and within five minutes, the laptop looked and worked just like the desktop.

It shortcuts a lot of stuff. You express everything in a standard way, and there are modules for the vast majority of what you would want to install and how. If you're fluent in Nix, you can do this very quickly. More importantly, you can effortlessly reinstall things as they were before.

It's easy to adapt for a variety of different platforms, including virtual machines.” — Brian

“It's way more portable than any other deployment tool. An organisation seldom has a whole fleet of identical machines: they will all be different. With Nix, you can have a single repo that describes all the machines, and it all just works together.” — Jorg

“With Nix, you have an isolated environment. Instead of having to do lots of steps to set up an application, you can simply take a configuration file and use a single command.

With NixOS, you define the whole system in your configuration. You don't need to look at the machine itself to know what it's doing; it's all there in the configuration.” — Lassulus

“If you have everything in Nix, then all of your configuration is just one big data structure that describes everything.

This makes it a lot easier to understand.” — Vincent

All of that is pretty cool. You can pre-configure things, and easily tweak and reuse those configurations in different situations. And you always know exactly where to look if you're investigating how something is set up. But here's the really good part:

“Whether it's software or hardware configuration, Nix is declarative. Everything is defined in a configuration. This means that you can use version control software such as Github to store your configurations, and be able to track the history of any changes. If something no longer works, you can see exactly what was changed, when and by whom. It's then simple to revert to an earlier version while you debug the changes. This advantage scales up from a single laptop to an entire cluster of servers.” — Ramses

For me, this was the moment when I said, "Yes! Bring on Nix!" As a software developer and troubleshooter, I've spent far too many hours trying to find out who changed what when a system is broken. And because many organisations turn this situation into a



blame game and witch hunt, nobody will admit to having changed anything.



With Nix, it's all there in the git history, and you can quietly revert to something that works while you fix the problem. To me, that definitely makes it worth spending the time to learn and implement Nix.

Nix is reliable

"Nix is a new way to compose software together. The industry is building more and more software on top of other software, like a tower, and that tower can get wobbly! Nix is like the scaffolding: it's a solid structure that connects all the software together. You can build more layers of software reliably, without the danger of your tower falling over. Each structure has its own location, instead of throwing everything together in a central bin directory."— Jonas



How many times have you, or a colleague, complained that it all worked until you installed x, y or z? This won't happen with Nix. You can install whatever you like, and what you had before will keep right on working.

Nix packages are built and stored in isolation

In Nix, each package is stored in a predefined isolated environment. What does this mean in practice?

“You can have different people doing different stuff, and they don't collide. You can have people working on different versions and different applications, and they don't stand on each other's feet.” — Jorg

“I do enjoy the simplicity of setting up a development environment with Nix.

It allows separation between your development environment and production environment. I don't want to dirty up my production system with development tools. I used to use virtual machines or Docker, but those are quite resource hungry.” — Matjaz

Nix roll back / roll forward facility

Both Nix and NixOS keep several generations of configurations, and allow you to roll back or roll forward at any time.

“In Unix/Linux, there is no efficient way of monitoring the configuration state before a change was made, and you can't revert to what was there before.” — Antony

“Rolling back is easy.” — Samuel

NixOS configurations are modular

“Because it's modular, you can overwrite and merge any kind of service that you want, and it will all work together.” - Jorg

Is Learning Nix Worthwhile?

Whenever you consider learning a new tool, you need to evaluate whether the time the tool will save you in the future is more than the time you'll take to gain the necessary skills.

In my opinion, this equation is definitely correct!

```
future_time_saved_with_nix > time_taken_to_learn_nix = true;
```



- No more trying to figure out who changed what when a system breaks
- No more hours wasted in setting up the software I need on a new machine
- No more figuring out why the software works on these machines and not those.
- No more making sure a new member of the team has exactly the same software versions as everyone else

So I'll wrap up this article, because I'm off to up my Nix skills. Thanks for reading!

SHARE



Introducing NixOS Anywhere

[PREVIOUS POST](#)

We don't need NixOS cloud images anymore

[NEXT POST](#)

Sign up to our website

Subscribe now to get notified about new articles!

EMAIL ADDRESS

SUBSCRIBE





GET IN TOUCH



Wombat's Book of Nix

Amy de Buitléir



Table of Contents

- 1. Introduction
 - 1.1. Why Nix?
 - 1.2. Why *flakes*?
 - 1.3. Prerequisites
 - 1.4. See an error? Or want more?
- 2. The Nix language
 - 2.1. Introducing the Nix language
 - 2.2. Data types
 - 2.2.1. Strings
 - 2.2.2. Integers
 - 2.2.3. Floating point numbers
 - 2.2.4. Boolean
 - 2.2.5. Paths
 - 2.2.6. Lists
 - 2.2.7. Attribute sets
 - 2.2.8. Functions
 - 2.3. Stop reading this chapter!
 - 2.4. The Nix REPL
 - 2.5. Variables
 - 2.5.1. Assignment
 - 2.5.2. Immutability
 - 2.6. Numeric operations
 - 2.6.1. Arithmetic operators
 - 2.6.2. Floating-point calculations
 - 2.7. String operations
 - 2.7.1. String concatenation
 - 2.7.2. String interpolation
 - 2.7.3. Useful built-in functions for strings
 - 2.8. Boolean operations
 - 2.9. Path operations
 - 2.9.1. Concatenating paths
 - 2.9.2. Concatenating a path + a string
 - 2.9.3. Concatenating a string + a path
 - 2.9.4. Useful built-in functions for paths
 - 2.10. List operations
 - 2.10.1. List concatenation
 - 2.10.2. Useful built-in functions for lists
 - 2.11. Attribute set operations
 - 2.11.1. Selection
 - 2.11.2. Query
 - 2.11.3. Modification
 - 2.11.4. Recursive attribute sets
 - 2.11.5. Useful built-in functions for attribute sets
 - 2.12. Functions



- 2.12.1. Anonymous functions
- 2.12.2. Named functions and function application
- 2.12.3. Multiple parameters using nested functions
- 2.12.4. Multiple parameters using attribute sets
- 2.13. Argument sets
 - 2.13.1. Set patterns
 - 2.13.2. Optional parameters
 - 2.13.3. Variadic attributes
 - 2.13.4. @-patterns
- 2.14. If expressions
- 2.15. Let expressions
- 2.16. With expressions
- 3. Hello, flake!
- 4. The hello-flake repo
- 5. Flake structure
 - 5.1. Description
 - 5.2. Inputs
 - 5.3. Outputs
- 6. A generic flake
- 7. Another look at hello-flake
- 8. Modifying the flake
 - 8.1. The Nix development shell
 - 8.2. Introducing a dependency
 - 8.3. Development workflows
 - 8.4. This all seems like a hassle!
- 9. A new flake from scratch
 - 9.1. Haskell
 - 9.1.1. A simple Haskell program
 - 9.1.2. (Optional) Testing before packaging
 - Some unsuitable shells
 - A suitable shell for a quick test
 - 9.1.3. The cabal file
 - 9.1.4. (Optional) Building and running with cabal-install
 - 9.1.5. The Nix flake
 - 9.2. Python
 - 9.2.1. A simple Python program
 - 9.2.2. A Python builder
 - 9.2.3. The Nix flake
- 10. Recipes
 - 10.1. Access to a top-level package from the Nixpkgs/NixOS repo
 - 10.1.1. From the command line
 - 10.1.2. In `shell.nix`
 - 10.1.3. In a Bash script
 - 10.2. Access to a package defined in a remote git repo
 - 10.2.1. In `shell.nix`



10.3. Access to a flake defined in a remote git repo

 10.3.1. In `shell.nix`

10.4. Access to a Haskell library package in the `nixpkgs` repo (without a `.cabal` file)

 10.4.1. In `shell.nix`

 10.4.2. In a Haskell script

10.5. Access to a Haskell package on your local computer

 10.5.1. In `shell.nix`

10.6. Access to a Haskell package on your local computer, with inter-dependencies

 10.6.1. In `shell.nix`

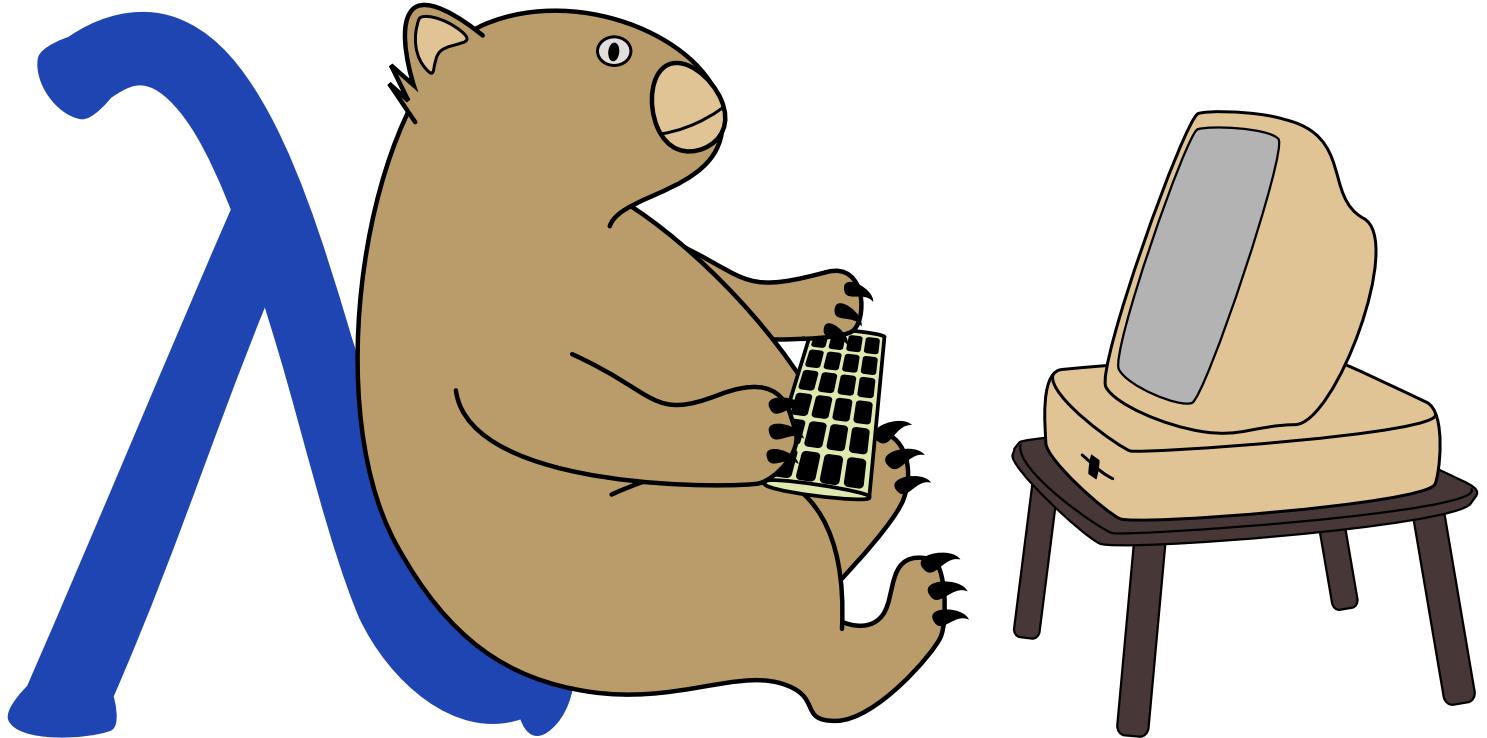
10.7. Access to a Python library package in the `nixpkgs` repo (without using a Python builder)

 10.7.1. In a Python script

10.8. Set an environment variable

 10.8.1. In `shell.nix`





This book is available [online](https://mhwombat.codeberg.page/nix-book/) (<https://mhwombat.codeberg.page/nix-book/>) and as a downloadable [PDF](#) (<https://codeberg.org/mhwombat/nix-book/raw/branch/pages/wombats-book-of-nix.pdf>).



1. Introduction

1.1. Why Nix?

If you've opened this PDF, you already have your own motivation for learning Nix. Here's how it helps me. As a researcher, I tend to work on a series of short-term projects, mostly demos and prototypes. For each one, I typically develop some software using a compiler, often with some open source libraries. Often I use other tools to analyse data or generate documentation, for example.

Problems would arise when handing off the project to colleagues; they would report errors when trying to build or run the project. Belatedly I would realise that my code relies on a library that they need to install. Or perhaps they had installed the library, but the version they're using is incompatible.

Using containers helped with the problem. However, I didn't want to *develop* in a container. I did all my development in my nice, familiar, environment with my custom aliases and shell prompt. and *then* I containerised the software. This added step was annoying for me, and if my colleague wanted to do some additional development, they would probably extract all of the source code from the container first anyway. Containers are great, but this isn't the ideal use case for them.

Nix allows me to work in my custom environment, but forces me to specify any dependencies. It automatically tracks the version of each dependency so that it can replicate the environment wherever and whenever it's needed.

1.2. Why *flakes*?

Flakes are labeled as an experimental feature, so it might seem safer to avoid them. However, they have been in use for years, and there is widespread adoption, so the aren't going away any time soon. Flakes are easier to understand, and offer more features than the traditional Nix approach. After weighing the pros and cons, I feel it's better to learn and use flakes; and this seems to be the general consensus.

1.3. Prerequisites

To follow along with the examples in this book, you will need access to a computer or (virtual machine) with Nix (or NixOS) installed and *flakes enabled*.

I recommend the installer from zero-to-nix.com (<https://zero-to-nix.com/start/install>). This installer automatically enables flakes.

More documentation (and another installer) available at nixos.org (<https://nixos.org/>).

To *enable flakes on an existing Nix or NixOS installation*, see the instructions in the [NixOS wiki](https://nixos.wiki/wiki/Flakes) (<https://nixos.wiki/wiki/Flakes>).

There are hyphenated and un-hyphenated versions of many Nix commands. For example, `nix-shell` and `nix shell` are two different commands. Don't confuse them!



Generally speaking, the un-hyphenated versions are for working directly with flakes, while the hyphenated versions are for everything else.

1.4. See an error? Or want more?

If notice an error, or you're interested in an area that isn't covered in this book, feel free to open an [issue](#) (<https://codeberg.org/mhwombat/nix-book/issues>).

2. The Nix language

2.1. Introducing the Nix language

Nix and NixOS use a functional programming language called *Nix* to specify how to build and install software, and how to configure system, user, and project-specific environments. (Yes, “Nix” is the name of both the package manager and the language it uses.)

Nix is a *functional* language. In a *procedural* language such as C or Java, the focus is on writing a series of *steps* (statements) to achieve a desired result. By contrast, in a functional language the focus is on *defining* the desired result.

In the case of Nix, the desired result is usually a *derivation*: a software package that has been built and made available for use. The Nix language has been designed for that purpose, and thus has some features you don’t typically find in general-purpose languages.

2.2. Data types

2.2.1. Strings

Strings are enclosed by double quotes ("), or *two* single quotes (').

```
"Hello, world!"
```

```
''This string contains "double quotes"'''
```

They can span multiple lines.

```
''Old pond
A frog jumps in
The sound of water
-- Basho'''
```

2.2.2. Integers

```
7
256
```

2.2.3. Floating point numbers

```
3.14
6.022e23
```



2.2.4. Boolean

The Boolean values in Nix are `true` and `false`.

2.2.5. Paths

File paths play an important role in building software, so Nix has a special data type for them. Paths may be absolute (e.g. `/bin/sh`) or relative (e.g. `./data/file1.csv`). Note that paths are not enclosed in quotation marks; they are not strings!

Enclosing a path in angle brackets, e.g. <nixpkgs> causes the directories listed in the environment variable NIX_PATH to be searched for the given file or directory name. These are called *lookup paths*.

2.2.6. Lists

List elements are enclosed in square brackets and separated by spaces (not commas). The elements need not be of the same type.

```
[ "apple" 123 ./build.sh false ]
```

Lists can be empty.

```
[]
```

List elements can be of any type, and can even be lists themselves.

```
[ [ 1 2 ] [ 3 4 ] ]
```

2.2.7. Attribute sets

Attribute sets associate keys with values. They are enclosed in curly brackets, and the associations are terminated by semi-colons. Note that the final semi-colon before the closing bracket is required.

```
{ name = "Professor Paws"; age = 10; species = "cat"; }
```

The keys of an attribute set must be strings. When the key is not a valid identifier, it must be enclosed in quotation marks.

```
{ abc = true; "123" = false; }
```

Attribute sets can be empty.

```
{}
```

Values of attribute sets can be of any type, and can even be attribute sets themselves.

```
{ name = { first = "Professor"; last = "Paws"; }; age = 10; species = "cat"; }
```

In Section 2.11.4, “Recursive attribute sets” you will be introduced to a special type of attribute set.



In some Nix documentation, and in many articles about Nix, attribute sets are simply called “sets”.

2.2.8. Functions

We'll learn how to write functions later in this chapter. For now, note that functions are “first-class values”, meaning that they can be treated like any other data type. For example, a function can be assigned to a variable, appear as an element in a list, or be associated with a key in an attribute set.

```
[ "apple" 123 ./build.sh false (x: x*x) ]  
{ name = "Professor Paws"; age = 10; species = "cat"; formula = (x: x*2); }
```

2.3. Stop reading this chapter!

When I first began using Nix, it seemed logical to start by learning the Nix language. However, after following an in-depth tutorial, I found that I didn't know how to do anything useful with the language! It wasn't until later that I understood what I was missing: a guide to the most useful library functions.

When working with Nix or NixOS, it's very rare that you'll want to write something from scratch. Instead, you'll use one of the many library functions that make things easier and shield you from the underlying complexity. Many of these functions are language-specific, and the documentation for them may be inadequate. Often the easiest (or only) way to learn to use them is to find an example that does something similar to what you want, and then modify the function parameters to suit your needs.

At this point you've learned enough of the Nix language to do the majority of common Nix tasks. So when I say "Stop reading this chapter!", I'm only half-joking. Instead I suggest that you *skim* the rest of this chapter, paying special attention to anything marked with ⓘ. Then move on to the following chapters where you will learn how to develop and package software using Nix. Afterward, come back to this chapter and read it in more detail.

While writing this book, I anticipated that readers would want to skip around, alternating between pure learning and learning-by-doing. I've tried to structure the book to support that; sometimes repeating information from earlier chapters that you might have skipped.

2.4. The Nix REPL

The Nix REPL [1] is an interactive environment for evaluating and debugging Nix code. It's also a good place to begin learning Nix. Enter it using the command `nix repl`. Within the REPL, type `:?` to see a list of available commands.



```
$# echo "$ nix repl"
Welcome to Nix 2.18.1. Type :? for help.

nix-repl> :?
The following commands are available:

<expr>           Evaluate and print expression
<x> = <expr>    Bind expression to variable
:a, :add <expr>  Add attributes from resulting set to scope
:b <expr>         Build a derivation
:bl <expr>        Build a derivation, creating GC roots in the
                   working directory
:e, :edit <expr> Open package or function in $EDITOR
:i <expr>          Build derivation, then install result into
                   current profile
:l, :load <path> Load Nix expression and add it to scope
:lf, :load-flake <ref> Load Nix flake and add it to scope
:p, :print <expr> Evaluate and print expression recursively
:q, :quit          Exit nix-repl
:r, :reload        Reload all files
:sh <expr>         Build dependencies of derivation, then start
                   nix-shell
:t <expr>          Describe result of evaluation
:u <expr>          Build derivation, then start nix-shell
:doc <expr>        Show documentation of a builtin function
:log <expr>        Show logs for a derivation
:te, :trace-enable [bool] Enable, disable or toggle showing traces for
                           errors
:?, :help           Brings up this help menu
```

A command that is useful to beginners is `:t`, which tells you the type of an expression.

Note that the command to exit the REPL is `:q` (or `:quit` if you prefer).

2.5. Variables

2.5.1. Assignment

You can declare variables in Nix and assign values to them.

```
nix-repl> a = 7
nix-repl> b = 3
nix-repl> a - b
4
```



The spaces before and after operators aren't always required. However, you can get unexpected results when you omit them, as shown in the following example. Nix allows hyphens (-) in variable names, so `a-b` is interpreted as the name of a variable rather than a subtraction operation.



```
nix-repl> a-b  
error: undefined variable 'a-b'  
  
at «string»:1:1:  
  
1| a-b  
| ^
```

2.5.2. Immutability

In Nix, values are *immutable*; once you assign a value to a variable, you cannot change it. You can, however, create a new variable with the same name, but in a different scope. Don't worry if you don't completely understand the previous sentence; we will see some examples in [`functions`], Section 2.15, "Let expressions", and Section 2.16, "With expressions".

In the Nix REPL, it may seem like the values of variables can be changed, in *apparent* contradiction to the previous paragraph. In truth, the REPL works some behind-the-scenes "magic", effectively creating a new nested scope with each assignment. This makes it much easier to experiment in the REPL.



```
nix-repl> x = 1  
  
nix-repl> x  
1  
  
nix-repl> x = x + 1 # creates a new variable called "x"; the original is no longer in scope  
  
nix-repl> x  
2
```

2.6. Numeric operations

2.6.1. Arithmetic operators

The usual arithmetic operators are provided.



```
nix-repl> 1 + 2    # addition  
3  
  
nix-repl> 5 - 3    # subtraction  
2  
  
nix-repl> 3 * 4    # multiplication  
12  
  
nix-repl> 6 / 2    # division  
3  
  
nix-repl> -1      # negation  
-1
```

As mentioned in Section 2.5, “Variables”, you can get unexpected results when you omit spaces around operators. Consider the following example.

```
nix-repl> 6/2  
/home/amy/codeberg/nix-book/6/2
```

What happened? Let’s use the `:t` command to find out the type of the expression.

```
nix-repl> :t 6/2  
a path
```



If an expression can be interpreted as a path, Nix will do so. This is useful, because paths are *far* more commonly used in Nix expressions than arithmetic operators. In this case, Nix interpreted `6/2` as a relative path from the current directory, which in the above example was `/home/amy/codeberg/nix-book`.

Adding a space after the `/` operator produces the expected result.

```
nix-repl> 6/ 2  
3
```

To avoid surprises and improve readability, I prefer to use spaces before and after all operators.

2.6.2. Floating-point calculations

Numbers without a decimal point are assumed to be integers. To ensure that a number is interpreted as a floating-point value, add a decimal point.



```
nix-repl> :t 5  
an integer  
  
nix-repl> :t 5.0  
a float  
  
nix-repl> :t 5.  
a float
```

In the example below, the first expression results in integer division (rounding down), while the second produces a floating-point result.

```
nix-repl> 5 / 3  
1
```

```
nix-repl> 5.0 / 3  
1.66667
```

2.7. String operations

2.7.1. String concatenation

String concatenation uses the `+` operator.

```
nix-repl> "Hello, " + "world!"  
"Hello, world!"
```

2.7.2. String interpolation

You can use the `${variable}` syntax to insert the value of a variable within a string.

```
nix-repl> name = "Wombat"  
  
nix-repl> "Hi, I'm ${name}."  
"Hi, I'm Wombat."
```

You cannot mix numbers and strings. Earlier we set `a = 7`, so the following expression fails.

```
nix-repl> "My favourite number is ${a}."  
error:  
... while evaluating a path segment  
  
at «string»:1:25:  
  
    1| "My favourite number is ${a}."  
          ^  
  
error: cannot coerce an integer to a string
```



Nix does provide functions for converting between types; we'll see these in [[_built_in_functions](#)].

2.7.3. Useful built-in functions for strings

Nix provides some built-in functions for working with strings; a few examples are shown below. For more information on these and other built-in functions, see the Nix Manual (<https://nixos.org/manual/nix/stable/language/builtins>).



How long is this string?

```
nix-repl> builtins.stringLength "supercalifragilisticexpialidocious"  
34
```

Given a starting position and a length, extract a substring. The first character of a string has index 0 .

```
nix-repl> builtins.substring 3 6 "hayneedlestack"
"needle"
```

Convert an expression to a string.

```
nix-repl> builtins.toString 7
"7"

nix-repl> builtins.toString (3*4 + 1)
"13"
```

2.8. Boolean operations

The usual boolean operators are available. Recall that earlier we set `a = 7` and `b = 3` .

```
nix-repl> a == 7          # equality test
true

nix-repl> b != 3          # inequality
false

nix-repl> a > 12          # greater than
false

nix-repl> b >= 2          # greater than or equal
true

nix-repl> a < b          # less than
false

nix-repl> b <= a          # less than or equal
true

nix-repl> !true           # logical negation
false

nix-repl> (3 * a == 21) && (a > b)    # logical AND
true

nix-repl> (b > a) || (b > 10)       # logical OR
false
```

One operator that might be unfamiliar to you is *logical implication*, which uses the symbol \rightarrow . The expression $u \rightarrow v$ is equivalent to `!u || v` .



```
nix-repl> u = false  
nix-repl> v = true  
  
nix-repl> u -> v  
true  
  
nix-repl> v -> u  
false
```

2.9. Path operations

Any expression that contains a forward slash (/) *not* followed by a space is interpreted as a path. To refer to a file or directory relative to the current directory, prefix it with ./ . You can specify the current directory as ./ .

```
nix-repl> ./file.txt  
/home/amy/codeberg/nix-book/file.txt  
  
nix-repl> ./  
/home/amy/codeberg/nix-book
```

2.9.1. Concatenating paths

Paths can be concatenated to produce a new path.

```
nix-repl> /home/wombat + /bin/sh  
/home/wombat/bin/sh  
  
nix-repl> :t /home/wombat + /bin/sh  
a path
```

Relative paths are made absolute when they are parsed, which occurs before concatenation. This is why the result in the example below is not /home/wombat/file.txt .



```
nix-repl> /home/wombat + ./file.txt  
/home/wombat/home/amy/codeberg/nix-book/file.txt
```

2.9.2. Concatenating a path + a string

A path can be concatenated with a string to produce a new path.

```
nix-repl> /home/wombat + "/file.txt"  
/home/wombat/file.txt  
  
nix-repl> :t /home/wombat + "/file.txt"  
a path
```



The Nix reference manual says that the string must not "have a string context" that refers to a store path. String contexts are beyond the scope of this book; for more information see <https://nixos.org/manual/nix/stable/language/operators#path-concatenation>.

2.9.3. Concatenating a string + a path

Strings can be concatenated with paths, but with a side-effect that may surprise you: if the path exists, the file is copied to the Nix store! The result is a string, not a path.

In the example below, the file `file.txt` is copied to `/nix/store/gp8ba25gpwvbqizqfr58jr014gmv1hd8-file.txt` (not, as you might expect, to `/home/wombat/nix/store/gp8ba25gpwvbqizqfr58jr014gmv1hd8-file.txt`).

```
nix-repl> "/home/wombat" + ./file.txt  
"/home/wombat/nix/store/gp8ba25gpwvbqizqfr58jr014gmv1hd8-file.txt"
```

The path must exist.

```
nix-repl> "/home/wombat" + ./no-such-file.txt  
error (ignored): error: end of string reached  
error: getting status of '/home/amy/codeberg/nix-book/no-such-file.txt': No such file or directory
```

2.9.4. Useful built-in functions for paths

Nix provides some built-in functions for working with paths; a few examples are shown below. For more information on these and other built-in functions, see the Nix Manual (<https://nixos.org/manual/nix/stable/language/builtins>).

Does the path exist?

```
nix-repl> builtins.pathExists ./index.html  
true  
  
nix-repl> builtins.pathExists /no/such/path  
false
```

Get a list of the files in a directory, along with the type of each file.

```
nix-repl> builtins.readDir .  
{ ".envrc" = "regular"; ".git" = "directory"; ".gitignore" = "regular"; Makefile = "regular"; images = "directory"; '
```

Read the contents of a file into a string.

```
nix-repl> builtins.readFile ./envrc  
"use nix\n"
```



2.10. List operations

2.10.1. List concatenation

Lists can be concatenated using the `++` operator.

```
nix-repl> [ 1 2 3 ] ++ [ "apple" "banana" ]  
[ 1 2 3 "apple" "banana" ]
```

2.10.2. Useful built-in functions for lists

Nix provides some built-in functions for working with lists; a few examples are shown below. For more information on these and other built-in functions, see the Nix Manual (<https://nixos.org/manual/nix/stable/language/builtins>).

Testing if an element appears in a list.

```
nix-repl> fruit = [ "apple" "banana" "cantaloupe" ]  
nix-repl> builtins.elem "apple" fruit  
true  
nix-repl> builtins.elem "broccoli" fruit  
false
```

Selecting an item from a list by index. The first element in a list has index 0 .

```
nix-repl> builtins.elemAt fruit 0  
"apple"  
nix-repl> builtins.elemAt fruit 2  
"cantaloupe"
```

Determining the number of elements in a list.

```
nix-repl> builtins.length fruit  
3
```

Accessing the first element of a list.

```
nix-repl> builtins.head fruit  
"apple"
```

Dropping the first element of a list.

```
nix-repl> builtins.tail fruit  
[ "banana" "cantaloupe" ]
```

Functions are useful for working with lists. Functions will be introduced in Section 2.12, “Functions”, but the following examples should be somewhat self-explanatory.

Using a function to filter (select elements from) a list.

```
nix-repl> numbers = [ 1 3 6 8 9 15 25 ]  
nix-repl> isBig = n: n > 10          # is the number "big" (greater than 10)?  
nix-repl> builtins.filter isBig numbers    # get just the "big" numbers  
[ 15 25 ]
```

Applying a function to all the elements in a list.



```
nix-repl> double = n: 2*n          # multiply by two
nix-repl> builtins.map double numbers  # double each element in the list
[ 2 6 12 16 18 30 50 ]
```

2.11. Attribute set operations

2.11.1. Selection

The `.` operator selects an attribute from a set.

```
nix-repl> animal = { name = { first = "Professor"; last = "Paws"; }; age = 10; species = "cat"; }

nix-repl> animal . age
10

nix-repl> animal . name . first
"Professor"
```

2.11.2. Query

We can use the `?` operator to find out if a set has a particular attribute.

```
nix-repl> animal ? species
true

nix-repl> animal ? bicycle
false
```

2.11.3. Modification

We can use the `//` operator to modify an attribute set. Recall that Nix values are immutable, so the result is a new value (the original is not modified). Attributes in the right-hand set take preference.

```
nix-repl> animal // { species = "tiger"; }
{ age = 10; name = { ... }; species = "tiger"; }
```

2.11.4. Recursive attribute sets

An ordinary attribute set cannot refer to its own elements. To do this, you need a *recursive* attribute set.

```
nix-repl> { x = 3; y = 4*x; }
error: undefined variable 'x'

at «string»:1:16:

  1| { x = 3; y = 4*x; }
   |           ^

nix-repl> rec { x = 3; y = 4*x; }
{ x = 3; y = 12; }
```



2.11.5. Useful built-in functions for attribute sets

Nix provides some built-in functions for working with attribute sets; a few examples are shown below. For more

information on these and other built-in functions, see the Nix Manual (<https://nixos.org/manual/nix/stable/language/builtins>).

Get an alphabetical list of the keys.

```
nix-repl> builtins.attrNames animal
[ "age" "name" "species" ]
```

Get the values associated with each key, in alphabetical order by the key names.

```
nix-repl> builtins.attrValues animal
[ 10 "Professor Paws" "cat" ]
```

What value is associated with a key?

```
nix-repl> builtins.getAttr "age" animal
10
```

Does the set have a value for a key?

```
nix-repl> builtins.hasAttr "name" animal
true
```

```
nix-repl> builtins.hasAttr "car" animal
false
```

Remove one or more keys and associated values from a set.

```
nix-repl> builtins.removeAttrs animal [ "age" "species" ]
{ name = "Professor Paws"; }
```

2.12. Functions

2.12.1. Anonymous functions

Functions are defined using the syntax *parameter*: *expression*, where the *expression* typically involves the *parameter*. Consider the following example.

```
nix-repl> x: x + 1
«lambda @ «string»:1:1»
```



We created a function that adds 1 to its input. However, it doesn't have a name, so we can't use it directly. Anonymous functions do have their uses, as we shall see shortly.

Note that the message printed by the Nix REPL when we created the function uses the term *lambda*. This derives from a branch of mathematics called *lambda calculus*. Lambda calculus was the inspiration for most functional languages such as Nix. Functional programmers often call anonymous functions "lambdas".

The Nix REPL confirms that the expression `x: x + 1` defines a function.

```
nix-repl> :t x: x + 1
a function
```

2.12.2. Named functions and function application

How can we use a function? Recall from Section 2.2.8, “Functions” that functions can be treated like any other data type. In particular, we can assign it to a variable.

```
nix-repl> f = x: x + 1
nix-repl> f
«lambda @ «string»:1:2»
```

Procedural languages such as C or Java often use parenthesis to apply a function to a value, e.g. `f(5)`. Nix, like lambda calculus and most functional languages, does not require parenthesis for function application. This reduces visual clutter when chaining a series of functions.

Now that our function has a name, we can use it.

```
nix-repl> f 5
6
```

2.12.3. Multiple parameters using nested functions

Functions in Nix always have a single parameter. To define a calculation that requires more than one parameter, we create functions that return functions!

```
nix-repl> add = a: (b: a+b)
```

We have created a function called `add`. When applied to a parameter `a`, it returns a new function that adds `a` to its input. Note that the expression `(b: a+b)` is an anonymous function. We never call it directly, so it doesn't need a name. Anonymous functions are useful after all!

I used parentheses to emphasise the inner function, but they aren't necessary. More commonly we would write the following.

```
nix-repl> add = a: b: a+b
```

If we only supply one parameter to `add`, the result is a new function rather than a simple value. Invoking a function without supplying all of the expected parameters is called *partial application*.

```
nix-repl> add 3          # Returns a function that adds 3 to any input
«lambda @ «string»:1:6»
```

Now let's apply `add 3` to the value `5`.

```
nix-repl> (add 3) 5
8
```



In fact, the parentheses aren't needed.

```
nix-repl> add 3 5  
8
```

If you've never used a functional programming language, this all probably seems very strange. Imagine that you want to add two numbers, but you have a very unusual calculator labeled "add". This calculator never displays a result, it only produces more calculators! If you enter the value `3` into the "add" calculator, it gives you a second calculator labeled "add 3". You then enter `5` into the "add 3" calculator, which displays the result of the addition, `8`.

With that image in mind, let's walk through the steps again in the REPL, but this time in more detail. The function `add` takes a single parameter `a`, and returns a new function that takes a single parameter `b`, and returns the value `a + b`. Let's apply `add` to the value `3`, and give the resulting new function a name, `g`.

```
nix-repl> g = add 3
```

The function `g` takes a single parameter and adds `3` to it. The Nix REPL confirms that `g` is indeed a function.

```
nix-repl> :t g  
a function
```

Now we can apply `g` to a number to get a new number.

```
nix-repl> g 5  
8
```

2.12.4. Multiple parameters using attribute sets

I said earlier that a function in Nix always has a single parameter. However, that parameter need not be a simple value; it could be a list or an attribute set. This approach is widely used in Nix, and the language has some special features to support it. This is an important topic, so we will cover it separately in Section 2.13, "Argument sets".

2.13. Argument sets

An attribute set that is used as a function parameter is often called an *argument set*.

2.13.1. Set patterns

To specify an attribute set as a function parameter, we use a *set pattern*, which has the form

```
{ _name1_, _name2_, ... }
```



Note that while the key-value associations in attribute sets are separated by semi-colons, the key names in the attribute set pattern are separated by commas. Here's an example of a function that has an attribute set as an input parameter.

```
nix-repl> greet = { first, last }: "Hello ${first} ${last}! May I call you ${first}?"  
nix-repl> greet { first="Amy"; last="de Buitléir"; }  
"Hello Amy de Buitléir! May I call you Amy?"
```

2.13.2. Optional parameters

We can make some values in an argument set optional by providing default values, using the syntax `name ? value`. This is illustrated below.

```
nix-repl> greet = { first, last ? "whatever-your-lastname-is", topic ? "Nix" }: "Hello ${first} ${last}! May I call you ${topic}?"  
nix-repl> greet { first="Amy"; }  
"Hello Amy whatever-your-lastname-is! May I call you Amy? Are you enjoying learning Nix?"  
nix-repl> greet { first="Amy"; topic="Mathematics"; }  
"Hello Amy whatever-your-lastname-is! May I call you Amy? Are you enjoying learning Mathematics?"
```

2.13.3. Variadic attributes

A function can allow the caller to supply argument sets that contain "extra" values. This is done with the special parameter `...`.

```
nix-repl> formatName = { first, last, ... }: "${first} ${last}"
```

One reason for doing this is to allow the caller to pass the same argument set to multiple functions, even though each function may not need all of the values.

```
nix-repl> person = { first="Joe"; last="Bloggs"; address="123 Main Street"; }  
nix-repl> formatName person  
"Joe Bloggs"
```

Another reason for allowing variadic arguments is when a function calls another function, supplying the same argument set. An example is shown in Section 2.13.4, “@-patterns”.

2.13.4. @-patterns

It can be convenient for a function to be able to reference the argument set as a whole. This is done using an `@-pattern`.

```
nix-repl> formatPoint = p@{ x, y, ... }: builtins.toXML p  
nix-repl> formatPoint { x=5; y=3; z=2; }  
<?xml version='1.0' encoding='utf-8'?>\n<expr>\n  <attrs>\n    <attr name=\"x\">\n      <int value=\"5\" />\n    </attr>
```

Alternatively, the `@-pattern` can appear *after* the argument set, as in the example below.

```
nix-repl> formatPoint = { x, y, ... } @ p: builtins.toXML p
```



An `@-pattern` is the only way a function can access variadic attributes, so they are often used together. In the example below, the function `greet` passes its argument set, including the variadic arguments, to the function `confirmAddress`.

```
nix-repl> confirmAddress = { address, ... }: "Do you still live at ${address}?"  
nix-repl> greet = args@{ first, last, ... }: "Hello ${first}. " + confirmAddress args  
nix-repl> greet person  
"Hello Joe. Do you still live at 123 Main Street?"
```

2.14. If expressions

The conditional construct in Nix is an *expression*, not a *statement*. Since expressions must have values in all cases, you must specify both the `then` and the `else` branch.

```
nix-repl> a = 7  
nix-repl> b = 3  
nix-repl> if a > b then "yes" else "no"  
"yes"
```

2.15. Let expressions

A `let` expression defines a value with a local scope.

```
nix-repl> let x = 3; in x*x  
9  
nix-repl> let x = 3; y = 2; in x*x + y  
11
```

You can also nest `let` expressions. The previous expression is equivalent to the following.

```
nix-repl> let x = 3; in let y = 2; in x*x + y  
11
```

A variable defined inside a `let` expression will "shadow" an outer variable with the same name.



```
nix-repl> x = 100  
nix-repl> let x = 3; in x*x  
9  
nix-repl> let x = 3; in let x = 7; in x+1  
8
```



A variable in a `let` expression can refer to another variable in the expression. This is similar to how recursive attribute sets work.

```
nix-repl> let x = 3; y = x + 1; in x*y  
12
```

2.16. With expressions

A `with` expression is somewhat similar to a `let` expression, but it brings all of the associations in an attribute set into scope.

```
nix-repl> point = { x1 = 3; x2 = 2; }

nix-repl> with point; x1*x1 + x2
11
```

Unlike a `let` expression, a variable defined inside a `with` expression will *not* "shadow" an outer variable with the same name.

```
nix-repl> name = "Amy"

nix-repl> animal = { name = "Professor Paws"; age = 10; species = "cat"; }

nix-repl> with animal; "Hello, " + name
>Hello, Amy
```

However, you can refer to the variable in the inner scope using the attribute selection operator (`.`).

```
nix-repl> with animal; "Hello, " + animal.name
>Hello, Professor Paws
```



3. Hello, flake!

Before learning to write Nix flakes, let's learn how to use them. I've created a simple example of a flake in this git [repository](https://codeberg.org/mhwombat/hello-flake) (<https://codeberg.org/mhwombat/hello-flake>). To run this flake, you don't need to install anything; simply run the command below. The first time you use a flake, Nix has to fetch and build it, which may take time. Subsequent invocations should be instantaneous.

```
$ nix run "git+https://codeberg.org/mhwombat/hello-flake"  
Hello from your flake!
```

That's a lot to type every time we want to use this package. Instead, we can enter a shell with the package available to us, using the `nix shell` command.

```
$ nix shell "git+https://codeberg.org/mhwombat/hello-flake"
```

In this shell, the command is on our `$PATH`, so we can execute the command by name.

```
$ hello-flake  
Hello from your flake!
```

Nix didn't *install* the package; it merely built and placed it in a directory called the "Nix store". Thus we can have multiple versions of a package without worrying about conflicts. We can find out the location of the executable, if we're curious.

```
$ which hello-flake  
/nix/store/qskl8ajlgnl654fhgsmv74yv8x9r3kzg-hello-flake/bin/hello-flake
```

Once we exit that shell, the `hello-flake` command is no longer directly available.

```
$ exit  
$ hello-flake  
sh: line 3: hello-flake: command not found
```

However, we can still run the command using the store path we found earlier. That's not particularly convenient, but it does demonstrate that the package remains in the store for future use.

```
/nix/store/0xbn2hi6h1m5h4kc02vwffs2cydrbc0r-hello-flake/bin/hello-flake
```



4. The hello-flake repo

Let's clone the repository and see how the flake is defined.

```
$ git clone https://codeberg.org/mhwombat/hello-flake
Cloning into 'hello-flake'...
$ cd hello-flake
$ ls
flake.lock
flake.nix
hello-flake
LICENSE
README.md
```

This is a simple repo with just a few files. Like most git repos, it includes `LICENSE`, which contains the software license, and `README.md` which provides information about the repo.

The `hello-flake` file is the executable we ran earlier. This particular executable is just a shell script, so we can view it. It's an extremely simple script with just two lines.

hello-flake

```
1 | #!/usr/bin/env sh
2 |
3 | echo "Hello from your flake!"
```

Now that we have a copy of the repo, we can execute this script directly.

```
$ ./hello-flake
Hello from your flake!
```

Not terribly exciting, I know. But starting with such a simple package makes it easier to focus on the flake system without getting bogged down in the details. We'll make this script a little more interesting later.

Let's look at another file. The file that defines how to package a flake is always called `flake.nix`.

flake.nix

NIX



```

1  {
2      # See https://github.com/mhwombat/nix-for-numbskulls/blob/main/flakes.md
3      # for a brief overview of what each section in a flake should or can contain.
4
5      description = "a very simple and friendly flake";
6
7      inputs = {
8          nixpkgs.url = "github:NixOS/nixpkgs";
9          flake-utils.url = "github:numtide/flake-utils";
10     };
11
12     outputs = { self, nixpkgs, flake-utils }:
13         flake-utils.lib.eachDefaultSystem (system:
14             let
15                 pkgs = import nixpkgs { inherit system; };
16                 in
17             {
18                 packages = rec {
19                     hello = pkgs.stdenv.mkDerivation rec {
20                         name = "hello-flake";
21
22                         src = ./;
23
24                         unpackPhase = "true";
25
26                         buildPhase = ":";
27
28                         installPhase =
29                             ''
30                             mkdir -p $out/bin
31                             cp $src/hello-flake $out/bin/hello-flake
32                             chmod +x $out/bin/hello-flake
33                         '';
34                     };
35                     default = hello;
36                 };
37
38                 apps = rec {
39                     hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };
40                     default = hello;
41                 };
42             }
43         );
44     }
}

```

If this is your first time seeing a flake definition, it probably looks intimidating. Flakes are written in the Nix language, introduced in an earlier chapter. However, you don't really need to know Nix to follow this example. For now, I'd like to focus on the inputs section.

```

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

```



There are just two entries, one for `nixpkgs` and one for `flake-utils`. The first one, `nixpkgs` refers to the collection of standard software packages that can be installed with the Nix package manager. The second, `flake-utils`, is a collection of utilities that simplify writing flakes. The important thing to note is that the `hello-flake` package *depends* on `nixpkgs` and `flake-utils`.

Finally, let's look at `flake.lock`, or rather, just part of it.

`flake.lock`

```
{
  "nodes": {
    "flake-utils": {
      "inputs": {
        "systems": "systems"
      },
      "locked": {
        "lastModified": 1681202837,
        "narHash": "sha256-H+Rh19JDwRtpVPAWp64F+r1EtXUWBAQW28eAi3SRSzg=",
        "owner": "numtide",
        "repo": "flake-utils",
        "rev": "cfacdce06f30d2b68473a46042957675eebb3401",
        "type": "github"
      },
      "original": {
        "owner": "numtide",
        "repo": "flake-utils",
        "type": "github"
      }
    },
    "nixpkgs": {
      "locked": {
        "lastModified": 1681665000,
        "narHash": "sha256-hDGTR59wC3qrQZFxVi2U3vTY+r02+0kbq080h01C4Nk=",
        "owner": "NixOS",
        "repo": "nixpkgs",
        "rev": "3a6205d9f79fe526be03d8c465403b118ca4cf37",
        "type": "github"
      },
      "original": {
        "owner": "NixOS",
        "repo": "nixpkgs",
        "type": "github"
      }
    },
    "root": {
      "inputs": {
        "flake-utils": "flake-utils",
        "nixpkgs": "nixpkgs"
      }
    }
  ...
}
```

LINENUMS

If `flake.nix` seemed intimidating, then this file looks like an invocation for Cthulhu. The good news is that this file is automatically generated; you never need to write it. It contains information about all of the dependencies for the flake, including where they came from, the exact version/revision, and hash. This lockfile *uniquely* specifies all flake dependencies, (e.g., version number, branch, revision, hash), so that *anyone, anywhere, any time, can re-create the exact same environment that the original developer used.*



No more complaints of “but it works on my machine!”. That is the benefit of using flakes.

5. Flake structure

The basic structure of a flake is shown below.

```
{  
    description = package description  
    inputs = dependencies  
    outputs = what the flake produces  
    nixConfig = advanced configuration options  
}
```

NIX

5.1. Description

The `description` part is self-explanatory; it's just a string. You probably won't need `nixConfig` unless you're doing something fancy. I'm going to focus on what goes into the `inputs` and `outputs` sections, and highlight some of the things I found confusing when I began using flakes.

5.2. Inputs

This section specifies the dependencies of a flake. It's an *attribute set*; it maps keys to values.

To ensure that a build is reproducible, the build step runs in a *pure* environment with no network access. Therefore, any external dependencies must be specified in the “inputs” section so they can be fetched in advance (before we enter the pure environment).

Each entry in this section maps an input name to a *flake reference*. This commonly takes the following form.

```
NAME.url = URL-LIKE-EXPRESSION
```

As a first example of a flake reference, all (almost all?) flakes depend on “nixpkgs”, which is a large Git repository of programs and libraries that are pre-packaged for Nix. We can write that as

```
nixpkgs.url = "github:NixOS/nixpkgs/nixos-version";
```

NIX

where `version` is replaced with the version number that you used to build the package, e.g. `22.11`. Information about the latest nixpkgs releases is available at <https://status.nixos.org/>. You can also write the entry without the version number

```
nixpkgs.url = "github:NixOS/nixpkgs/nixos";
```

NIX

or more simply,

```
nixpkgs.url = "nixpkgs";
```

NIX

You might be concerned that omitting the version number would make the build non-reproducible. If someone else builds the flake, could they end up with a different version of nixpkgs? No! remember that the lockfile (`flake.lock`) *uniquely* specifies all flake inputs.

Git and Mercurial repositories are the most common type of flake reference, as in the examples below.



A Git repository

```
git+https://github.com/NixOS/patchelf
```

A specific branch of a Git repository

```
git+https://github.com/NixOS/patchelf?ref=master
```

A specific revision of a Git repository

```
git+https://github.com/NixOS/patchelf?ref=master&rev=f34751b88bd07d7f44f5cd3200fb4122bf916c7e
```

A tarball

```
https://github.com/NixOS/patchelf/archive/master.tar.gz
```

You can find more examples of flake references in the [Nix Reference Manual](#) (<https://nixos.org/manual/nix/stable/command-ref/new-cli/nix3-flake.html#examples>).

Although you probably won't need to use it, there is another syntax for flake references that you might encounter. This example

 NIX
inputs.import-cargo = {
 type = "github";
 owner = "edolstra";
 repo = "import-cargo";
};

is equivalent to

NIX
inputs.import-cargo.url = "github:edolstra/import-cargo";

Each of the `inputs` is fetched, evaluated and passed to the `outputs` function as a set of attributes with the same name as the corresponding input.

5.3. Outputs

This section is a function that essentially returns the recipe for building the flake.

We said above that `inputs` are passed to the `outputs`, so we need to list them as parameters. This example references the `import-cargo` dependency defined in the previous example.

 NIX
outputs = { self, nixpkgs, import-cargo }: {
 definitions for outputs
};

So what actually goes in the highlighted section? That depends on the programming languages your software is written in, the build system you use, and more. There are Nix functions and tools that can simplify much of this, and new, easier-to-use ones are released regularly. We'll look at some of these in the next section.

6. A generic flake

The previous section presented a very high-level view of flakes, focusing on the basic structure. In this section, we will add a bit more detail.

Flakes are written in the Nix programming language, which is a functional language. As with most programming languages, there are many ways to achieve the same result. Below is an example you can follow when writing your own flakes. I'll explain the example in some detail.

```
{
  description = "brief package description";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
    ...other dependencies... ❶
  };

  outputs = { self, nixpkgs, flake-utils, ...other dependencies... ❷ }:
    flake-utils.lib.eachDefaultSystem (system: ❸
      let
        pkgs = import nixpkgs { inherit system; };
        python = pkgs.python3;
      in
      {
        devShells = rec {
          default = pkgs.mkShell {
            packages = [ packages needed for development shell; ❹ ]));
          ];
        };

        packages = rec {
          myPackageName = package definition; ❺
          default = myPackageName;
        };

        apps = rec {
          myPackageName = flake-utils.lib.mkApp { drv = self.packages.${system}.myPackageName; };
          default = myPackageName;
        };
      };
    );
  );
}
```

We discussed how to specify flake inputs ❶ in the previous section, so this part of the flake should be familiar. Remember also that any dependencies in the input section should also be listed at the beginning of the outputs section ❷ .

Now it's time to look at the content of the output section. If we want the package to be available for multiple systems (e.g., “x86_64-linux”, “aarch64-linux”, “x86_64-darwin”, and “aarch64-darwin”), we need to define the output for each of those systems. Often the definitions are identical, apart from the name of the system. The `eachDefaultSystem` function ❸ provided by `flake-utils` allows us to write a single definition using a variable for the system name. The function then iterates over all default systems to generate the outputs for each one.

The `devShells` variable specifies the environment that should be available when doing development on the package. If you don't need a special development environment, you can omit this section. At ❹ you would list any tools (e.g., compilers and language-specific build tools) you want to have available in a development shell. If the compiler needs



access to language-specific packages, there are Nix functions to assist with that. These functions are very language-specific, and not always well-documented. We will see examples for some languages later in the tutorial. In general, I recommend that you do a web search for “nix language”, and try to find resources that were written or updated recently.

The `packages` variable defines the packages that this flake provides. The package definition ❷ depends on the programming languages your software is written in, the build system you use, and more. There are Nix functions and tools that can simplify much of this, and new, easier-to-use ones are released regularly. Again, I recommend that you do a web search for “nix language”, and try to find resources that were written or updated recently.

The `apps` variable identifies any applications provided by the flake. In particular, it identifies the default executable ❸ that `nix run` will run if you don’t specify an app.

Below is a list of some functions that are commonly used in this section.

General-purpose

The standard environment provides `mkDerivation`, which is especially useful for the typical `./configure; make; make install` scenario. It’s customisable.

Python

`buildPythonApplication`, `buildPythonPackage`.

Haskell

`mkDerivation` (Haskell version, which is a wrapper around the standard environment version), `developPackage`, `callCabal2Nix`.



7. Another look at hello-flake

Now that we have a better understanding of the structure of `flake.nix`, let's have a look at the one we saw earlier, in the `hello-flake` repo. If you compare this flake definition to the colour-coded template presented in the previous section, most of it should look familiar.

`flake.nix`

```
{  
  description = "a very simple and friendly flake";  
  
  inputs = {  
    nixpkgs.url = "github:NixOS/nixpkgs";  
    flake-utils.url = "github:numtide/flake-utils";  
  };  
  
  outputs = { self, nixpkgs, flake-utils }:  
    flake-utils.lib.eachDefaultSystem (system:  
      let  
        pkgs = import nixpkgs { inherit system; };  
      in  
      {  
        packages = rec {  
          hello =  
            . . .  
            SOME UNFAMILIAR STUFF  
            . . .  
        };  
        default = hello;  
      };  
  
      apps = rec {  
        hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };  
        default = hello;  
      };  
    );  
  );  
}
```

This `flake.nix` doesn't have a `devShells` section, because development on the current version doesn't require anything beyond the “bare bones” linux commands. Later we will add a feature that requires additional development tools.

Now let's look at the section I labeled `SOME UNFAMILIAR STUFF` and see what it does.



```
packages = rec {
    hello = pkgs.stdenv.mkDerivation rec { ❶
        name = "hello-flake";

        src = ./.; ❷

        unpackPhase = "true";

        buildPhase = ":";

        installPhase =
        ''
            mkdir -p $out/bin ❸
            cp $src/hello-flake $out/bin/hello-flake ❹
            chmod +x $out/bin/hello-flake ❺
        '';
    };
};
```

This flake uses `mkDerivation` ❶ which is a very useful general-purpose package builder provided by the Nix standard environment. It's especially useful for the typical `./configure; make; make install` scenario, but for this flake we don't even need that.

The `name` variable is the name of the flake, as it would appear in a package listing if we were to add it to `Nixpkgs` or another package collection. The `src` variable ❷ supplies the location of the source files, relative to `flake.nix`. When a flake is accessed for the first time, the repository contents are fetched in the form of a tarball. The `unpackPhase` variable indicates that we do want the tarball to be unpacked.

The `buildPhase` variable is a sequence of Linux commands to build the package. Typically, building a package requires compiling the source code. However, that's not required for a simple shell script. So `buildPhase` consists of a single command, `:`, which is a no-op or "do nothing" command.

The `installPhase` variable is a sequence of Linux commands that will do the actual installation. In this case, we create a directory ❸ for the installation, copy the `hello-flake` script there ❹, and make the script executable ❺. The environment variable `$src` refers to the source directory, which we specified earlier ❷.

Earlier we said that the build step runs in a pure environment to ensure that builds are reproducible. This means no Internet access; indeed no access to any files outside the build directory. During the build and install phases, the only commands available are those provided by the Nix standard environment and the external dependencies identified in the `inputs` section of the flake.

I've mentioned the Nix standard environment before, but I didn't explain what it is. The standard environment, or `stdenv`, refers to the functionality that is available during the build and install phases of a Nix package (or flake). It includes the commands listed below^[2].



- The GNU C Compiler, configured with C and C++ support.
- GNU coreutils (contains a few dozen standard Unix commands).
- GNU findutils (contains find).
- GNU diffutils (contains diff, cmp).
- GNU sed.

- GNU grep.
- GNU awk.
- GNU tar.
- gzip, bzip2 and xz.
- GNU Make.
- Bash.
- The patch command.
- On Linux, stdenv also includes the patchelf utility.

Only a few environment variables are available. The most interesting ones are listed below.

- `$name` is the package name.
- `$src` refers to the source directory.
- `$out` is the path to the location in the Nix store where the package will be added.
- `$system` is the system that the package is being built for.
- `$PWD` and `$TMP` both point to a temporary build directories
- `$HOME` and `$PATH` point to nonexistent directories, so the build cannot rely on them.



8. Modifying the flake

8.1. The Nix development shell

Let's make a simple modification to the script. This will give you an opportunity to check your understanding of flakes.

The first step is to enter a development shell.

```
$ nix develop
```

The `flake.nix` file specifies all of the tools that are needed during development of the package. The `nix develop` command puts us in a shell with those tools. As it turns out, we didn't need any extra tools (beyond the standard environment) for development yet, but that's usually not the case. Also, we will soon need another tool.

A development environment only allows you to *develop* the package. Don't expect the package *outputs* (e.g. executables) to be available until you build them. However, our script doesn't need to be compiled, so can't we just run it?

```
$ hello-flake
bash: line 16: hello-flake: command not found
```

That worked before; why isn't it working now? Earlier we used `nix shell` to enter a *runtime* environment where `hello-flake` was available and on the `$PATH`. This time we entered a *development* environment using the `nix develop` command. Since the flake hasn't been built yet, the executable won't be on the `$PATH`. We can, however, run it by specifying the path to the script.

```
$ ./hello-flake
Hello from your flake!
```

We can also build the flake using the `nix build` command, which places the build outputs in a directory called `result`.

```
$ nix build
$ result/bin/hello-flake
Hello from your flake!
```

Rather than typing the full path to the executable, it's more convenient to use `nix run`.

```
$ nix run
Hello from your flake!
```

Here's a summary of the more common Nix commands.



command	Action
<code>nix develop</code>	Enters a <i>development</i> shell with all the required development tools (e.g. compilers and linkers) available (as specified by <code>flake.nix</code>).
<code>nix shell</code>	Enters a <i>runtime</i> shell where the flake's executables are available on the <code>\$PATH</code> .

command	Action
nix build	Builds the flake and puts the output in a directory called <code>result</code> .
nix run	Runs the flake's default executable, rebuilding the package first if needed. Specifically, it runs the version in the Nix store, not the version in <code>result</code> .

8.2. Introducing a dependency

Now we're ready to make the flake a little more interesting. Instead of using the `echo` command in the script, we can use the Linux `cowsay` command. Here's the `hello-flake` file, with the modified line highlighted.

hello-flake

```
#!/usr/bin/env sh
cowsay "Hello from your flake!"
```

NIX

Let's test the modified script.

```
$ ./hello-flake
./hello-flake: line 3: cowsay: command not found
```

What went wrong? Remember that we are in a *development* shell. Since `flake.nix` didn't define the `devShells` variable, the development shell only includes the Nix standard environment. In particular, the `cowsay` command is not available.

To fix the problem, we can modify `flake.nix`. We don't need to add `cowsay` to the `inputs` section because it's included in `nixpkgs`, which is already an input. However, we also want it to be available in a development shell. The highlighted modifications below will accomplish that.

flake.nix

NIX


```
{  
  # See https://github.com/mhwombat/nix-for-numbskulls/blob/main/flakes.md  
  # for a brief overview of what each section in a flake should or can contain.  
  
  description = "a very simple and friendly flake";  
  
  inputs = {  
    nixpkgs.url = "github:NixOS/nixpkgs";  
    flake-utils.url = "github:numtide/flake-utils";  
  };  
  
  outputs = { self, nixpkgs, flake-utils }:  
    flake-utils.lib.eachDefaultSystem (system:  
      let  
        pkgs = import nixpkgs { inherit system; };  
        in  
        {  
          devShells = rec {  
            default = pkgs.mkShell {  
              packages = [ pkgs.cowsay ];  
            };  
          };  
  
          packages = rec {  
            hello = pkgs.stdenv.mkDerivation rec {  
              name = "hello-flake";  
  
              src = ./.;  
  
              unpackPhase = "true";  
  
              buildPhase = ":";  
  
              installPhase =  
                '';  
                mkdir -p $out/bin  
                cp $src/hello-flake $out/bin/hello-flake  
                chmod +x $out/bin/hello-flake  
              '';  
            };  
            default = hello;  
          };  
  
          apps = rec {  
            hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };  
            default = hello;  
          };  
        };  
    );  
};  
}
```



Now we restart the development shell and see that the `cowsay` command is available and the script works. Because we've updated source files but haven't `git committed` the new version, we get a warning message about it being "dirty". It's just a warning, though; the script runs correctly.

```
$ nix develop
warning: Git tree '/home/amy/codeberg/nix-book/source/modify-hello-flake/hello-flake' is dirty
$ which cowsay # is it available now?
/nix/store/gfi27h4y5n4aralcxrc0377p8mjb1cvb-cowsay-3.7.0/bin/cowsay
$ ./hello-flake

< Hello from your flake! >
-----
 \  ^__^
  \  (oo)\_____
    (__)\       )\/\
        ||----w |
        ||     ||
```

Alternatively, we could use `nix run`.

```
$ nix run
warning: Git tree '/home/amy/codeberg/nix-book/source/modify-hello-flake/hello-flake' is dirty

< Hello from your flake! >
-----
 \  ^__^
  \  (oo)\_____
    (__)\       )\/\
        ||----w |
        ||     ||
```

Note, however, that `nix run` rebuilt the package in the Nix store and ran *that*. It did not alter the copy in the `result` directory, as we'll see next.

```
$ cat result/bin/hello-flake
#!/nix/store/zlf0f88vj30sc7567b80152d19pbcdmy2-bash-5.2-p15/bin/sh

echo "Hello from your flake!"
```

If we want to update the version in `result`, we need `nix build` again.

```
$ nix build
warning: Git tree '/home/amy/codeberg/nix-book/source/modify-hello-flake/hello-flake' is dirty
$ cat result/bin/hello-flake
#!/nix/store/zlf0f88vj30sc7567b80152d19pbcdmy2-bash-5.2-p15/bin/sh

cowsay "Hello from your flake!"
```

Let's `git commit` the changes and verify that the warning goes away. We don't need to `git push` the changes until we're ready to share them.



```
$ git commit hello-flake flake.nix -m 'added bovine feature'  
[main c264cad] added bovine feature  
2 files changed, 7 insertions(+), 1 deletion(-)  
$ nix run  
  
< Hello from your flake! >  
-----  
 \ ^__^  
  \  (oo)\_____  
   (__)\       )\/\  
    ||----w |  
    ||     ||
```

8.3. Development workflows

If you're getting confused about when to use the different commands, it's because there's more than one way to use Nix. I tend to think of it as two different development workflows.

My usual, *high-level workflow* is quite simple.

1. `nix run` to re-build (if necessary) and run the executable.
2. Fix any problems in `flake.nix` or the source code.
3. Repeat until the package works properly.

In the high-level workflow, I don't use a development shell because I don't need to directly invoke development tools such as compilers and linkers. Nix invokes them for me according to the output definition in `flake.nix`.

Occasionally I want to work at a lower level, and invoke compiler, linkers, etc. directly. Perhaps want to work on one component without rebuilding the entire package. Or perhaps I'm confused by some error message, so I want to temporarily bypass Nix and work directly with the compiler. In this case I temporarily switch to a *low-level workflow*.

1. `nix develop` to enter a development shell with any development tools I need (e.g. compilers, linkers, documentation generators).
2. Directly invoke tools such as compilers.
3. Fix any problems in `flake.nix` or the source code.
4. Directly invoke the executable. Note that the location of the executable depends on the development tools – It probably isn't `result`!
5. Repeat until the package works properly.

I generally only use `nix build` if I just want to build the package but not execute anything (perhaps it's just a library).



8.4. This all seems like a hassle!

It is a bit annoying to modify `flake.nix` and either rebuild or reload the development environment every time you need another tool. However, this Nix way of doing things ensures that all of your dependencies, down to the exact versions, are captured in `flake.lock`, and that anyone else will be able to reproduce the development environment.

9. A new flake from scratch

At last we are ready to create a flake from scratch! The sections in this chapter are very similar; read the one for your language of choice. If you're interested in a language that I haven't covered, feel free to suggest it by creating an [issue](#) (<https://codeberg.org/mhwombat/nix-book/issues>).

9.1. Haskell

Start with an empty directory and create a git repository.

```
$ mkdir hello-haskell
$ cd hello-haskell
$ git init
Initialized empty Git repository in /home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-haskell/.git/
```

9.1.1. A simple Haskell program

Next, we'll create a simple Haskell program.

Main.hs

```
1 import Network.HostName
2
3 main :: IO ()
4 main = do
5   putStrLn "Hello from Haskell inside a Nix flake!"
6   h <- getHostName
7   putStrLn $ "Your hostname is: " ++ h
```

HASKELL

9.1.2. (Optional) Testing before packaging

Before we package the program, let's verify that it runs. We're going to need a Haskell compiler. By now you've probably figured out that we can write a `flake.nix` and define a development shell that includes Haskell. We'll do that shortly, but first I want to show you a handy shortcut. We can launch a *temporary* shell with any Nix packages we want. This is convenient when you just want to try out some new software and you're not sure if you'll use it again. It's also convenient when you're not ready to write `flake.nix` (perhaps you're not sure what tools and packages you need), and you want to experiment a bit first.

The command to enter a temporary shell is

```
nix-shell -p packages
```

If there are multiple packages, they should be separated by spaces.



The command used here is `nix-shell` with a hyphen, not `nix shell` with a space; those are two different commands. In fact there are hyphenated and non-hyphenated versions of many Nix commands, and yes, it's confusing. The non-hyphenated commands were introduced when support for flakes was added to Nix. I predict that eventually all hyphenated commands will be replaced with non-hyphenated versions. Until then, a useful rule of thumb is that non-hyphenated commands are for working directly with flakes; hyphenated commands are for everything else.

Some unsuitable shells



In this section, we will try commands that fail in subtle ways. Examining these failures will give you a much better understanding of Haskell development with Nix, and help you avoid (or at least diagnose) similar problems in future. If you're impatient, you can skip to the next section to see the right way to do it. You can come back to this section later to learn more.

Let's enter a shell with the Glasgow Haskell Compiler ("ghc") and try to run the program.

```
$ nix-shell -p ghc
$ runghc Main.hs

Main.hs:1:1: error:
  Could not find module 'Network.HostName'
  Use -v (or `:set -v` in ghci) to see a list of the files searched for.
|
1 | import Network.HostName
| ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

The error message tells us that we need the module `Network.HostName`. That module is provided by the Haskell package called `hostname`. Let's exit that shell and try again, this time adding the `hostname` package.

```
$ exit
$ nix-shell -p "[ghc hostname]"
$ runghc Main.hs

Main.hs:1:1: error:
  Could not find module 'Network.HostName'
  Use -v (or `:set -v` in ghci) to see a list of the files searched for.
|
1 | import Network.HostName
| ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

That reason that failed is that we asked for the wrong package. The Nix package `hostname` isn't the Haskell package we wanted, it's a different package entirely (an alias for `hostname-net-tools`.) The package we want is in the *package set* called `haskellPackages`, so we can refer to it as `haskellPackages.hostname`.

Let's try that again, with the correct package.

```
$ exit
$ nix-shell -p "[ghc haskellPackages.hostname]"
$ runghc Main.hs

Main.hs:1:1: error:
  Could not find module 'Network.HostName'
  Use -v (or `:set -v` in ghci) to see a list of the files searched for.
|
1 | import Network.HostName
| ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Now what's wrong? The syntax we used in the `nix-shell` command above is fine, but it doesn't make the package *available to GHC!*

A suitable shell for a quick test



Consider the Haskell "pandoc" package, which provides both an executable (the Nix package `pandoc`) and a library (the Nix package `haskellPackages.pandoc`). There are several different shells we could create involving both Pandoc and GHC, and it's important to understand the differences between them.

<code>nix-shell -p "[ghc pandoc]"</code>	Makes the Pandoc <i>executable</i> available at the command line, but the <i>library</i> won't be visible to GHC.
<code>nix-shell -p "haskellPackages.ghcWithPackages (pkgs: with pkgs; [pandoc])"</code>	Makes the Pandoc <i>library</i> visible to GHC, but we won't be able to run the <i>executable</i> .
<code>nix-shell -p "[pandoc (haskellPackages.ghcWithPackages (pkgs: with pkgs; [pandoc]))]"</code>	Makes the Pandoc <i>executable</i> available at the command line, and the <i>library</i> visible to GHC.

Now we can create a shell that can run the program.

```
$ nix-shell -p "haskellPackages.ghcWithPackages (pkgs: with pkgs; [ hostname ])"  
$ runghc Main.hs  
Hello from Haskell inside a Nix flake!  
Your hostname is: wombat11k
```

Success! Now we know the program works.

9.1.3. The cabal file

It's time to write a Cabal file for this program. This is just an ordinary Cabal file; we don't need to do anything special for Nix.

hello-flake-haskell.cabal



CABAL

```
1 cabal-version: 3.0
2 name: hello-flake-haskell
3 version: 1.0.0
4 synopsis: A simple demonstration using a Nix flake to package a Haskell program that prints a greeting.
5 description:
6   For more information and a tutorial on how to use this package,
7   please see the README at <https://codeberg.org/mhwombat/hello-flake-haskell#readme>.
8 homepage: https://codeberg.org/mhwombat/hello-flake-haskell
9 bug-reports: https://codeberg.org/mhwombat/hello-flake-haskell/issues
10 license: GPL-3.0-only
11 license-file: LICENSE
12 author: Amy de Buitléir
13 maintainer: amy@nualeargais.ie
14 copyright: (c) 2023 Amy de Buitléir
15 category: Text
16 build-type: Simple
17
18 executable hello-flake-haskell
19   main-is: Main.hs
20   build-depends:
21     base,
22     hostname
23 -- NOTE: Best practice is to specify version constraints for the packages we depend on.
24 -- However, I'm confident that this package will only be used as a Nix flake.
25 -- Nix will automatically ensure that anyone running this program is using the
26 -- same library versions that I used to build it.
```

9.1.4. (Optional) Building and running with cabal-install

At this point, I would normally write `flake.nix` and use Nix to build the program. I'll cover that in the next section. However, it's useful to know how to build the package manually in a Nix environment, without using a Nix flake. When you're new to Nix, this can help you differentiate between problems in your flake definition and problems in your Cabal file.

```
$ cabal build
sh: line 35: cabal: command not found
```

Aha! We need `cabal-install` in our shell. Rather than launch another shell-within-a-shell, let's exit create a new one.



```
$ exit
$ nix-shell -p "[ cabal-install (haskellPackages.ghcWithPackages (pkgs: with pkgs; [ hostname ]))]"
$ cabal build
Warning: The package list for 'hackage.haskell.org' is 24 days old.
Run 'cabal update' to get the latest list of available packages.
Resolving dependencies...
Build profile: -w ghc-9.4.8 -O1
In order, the following will be built (use -v for more details):
- hello-flake-haskell-1.0.0 (exe:hello-flake-haskell) (first run)
Configuring executable 'hello-flake-haskell' for hello-flake-haskell-1.0.0..
Warning: Packages using 'cabal-version: >= 1.10' and before 'cabal-version: 3.4' must specify the 'default-language' field for each component (e.g. Haskell98 or Haskell2010). If a component uses different languages in different modules then list the other ones in the 'other-languages' field.
Warning: The 'license-file' field refers to the file 'LICENSE' which does not exist.
Preprocessing executable 'hello-flake-haskell' for hello-flake-haskell-1.0.0..
Building executable 'hello-flake-haskell' for hello-flake-haskell-1.0.0..
[1 of 1] Compiling Main           ( Main.hs, /home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-haskell-1.0.0/Main.hs )
[2 of 2] Linking /home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-haskell/dist-newstyle/build/x86_64-linux-gnu/hello-haskell-1.0.0/hello-haskell
$ cabal run
Hello from Haskell inside a Nix flake!
Your hostname is: wombat11k
$ exit
```

After a lot of output messages, the build succeeds and the program runs.

9.1.5. The Nix flake

Now we should write `flake.nix`. We already know how to write most of the flake from the examples we did earlier. The two parts that would be different are the development shell and the package builder.

However, there's a simpler way, using `haskell-flake`.

`flake.nix`

```
1 {
2   description = "a flake using Haskell";
3
4   # This example uses haskell-flake to make things simpler.
5   # See https://haskell.flake.page/ for more information and examples.
6
7   inputs = {
8     nixpkgs.url = "github:nixos/nixpkgs/nixpkgs-unstable";
9     flake-parts.url = "github:hercules-ci/flake-parts";
10    haskell-flake.url = "github:srid/haskell-flake";
11  };
12  outputs = inputs@{ self, nixpkgs, flake-parts, ... }: {
13    flake-parts.lib.mkFlake { inherit inputs; } {
14      systems = nixpkgs.lib.systems.flakeExposed;
15      imports = [ inputs.haskell-flake.flakeModule ];
16
17      perSystem = { self', pkgs, ... }: {
18        haskellProjects.default = {};
19
20        # haskell-flake doesn't set the default package, but you can do it here.
21        packages.default = self'.packages.hello-flake-haskell;
22      };
23    };
24 }
```

NIX



The above definition will work for most of your haskell projects; simply change the `description` and the package name in `packages.default`. Let's try out the new flake.

```
$ nix run
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-haskell' is dirty
error: getting status of '/nix/store/0ccnxa25whszw7mgbgyzdm4nqc0zwnm8-source/flake.nix': No such file or directory
```

Why can't it find `flake.nix`? Nix flakes only "see" files that are part of the repository. We need to add all of the important files to the repo before building or running the flake.

```
$ git add flake.nix hello-flake-haskell.cabal Main.hs
$ nix run
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-haskell' is dirty
warning: creating lock file '/home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-haskell/flake.lock'
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-haskell' is dirty
these 2 derivations will be built:
  /nix/store/qhb3mvp8i87n58iwi3ldkwpin2m9zgya-source-hello-flake-haskell-sdist.tar.gz.drv
  /nix/store/8qdbmfms1h0b60aqdxfk28fmdnlkcm1l-hello-flake-haskell-1.0.0.drv
building '/nix/store/qhb3mvp8i87n58iwi3ldkwpin2m9zgya-source-hello-flake-haskell-sdist.tar.gz.drv'...
error: builder for '/nix/store/qhb3mvp8i87n58iwi3ldkwpin2m9zgya-source-hello-flake-haskell-sdist.tar.gz.drv' failed \n
      last 7 log lines:
        > unpacking source archive /nix/store/gg6b20p83m5mqcfp1qr0w37bjhz3k33y-source-hello-flake-haskell
        > source root is source-hello-flake-haskell
        > Config file path source is default config file.
        > Config file not found: /build/source-hello-flake-haskell/.config/cabal/config
        > Writing default configuration to
        > /build/source-hello-flake-haskell/.config/cabal/config
        > /build/source-hello-flake-haskell/.LICENSE: withBinaryFile: does not exist (No such file or directory)
        For full logs, run 'nix log /nix/store/qhb3mvp8i87n58iwi3ldkwpin2m9zgya-source-hello-flake-haskell-sdist.tar.gz.drv'
error: 1 dependencies of derivation '/nix/store/8qdbmfms1h0b60aqdxfk28fmdnlkcm1l-hello-flake-haskell-1.0.0.drv' failed
```

We'd like to share this package with others, but first we should do some cleanup. When the package was built (automatically by the `nix run` command), it created a `flake.lock` file. We need to add this to the repo, and commit all important files.

```
$ git add flake.lock
$ git commit -a -m 'initial commit'
[master (root-commit) 666b827] initial commit
 4 files changed, 137 insertions(+)
create mode 100644 Main.hs
create mode 100644 flake.lock
create mode 100644 flake.nix
create mode 100644 hello-flake-haskell.cabal
```

You can test that your package is properly configured by going to another directory and running it from there.



```
$ cd ..
$ nix run ./hello-haskell
these 2 derivations will be built:
  /nix/store/qhb3mvp8i87n58iwi3ldkwpin2m9zgya-source-hello-flake-haskell-sdist.tar.gz.drv
  /nix/store/8qdbmfms1h0b60aqdxfk28fmdnlkcm1l-hello-flake-haskell-1.0.0.drv
building '/nix/store/qhb3mvp8i87n58iwi3ldkwpin2m9zgya-source-hello-flake-haskell-sdist.tar.gz.drv'...
error: builder for '/nix/store/qhb3mvp8i87n58iwi3ldkwpin2m9zgya-source-hello-flake-haskell-sdist.tar.gz.drv' failed \n
last 7 log lines:
> unpacking source archive /nix/store/gg6b20p83m5mqcfp1qr0w37bjhz3k33y-source-hello-flake-haskell
> source root is source-hello-flake-haskell
> Config file path source is default config file.
> Config file not found: /build/source-hello-flake-haskell/.config/cabal/config
> Writing default configuration to
> /build/source-hello-flake-haskell/.config/cabal/config
> /build/source-hello-flake-haskell/.LICENSE: withBinaryFile: does not exist (No such file or directory)
For full logs, run 'nix log /nix/store/qhb3mvp8i87n58iwi3ldkwpin2m9zgya-source-hello-flake-haskell-sdist.tar.gz.drv'
error: 1 dependencies of derivation '/nix/store/8qdbmfms1h0b60aqdxfk28fmdnlkcm1l-hello-flake-haskell-1.0.0.drv' failed
```

If you move the project to a public repo, anyone can run it. Recall from the beginning of the tutorial that you were able to run `hello-flake` directly from my repo with the following command.

```
nix run "git+https://codeberg.org/mhwombat/hello-flake"
```

Modify the URL accordingly and invite someone else to run your new Haskell flake.

9.2. Python

Start with an empty directory and create a git repository.

```
$ mkdir hello-python
$ cd hello-python
$ git init
Initialized empty Git repository in /home/amy/codeberg/nix-book/source/new-flake/python-flake/hello-python/.git/
```

9.2.1. A simple Python program

Next, we'll create a simple Python program.

hello.py

```
1 #!/usr/bin/env python
2
3 def main():
4     print("Hello from inside a Python program built with a Nix flake!")
5
6 if __name__ == "__main__":
7     main()
```

PYTHON



Before we package the program, let's verify that it runs. We're going to need Python. By now you've probably figured out that we can write a `flake.nix` and define a development shell that includes Python. We'll do that shortly, but first I want to show you a handy shortcut. We can launch a *temporary* shell with any Nix packages we want. This is convenient when you just want to try out some new software and you're not sure if you'll use it again. It's also convenient when you're not ready to write `flake.nix` (perhaps you're not sure what tools and packages you need), and you want to experiment a bit first.

The command to enter a temporary shell is

```
nix-shell -p packages
```

If there are multiple packages, they should be separated by spaces.

 The command used here is `nix-shell` with a hyphen, not `nix shell` with a space; those are two different commands. In fact there are hyphenated and non-hyphenated versions of many Nix commands, and yes, it's confusing. The non-hyphenated commands were introduced when support for flakes was added to Nix. I predict that eventually all hyphenated commands will be replaced with non-hyphenated versions. Until then, a useful rule of thumb is that non-hyphenated commands are for working directly with flakes; hyphenated commands are for everything else.

Let's enter a shell with Python so we can test the program.

```
$ nix-shell -p python3
$ python hello.py
Hello from inside a Python program built with a Nix flake!
```

9.2.2. A Python builder

Next, create a Python script to build the package. We'll use Python's setuptools, but you can use other build tools. For more information on setuptools, see the [Python Packaging User Guide](#)

(<https://packaging.python.org/en/latest/guides/distributing-packages-using-setuptools/>), especially the section on [setup args](#) (<https://packaging.python.org/en/latest/guides/distributing-packages-using-setuptools/#setup-args>).

setup.py

```
1 #!/usr/bin/env python
2
3 from setuptools import setup
4
5 setup(
6     name='hello-flake-python',
7     version='0.1.0',
8     py_modules=['hello'],
9     entry_points={
10         'console_scripts': ['hello-flake-python = hello:main']
11     },
12 )
```

PYTHON

We won't write `flake.nix` just yet. First we'll try building the package manually.

```
$ python -m build
/nix/store/qp5zys77biz7imbk6yy85q5pdv7qk84j-python3-3.11.6/bin/python: No module named build
```



The missing module error happens because we don't have `build` available in the temporary shell. We can fix that by adding "build" to the temporary shell. When you need support for both a language and some of its packages, it's best to use one of the Nix functions that are specific to the programming language and build system. For Python, we can use the `withPackages` function.

```
$ nix-shell -p "python3.withPackages (ps: with ps; [ build ])"
```

Note that we're now inside a temporary shell inside the previous temporary shell! To get back to the original shell, we have to `exit` twice. Alternatively, we could have done `exit` followed by the `nix-shell` command.

```
$ python -m build
```

After a lot of output messages, the build succeeds.

9.2.3. The Nix flake

Now we should write `flake.nix`. We already know how to write most of the flake from the examples we did earlier. The two parts that will be different are the development shell and the package builder.

Let's start with the development shell. It seems logical to write something like the following.

```
devShells = rec {
  default = pkgs.mkShell {
    packages = [ (python.withPackages (ps: with ps; [ build ])) ];
  };
};
```

Note that we need the parentheses to prevent `python.withPackages` and the argument from being processed as two separate tokens. Suppose we wanted to work with `virtualenv` and `pip` instead of `build`. We could write something like the following.

```
devShells = rec {
  default = pkgs.mkShell {
    packages = [
      # Python plus helper tools
      (python.withPackages (ps: with ps; [
        virtualenv # Virtualenv
        pip # The pip installer
      ]))
    ];
  };
};
```

For the package builder, we can use the `buildPythonApplication` function.

```
packages = rec {
  hello = python.pkgs.buildPythonApplication {
    name = "hello-flake-python";
    buildInputs = with python.pkgs; [ pip ];
    src = ./;
  };
  default = hello;
};
```



If you put all the pieces together, your `flake.nix` should look something like this.

flake.nix

```

1 {
2 # See https://github.com/mhwombat/nix-for-numbskulls/blob/main/flakes.md
3 # for a brief overview of what each section in a flake should or can contain.
4
5 description = "a very simple and friendly flake written in Python";
6
7 inputs = {
8   nixpkgs.url = "github:NixOS/nixpkgs";
9   flake-utils.url = "github:numtide/flake-utils";
10 };
11
12 outputs = { self, nixpkgs, flake-utils }:
13   flake-utils.lib.eachDefaultSystem (system:
14     let
15       pkgs = import nixpkgs { inherit system; };
16       python = pkgs.python3;
17     in
18     {
19       devShells = rec {
20         default = pkgs.mkShell {
21           packages = [
22             # Python plus helper tools
23             (python.withPackages (ps: with ps; [
24               virtualenv # Virtualenv
25               pip # The pip installer
26             ]))
27           ];
28         };
29       };
30
31       packages = rec {
32         hello = python.pkgs.buildPythonApplication {
33           name = "hello-flake-python";
34
35           buildInputs = with python.pkgs; [ pip ];
36
37           src = ./;
38         };
39         default = hello;
40       };
41
42       apps = rec {
43         hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };
44         default = hello;
45       };
46     };
47   );
48 }

```

Let's try out the new flake.



```
$ nix run
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/python-flake/hello-python' is dirty
error: getting status of '/nix/store/0ccnxa25whszw7mgbgyzdm4nqc0zwnm8-source/flake.nix': No such file or directory
```

Why can't it find `flake.nix`? Nix flakes only “see” files that are part of the repository. We need to add all of the important files to the repo before building or running the flake.

```
$ git add flake.nix setup.py hello.py
$ nix run
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/python-flake/hello-python' is dirty
warning: creating lock file '/home/amy/codeberg/nix-book/source/new-flake/python-flake/hello-python/flake.lock'
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/python-flake/hello-python' is dirty
Hello from inside a Python program built with a Nix flake!
```

We'd like to share this package with others, but first we should do some cleanup. When the package was built (automatically by the `nix run` command), it created a `flake.lock` file. We need to add this to the repo, and commit all important files.

```
$ git add flake.lock
$ git commit -a -m 'initial commit'
[master (root-commit) ddb5606] initial commit
 4 files changed, 127 insertions(+)
 create mode 100644 flake.lock
 create mode 100644 flake.nix
 create mode 100644 hello.py
 create mode 100644 setup.py
```

You can test that your package is properly configured by going to another directory and running it from there.

```
$ cd ..
$ nix run ./hello-python
Hello from inside a Python program built with a Nix flake!
```

If you move the project to a public repo, anyone can run it. Recall from the beginning of the tutorial that you were able to run `hello-flake` directly from my repo with the following command.

```
nix run "git+https://codeberg.org/mhwombat/hello-flake"
```

Modify the URL accordingly and invite someone else to run your new Python flake.



10. Recipes

This chapter provides examples of how to use Nix in a variety of scenarios. Multiple types of recipes are provided are provided for some scenarios; comparing the different recipes will help you better understand Nix.

- An "*ad hoc*" shell is useful when you want to quickly create an environment for a one-off task.
- A *traditional nix shell* is useful when you want to define an environment that you will use more than once.
- *Nix flakes* are the recommended approach for development projects.
- You can use `nix-shell` to run scripts in arbitrary languages, providing the necessary dependencies. This is particularly convenient for standalone scripts because you don't need to create a repo and write a separate `flake.nix`. The script should start with two "*shebang*" (`#!`) commands. The first should invoke `nix-shell`. The second should declare the script interpreter and any dependencies.

10.1. Access to a top-level package from the Nixpkgs/NixOS repo

Ex: Access two packages from nixpkgs: hello and cowsay.

10.1.1. From the command line

```
$ nix-shell -p "[hello cowsay]"
$ hello
Hello, world!
$ cowsay "moo"
-----
< moo >
-----
 \   ^__^
  \  (oo)\_____
    (__)\       )\/\
        ||----w |
        ||     ||
```

10.1.2. In `shell.nix`

`shell.nix`

```
1 | with (import <nixpkgs> {});
2 | mkShell {
3 |   buildInputs = [
4 |     hello
5 |     cowsay
6 |   ];
7 | }
```

NIX

Here's a demonstration using the shell.



```
$ nix-shell
$ hello
Hello, world!
$ cowsay "moo"

< moo >
-----
 \  ^__^
  \  (oo)\_____
   (__)\       )\/\
    ||----w |
     ||     ||
```

10.1.3. In a Bash script

Script

```
1 #! /usr/bin/env nix-shell
2 #! nix-shell -i bash -p "[hello cowsay]"
3 hello
4 cowsay "Pretty cool, huh?"
```

BASH

Output

```
Hello, world!

< Pretty cool, huh? >
-----
 \  ^__^
  \  (oo)\_____
   (__)\       )\/\
    ||----w |
     ||     ||
```

10.2. Access to a package defined in a remote git repo

Ex: Access a package called `hello-nix`, which is defined in a remote git repo on codeberg. To use a package from GitHub, GitLab, or any other public platform, modify the URL.

10.2.1. In `shell.nix`

shell.nix

```
1 with (import <nixpkgs> {});
2 let
3   hello-nix = import (builtins.fetchGit {
4     url = "https://codeberg.org/mhwombat/hello-nix";
5     rev = "aa2c87f8b89578b069b09fdb2be30a0c9d8a77d8";
6   });
7   in
8   mkShell {
9     buildInputs = [ hello-nix ];
10 }
```

NIX



Here's a demonstration using the shell.

```
$ nix-shell
$ hello-nix
Hello from your nix package!
```

10.3. Access to a flake defined in a remote git repo

Ex: Access a flake called `hello-flake`, which is defined in a remote git repo on codeberg. To use a package from GitHub, GitLab, or any other public platform, modify the URL.

10.3.1. In `shell.nix`

`shell.nix`

```
1 with (import <nixpkgs> {});
2 let
3   hello-flake = ( builtins.getFlake
4     git+https://codeberg.org/mhwombat/hello-flake?ref=main&rev=3aa43dbe7be878dde7b2bdcbe992fe17
5     ).packages.${builtins.currentSystem}.default;
6 in
7 mkShell {
8   buildInputs = [
9     hello-flake
10  ];
11 }
```

NIX

Here's a demonstration using the shell.

```
$ nix-shell
$ hello-flake
Hello from your flake!
```

10.4. Access to a Haskell library package in the `nixpkgs` repo (without a `.cabal` file)

Occasionally you might want to run a short Haskell program that depends on a Haskell library, but you don't want to bother writing a cabal file.

Example: Access the `containers` package from the `haskellPackages` set in the `nixpkgs` repo.

10.4.1. In `shell.nix`

`shell.nix`

```
1 with (import <nixpkgs> {});
2 let
3   customGhc = haskellPackages.ghcWithPackages (pkgs: with pkgs; [ containers ]);
4 in
5 mkShell {
6   buildInputs = [
7     customGhc
8   ];
9 }
```

NIX


Here's a short Haskell program that uses it.

`Main.hs`

HASKELL

```

1 import Data.Map
2
3 m :: Map String Int
4 m = fromList [("cats", 3), ("dogs", 2)]
5
6 main :: IO ()
7 main = do
8   let cats = findWithDefault 0 "cats" m
9   let dogs = findWithDefault 0 "dogs" m
10  let zebras = findWithDefault 0 "zebras" m
11  print $ "I have " ++ show cats ++ " cats, " ++ show dogs ++ " dogs, and " ++ show zebras ++ " zebras."

```

Here's a demonstration using the program.

```

$ nix-shell
$ runghc Main.hs
"I have 3 cats, 2 dogs, and 0 zebras."

```

10.4.2. In a Haskell script

Script

```

1 #! /usr/bin/env nix-shell
2 #! nix-shell -p "haskellPackages.ghcWithPackages (p: [p.containers])"
3 #! nix-shell -i runghc
4
5 import Data.Map
6
7 m :: Map String Int
8 m = fromList [("cats", 3), ("dogs", 2)]
9
10 main :: IO ()
11 main = do
12   let cats = findWithDefault 0 "cats" m
13   let dogs = findWithDefault 0 "dogs" m
14   let zebras = findWithDefault 0 "zebras" m
15   print $ "I have " ++ show cats ++ " cats, " ++ show dogs ++ " dogs, and " ++ show zebras ++ " zebras."

```

HASkELL

Output

```
"I have 3 cats, 2 dogs, and 0 zebras."
```

10.5. Access to a Haskell package on your local computer

Ex: Access three Haskell packages (`pandoc-linear-table`, `pandoc-logic-proof`, and `pandoc-columns`) that are on my hard drive.

10.5.1. In `shell.nix`

shell.nix



NIX

```

1 with (import <nixpkgs> {});
2 let
3   pandoc-linear-table = haskellPackages.callCabal2nix "pandoc-linear-table" /home/amy/github/pandoc-linear-table
4   pandoc-logic-proof = haskellPackages.callCabal2nix "pandoc-logic-proof" /home/amy/github/pandoc-logic-proof {}
5   pandoc-columns = haskellPackages.callCabal2nix "pandoc-columns" /home/amy/github/pandoc-columns {};
6 in
7 mkShell {
8   buildInputs = [
9     pandoc
10    pandoc-linear-table
11    pandoc-logic-proof
12    pandoc-columns
13  ];
14 }

```

10.6. Access to a Haskell package on your local computer, with inter-dependencies

Ex: Access four Haskell packages (`pandoc-linear-table` , `pandoc-logic-proof` , `pandoc-columns` and `pandoc-maths-web`) that are on my hard drive. The fourth package depends on the first three to build.

10.6.1. In `shell.nix`

shell.nix

```

1 with (import <nixpkgs> {});
2 let
3   pandoc-linear-table = haskellPackages.callCabal2nix "pandoc-linear-table" /home/amy/github/pandoc-linear-table
4   pandoc-logic-proof = haskellPackages.callCabal2nix "pandoc-logic-proof" /home/amy/github/pandoc-logic-proof {}
5   pandoc-columns = haskellPackages.callCabal2nix "pandoc-columns" /home/amy/github/pandoc-columns {};
6   pandoc-maths-web = haskellPackages.callCabal2nix "pandoc-maths-web" /home/amy/github/pandoc-maths-web
7   {
8     inherit pandoc-linear-table pandoc-logic-proof pandoc-columns;
9   };
10 in
11 mkShell {
12   buildInputs = [
13     pandoc
14     pandoc-linear-table
15     pandoc-logic-proof
16     pandoc-columns
17     pandoc-maths-web
18   ];
19 }

```

10.7. Access to a Python library package in the nixpkgs repo (without using a Python builder)

Occasionally you might want to run a short Python program that depends on a Python library, but you don't want to bother configuring a builder.

Example: Access the `html_sanitizer` package from the `python3nnPackages` set in the nixpkgs repo.

10.7.1. In a Python script

Script



PYTHON

```
1 #! /usr/bin/env nix-shell
2 #! nix-shell -i python3 -p python3Packages.html-sanitizer
3
4 from html_sanitizer import Sanitizer
5 sanitizer = Sanitizer() # default configuration
6
7 original='<span style="font-weight:bold">some text</span>'
8 print('original: ', original)
9
10 sanitized=sanitizer.sanitize(original)
11 print('sanitized: ', sanitized)
```

Output

```
original: <span style="font-weight:bold">some text</span>
sanitized: <strong>some text</strong>
```

10.8. Set an environment variable

Ex: Set the value of the environment variable FOO to “bar”

10.8.1. In `shell.nix`

`shell.nix`

```
1 with (import <nixpkgs> {});
2 mkShell {
3   shellHook = ''
4     export FOO="bar"
5   '';
6 }
```

NIX

1. REPL is an acronym for (Read-Eval-Print-Loop).

2. For more information on the standard environment, see the [Nixpkgs manual](https://nixos.org/manual/nixpkgs/stable/#sec-tools-of-stdenv) (<https://nixos.org/manual/nixpkgs/stable/#sec-tools-of-stdenv>)

Last updated 2023-12-03 19:55:00 GMT



Graham Christensen

Erase your darlings

immutable infrastructure for mutable systems

posted on April 13 2020

I erase my systems at every boot.

Over time, a system collects state on its root partition. This state lives in assorted directories like `/etc` and `/var`, and represents every under-documented or out-of-order step in bringing up the services.

“Right, run `myapp-init`.”

These small, inconsequential “oh, oops” steps are the pieces that get lost and don’t appear in your runbooks.

“Just download ca-certificates to … to fix …”

Each of these quick fixes leaves you doomed to repeat history in three years when you’re finally doing that dreaded RHEL 7 to RHEL 8 upgrade.

“Oh, touch `/etc/ipsec.secrets` or the l2tp tunnel won’t work.”

Immutable infrastructure gets us *so close*

Immutable infrastructure is a wonderfully effective method of eliminating so many of these forgotten steps. Leaning in to the pain by deleting and replacing your servers on a weekly or monthly basis means you are constantly testing and exercising your automation and runbooks.

The nugget here is the regular and indiscriminate removal of system state. Destroying the whole server doesn’t leave you much room to forget the little



tweaks you made along the way.

These techniques work great when you meet two requirements:

- you can provision and destroy servers with an API call
- the servers aren't inherently stateful

Long running servers

There are lots of cases in which immutable infrastructure *doesn't* work, and the dirty secret is **those servers need good tools the most**.

Long-running servers cause long outages. Their runbooks are outdated and incomplete. They accrete tweaks and turn in to an ossified, brittle snowflake — except its arms are load-bearing.

Let's bring the ideas of immutable infrastructure to these systems too. Whether this system is embedded in a stadium's jumbotron, in a datacenter, or under your desk, we *can* keep the state under control.

FHS isn't enough

The hard part about applying immutable techniques to long running servers is knowing exactly where your application state ends and the operating system, software, and configuration begin.

This is hard because legacy operating systems and the Filesystem Hierarchy Standard poorly separate these areas of concern. For example, `/var/lib` is for state information, but how much of this do you actually care about tracking? What did you configure in `/etc` on purpose?



The answer is probably not a lot.

You may not care, but all of this accumulation of junk is a tarpit. Everything becomes harder: replicating production, testing changes, undoing mistakes.

New computer smell

Getting a new computer is this moment of cleanliness. The keycaps don't have oils on them, the screen is perfect, and the hard drive is fresh and unspoiled — for about an hour or so.

Let's get back to that.

How is this possible?

NixOS can boot with only two directories: `/boot`, and `/nix`.

`/nix` contains read-only system configurations, which are specified by your `configuration.nix` and are built and tracked as system generations. These never change. Once the files are created in `/nix`, the only way to change the config's contents is to build a new system configuration with the contents you want.

Any configuration or files created on the drive outside of `/nix` is state and cruft. We can lose everything outside of `/nix` and `/boot` and have a healthy system. My technique is to explicitly opt in and *choose* which state is important, and only keep that.

How this is possible comes down to the boot sequence.

For NixOS, the bootloader follows the same basic steps as a standard Linux distribution: the kernel starts with an initial ramdisk, and the initial ramdisk mounts the system disks.

And here is where the similarities end.

NixOS's early startup

NixOS configures the bootloader to pass some extra information: a specific system configuration. This is the secret to NixOS's bootloader rollbacks, and also the key to erasing our disk on each boot. The parameter is named `systemConfig`.

On every startup the very early boot stage knows what the system's configuration should be: the entire system configuration is stored in the read-only `/nix/store`,



and the directory passed through `systemConfig` has a reference to the config. Early boot then manipulates `/etc` and `/run` to match the chosen setup. Usually this involves swapping out a few symlinks.

If `/etc` simply doesn't exist, however, early boot *creates* `/etc` and moves on like it were any other boot. It also *creates* `/var`, `/dev`, `/home`, and any other core directories that must be present.

Simply speaking, an empty `/` is *not surprising* to NixOS. In fact, the NixOS netboot, EC2, and installation media all start out this way.

Opting out

Before we can opt in to saving data, we must opt out of saving data *by default*. I do this by setting up my filesystem in a way that lets me easily and safely erase the unwanted data, while preserving the data I do want to keep.

My preferred method for this is using a ZFS dataset and rolling it back to a blank snapshot before it is mounted. A partition of any other filesystem would work just as well too, running `mkfs` at boot, or something similar. If you have a lot of RAM, you could skip the erase step and make `/` a tmpfs.

Opting out with ZFS

When installing NixOS, I partition my disk with two partitions, one for the boot partition, and another for a ZFS pool. Then I create and mount a few datasets.

My root dataset:

```
# zfs create -p -o mountpoint=legacy rpool/local/root
```



Before I even mount it, I **create a snapshot while it is totally blank**:

```
# zfs snapshot rpool/local/root@blank
```

And then mount it:

```
# mount -t zfs rpool/local/root /mnt
```

Then I mount the partition I created for the /boot:

```
# mkdir /mnt/boot
# mount /dev/the-boot-partition /mnt/boot
```

Create and mount a dataset for /nix:

```
# zfs create -p -o mountpoint=legacy rpool/local/nix
# mkdir /mnt/nix
# mount -t zfs rpool/local/nix /mnt/nix
```

And a dataset for /home:

```
# zfs create -p -o mountpoint=legacy rpool/safe/home
# mkdir /mnt/home
# mount -t zfs rpool/safe/home /mnt/home
```

And finally, a dataset explicitly for state I want to persist between boots:

```
# zfs create -p -o mountpoint=legacy rpool/safe/persist
# mkdir /mnt/persist
# mount -t zfs rpool/safe/persist /mnt/persist
```

Note: in my systems, datasets under rpool/local are never backed up, and datasets under rpool/safe are.

And now safely erasing the root dataset on each boot is very easy: after devices are made available, roll back to the blank snapshot:

```
{
    boot.initrd.postDeviceCommands = lib.mkAfter ''
        zfs rollback -r rpool/local/root@blank
    '';
}
```



I then finish the installation as normal. If all goes well, your next boot will start

with an empty root partition but otherwise be configured exactly as you specified.

Opting in

Now that I'm keeping no state, it is time to specify what I do want to keep. My choices here are different based on the role of the system: a laptop has different state than a server.

Here are some different pieces of state and how I preserve them. These examples largely use reconfiguration or symlinks, but using ZFS datasets and mount points would work too.

Wireguard private keys

Create a directory under `/persist` for the key:

```
# mkdir -p /persist/etc/wireguard/
```

And use Nix's wireguard module to generate the key there:

```
{
  networking.wireguard.interfaces.wg0 = {
    generatePrivateKeyFile = true;
    privateKeyFile = "/persist/etc/wireguard/wg0";
  };
}
```

NetworkManager connections

Create a directory under `/persist`, mirroring the `/etc` structure:



```
# mkdir -p /persist/etc/NetworkManager/system-connections
```

And use Nix's `etc` module to set up the symlink:

```
{
  etc."NetworkManager/system-connections" = {
```

```
source = "/persist/etc/NetworkManager/system-connections/";  
};  
}
```

Bluetooth devices

Create a directory under `/persist`, mirroring the `/var` structure:

```
# mkdir -p /persist/var/lib/bluetooth
```

And then use systemd's tmpfiles.d rules to create a symlink from `/var/lib/bluetooth` to my persisted directory:

```
{  
    systemd.tmpfiles.rules = [  
        "L /var/lib/bluetooth - - - - /persist/var/lib/bluetooth"  
    ];  
}
```

SSH host keys

Create a directory under `/persist`, mirroring the `/etc` structure:

```
# mkdir -p /persist/etc/ssh
```

And use Nix's openssh module to create and use the keys in that directory:

```
{  
    services.openssh = {  
        enable = true;  
        hostKeys = [  
            {  
                path = "/persist/ssh/ssh_host_ed25519_key";  
                type = "ed25519";  
            }  
        ];  
    };  
}
```



```

        path = "/persist/ssh/ssh_host_rsa_key";
        type = "rsa";
        bits = 4096;
    }
];
};

}

```

ACME certificates

Create a directory under `/persist`, mirroring the `/var` structure:

```
# mkdir -p /persist/var/lib/acme
```

And then use systemd's tmpfiles.d rules to create a symlink from `/var/lib/acme` to my persisted directory:

```
{
    systemd.tmpfiles.rules = [
        "L /var/lib/acme - - - /persist/var/lib/acme"
    ];
}
```

Answering the question “what am I about to lose?”

I found this process a bit scary for the first few weeks: was I losing important data each reboot? No, I wasn't.

If you're worried and want to know what state you'll lose on the next boot, you can list the files on your root filesystem and see if you're missing something important

```
# tree -x /
├── bin
│   └── sh -> /nix/store/97zzcs494vn5k2yw-dash-0.5.10.2/bin/dash
└── dev
```



```
|── etc
|   └── asound.conf -> /etc/static/asound.conf
... snip ...
```

ZFS can give you a similar answer:

```
# zfs diff rpool/local/root@blank
M      /
+
+      /nix
+      /etc
+      /root
+      /var/lib/is-nix-channel-up-to-date
+      /etc/pki/fwupd
+      /etc/pki/fwupd-metadata
...
... snip ...
```

Your stateless future

You may bump in to new state you meant to be preserving. When I'm adding new services, I think about the state it is writing and whether I care about it or not. If I care, I find a way to redirect its state to `/persist`.

Take care to reboot these machines on a somewhat regular basis. It will keep things agile, proving your system state is tracked correctly.

This technique has given me the “new computer smell” on every boot without the datacenter full of hardware, and even on systems that do carry important state. I have deployed this strategy to systems in the large and small: build farm servers, database servers, my NAS and home server, my raspberry pi garage door opener, and laptops.



NixOS enables powerful new deployment models in so many ways, allowing for systems of all shapes and sizes to be managed properly and consistently. I think this model of ephemeral roots is yet another example of this flexibility and power. I would like to see this partitioning scheme become a reference architecture and take us out of this eternal tarpit of legacy.

Posts

- [NixOS on the Framework](#)
- [Flakes are such an obviously good thing](#)
- [Erase your darlings](#)
- [ZFS Datasets for NixOS](#)
- [Optimising Docker Layers for Better Caching with Nix](#)
- [an EPYC NixOS build farm](#)
- [cache.nixos.org, now more local!](#)
- [Prometheus and the NixOS System Version](#)
- [NixOS on a Dell 9560](#)
- [How to use a NixOS Linux Server for Time Machine Backups](#)
- [Pip Install with Docker and Fixing the ascii decode error](#)
- [Packer - Create AMI with EBS Volumes with VolumeType](#)
- [Enable certificate revocation in Chrome](#)
- [How to delete all \(or most\) jobs from a beanstalk tube from the shell](#)
- [Why a MySQL Slave Created from an LVM Snapshot Would Mark Tables Corrupt](#)
- [Listing Users with Database Access](#)

About

Graham works on [NixOS](#).

- **E-Mail:** graham@grahamc.com
- **Phone:** +1-407-670-9980
- **GitHub:** github.com/grahamc
- **Twitter:** [@grhmc](https://twitter.com/@grhmc)



© 2023 Graham Christensen. All Rights Reserved. · [Subscribe to RSS](#)

[Are you looking for IT consultancy services in the NixOS, Linux, Kubernetes, DevOps space? Check out Taserud Consulting AB.](#)

- [~elis/](#)
- [./about/](#)
- [./work/](#)
- [./talks/](#)
- [./blog/](#)

NixOS *: tmpfs as root

2020-05-02 · 6 minutes read · [NixOSLinuxtmpfsImpermanence](#)

This post covers both EFI and legacy boot setups.

One fairly unique property of NixOS is the ability to boot with only `/boot` and `/nix`. Nothing else is actually required. This supports doing all sorts of weird things with your root file system.

One way is to do like [Graham's post "erase your darlings"](#) describes and empty your root file system each boot using ZFS snapshots. This way have some cool things that you could do on top of his setup, such as doing snapshots when it's running and roll-back to empty on boot. That way you actually can go back to recover files you lost but still have an empty state.

Another way is to go the tmpfs way which is probably why you're here. So I'm going to walk through the install with tmpfs as root file system.

Step 1 - Partitioning on EFI

I'm going to do the most basic setup when it comes to file systems, just a `/boot` as `fat32` and `/nix` as `ext4` without encryption. If you want to have another file system or use LUKS or something it should be trivial to just format the drive differently and mount it.

```
# Defining a helper variable to make the following
# commands shorter.
DISK=/dev/disk/by-id/ata-VENDOR-ID-OF-THE-DRIVE

# Create partition table
parted $DISK -- mklabel gpt

# Create a /boot as $DISK-part1
parted $DISK -- mkpart ESP fat32 1MiB 512MiB
parted $DISK -- set 1 boot on

# Create a /nix as $DISK-part2
parted $DISK -- mkpart Nix 512MiB 100%
```



Step 1 - Partitioning for legacy boot

I'm going to do the most basic setup when it comes to file systems, just a `/boot` as ext4 and `/nix` as ext4 without encryption. If you want to have another file system or use LUKS or something it should be trivial to just format the drive differently and mount it.

```
# Defining a helper variable to make the following
# commands shorter.
DISK=/dev/disk/by-id/ATA-VENDOR-ID-OF-THE-DRIVE

# Create partition table
parted $DISK -- mklabel msdos

# Create a /boot as $DISK-part1
parted $DISK -- mkpart primary ext4 1M 512M
parted $DISK -- set 1 boot on

# Create a /nix as $DISK-part2
parted $DISK -- mkpart primary ext4 512MiB 100%
```

Step 2 - Creating the file systems

This is fairly straight forward in my example:

```
# /boot partition for EFI
mkfs.vfat $DISK-part1

# /boot partition for legacy boot
mkfs.ext4 $DISK-part1

# /nix partition
mkfs.ext4 $DISK-part2
```

Step 3 - Mounting the file systems

Here we do one of the neat tricks, we mount tmpfs instead of a partition and then we mount the partitions we just created.

```
# Mount your root file system
mount -t tmpfs none /mnt

# Create directories
mkdir -p /mnt/{boot,nix,etc/nixos,var/log}

# Mount /boot and /nix
mount $DISK-part1 /mnt/boot
mount $DISK-part2 /mnt/nix

# Create a directory for persistent directories
mkdir -p /mnt/nix/persist/{etc/nixos,var/log}

# Bind mount the persistent configuration / logs
mount -o bind /mnt/nix/persist/etc/nixos /mnt/etc/nixos
mount -o bind /mnt/nix/persist/var/log /mnt/var/log
```



Step 4 - Configuration

Then go ahead and do a `nixos-generate-config --root /mnt` to get a basic configuration for

your system.

Step 4.1 - Configure disks

One thing you *want* to do that isn't needed when you install on a normal file system is that you want to set options for your root file system.

So go ahead, open up `hardware-configuration.nix` and add the following options to your root file system.

The `size` is something you can adjust depending on how much garbage you are willing to store in ram until you run out of space on your root. 2G is usually big enough for most of my systems.

```
{
  # ...

  fileSystems."/" = {
    device = "none";
    fsType = "tmpfs";
    options = [ "defaults" "size=2G" "mode=755" ];
  };

  # ...
}
```

Step 4.2 - Configure users

When you have a system with a tmpfs root you have to configure all users and passwords in `configuration.nix`, otherwise you won't have any user or a password on the next boot.

You probably want to have immutable users as well since it doesn't make any sense to have mutability of users if it's going to reset anyways.

Note: Don't use the options `password` or `hashedPassword` for users because it won't work. It has to be the options named `initialPassword` or `initialHashedPassword`.

```
{
  # ...

  # Don't allow mutation of users outside of the config.
  users.mutableUsers = false;

  # Set a root password, consider using initialHashedPassword instead.
  #
  # To generate a hash to put in initialHashedPassword
  # you can do this:
  # $ nix-shell --run 'mkpasswd -m SHA-512 -s' -p mkpasswd
  users.users.root.initialPassword = "hunter2";

  # ...
}
```



Make sure to add your own user with a password. The password for the root user is of course optional. But it may be quite useful.

Step 4.3 - Configure persistent files / directories

You probably want to have some more persistent files than the two bind mounts we already have created during the setup.

Adding persistent files from etc:

```
{
# ...

# machine-id is used by systemd for the journal, if you don't
# persist this file you won't be able to easily use journalctl to
# look at journals for previous boots.
environment.etc."machine-id".source
= "/nix/persist/etc/machine-id";

# if you want to run an openssh daemon, you may want to store the
# host keys across reboots.
#
# For this to work you will need to create the directory yourself:
# $ mkdir /nix/persist/etc/ssh
environment.etc."ssh/ssh_host_rsa_key".source
= "/nix/persist/etc/ssh/ssh_host_rsa_key";
environment.etc."ssh/ssh_host_rsa_key.pub".source
= "/nix/persist/etc/ssh/ssh_host_rsa_key.pub";
environment.etc."ssh/ssh_host_ed25519_key".source
= "/nix/persist/etc/ssh/ssh_host_ed25519_key";
environment.etc."ssh/ssh_host_ed25519_key.pub".source
= "/nix/persist/etc/ssh/ssh_host_ed25519_key.pub";

# ...
}
```

From here you can probably figure out how to do more bind-mounts and symbolic links in `/etc` for the files you want to live across reboots.

A useful tool to discovering files in your tmpfs is `ncdu`, I tend to run `sudo ncd x /` to walk around the directory tree to see if there's something I want to make persistent.

You may want to make your `/home` persistent, that can be done by a mount or bind-mount. I have that as tmpfs as well, but that is probably enough content for it's own post.

Update: Now there's a follow-up post for putting tmpfs on `/home` as well, it's located  here: [NixOS *: tmpfs as home](#).

Step 4.4 - Configure the boot loader

Here you can choose your boot loader.

```
{
# ...
```

```
# Use systemd boot (EFI only)
boot.loader.systemd-boot.enable = true;
boot.loader.efi.canTouchEfiVariables = true;

# Use the GRUB 2 boot loader (Both EFI and legacy boot supported).
boot.loader.grub.enable = true;

# This is for GRUB in EFI mode
boot.loader.grub.efiSupport = true;
boot.loader.grub.device = "nodev";

# This is for GRUB for legacy boot
boot.loader.grub.version = 2;
boot.loader.grub.device = "/dev/sda";

# ...
}
```

Step 4.5 - Configure the rest of your system

You should make sure to go through `configuration.nix` to make all necessary configuration you want such as boot loader, hostname, timezone, keymap, personal user, network settings, etc.

Step 5 - Perform install

Perform the actual install. We can ignore setting the roots password at the end of the install since it won't be there after reboot anyway.

```
nixos-install --no-root-passwd
```

Copyright © 2010 Elis Hirwing.



Haumea

Filesystem-based module system for Nix

Haumea is not related to or a replacement for NixOS modules. It is closer to the module systems of traditional programming languages, with support for file hierarchy and visibility.

In short, haumea maps a directory of Nix files into an attribute set:

From	To
<pre> - foo/ - bar.nix - baz.nix - __internal.nix - bar.nix - _utils/ - foo.nix</pre>	<pre>{ foo = { bar = <...>; baz = <...>; }; bar = <...>; }</pre>

Haumea's source code is hosted on [GitHub](#) under the [MPL-2.0](#) license. Haumea bootstraps itself. You can see the entire implementation in the [src](#) directory.

Why Haumea?

- No more manual imports

Manually importing files can be tedious, especially when there are many of them. Haumea takes care of all of that by automatically importing the files into an attribute set.

- Modules

Haumea takes inspiration from traditional programming languages. Visibility makes it easy to create utility modules, and haumea makes self-referencing and creating fixed points a breeze with the introduction of `self`, `super`, and `root`.

- Organized directory layout

What you see is what you get. By default¹, the file tree will look exactly like the resulting attribute set.



- Extensibility

Changing how the files are loaded is as easy as specifying a `loader`, and the `transformer` option makes it possible to extensively manipulate the tree.

→ [Getting Started](#)

¹ Unless you are doing transformer magic



Getting Started

Haumea comes with a template for a simple Nix library. You can try out the template with:

```
nix flake init -t github:nix-community/haumea
```

This will generate `flake.nix` and some other relevant files in the current directory. Or if you want to create a new directory for this, run:

```
nix flake new <dir> -t github:nix-community/haumea
```

You can use haumea without the template by adding it to your flake inputs:

```
inputs = {
  haumea = {
    url = "github:nix-community/haumea/v0.2.2";
    inputs.nixpkgs.follows = "nixpkgs";
  };
  nixpkgs.url = "github:nix-community/nixpkgs.lib";
};
```

Haumea is pinned to a tag here so potential breaking changes in the main branch wouldn't break downstream consumers. See the [Versioning](#) chapter for information.

The rest of this chapter will be using this template.

In `flake.nix`, the main thing you want to look at is `lib`:

```
lib = haumea.lib.load {
  src = ./src;
  inputs = {
    inherit (nixpkgs) lib;
  };
};
```



`haumea.lib.load` is the main entry point of haumea. It loads a directory (`./src`) of Nix files¹ into an attribute set. You can see the result of this by running `nix eval .#lib`:

```
{ answer = 42; }
```

If you open `src/answer.nix`, you can see that it is a lambda that returns 42.

```
{ lib }:
```

```
lib.id 42
```

The `lib` here is provided by the `inputs` option of `load`:

```
inputs = {
  inherit (nixpkgs) lib;
};
```

The `lib.id 42` in `answer.nix` becomes `nixpkgs.lib.id 42`, which evaluates to 42.

By default, the file doesn't have to specify the inputs that it does not use, or even specify any inputs at all if it is not using any inputs. Both `{ }: 42` and `42` are valid in this case and will do exactly the same thing.

`self`, `super`, and `root` are special inputs that are always available. You might already be familiar with them based on the names. These names are reserved, haumea will throw an error if you try to override them.

The documentation for `load` explains this more thoroughly and talks about some workarounds.

`checks` works basically the same, just with `loadEvalTests` instead of `load`. You can run the checks with `nix flake check`.

¹ Non-Nix files can also be loaded using `matchers`



Versioning

Haumea follows [semantic versioning](#). Breaking changes can happen on the main branch at any time, so it is recommended to pin haumea to a specific tag.

A list of available versions can be found on the [GitHub releases](#) page.



API Reference

The following sections documents everything in the library.

If you are using haumea with flakes, that would be `haumea.lib`.



load

Source: [src/load.nix](#)

Type: `{ src, loader?, inputs?, transformer? } -> { ... }`

Arguments:

- `src : Path`

The directory to load files from.

- (optional) `loader : ({ self, super, root, ... } -> Path -> a) | [Matcher]`

Loader for the files, defaults to `loaders.default`. It can be either a function that loads of Nix file, or a list of `matchers` that allows you to load any type of file.

- (optional) `inputs : { ... }`

Extra inputs to be passed to the loader.

`self`, `super`, and `root` are reserved names that cannot be passed as an input. To work around that, remove them using `removeAttrs`, or pass them by overriding the loader.

- (optional) `transformer : (cursor : [String]) -> { ... } -> a` or a list of transformers

Module transformer, defaults to `[]` (no transformation). This will transform each directory module in `src`, including the root. `cursor` represents the position of the directory being transformed, where `[]` means root and `["foo" "bar"]` means `root.foo.bar`.

Files found in `src` are loaded into an attribute set with the specified `loader`. As an example, the entirety of haumea's API is `load` ed from the `src` directory.

For a directory like this:



```
src
└── foo/
    ├── bar.nix
    ├── baz.nix
    └── __internal.nix
└── bar.nix
└── _utils/
    └── foo.nix
```

The output will look like this:

```
{
  foo = {
    bar = <...>;
    baz = <...>;
  };
  bar = <...>;
}
```

Notice that there is no `_utils`. This is because files and directories that start with `_` are only visible inside the directory being loaded, and will not be present in the final output.

Similarly, files and directories that start with `__` are only visible if they are in the same directory, meaning `foo/__internal.nix` is only accessible if it is being accessed from within `foo`.

By default, the specified `inputs`, in addition to `self`, `super`, and `root`, will be passed to the file being loaded, if the file is a function.

- `self` represents the current file.
- `super` represents the directory the file is in.
- `root` represents the root of the `src` directory being loaded.

Continuing the example above, this is the content of `foo/bar.nix` (`super` and `root` are unused, they are just here for demonstration purposes):

```
{ self, super, root }:

{
  a = 42;
  b = self.a * 2;
}
```



`self.a` will be `42`, which will make `b 84`. Accessing `self.b` here would cause infinite recursion, and accessing anything else would fail due to missing attributes.

```
super will be { bar = self; baz = <...>; internal = <...>; }.
```

And `root` will be:

```
{
  # foo = super;
  foo = {
    bar = <...>;
    baz = <...>;
    internal = <...>;
  };
  baz = <...>;
  utils.foo = <...>;
}
```

Note that this is different from the return value of `load . foo.internal` is accessible here because it is being accessed from within `foo`. Same for `utils`, which is accessible from all files within `src`, the directory being loaded.



loadEvalTests

Source: [src/loadEvalTests.nix](#)

Type: `{ src, loader?, inputs? } -> { }`

A wrapper around `load` to run eval tests using `runTests`.

The accepted arguments are exactly the same as `load`.

This function will throw an error if at least one test failed, otherwise it will always return `{ }` (an empty attribute set).

As an example, haumea's `tests` are loaded with `loadEvalTests`.

Alternatively, `namaka` provides utilities for snapshot testing, which can save you some time from writing reference values.



Loaders

loaders.callPackage

Source: <src/loaders/callPackage.nix>

Type: `{ self, super, root, ... } -> Path -> a`

A wrapper around `callPackageWith`. It adds `override` and `overrideDerivation` to the output (as `makeOverridable` does), and requires the file being loaded to be a function that returns an attribute set. Unlike `loaders.default`, it will respect optional function arguments, as they can be overridden with the added `override` attribute.

loaders.default

Source: <src/loaders/default.nix>

Type: `{ self, super, root, ... } -> Path -> a`

This is the default loader. It imports the file, and provides it the necessary inputs if the file is a function.

Default values of optional function arguments will be ignored, e.g. for `{ foo ? "bar" }:foo`, `"bar"` will be ignored, and it requires `inputs` to contain `foo`. For that reason, although not strictly forbidden, optional arguments are discouraged since they are no-ops.

loaders.path

Source: <src/loaders/path.nix>

Type: `{ ... } -> Path -> Path`

This loader will simply return the path of the file without `import`ing it.



loaders.scoped

Source: [src/loaders/scoped.nix](https://nix-community.github.io/haumea/src/loaders/scoped.nix)

Type: { self, super, root, ... } -> Path -> a

This is like `loaders.default`, except it uses `scopedImport` instead of `import`. With this loader, you don't have to explicitly declare the inputs with a lambda, since `scopedImport` will take care of it as if the file being loaded is wrapped with `with inputs;`.

loaders.verbatim

Source: [src/loaders/verbatim.nix](https://nix-community.github.io/haumea/src/loaders/verbatim.nix)

Type: { ... } -> Path -> a

This loader will simply `import` the file without providing any input. It is useful when the files being loaded are mostly functions that don't require any external input.



Matchers

Type: `{ matches : String -> Bool, loader : { self, super, root, ... } -> Path -> a }`

Matchers allows non-Nix files to be loaded.

This is used for the `loader` option of `load`, which will find the first matcher where `matches` returns `true`, and use its `loader` to load the file.

`matches` takes the name of the file with (up to 2) extra preceding `_`s removed. For both `bar.nix` and `foo/_bar.nix`, the string `matches` gets will be `bar.nix`.

`loader` works exactly like passing a function to the `loader` option, the only difference is that the matcher interface allows loading non-Nix files.

When using matchers, the attribute name will be the file name without its extension, which will be `foo` for all of the following files:

- `foo.nix`
- `bar/_foo.nix`
- `baz/foo`

Only the last file extension is removed, so `far.bar.baz` will have an attribute name of `foo.bar`.

matchers.always

Source: [src/matchers/always.nix](https://github.com/nix-community/haumea/blob/main/src/matchers/always.nix)

Type: `({ self, super, root, ... } -> Path -> a }) -> Matcher`

Matches any file name. This can be used as the last matcher as a catch-all.



matchers.extension

Source: [src/matchers/extension.nix](https://github.com/nix-community/haumea/blob/main/src/matchers/extension.nix)

Type: `String -> ({ self, super, root, ... } -> Path -> a }) -> Matcher`

Matches files with the given extension. `matchers.extension "foo"` matches `a.foo` and `a.b.foo`, but not `.foo`.

matchers.json

Source: <src/matchers/json.nix>

Type: `Matcher`

Matches all JSON files and loads them using `lib.importJSON`.

matchers.nix

Source: <src/matchers/nix.nix>

Type: `({ self, super, root, ... } -> Path -> a}) -> Matcher`

Matches files that end in `.nix`. This is equivalent to `matchers.extension "nix"`.

This is the default matcher if no matchers are defined.

matchers.regex

Source: <src/matchers/regex.nix>

Type: `(regex : String) -> ([String] -> { self, super, root, ... } -> Path -> a}) -> Matcher`

Matches the file name using the given regex. Instead of a regular loader, the function will also take the regex matches returned by `builtins.match`, as shown in the type signature  (`[String]`).

matchers.toml

Source: <src/matchers/toml.nix>

Type: `Matcher`

Matches all TOML files and loads them using `lib.importTOML`.



Transformers

transformers.hoistAttrs

Source: <src/transformers/hoistAttrs.nix>

Type: `(from : String) -> (to : String) -> [String] -> { ... } -> { ... }`

This transformer will hoist any attribute of type Attrs with key `${from}` up the chain. When the root node is reached, it will be renamed to an attribute of type Attrs with key `${to}` and as such presented back to the consumer.

Neighbouring lists are concatenated (`recursiveUpdate`) during hoisting. Root doesn't concat `${from}` declarations, use `${to}` at the root.

This can be used to declare `options` locally at the leaves of the configuration tree, where the NixOS module system would not otherwise tolerate them.

transformers.hoistLists

Source: <src/transformers/hoistLists.nix>

Type: `(from : String) -> (to : String) -> [String] -> { ... } -> { ... }`

This transformer will hoist any attribute of type List with key `${from}` up the chain. When the root node is reached, it will be renamed to an attribute of type List with key `${to}` and as such presented back to the consumer.

Neighbouring lists are concatenated (`++`) during hoisting. Root doesn't concat `${from}` declarations, use `${to}` at the root.

This can be used to declare `imports` locally at the leaves of the configuration tree, where the NixOS module system would not otherwise tolerate them.



transformers.liftDefault

Source: [src/transfomers/liftDefault.nix](#)

Type: [String] -> { ... } -> { ... }

This transformer will lift the contents of `default` into the module. It will fail if `default` is not an attribute set, or has any overlapping attributes with the module.

transformers.prependUnderscore

Source: [src/transfomers/prependUnderscore.nix](#)

Type: [String] -> { ... } -> { ... }

This transformer prepends `_` to attributes that are not valid identifiers, e.g. `42` -> `_42`. Attributes that are already valid identifiers (e.g. `foo`) are left unchanged.



Contributing to Haumea

Unless explicitly stated, all contributions are licensed under [MPL-2.0](#), the license used by haumea.

Making Changes to the API

This doesn't apply to bug fixes.

- Discuss before opening a pull request, so your work doesn't go to waste. Anything from GitHub issues to Matrix discussions is fine.
- Update documentation accordingly. Everything in `haumea.lib` should be documented.
- Add [tests](#) when necessary. Test your changes with `nix flake check`. Make sure new files are added to git.

Documentation

Documentation sits in the `docs` directory. You can get started with `nix develop ./docs`, which will start up [mdbook](#) and serve the documentation on localhost.

Scope

Haumea only depends on `nixpkgs.lib`. Features that depend on the rest of `nixpkgs` should not be added. However, changes that are specific to, but don't depend on `nixpkgs` are allowed.



Style

- Format all Nix files with `nixpkgs-fmt`.
- `with` should be avoided unless absolutely necessary, `let inherit` is preferred at almost all times.
- `rec` attribute sets should be avoided at most times, use `self` or `let-in` instead.

See Also

- [namaka](#) is a snapshot testing tool for Nix built on top of haumea with an interface similar to [loadEvalTests](#).
- [paisano](#) defines a directory structure for auto-importing. It is currently being reimplemented with haumea on the [paisano-haumea](#) branch.
- [std](#) is a more full-featured framework based on the directory structure defined by paisano. Paisano was originally split out of std as a way to modularize std.
- [hive](#) is a project built on top of haumea and paisano with a focus on building system and home configurations.



Changelog

v0.2.2 - 2023-05-26

Features

- New `book`
- load: `loader` now also accepts a list of matchers for loading non-Nix files ([#10](#))

The following matchers and functions available under `matchers`:

- `always` always matches the file regardless of its file name
- `extension` matches the file by its extension
- `json` loads all JSON files
- `nix` is the default matcher if the `loader` is a function and not a list of matchers
- `regex` matches the file using the given regex
- `toml` loads all TOML files

v0.2.1 - 2023-04-19

Features

- `loaders.scoped` to utilize `scopedImport` instead of `import` for loading files

v0.2.0 - 2023-04-10



Breaking Changes

- Transformers now accept a cursor as an argument. The type signature of `transformer` have changed from `{ ... } -> a` to `[String] -> { ... } -> a`

Features

- `transformers.hoistAttrs` and `transformers.hoistLists` bring a specific attribute name at all levels to the root.
- load: `transformer` now also accepts a list or a nested list of functions.

v0.1.1 - 2023-04-07

Features

- load: add transformer option
- transformers: add liftDefault

v0.1.0 - 2023-04-01

First release



just

[crates.io v1.23.0](#)  CI passing [downloads 4.3M](#)  chat 69 online [Say Thanks !](#)

`just` is a handy way to save and run project-specific commands.

This readme is also available as a [book](#).

(中文文档在 [这里](#), 快看·来!)

Commands, called *recipes*, are stored in a file called `justfile` with syntax inspired by `make`:

```
alias b := build
host := `uname -a`
# build main
build:
    cc *.c -o main

# test everything
test-all: build
    ./test --all

# run a specific test
test TEST: build
    ./test --test {{TEST}}
```

```
: just -l
Available recipes:
  build      # build main
  b          # alias for `build`
  test TEST # run a specific test
  test-all   # test everything
:
```

You can then run them with `just RECIPE`:

```
$ just test-all
cc *.c -o main
./test --all
Yay, all your tests passed!
```



`just` has a ton of useful features, and many improvements over `make`:

- `just` is a command runner, not a build system, so it avoids much of `make`'s complexity and idiosyncrasies. No need for `.PHONY` recipes!

- Linux, MacOS, and Windows are supported with no additional dependencies. (Although if your system doesn't have an `sh`, you'll need to [choose a different shell](#).)
- Errors are specific and informative, and syntax errors are reported along with their source context.
- Recipes can accept [command line arguments](#).
- Wherever possible, errors are resolved statically. Unknown recipes and circular dependencies are reported before anything runs.
- `just` [loads .env files](#), making it easy to populate environment variables.
- Recipes can be [listed from the command line](#).
- Command line completion scripts are [available for most popular shells](#).
- Recipes can be written in [arbitrary languages](#), like Python or NodeJS.
- `just` can be invoked from any subdirectory, not just the directory that contains the `justfile`.
- And [much more!](#)

If you need help with `just` please feel free to open an issue or ping me on [Discord](#). Feature requests and bug reports are always welcome!



Installation



Prerequisites

`just` should run on any system with a reasonable `sh`, including Linux, MacOS, and the BSDs.

On Windows, `just` works with the `sh` provided by [Git for Windows](#), [GitHub Desktop](#), or [Cygwin](#).

If you'd rather not install `sh`, you can use the `shell` setting to use the shell of your choice.

Like PowerShell:

```
# use PowerShell instead of sh:  
set shell := ["powershell.exe", "-c"]  
  
hello:  
  Write-Host "Hello, world!"
```

...or `cmd.exe`:

```
# use cmd.exe instead of sh:  
set shell := ["cmd.exe", "/c"]  
  
list:  
  dir
```

You can also set the shell using command-line arguments. For example, to use PowerShell, launch `just` with `--shell powershell.exe --shell-arg -c`.

(PowerShell is installed by default on Windows 7 SP1 and Windows Server 2008 R2 S1 and later, and `cmd.exe` is quite fiddly, so PowerShell is recommended for most Windows users.)



Packages

Operating System	Package Manager	Package	Command
Various	Cargo	just	<code>cargo install just</code>
Microsoft Windows	Scoop	just	<code>scoop install just</code>
Various	Homebrew	just	<code>brew install just</code>
macOS	MacPorts	just	<code>port install just</code>
Arch Linux	pacman	just	<code>pacman -S just</code>
Various	Nix	just	<code>nix-env -iA nixpkgs.just</code>
NixOS	Nix	just	<code>nix-env -iA nixos.just</code>
Solus	eopkg	just	<code>eopkg install just</code>
Void Linux	XBPS	just	<code>xbps-install -S just</code>
FreeBSD	pkg	just	<code>pkg install just</code>
Alpine Linux	apk-tools	just	<code>apk add just</code>
Fedora Linux	DNF	just	<code>dnf install just</code>
Gentoo Linux	Portage	guru/dev-build/just	<code>eselect repository enable guru</code> <code>emerge --sync guru</code> <code>emerge dev-build/just</code>
Various	Conda	just	<code>conda install -c conda-forge just</code>
Microsoft Windows	Chocolatey	just	<code>choco install just</code>
Various	Snap	just	<code>snap install --edge --classic just</code>
Various	asdf	just	<code>asdf plugin add just</code> <code>asdf install just <version></code>
Debian and Ubuntu derivatives	MPR	just	<code>git clone https://mpr.makedeb.org/just</code> <code>cd just</code> <code>makedeb -si</code>
Debian and Ubuntu	Prebuilt-MPR	just	You must have the Prebuilt-MPR set up on your system in order to run this



Operating System	Package Manager	Package	Command
derivatives			command. <code>sudo apt install just</code>
Microsoft Windows	Windows Package Manager	Casey/Just	<code>winget install --id Casey.Just --exact</code>



Packaging status	
Alpine Linux 3.13	0.8.3
Alpine Linux 3.14	0.9.5
Alpine Linux 3.15	0.10.3
Alpine Linux 3.16	1.1.3
Alpine Linux 3.17	1.8.0
Alpine Linux 3.18	1.13.0
Alpine Linux 3.19	1.16.0
Alpine Linux Edge	1.23.0
ALT Sisyphus	1.22.1
Arch	1.23.0
Arch Linux 32 i686	1.23.0
Arch Linux 32 pentium4	1.23.0
Arch Linux ARM aarch64	1.23.0
AUR	0.9.4.r322...
Chocolatey	1.23.0
FreeBSD Ports	1.23.0
Gentoo overlay GURU	1.23.0
GNU Guix	1.14.0
Homebrew	1.23.0
LiGurOS stable	1.16.0
LiGurOS develop	1.23.0
MacPorts	1.23.0
Manjaro Stable	1.22.1
Manjaro Testing	1.23.0
Manjaro Unstable	1.23.0
MPR	1.23.0
nixpkgs stable 21.11	0.10.3
nixpkgs stable 22.05	1.1.3
nixpkgs stable 22.11	1.8.0
nixpkgs stable 23.05	1.13.0
nixpkgs stable 23.11	1.23.0
nixpkgs unstable	1.23.0
OpenPKG	1.11.0
openSUSE Tumbleweed	1.23.0
Parabola	1.23.0
Solus	1.9.0
Void Linux x86_64	1.23.0
Wikidata	1.23.0
winget	1.23.0



Pre-Built Binaries

Pre-built binaries for Linux, MacOS, and Windows can be found on [the releases page](#).

You can use the following command on Linux, MacOS, or Windows to download the latest release, just replace `DEST` with the directory where you'd like to put `just`:

```
curl --proto '=https' --tlsv1.2 -sSf https://just.systems/install.sh | bash -s  
-- --to DEST
```

For example, to install `just` to `~/bin`:

```
# create ~/bin  
mkdir -p ~/bin  
  
# download and extract just to ~/bin/just  
curl --proto '=https' --tlsv1.2 -sSf https://just.systems/install.sh | bash -s  
-- --to ~/bin  
  
# add `~/bin` to the paths that your shell searches for executables  
# this line should be added to your shells initialization file,  
# e.g. `~/.bashrc` or `~/.zshrc`  
export PATH="$PATH:$HOME/bin"  
  
# just should now be executable  
just --help
```

Note that `install.sh` may fail on GitHub actions, or in other environments where many machines share IP addresses. `install.sh` calls GitHub APIs in order to determine the latest version of `just` to install, and those API calls are rate-limited on a per-IP basis. To make `install.sh` more reliable in such circumstances, pass a specific tag to install with `--tag`.



GitHub Actions

With [extractions/setup-just](#):

```
- uses: extractions/setup-just@v1
  with:
    just-version: 0.8 # optional semver specification, otherwise latest
```

With [taiki-e/install-action](#):

```
- uses: taiki-e/install-action@just
```



Release RSS Feed

An [RSS feed](#) of just releases is available [here](#).



Node.js Installation

[just-install](#) can be used to automate installation of `just` in Node.js applications.

`just` is a great, more robust alternative to npm scripts. If you want to include `just` in the dependencies of a Node.js application, `just-install` will install a local, platform-specific binary as part of the `npm install` command. This removes the need for every developer to install `just` independently using one of the processes mentioned above. After installation, the `just` command will work in npm scripts or with npx. It's great for teams who want to make the set up process for their project as easy as possible.

For more information, see the [just-install README file](#).



Backwards Compatibility

With the release of version 1.0, `just` features a strong commitment to backwards compatibility and stability.

Future releases will not introduce backwards incompatible changes that make existing `justfile`s stop working, or break working invocations of the command-line interface.

This does not, however, preclude fixing outright bugs, even if doing so might break `justfiles` that rely on their behavior.

There will never be a `just 2.0`. Any desirable backwards-incompatible changes will be opt-in on a per-`justfile` basis, so users may migrate at their leisure.

Features that aren't yet ready for stabilization are gated behind the `--unstable` flag.

Features enabled by `--unstable` may change in backwards incompatible ways at any time.

Unstable features can also be enabled by setting the environment variable `JUST_UNSTABLE` to any value other than `false`, `0`, or the empty string.



Editor Support

`justfile` syntax is close enough to `make` that you may want to tell your editor to use `make` syntax highlighting for `just`.



Vim and Neovim

vim-just

The [vim-just](#) plugin provides syntax highlighting for `justfile`s.

Install it with your favorite package manager, like [Plug](#):

```
call plug#begin()  
  
Plug 'NoahTheDuke/vim-just'  
  
call plug#end()
```

Or with Vim's built-in package support:

```
mkdir -p ~/.vim/pack/vendor/start  
cd ~/.vim/pack/vendor/start  
git clone https://github.com/NoahTheDuke/vim-just.git
```

tree-sitter-just

[tree-sitter-just](#) is an [Nvim Treesitter](#) plugin for Neovim.

Makefile Syntax Highlighting

Vim's built-in makefile syntax highlighting isn't perfect for `justfile`s, but it's better than nothing. You can put the following in `~/.vim/filetype.vim`:

```
if exists("did_load_filetypes")
  finish
endif

augroup filetypeetect
  au BufNewFile,BufRead justfile setf make
augroup END
```

Or add the following to an individual `justfile` to enable `make` mode on a per-file basis:

```
# vim: set ft=make :
```



Emacs

[just-mode](#) provides syntax highlighting and automatic indentation of `justfile`s. It is available on [MELPA](#) as [just-mode](#).

[justl](#) provides commands for executing and listing recipes.

You can add the following to an individual `justfile` to enable `make` mode on a per-file basis:

```
# Local Variables:  
# mode: makefile  
# End:
```



Visual Studio Code

An extension for VS Code by [skellock](#) is [available here \(repository\)](#), but is no longer actively developed.

You can install it from the command line by running:

```
code --install-extension skellock.just
```

An more recently active fork by [sclu1034](#) is available [here](#).



JetBrains IDEs

A plugin for JetBrains IDEs by [linux_china](#) is [available here](#).



Kakoune

Kakoune supports `justfile` syntax highlighting out of the box, thanks to TeddyDD.



Helix

[Helix](#) supports `justfile` syntax highlighting out-of-the-box since version 23.05.



Sublime Text

The [Just package](#) by [nk9](#) with `just` syntax and some other tools is available on [PackageControl](#).



Micro

[Micro](#) supports Justfile syntax highlighting out of the box, thanks to [tomodachi94](#).



Other Editors

Feel free to send me the commands necessary to get syntax highlighting working in your editor of choice so that I may include them here.



Quick Start

See [the installation section](#) for how to install `just` on your computer. Try running `just --version` to make sure that it's installed correctly.

For an overview of the syntax, check out [this cheatsheet](#).

Once `just` is installed and working, create a file named `justfile` in the root of your project with the following contents:

```
recipe-name:  
  echo 'This is a recipe!'  
  
# this is a comment  
another-recipe:  
  @echo 'This is another recipe.'
```

When you invoke `just` it looks for file `justfile` in the current directory and upwards, so you can invoke it from any subdirectory of your project.

The search for a `justfile` is case insensitive, so any case, like `Justfile`, `JUSTFILE`, or `JuStFiLe`, will work. `just` will also look for files with the name `.justfile`, in case you'd like to hide a `justfile`.

Running `just` with no arguments runs the first recipe in the `justfile`:

```
$ just  
echo 'This is a recipe!'  
This is a recipe!
```

One or more arguments specify the recipe(s) to run:

```
$ just another-recipe  
This is another recipe.
```

`just` prints each command to standard error before running it, which is why `echo 'This is a recipe!'` was printed. This is suppressed for lines starting with `@`, which is why `echo 'This is another recipe.'` was not printed.

Recipes stop running if a command fails. Here `cargo publish` will only run if `cargo test` succeeds:

```
publish:  
  cargo test  
  # tests passed, time to publish!  
  cargo publish
```

Recipes can depend on other recipes. Here the `test` recipe depends on the `build` recipe, so `build` will run before `test`:

```
build:  
  cc main.c foo.c bar.c -o main  
  
test: build  
  ./test  
  
sloc:  
  @echo "`wc -l *.c` lines of code"  
  
$ just test  
cc main.c foo.c bar.c -o main  
./test  
testing... all tests passed!
```

Recipes without dependencies will run in the order they're given on the command line:

```
$ just build sloc  
cc main.c foo.c bar.c -o main  
1337 lines of code
```

Dependencies will always run first, even if they are passed after a recipe that depends on them:

```
$ just test build  
cc main.c foo.c bar.c -o main  
./test  
testing... all tests passed!
```



Examples

A variety of example `justfile`s can be found in the [examples directory](#).



Features



The Default Recipe

When `just` is invoked without a recipe, it runs the first recipe in the `justfile`. This recipe might be the most frequently run command in the project, like running the tests:

```
test:  
  cargo test
```

You can also use dependencies to run multiple recipes by default:

```
default: lint build test  
  
build:  
  echo Building...  
  
test:  
  echo Testing...  
  
lint:  
  echo Linting...
```

If no recipe makes sense as the default recipe, you can add a recipe to the beginning of your `justfile` that lists the available recipes:

```
default:  
  just --list
```



Listing Available Recipes

Recipes can be listed in alphabetical order with `just --list`:

```
$ just --list
Available recipes:
  build
  test
  deploy
  lint
```

`just --summary` is more concise:

```
$ just --summary
build test deploy lint
```

Pass `--unsorted` to print recipes in the order they appear in the `justfile`:

```
test:
  echo 'Testing!'

build:
  echo 'Building!'
```

```
$ just --list --unsorted
Available recipes:
  test
  build
```

```
$ just --summary --unsorted
test build
```

If you'd like `just` to default to listing the recipes in the `justfile`, you can use this as your default recipe:

```
default:
  @just --list
```



Note that you may need to add `--justfile {{justfile()}}` to the line above above. Without it, if you executed `just -f /some/distant/justfile -d .` or `just -f ./non-standard-justfile`, the plain `just --list` inside the recipe would not necessarily use the file you provided. It would try to find a `justfile` in your current path, maybe even resulting in a `No justfile found` error.

The heading text can be customized with `--list-heading`:

```
$ just --list --list-heading '$Cool stuff...\n'  
Cool stuff...  
  test  
  build
```

And the indentation can be customized with `--list-prefix`:

```
$ just --list --list-prefix ....  
Available recipes:  
....test  
....build
```

The argument to `--list-heading` replaces both the heading and the newline following it, so it should contain a newline if non-empty. It works this way so you can suppress the heading line entirely by passing the empty string:

```
$ just --list --list-heading ''  
  test  
  build
```



Aliases

Aliases allow recipes to be invoked on the command line with alternative names:

```
alias b := build
```

```
build:  
  echo 'Building!'
```

```
$ just b  
echo 'Building!'  
Building!
```



Settings

Settings control interpretation and execution. Each setting may be specified at most once, anywhere in the `justfile`.

For example:

```
set shell := ["zsh", "-cu"]

foo:
  # this line will be run as `zsh -cu 'ls **/*.txt'`
  ls **/*.txt
```

Table of Settings

Name	Value	Default	Description
allow-duplicate-recipes	boolean	false	Allow recipes appearing later in a <code>justfile</code> to override earlier recipes with the same name.
dotenv-filename	string	-	Load a <code>.env</code> file with a custom name, if present.
dotenv-load	boolean	false	Load a <code>.env</code> file, if present.
dotenv-path	string	-	Load a <code>.env</code> file from a custom path, if present. Overrides <code>dotenv-filename</code> .
export	boolean	false	Export all variables as environment variables.
fallback	boolean	false	Search <code>justfile</code> in parent directory if the first recipe on the command line is not found.
ignore-comments	boolean	false	Ignore recipe lines beginning with <code>#</code> .
positional-arguments	boolean	false	Pass positional arguments.
shell	[COMMAND, ARGS...]	-	Set the command used to invoke recipes and evaluate backticks.
tempdir	string	-	Create temporary directories in <code>tempdir</code> instead of the system default temporary directory.



Name	Value	Default	Description
windows-powershell	boolean	false	Use PowerShell on Windows as default shell. (Deprecated. Use windows-shell instead.)
windows-shell	[COMMAND, ARGS...]	-	Set the command used to invoke recipes and evaluate backticks.

Boolean settings can be written as:

```
set NAME
```

Which is equivalent to:

```
set NAME := true
```

Allow Duplicate Recipes

If `allow-duplicate-recipes` is set to `true`, defining multiple recipes with the same name is not an error and the last definition is used. Defaults to `false`.

```
set allow-duplicate-recipes
```

```
@foo:
echo foo
```

```
@foo:
echo bar
```

```
$ just foo
bar
```

Dotenv Settings

If `dotenv-load`, `dotenv-filename` or `dotenv-path` is set, `just` will load environment variables from a file.



If `dotenv-path` is set, `just` will look for a file at the given path.

Otherwise, `just` looks for a file named `.env` by default, unless `dotenv-filename` set, in which case the value of `dotenv-filename` is used. This file can be located in the same directory as your `justfile` or in a parent directory.

The loaded variables are environment variables, not `just` variables, and so must be accessed using `$VARIABLE_NAME` in recipes and backticks.

For example, if your `.env` file contains:

```
# a comment, will be ignored
DATABASE_ADDRESS=localhost:6379
SERVER_PORT=1337
```

And your `justfile` contains:

```
set dotenv-load

serve:
  @echo "Starting server with database $DATABASE_ADDRESS on port $SERVER_PORT..."
  ./server --database $DATABASE_ADDRESS --port $SERVER_PORT
```

`just serve` will output:

```
$ just serve
Starting server with database localhost:6379 on port 1337...
./server --database $DATABASE_ADDRESS --port $SERVER_PORT
```

Export

The `export` setting causes all `just` variables to be exported as environment variables. Defaults to `false`.

```
set export

a := "hello"

@foo b:
  echo $a
  echo $b

$ just foo goodbye
hello
goodbye
```



Positional Arguments

If `positional-arguments` is `true`, recipe arguments will be passed as positional arguments

to commands. For linewise recipes, argument `$0` will be the name of the recipe.

For example, running this recipe:

```
set positional-arguments

@foo bar:
    echo $0
    echo $1
```

Will produce the following output:

```
$ just foo hello
foo
hello
```

When using an `sh`-compatible shell, such as `bash` or `zsh`, `$@` expands to the positional arguments given to the recipe, starting from one. When used within double quotes as `"$@"`, arguments including whitespace will be passed on as if they were double-quoted. That is, `"$@"` is equivalent to `"$1" " $2" ...`. When there are no positional parameters, `"$@"` and `$@` expand to nothing (i.e., they are removed).

This example recipe will print arguments one by one on separate lines:

```
set positional-arguments

@test *args='':
    bash -c 'while (( "$#" )); do echo - $1; shift; done' -- "$@"
```

Running it with *two* arguments:

```
$ just test foo "bar baz"
- foo
- bar baz
```

Shell



The `shell` setting controls the command used to invoke recipe lines and backticks. Shebang recipes are unaffected.

```
# use python3 to execute recipe lines and backticks
set shell := ["python3", "-c"]

# use print to capture result of evaluation
foos := `print("foo" * 4`

foo:
    print("Snake snake snake snake.")
    print("{{foos}}")
```

just passes the command to be executed as an argument. Many shells will need an additional flag, often `-c`, to make them evaluate the first argument.

Windows Shell

just uses `sh` on Windows by default. To use a different shell on Windows, use `windows-shell`:

```
set windows-shell := ["powershell.exe", "-NoLogo", "-Command"]

hello:
    Write-Host "Hello, world!"
```

See [powershell.just](#) for a justfile that uses PowerShell on all platforms.

Windows PowerShell

set windows-powershell uses the legacy `powershell.exe` binary, and is no longer recommended. See the `windows-shell` setting above for a more flexible way to control which shell is used on Windows.

just uses `sh` on Windows by default. To use `powershell.exe` instead, set `windows-powershell` to true.

```
set windows-powershell := true

hello:
    Write-Host "Hello, world!"
```



Python 3

```
set shell := ["python3", "-c"]
```

Bash

```
set shell := ["bash", "-uc"]
```

Z Shell

```
set shell := ["zsh", "-uc"]
```

Fish

```
set shell := ["fish", "-c"]
```

Nushell

```
set shell := ["nu", "-c"]
```

If you want to change the default table mode to `light`:

```
set shell := ['nu', '-m', 'light', '-c']
```

Nushell was written in Rust, and has cross-platform support for Windows / macOS and Linux.



Documentation Comments

Comments immediately preceding a recipe will appear in `just --list`:

```
# build stuff
build:
    ./bin/build

# test stuff
test:
    ./bin/test

$ just --list
Available recipes:
    build # build stuff
    test # test stuff
```



Variables and Substitution

Variables, strings, concatenation, path joining, and substitution using `{{...}}` are supported:

```
tmpdir := `mktemp -d`  
version := "0.2.7"  
tardir := tmpdir / "awesomesauce-" + version  
tarball := tardir + ".tar.gz"  
  
publish:  
  rm -f {{tarball}}  
  mkdir {{tardir}}  
  cp README.md *.c {{tardir}}  
  tar zcvf {{tarball}} {{tardir}}  
  scp {{tarball}} me@server.com:release/  
  rm -rf {{tarball}} {{tardir}}
```

Joining Paths

The `/` operator can be used to join two strings with a slash:

```
foo := "a" / "b"
```

```
$ just --evaluate foo  
a/b
```

Note that a `/` is added even if one is already present:

```
foo := "a/"  
bar := foo / "b"
```

```
$ just --evaluate bar  
a//b
```

Absolute paths can also be constructed^{1.5.0}:

```
foo := / "b"
```

```
$ just --evaluate foo  
/b
```

The `/` operator uses the `/` character, even on Windows. Thus, using the `/` operator should



be avoided with paths that use universal naming convention (UNC), i.e., those that start with `\?`, since forward slashes are not supported with UNC paths.

Escaping {{

To write a recipe containing `{{`, use `{{{`:

```
braces:  
echo 'I {{{LOVE}} curly braces!'
```

(An unmatched `}}` is ignored, so it doesn't need to be escaped.)

Another option is to put all the text you'd like to escape inside of an interpolation:

```
braces:  
echo '{{'I {{LOVE}} curly braces!'}}'
```

Yet another option is to use `{} "{{" }}`:

```
braces:  
echo 'I {{ "{{" }}LOVE}} curly braces!'
```



Strings

Double-quoted strings support escape sequences:

```
string-with-tab          := "\t"
string-with-newline      := "\n"
string-with-carriage-return := "\r"
string-with-double-quote  := """
string-with-slash        := "\\"
string-with-no-newline    := "\"
"

$ just --evaluate
"tring-with-carriage-return := "
string-with-double-quote   := """
string-with-newline        := "
"
string-with-no-newline      := ""
string-with-slash           := \
string-with-tab             := "      "
```

Strings may contain line breaks:

```
single := '
hello
'

double := "
goodbye
"
```

Single-quoted strings do not recognize escape sequences:

```
escapes := '\t\n\r\"\\'
```

```
$ just --evaluate
escapes := "\t\n\r\"\\"
```

Indented versions of both single- and double-quoted strings, delimited by triple single- or triple double-quotes, are supported. Indented string lines are stripped of a leading line break, and leading whitespace common to all non-blank lines:



```
# this string will evaluate to `foo\nbar\n`  
x := '''  
  foo  
  bar  
'''  
  
# this string will evaluate to `abc\n  wuv\nxyz\n`  
y := """  
  abc  
    wuv  
  xyz  
"""
```

Similar to unindented strings, indented double-quoted strings process escape sequences, and indented single-quoted strings ignore escape sequences. Escape sequence processing takes place after unindentation. The unindentation algorithm does not take escape-sequence produced whitespace or newlines into account.



Ignoring Errors

Normally, if a command returns a non-zero exit status, execution will stop. To continue execution after a command, even if it fails, prefix the command with `-`:

```
foo:  
-cat foo  
echo 'Done!'
```

```
$ just foo  
cat foo  
cat: foo: No such file or directory  
echo 'Done!'  
Done!
```



Functions

`just` provides a few built-in functions that might be useful when writing recipes.

System Information

- `arch()` — Instruction set architecture. Possible values are: "aarch64", "arm", "asmjs", "hexagon", "mips", "msp430", "powerpc", "powerpc64", "s390x", "sparc", "wasm32", "x86", "x86_64", and "xcore".
- `num_cpus()` — Number of logical CPUs.
- `os()` — Operating system. Possible values are: "android", "bitrig", "dragonfly", "emscripten", "freebsd", "haiku", "ios", "linux", "macos", "netbsd", "openbsd", "solaris", and "windows".
- `os_family()` — Operating system family; possible values are: "unix" and "windows".

For example:

```
system-info:  
@echo "This is an {{arch()}} machine".
```

```
$ just system-info  
This is an x86_64 machine
```

The `os_family()` function can be used to create cross-platform `justfile`s that work on various operating systems. For an example, see [cross-platform.just](#) file.

Environment Variables

- `env_var(key)` — Retrieves the environment variable with name `key`, aborting if it is not present.

```
home_dir := env_var('HOME')
```

```
test:  
echo "{{home_dir}}"
```

```
$ just  
/home/user1
```

- `env_var_or_default(key, default)` — Retrieves the environment variable with name `key`, returning `default` if it is not present.



- `env(key)` 1.15.0 — Alias for `env_var(key)` .
- `env(key, default)` 1.15.0 — Alias for `env_var_or_default(key, default)` .

Invocation Directory

- `invocation_directory()` - Retrieves the absolute path to the current directory when `just` was invoked, before `just` changed it (`chdir'd`) prior to executing commands. On Windows, `invocation_directory()` uses `cygpath` to convert the invocation directory to a Cygwin-compatible `/`-separated path. Use `invocation_directory_native()` to return the verbatim invocation directory on all platforms.

For example, to call `rustfmt` on files just under the “current directory” (from the user/invoker’s perspective), use the following rule:

```
rustfmt:  
  find {{invocation_directory()}} -name \*.rs -exec rustfmt {} \;
```

Alternatively, if your command needs to be run from the current directory, you could use (e.g.):

```
build:  
  cd {{invocation_directory()}}; ./some_script_that_needs_to_be_run_from_here
```

- `invocation_directory_native()` - Retrieves the absolute path to the current directory when `just` was invoked, before `just` changed it (`chdir'd`) prior to executing commands.

Justfile and Justfile Directory

- `justfile()` - Retrieves the path of the current `justfile` .
- `justfile_directory()` - Retrieves the path of the parent directory of the current `justfile` .

For example, to run a command relative to the location of the current `justfile` :

```
script:  
  ./{{justfile_directory()}}/scripts/some_script
```

Just Executable



- `just_executable()` - Absolute path to the `just` executable.

For example:

```
executable:  
@echo The executable is at: {{just_executable()}}
```

```
$ just  
The executable is at: /bin/just
```

Just Process ID

- `just_pid()` - Process ID of the `just` executable.

For example:

```
pid:  
@echo The process ID is: {{ just_pid() }}
```

```
$ just  
The process ID is: 420
```

String Manipulation

- `quote(s)` - Replace all single quotes with '`\'`' and prepend and append single quotes to `s`. This is sufficient to escape special characters for many shells, including most Bourne shell descendants.
- `replace(s, from, to)` - Replace all occurrences of `from` in `s` to `to`.
- `replace_regex(s, regex, replacement)` - Replace all occurrences of `regex` in `s` to `replacement`. Regular expressions are provided by the [Rust regex crate](#). See the [syntax documentation](#) for usage examples. Capture groups are supported. The `replacement` string uses [Replacement string syntax](#).
- `trim(s)` - Remove leading and trailing whitespace from `s`.
- `trim_end(s)` - Remove trailing whitespace from `s`.
- `trim_end_match(s, pat)` - Remove suffix of `s` matching `pat`.
- `trim_end_matches(s, pat)` - Repeatedly remove suffixes of `s` matching `pat`.
- `trim_start(s)` - Remove leading whitespace from `s`.
- `trim_start_match(s, pat)` - Remove prefix of `s` matching `pat`.
- `trim_start_matches(s, pat)` - Repeatedly remove prefixes of `s` matching `pat`.



Case Conversion

- `capitalize(s)` 1.7.0 - Convert first character of `s` to uppercase and the rest to lowercase.
- `kebabcase(s)` 1.7.0 - Convert `s` to kebab-case .
- `lowercamelcase(s)` 1.7.0 - Convert `s` to lowerCamelCase .
- `lowercase(s)` - Convert `s` to lowercase.
- `shoutykebabcase(s)` 1.7.0 - Convert `s` to SHOUTY-KEBAB-CASE .
- `shoutysnakecase(s)` 1.7.0 - Convert `s` to SHOUTY_SNAKE_CASE .
- `snakecase(s)` 1.7.0 - Convert `s` to snake_case .
- `titlecase(s)` 1.7.0 - Convert `s` to Title Case .
- `uppercamelcase(s)` 1.7.0 - Convert `s` to UpperCamelCase .
- `uppercase(s)` - Convert `s` to uppercase.

Path Manipulation

Fallible

- `absolute_path(path)` - Absolute path to relative `path` in the working directory. `absolute_path("./bar.txt")` in directory `/foo` is `/foo/bar.txt` .
- `canonicalize(path)` - Canonicalize `path` by resolving symlinks and removing `..`, `..`, and extra `/`s where possible.
- `extension(path)` - Extension of `path`. `extension("/foo/bar.txt")` is `.txt` .
- `file_name(path)` - File name of `path` with any leading directory components removed. `file_name("/foo/bar.txt")` is `bar.txt` .
- `file_stem(path)` - File name of `path` without extension. `file_stem("/foo/bar.txt")` is `bar` .
- `parent_directory(path)` - Parent directory of `path`. `parent_directory("/foo/bar.txt")` is `/foo` .
- `without_extension(path)` - `path` without extension. `without_extension("/foo/bar.txt")` is `/foo/bar` .



These functions can fail, for example if a path does not have an extension, which will halt execution.

Infallible

- `clean(path)` - Simplify `path` by removing extra path separators, intermediate `.` .

- components, and `..` where possible. `clean("foo//bar")` is `foo/bar`, `clean("foo/..")` is `..`, `clean("foo/./bar")` is `foo/bar`.
- `join(a, b...)` - This function uses `/` on Unix and `\` on Windows, which can lead to unwanted behavior. The `/` operator, e.g., `a / b`, which always uses `/`, should be considered as a replacement unless `\`s are specifically desired on Windows. Join path `a` with path `b`. `join("foo/bar", "baz")` is `foo/bar/baz`. Accepts two or more arguments.

Filesystem Access

- `path_exists(path)` - Returns `true` if the path points at an existing entity and `false` otherwise. Traverses symbolic links, and returns `false` if the path is inaccessible or points to a broken symlink.

Error Reporting

- `error(message)` - Abort execution and report error `message` to user.

UUID and Hash Generation

- `sha256(string)` - Return the SHA-256 hash of `string` as a hexadecimal string.
- `sha256_file(path)` - Return the SHA-256 hash of the file at `path` as a hexadecimal string.
- `uuid()` - Generate a random version 4 UUID.

Semantic Versions

- `semver_matches(version, requirement)` 1.16.0 - Check whether a semantic version, e.g., `"0.1.0"` matches a requirement, e.g., `">=0.1.0"`, returning `"true"` if so and `"false"` otherwise.

Xdg Directories 1.23.0



These functions return paths to user-specific directories for things like configuration, data, caches, executables, and the user's home directory. These functions follow the [Xdg Base Directory Specification](#), and are implemented with the `dirs` crate.

- `cache_directory()` - The user-specific cache directory.
- `config_directory()` - The user-specific configuration directory.

- `config_local_directory()` - The local user-specific configuration directory.
- `data_directory()` - The user-specific data directory.
- `data_local_directory()` - The local user-specific data directory.
- `executable_directory()` - The user-specific executable directory.
- `home_directory()` - The user's home directory.



Recipe Attributes

Recipes may be annotated with attributes that change their behavior.

Name	Description
[confirm] 1.17.0	Require confirmation prior to executing recipe.
[confirm("prompt")] 1.23.0	Require confirmation prior to executing recipe with a custom prompt.
[linux] 1.8.0	Enable recipe on Linux.
[macos] 1.8.0	Enable recipe on MacOS.
[no-cd] 1.9.0	Don't change directory before executing recipe.
[no-exit-message] 1.7.0	Don't print an error message if recipe fails.
[no-quiet] 1.23.0	Override globally quiet recipes and always echo out the recipe.
[private] 1.10.0	See Private Recipes .
[unix] 1.8.0	Enable recipe on Unixes. (Includes MacOS).
[windows] 1.8.0	Enable recipe on Windows.

A recipe can have multiple attributes, either on multiple lines:

```
[no-cd]
[private]
foo:
    echo "foo"
```

Or separated by commas on a single line^{1.14.0}:

```
[no-cd, private]
foo:
    echo "foo"
```



Enabling and Disabling Recipes^{1.8.0}

The [linux] , [macos] , [unix] , and [windows] attributes are configuration attributes. By default, recipes are always enabled. A recipe with one or more configuration attributes will only be enabled when one or more of those configurations is active.

This can be used to write `justfile`s that behave differently depending on which operating

system they run on. The `run` recipe in this `justfile` will compile and run `main.c`, using a different C compiler and using the correct output binary name for that compiler depending on the operating system:

```
[unix]
run:
  cc main.c
  ./a.out
```

```
[windows]
run:
  cl main.c
  main.exe
```

Disabling Changing Directory^{1.9.0}

`just` normally executes recipes with the current directory set to the directory that contains the `justfile`. This can be disabled using the `[no-cd]` attribute. This can be used to create recipes which use paths relative to the invocation directory, or which operate on the current directory.

For example, this `commit` recipe:

```
[no-cd]
commit file:
  git add {{file}}
  git commit
```

Can be used with paths that are relative to the current directory, because `[no-cd]` prevents `just` from changing the current directory when executing `commit`.

Requiring Confirmation for Recipes^{1.17.0}

`just` normally executes all recipes unless there is an error. The `[confirm]` attribute allows recipes require confirmation in the terminal prior to running. This can be overridden by passing `--yes` to `just`, which will automatically confirm any recipes marked by this attribute.

Recipes dependent on a recipe that requires confirmation will not be run if the relied upon recipe is not confirmed, as well as recipes passed after any recipe that requires confirmation.



```
[confirm]
delete all:
  rm -rf *
```

Custom Confirmation Prompt^{1.23.0}

The default confirmation prompt can be overridden with [confirm(PROMPT)] :

```
[confirm("Are you sure you want to delete everything?")]
delete-everything:
  rm -rf *
```



Command Evaluation Using Backticks

Backticks can be used to store the result of commands:

```
localhost := `dumpinterfaces | cut -d: -f2 | sed 's/\/.*/' | sed 's/ //g'`  
  
serve:  
./serve {{localhost}} 8080
```

Indented backticks, delimited by three backticks, are de-indented in the same manner as indented strings:

```
# This backtick evaluates the command `echo foo\n echo bar\n`, which produces  
# the value `foo\nbar\n`.  
stuff := ``  
    echo foo  
    echo bar  
```
```

See the [Strings](#) section for details on unindenting.

Backticks may not start with `#!`. This syntax is reserved for a future upgrade.



## Conditional Expressions

`if / else` expressions evaluate different branches depending on if two expressions evaluate to the same value:

```
foo := if "2" == "2" { "Good!" } else { "1984" }
```

```
bar:
 @echo "{{foo}}"
```

```
$ just bar
Good!
```

It is also possible to test for inequality:

```
foo := if "hello" != "goodbye" { "xyz" } else { "abc" }
```

```
bar:
 @echo {{foo}}
```

```
$ just bar
xyz
```

And match against regular expressions:

```
foo := if "hello" =~ 'hel+o' { "match" } else { "mismatch" }
```

```
bar:
 @echo {{foo}}
```

```
$ just bar
match
```

Regular expressions are provided by the [regex crate](#), whose syntax is documented on [docs.rs](#). Since regular expressions commonly use backslash escape sequences, consider using single-quoted string literals, which will pass slashes to the regex parser unmolested.



Conditional expressions short-circuit, which means they only evaluate one of their branches. This can be used to make sure that backtick expressions don't run when they shouldn't.

```
foo := if env_var("RELEASE") == "true" { `get-something-from-release-database`
} else { "dummy-value" }
```

Conditionals can be used inside of recipes:

```
bar foo:
echo {{ if foo == "bar" { "hello" } else { "goodbye" } }}
```

Note the space after the final `}`! Without the space, the interpolation will be prematurely closed.

Multiple conditionals can be chained:

```
foo := if "hello" == "goodbye" {
 "xyz"
} else if "a" == "a" {
 "abc"
} else {
 "123"
}
```

```
bar:
@echo {{foo}}
```

```
$ just bar
abc
```



## Stopping execution with error

Execution can be halted with the `error` function. For example:

```
foo := if "hello" == "goodbye" {
 "xyz"
} else if "a" == "b" {
 "abc"
} else {
 error("123")
}
```

Which produce the following error when run:

```
error: Call to function `error` failed: 123
16 | error("123")
```



## Setting Variables from the Command Line

Variables can be overridden from the command line.

```
os := "linux"

test: build
 ./test --test {{os}}

build:
 ./build {{os}}

$ just
./build linux
./test --test linux
```

Any number of arguments of the form `NAME=VALUE` can be passed before recipes:

```
$ just os=plan9
./build plan9
./test --test plan9
```

Or you can use the `--set` flag:

```
$ just --set os bsd
./build bsd
./test --test bsd
```



## Getting and Setting Environment Variables

### Exporting just Variables

Assignments prefixed with the `export` keyword will be exported to recipes as environment variables:

```
export RUST_BACKTRACE := "1"

test:
 # will print a stack trace if it crashes
 cargo test
```

Parameters prefixed with a `$` will be exported as environment variables:

```
test $RUST_BACKTRACE="1":
 # will print a stack trace if it crashes
 cargo test
```

Exported variables and parameters are not exported to backticks in the same scope.

```
export WORLD := "world"
This backtick will fail with "WORLD: unbound variable"
BAR := `echo hello $WORLD`

Running `just a foo` will fail with "A: unbound variable"
a $A $B=`echo $A`:
 echo $A $B
```

When `export` is set, all `just` variables are exported as environment variables.

### Getting Environment Variables from the environment

Environment variables from the environment are passed automatically to the recipes.



```
print_home_folder:
 echo "HOME is: '${HOME}'"

$ just
HOME is '/home/myuser'
```

## Setting just Variables from Environment Variables

Environment variables can be propagated to `just` variables using the functions `env_var()` and `env_var_or_default()`. See [environment-variables](#).



## Recipe Parameters

Recipes may have parameters. Here recipe `build` has a parameter called `target`:

```
build target:
 @echo 'Building {{target}}...'
 cd {{target}} && make
```

To pass arguments on the command line, put them after the recipe name:

```
$ just build my-awesome-project
Building my-awesome-project...
cd my-awesome-project && make
```

To pass arguments to a dependency, put the dependency in parentheses along with the arguments:

```
default: (build "main")

build target:
 @echo 'Building {{target}}...'
 cd {{target}} && make
```

Variables can also be passed as arguments to dependencies:

```
target := "main"

_build version:
 @echo 'Building {{version}}...'
 cd {{version}} && make

build: (_build target)
```

A command's arguments can be passed to dependency by putting the dependency in parentheses along with the arguments:

```
build target:
 @echo "Building {{target}}..."

push target: (build target)
 @echo 'Pushing {{target}}...'
```



Parameters may have default values:

```
default := 'all'

test target tests=default:
@echo 'Testing {{target}}:{{tests}}...'
./test --tests {{tests}} {{target}}
```

Parameters with default values may be omitted:

```
$ just test server
Testing server:all...
./test --tests all server
```

Or supplied:

```
$ just test server unit
Testing server:unit...
./test --tests unit server
```

Default values may be arbitrary expressions, but concatenations or path joins must be parenthesized:

```
arch := "wasm"

test triple=(arch + "-unknown-unknown") input=(arch / "input.dat"):
./test {{triple}}
```

The last parameter of a recipe may be variadic, indicated with either a `+` or a `*` before the argument name:

```
backup +FILES:
scp {{FILES}} me@server.com:
```

Variadic parameters prefixed with `+` accept *one or more* arguments and expand to a string containing those arguments separated by spaces:

```
$ just backup FAQ.md GRAMMAR.md
scp FAQ.md GRAMMAR.md me@server.com:
FAQ.md 100% 1831 1.8KB/s 00:00
GRAMMAR.md 100% 1666 1.6KB/s 00:00
```



Variadic parameters prefixed with `*` accept *zero or more* arguments and expand to a string containing those arguments separated by spaces, or an empty string if no arguments are present:

```
commit MESSAGE *FLAGS:
git commit {{FLAGS}} -m "{{MESSAGE}}"
```

Variadic parameters can be assigned default values. These are overridden by arguments passed on the command line:

```
test +FLAGS='-q':
cargo test {{FLAGS}}
```

{{...}} substitutions may need to be quoted if they contain spaces. For example, if you have the following recipe:

```
search QUERY:
lynx https://www.google.com/?q={{QUERY}}
```

And you type:

```
$ just search "cat toupee"
```

just will run the command `lynx https://www.google.com/?q=cat toupee`, which will get parsed by sh as `lynx`, `https://www.google.com/?q=cat`, and `toupee`, and not the intended `lynx` and `https://www.google.com/?q=cat toupee`.

You can fix this by adding quotes:

```
search QUERY:
lynx 'https://www.google.com/?q={{QUERY}}'
```

Parameters prefixed with a \$ will be exported as environment variables:

```
foo $bar:
echo $bar
```



## Running Recipes at the End of a Recipe

Normal dependencies of a recipes always run before a recipe starts. That is to say, the dependee always runs before the depender. These dependencies are called “prior dependencies”.

A recipe can also have subsequent dependencies, which run after the recipe and are introduced with an `&&`:

```
a:
 echo 'A!'

b: a && c d
 echo 'B!'

c:
 echo 'C!'

d:
 echo 'D!'
```

...running *b* prints:

```
$ just b
echo 'A!'
A!
echo 'B!'
B!
echo 'C!'
C!
echo 'D!'
D!
```



## Running Recipes in the Middle of a Recipe

`just` doesn't support running recipes in the middle of another recipe, but you can call `just` recursively in the middle of a recipe. Given the following `justfile`:

```
a:
 echo 'A!'

b: a
 echo 'B start!'
 just c
 echo 'B end!'

c:
 echo 'C!'
```

...running *b* prints:

```
$ just b
echo 'A!'
A!
echo 'B start!'
B start!
echo 'C!'
C!
echo 'B end!'
B end!
```

This has limitations, since recipe `c` is run with an entirely new invocation of `just`: Assignments will be recalculated, dependencies might run twice, and command line arguments will not be propagated to the child `just` process.



## Writing Recipes in Other Languages

Recipes that start with `#!` are called shebang recipes, and are executed by saving the recipe body to a file and running it. This lets you write recipes in different languages:

```
polyglot: python js perl sh ruby nu

python:
#!/usr/bin/env python3
print('Hello from python!')

js:
#!/usr/bin/env node
console.log('Greetings from JavaScript!')

perl:
#!/usr/bin/env perl
print "Larry Wall says Hi!\n";

sh:
#!/usr/bin/env sh
hello='Yo'
echo "$hello from a shell script!"

nu:
#!/usr/bin/env nu
let hello = 'Hola'
echo $($hello) from a nushell script!

ruby:
#!/usr/bin/env ruby
puts "Hello from ruby!"

$ just polyglot
Hello from python!
Greetings from JavaScript!
Larry Wall says Hi!
Yo from a shell script!
Hola from a nushell script!
Hello from ruby!
```



On Unix-like operating systems, including Linux and MacOS, shebang recipes are executed by saving the recipe body to a file in a temporary directory, marking the file as executable, and executing it. The OS then parses the shebang line into a command line and invokes it, including the path to the file. For example, if a recipe starts with `#!/usr/bin/env bash`, the final command that the OS runs will be something like `/usr/bin/env bash /tmp/PATH_TO_SAVED_RECIPE_BODY`. Keep in mind that different operating systems split shebang lines differently.

Windows does not support shebang lines. On Windows, `just` splits the shebang line into a command and arguments, saves the recipe body to a file, and invokes the `split` command and arguments, adding the path to the saved recipe body as the final argument. For example, on Windows, if a recipe starts with `#! py`, the final command the OS runs will be something like `py C:\Temp\PATH_TO_SAVED_RECIPE_BODY`.



## Safer Bash Shebang Recipes

If you're writing a `bash` shebang recipe, consider adding `set -euxo pipefail`:

```
foo:
#!/usr/bin/env bash
set -euxo pipefail
hello='Yo'
echo "$hello from Bash!"
```

It isn't strictly necessary, but `set -euxo pipefail` turns on a few useful features that make `bash` shebang recipes behave more like normal, linewise `just` recipe:

- `set -e` makes `bash` exit if a command fails.
- `set -u` makes `bash` exit if a variable is undefined.
- `set -x` makes `bash` print each script line before it's run.
- `set -o pipefail` makes `bash` exit if a command in a pipeline fails. This is `bash`-specific, so isn't turned on in normal linewise `just` recipes.

Together, these avoid a lot of shell scripting gotchas.

## Shebang Recipe Execution on Windows

On Windows, shebang interpreter paths containing a `/` are translated from Unix-style paths to Windows-style paths using `cygpath`, a utility that ships with [Cygwin](#).

For example, to execute this recipe on Windows:

```
echo:
#!/bin/sh
echo "Hello!"
```

The interpreter path `/bin/sh` will be translated to a Windows-style path using `cygpath` before being executed.



If the interpreter path does not contain a `/` it will be executed without being translated. This is useful if `cygpath` is not available, or you wish to pass a Windows-style path to the interpreter.

## Setting Variables in a Recipe

Recipe lines are interpreted by the shell, not `just`, so it's not possible to set `just` variables in the middle of a recipe:

```
foo:
 x := "hello" # This doesn't work!
 echo {{x}}
```

It is possible to use shell variables, but there's another problem. Every recipe line is run by a new shell instance, so variables set in one line won't be set in the next:

```
foo:
 x=hello && echo $x # This works!
 y=bye
 echo $y # This doesn't, `y` is undefined here!
```

The best way to work around this is to use a shebang recipe. Shebang recipe bodies are extracted and run as scripts, so a single shell instance will run the whole thing:

```
foo:
 #!/usr/bin/env bash
 set -euxo pipefail
 x=hello
 echo $x
```



## Sharing Environment Variables Between Recipes

Each line of each recipe is executed by a fresh shell, so it is not possible to share environment variables between recipes.

## Using Python Virtual Environments

Some tools, like [Python's venv](#), require loading environment variables in order to work, making them challenging to use with `just`. As a workaround, you can execute the virtual environment binaries directly:

```
venv:
 [-d foo] || python3 -m venv foo

run: venv
 ./foo/bin/python3 main.py
```



## Changing the Working Directory in a Recipe

Each recipe line is executed by a new shell, so if you change the working directory on one line, it won't have an effect on later lines:

```
foo:
 pwd # This `pwd` will print the same directory...
 cd bar
 pwd # ...as this `pwd`!
```

There are a couple ways around this. One is to call `cd` on the same line as the command you want to run:

```
foo:
 cd bar && pwd
```

The other is to use a shebang recipe. Shebang recipe bodies are extracted and run as scripts, so a single shell instance will run the whole thing, and thus a `pwd` on one line will affect later lines, just like a shell script:

```
foo:
 #!/usr/bin/env bash
 set -euxo pipefail
 cd bar
 pwd
```



## Indentation

Recipe lines can be indented with spaces or tabs, but not a mix of both. All of a recipe's lines must have the same type of indentation, but different recipes in the same `justfile` may use different indentation.

Each recipe must be indented at least one level from the `recipe-name` but after that may be further indented.

Here's a justfile with a recipe indented with spaces, represented as `·`, and tabs, represented as `→`.

```
set windows-shell := ["pwsh", "-NoLogo", "-NoProfileLoadTime", "-Command"]

set ignore-comments

list-space directory:
··#!pwsh
··foreach ($item in $(Get-ChildItem {{directory}})) {
····echo $item.Name
··}
··echo ""

indentation nesting works even when newlines are escaped
list-tab directory:
→ @foreach ($item in $(Get-ChildItem {{directory}})) { \
→ → echo $item.Name \
→ }
→ @echo ""

PS > just list-space ~
Desktop
Documents
Downloads

PS > just list-tab ~
Desktop
Documents
Downloads
```



## Multi-Line Constructs

Recipes without an initial shebang are evaluated and run line-by-line, which means that multi-line constructs probably won't do what you want.

For example, with the following `justfile`:

```
conditional:
 if true; then
 echo 'True!'
 fi
```

The extra leading whitespace before the second line of the `conditional` recipe will produce a parse error:

```
$ just conditional
error: Recipe line has extra leading whitespace
|
3 | echo 'True!'
| ^^^^^^
```

To work around this, you can write conditionals on one line, escape newlines with slashes, or add a shebang to your recipe. Some examples of multi-line constructs are provided for reference.

### if statements

```
conditional:
 if true; then echo 'True!'; fi
```

```
conditional:
 if true; then \
 echo 'True!'; \
 fi
```

```
conditional:
#!/usr/bin/env sh
 if true; then
 echo 'True!'
 fi
```



### for loops

```
for:
 for file in `ls .`; do echo $file; done

for:
 for file in `ls .`; do \
 echo $file; \
 done

for:
 #!/usr/bin/env sh
 for file in `ls .`; do
 echo $file
 done
```

## while loops

```
while:
 while `server-is-dead`; do ping -c 1 server; done

while:
 while `server-is-dead`; do \
 ping -c 1 server; \
 done

while:
 #!/usr/bin/env sh
 while `server-is-dead`; do
 ping -c 1 server
 done
```

## Outside Recipe Bodies

Parenthesized expressions can span multiple lines:



```
abc := ('a' +
 'b'
 + 'c')

abc2 := (
 'a' +
 'b' +
 'c'
)

foo param=('foo'
 + 'bar'
):
echo {{param}>

bar: (foo
 'Foo'
)
echo 'Bar!'
```

Lines ending with a backslash continue on to the next line as if the lines were joined by whitespace<sup>1.15.0</sup>:

```
a := 'foo' + \
 'bar'

foo param1 \
 param2='foo' \
 *varparam='': dep1 \
 (dep2 'foo')
echo {{param1}} {{param2}} {{varparam}>

dep1: \
 # this comment is not part of the recipe body
echo 'dep1'

dep2 \
param:
echo 'Dependency with parameter {{param}}'
```

Backslash line continuations can also be used in interpolations. The line following the backslash must start with the same indentation as the recipe body, although additional indentation is accepted.



```
recipe:
echo '{{ \
"This interpolation " + \
"has a lot of text." \
}}'
echo 'back to recipe body'
```



## Command Line Options

`just` supports a number of useful command line options for listing, dumping, and debugging recipes and variables:

```
$ just --list
Available recipes:
 js
 perl
 polyglot
 python
 ruby
$ just --show perl
perl:
 #!/usr/bin/env perl
 print "Larry Wall says Hi!\n";
$ just --show polyglot
polyglot: python js perl sh ruby
```

Run `just --help` to see all the options.



## Private Recipes

Recipes and aliases whose name starts with a `_` are omitted from `just --list`:

```
test: _test-helper
./bin/test

_test-helper:
./bin/super-secret-test-helper-stuff
```

```
$ just --list
Available recipes:
 test
```

And from `just --summary`:

```
$ just --summary
test
```

The `[private]` attribute<sup>1.10.0</sup> may also be used to hide recipes or aliases without needing to change the name:

```
[private]
foo:

[private]
alias b := bar

bar:
```

```
$ just --list
Available recipes:
 bar
```

This is useful for helper recipes which are only meant to be used as dependencies of other recipes.



## Quiet Recipes

A recipe name may be prefixed with `@` to invert the meaning of `@` before each line:

```
@quiet:
 echo hello
 echo goodbye
 #@ all done!
```

Now only the lines starting with `@` will be echoed:

```
$ just quiet
hello
goodbye
all done!
```

All recipes in a Justfile can be made quiet with `set quiet`:

```
set quiet

foo:
 echo "This is quiet"

@foo2:
 echo "This is also quiet"
```

The `[no-quiet]` attribute overrides this setting:

```
set quiet

foo:
 echo "This is quiet"

[no-quiet]
foo2:
 echo "This is not quiet"
```

Shebang recipes are quiet by default:

```
foo:
 #!/usr/bin/env bash
 echo 'Foo!'
```

```
$ just foo
Foo!
```



Adding @ to a shebang recipe name makes `just` print the recipe before executing it:

```
@bar:
#!/usr/bin/env bash
echo 'Bar!'
```

```
$ just bar
#!/usr/bin/env bash
echo 'Bar!'
Bar!
```

`just` normally prints error messages when a recipe line fails. These error messages can be suppressed using the `[no-exit-message]`<sup>1.7.0</sup> attribute. You may find this especially useful with a recipe that wraps a tool:

```
git *args:
@git {{args}}
```

```
$ just git status
fatal: not a git repository (or any of the parent directories): .git
error: Recipe `git` failed on line 2 with exit code 128
```

Add the attribute to suppress the exit error message when the tool exits with a non-zero code:

```
[no-exit-message]
git *args:
@git {{args}}
```

```
$ just git status
fatal: not a git repository (or any of the parent directories): .git
```



## Selecting Recipes to Run With an Interactive Chooser

The `--choose` subcommand makes `just` invoke a chooser to select which recipes to run. Choosers should read lines containing recipe names from standard input and print one or more of those names separated by spaces to standard output.

Because there is currently no way to run a recipe that requires arguments with `--choose`, such recipes will not be given to the chooser. Private recipes and aliases are also skipped.

The chooser can be overridden with the `--chooser` flag. If `--chooser` is not given, then `just` first checks if `$JUST_CHOOSER` is set. If it isn't, then the chooser defaults to `fzf`, a popular fuzzy finder.

Arguments can be included in the chooser, i.e. `fzf --exact`.

The chooser is invoked in the same way as recipe lines. For example, if the chooser is `fzf`, it will be invoked with `sh -cu 'fzf'`, and if the shell, or the shell arguments are overridden, the chooser invocation will respect those overrides.

If you'd like `just` to default to selecting recipes with a chooser, you can use this as your default recipe:

```
default:
@just --choose
```



## Invoking justfiles in Other Directories

If the first argument passed to `just` contains a `/`, then the following occurs:

1. The argument is split at the last `/`.
2. The part before the last `/` is treated as a directory. `just` will start its search for the `justfile` there, instead of in the current directory.
3. The part after the last slash is treated as a normal argument, or ignored if it is empty.

This may seem a little strange, but it's useful if you wish to run a command in a `justfile` that is in a subdirectory.

For example, if you are in a directory which contains a subdirectory named `foo`, which contains a `justfile` with the recipe `build`, which is also the default recipe, the following are all equivalent:

```
$ (cd foo && just build)
$ just foo/build
$ just foo/
```

Additional recipes after the first are sought in the same `justfile`. For example, the following are both equivalent:

```
$ just foo/a b
$ (cd foo && just a b)
```

And will both invoke recipes `a` and `b` in `foo/justfile`.



## Imports

One `justfile` can include the contents of another using `import` statements.

If you have the following `justfile`:

```
import 'foo/bar.just'

a: b
@echo A
```

And the following text in `foo/bar.just`:

```
b:
@echo B
```

`foo/bar.just` will be included in `justfile` and recipe `b` will be defined:

```
$ just b
B
$ just a
B
A
```

The `import` path can be absolute or relative to the location of the `justfile` containing it. A leading `~/` in the import path is replaced with the current users home directory.

Justfiles are insensitive to order, so included files can reference variables and recipes defined after the `import` statement.

Imported files can themselves contain `import`s, which are processed recursively.

When `allow-duplicate-recipes` is set, recipes in parent modules override recipes in imports.

Imports may be made optional by putting a `?` after the `import` keyword:

```
import? 'foo/bar.just'
```



Missing source files for optional imports do not produce an error.

## Modules<sup>1.19.0</sup>

A `justfile` can declare modules using `mod` statements. `mod` statements are currently unstable, so you'll need to use the `--unstable` flag, or set the `JUST_UNSTABLE` environment variable to use them.

If you have the following `justfile`:

```
mod bar

a:
 @echo A
```

And the following text in `bar.just`:

```
b:
 @echo B
```

`bar.just` will be included in `justfile` as a submodule. Recipes, aliases, and variables defined in one submodule cannot be used in another, and each module uses its own settings.

Recipes in submodules can be invoked as subcommands:

```
$ just --unstable bar b
B
```

Or with path syntax:

```
$ just --unstable bar::b
B
```

If a module is named `foo`, just will search for the module file in `foo.just`, `foo/mod.just`, `foo/justfile`, and `foo/.justfile`. In the latter two cases, the module file may have any capitalization.



Module statements may be of the form:

```
mod foo 'PATH'
```

Which loads the module's source file from `PATH`, instead of from the usual locations. A leading `~`/ in `PATH` is replaced with the current user's home directory.

Environment files are only loaded for the root justfile, and loaded environment variables are available in submodules. Settings in submodules that affect environment file loading are ignored.

Recipes in submodules without the `[no-cd]` attribute run with the working directory set to the directory containing the submodule source file.

`justfile()` and `justfile_directory()` always return the path to the root justfile and the directory that contains it, even when called from submodule recipes.

Modules may be made optional by putting a `?` after the `mod` keyword:

```
mod? foo
```

Missing source files for optional modules do not produce an error.

Optional modules with no source file do not conflict, so you can have multiple `mod` statements with the same name, but with different source file paths, as long as at most one source file exists:

```
mod? foo 'bar.just'
mod? foo 'baz.just'
```

See the [module stabilization tracking issue](#) for more information.



## Hiding justfiles

`just` looks for `justfiles` named `justfile` and `.justfile`, which can be used to keep a `justfile` hidden.



## Just Scripts

By adding a shebang line to the top of a `justfile` and making it executable, `just` can be used as an interpreter for scripts:

```
$ cat > script <<EOF
#!/usr/bin/env just --justfile

foo:
 echo foo
EOF
$ chmod +x script
$./script foo
echo foo
foo
```

When a script with a shebang is executed, the system supplies the path to the script as an argument to the command in the shebang. So, with a shebang of `#!/usr/bin/env just --justfile`, the command will be `/usr/bin/env just --justfile PATH_TO_SCRIPT`.

With the above shebang, `just` will change its working directory to the location of the script. If you'd rather leave the working directory unchanged, use `#!/usr/bin/env just --working-directory . --justfile`.

Note: Shebang line splitting is not consistent across operating systems. The previous examples have only been tested on macOS. On Linux, you may need to pass the `-s` flag to `env`:

```
#!/usr/bin/env -s just --justfile

default:
 echo foo
```



## Formatting and dumping justfiles

Each `justfile` has a canonical formatting with respect to whitespace and newlines.

You can overwrite the current `justfile` with a canonically-formatted version using the currently-unstable `--fmt` flag:

```
$ cat justfile
A lot of blank lines
```

```
some-recipe:
 echo "foo"
$ just --fmt --unstable
$ cat justfile
A lot of blank lines

some-recipe:
 echo "foo"
```

Invoking `just --fmt --check --unstable` runs `--fmt` in check mode. Instead of overwriting the `justfile`, `just` will exit with an exit code of 0 if it is formatted correctly, and will exit with 1 and print a diff if it is not.

You can use the `--dump` command to output a formatted version of the `justfile` to stdout:

```
$ just --dump > formatted-justfile
```

The `--dump` command can be used with `--dump-format json` to print a JSON representation of a `justfile`.



## Fallback to parent justfiles

If a recipe is not found in a `justfile` and the `fallback` setting is set, `just` will look for `justfile`s in the parent directory and up, until it reaches the root directory. `just` will stop after it reaches a `justfile` in which the `fallback` setting is `false` or unset.

As an example, suppose the current directory contains this `justfile`:

```
set fallback
foo:
 echo foo
```

And the parent directory contains this `justfile`:

```
bar:
 echo bar
```

```
$ just bar
Trying ../justfile
echo bar
bar
```



## Avoiding Argument Splitting

Given this `justfile`:

```
foo argument:
 touch {{argument}}
```

The following command will create two files, `some` and `argument.txt`:

```
$ just foo "some argument.txt"
```

The user's shell will parse `"some argument.txt"` as a single argument, but when `just` replaces `touch {{argument}}` with `touch some argument.txt`, the quotes are not preserved, and `touch` will receive two arguments.

There are a few ways to avoid this: quoting, positional arguments, and exported arguments.

### Quoting

Quotes can be added around the `{{argument}}` interpolation:

```
foo argument:
 touch '{{argument}}'
```

This preserves `just`'s ability to catch variable name typos before running, for example if you were to write `{}{argument}`, but will not do what you want if the value of `argument` contains single quotes.

### Positional Arguments

The `positional-arguments` setting causes all arguments to be passed as positional arguments, allowing them to be accessed with `$1`, `$2`, ..., and `$@`, which can be then double-quoted to avoid further splitting by the shell:



```
set positional-arguments

foo argument:
 touch "$1"
```

This defeats `just`'s ability to catch typos, for example if you type `$2`, but works for all possible values of `argument`, including those with double quotes.

## Exported Arguments

All arguments are exported when the `export` setting is set:

```
set export

foo argument:
 touch "$argument"
```

Or individual arguments may be exported by prefixing them with `$`:

```
foo $argument:
 touch "$argument"
```

This defeats `just`'s ability to catch typos, for example if you type `$argumant`, but works for all possible values of `argument`, including those with double quotes.



## Configuring the Shell

There are a number of ways to configure the shell for linewise recipes, which are the default when a recipe does not start with a `#!` shebang. Their precedence, from highest to lowest, is:

1. The `--shell` and `--shell-arg` command line options. Passing either of these will cause `just` to ignore any settings in the current justfile.
2. `set windows-shell := [...]`
3. `set windows-powershell` (deprecated)
4. `set shell := [...]`

Since `set windows-shell` has higher precedence than `set shell`, you can use `set windows-shell` to pick a shell on Windows, and `set shell` to pick a shell for all other platforms.



## Changelog

A changelog for the latest release is available in [CHANGELOG.md](#). Changelogs for previous releases are available on [the releases page](#). `just --changelog` can also be used to make a `just` binary print its changelog.



## Miscellanea



## Companion Tools

Tools that pair nicely with `just` include:

- [watchexec](#) — a simple tool that watches a path and runs a command whenever it detects modifications.



## Shell Alias

For lightning-fast command running, put `alias j=just` in your shell's configuration file.

In `bash`, the aliased command may not keep the shell completion functionality described in the next section. Add the following line to your `.bashrc` to use the same completion function as `just` for your aliased command:

```
complete -F _just -o bashdefault -o default j
```



## Shell Completion Scripts

Shell completion scripts for Bash, Zsh, Fish, PowerShell, and Elvish are available in the [completions](#) directory. Please refer to your shell's documentation for how to install them.

The `just` binary can also generate the same completion scripts at runtime, using the `--completions` command:

```
$ just --completions zsh > just.zsh
```

*macOS Note:* Recent versions of macOS use zsh as the default shell. If you use Homebrew to install `just`, it will automatically install the most recent copy of the zsh completion script in the Homebrew zsh directory, which the built-in version of zsh doesn't know about by default. It's best to use this copy of the script if possible, since it will be updated whenever you update `just` via Homebrew. Also, many other Homebrew packages use the same location for completion scripts, and the built-in zsh doesn't know about those either. To take advantage of `just` completion in zsh in this scenario, you can set `fpath` to the Homebrew location before calling `compinit`. Note also that Oh My Zsh runs `compinit` by default. So your `.zshrc` file could look like this:

```
Init Homebrew, which adds environment variables
eval "$(brew shellenv)"

fpath=($HOME_BREW_PREFIX/share/zsh/site-functions $fpath)

Then choose one of these options:
1. If you're using Oh My Zsh, you can initialize it here
source $ZSH/oh-my-zsh.sh

2. Otherwise, run compinit yourself
autoload -U compinit
compinit
```



## Grammar

A non-normative grammar of `justfile`s can be found in [GRAMMAR.md](#).



## just.sh

Before `just` was a fancy Rust program it was a tiny shell script that called `make`. You can find the old version in [contrib/just.sh](#).



## User justfiles

If you want some recipes to be available everywhere, you have a few options.

First, create a `justfile` in `~/.user.justfile` with some recipes.

### Recipe Aliases

If you want to call the recipes in `~/.user.justfile` by name, and don't mind creating an alias for every recipe, add the following to your shell's initialization script:

```
for recipe in `just --justfile ~/.user.justfile --summary`; do
 alias $recipe="just --justfile ~/.user.justfile --working-directory .
$recipe"
done
```

Now, if you have a recipe called `foo` in `~/.user.justfile`, you can just type `foo` at the command line to run it.

It took me way too long to realize that you could create recipe aliases like this. Notwithstanding my tardiness, I am very pleased to bring you this major advance in `justfile` technology.

### Forwarding Alias

If you'd rather not create aliases for every recipe, you can create a single alias:

```
alias .j='just --justfile ~/.user.justfile --working-directory .'
```

Now, if you have a recipe called `foo` in `~/.user.justfile`, you can just type `.j foo` at the command line to run it.

I'm pretty sure that nobody actually uses this feature, but it's there.

↖\_(ツ)\_↗



### Customization

You can customize the above aliases with additional options. For example, if you'd prefer to have the recipes in your `justfile` run in your home directory, instead of the current directory:

```
alias .j='just --justfile ~/.user.justfile --working-directory ~'
```



## Node.js package.json Script Compatibility

The following export statement gives `just` recipes access to local Node module binaries, and makes `just` recipe commands behave more like `script` entries in Node.js `package.json` files:

```
export PATH := "./node_modules/.bin:" + env_var('PATH')
```



## Alternatives and Prior Art

There is no shortage of command runners! Some more or less similar alternatives to `just` include:

- [make](#): The Unix build tool that inspired `just`. There are a few different modern day descendants of the original `make`, including [FreeBSD Make](#) and [GNU Make](#).
- [task](#): A YAML-based command runner written in Go.
- [maid](#): A Markdown-based command runner written in JavaScript.
- [microsoft/just](#): A JavaScript-based command runner written in JavaScript.
- [cargo-make](#): A command runner for Rust projects.
- [mmake](#): A wrapper around `make` with a number of improvements, including remote includes.
- [robo](#): A YAML-based command runner written in Go.
- [mask](#): A Markdown-based command runner written in Rust.
- [makesure](#): A simple and portable command runner written in AWK and shell.
- [haku](#): A make-like command runner written in Rust.



# Contributing

just welcomes your contributions! just is released under the maximally permissive [CC0](#) public domain dedication and fallback license, so your changes must also be released under this license.



## Janus

[Janus](#) is a tool that collects and analyzes `justfile`s, and can determine if a new version of `just` breaks or changes the interpretation of existing `justfile`s.

Before merging a particularly large or gruesome change, Janus should be run to make sure that nothing breaks. Don't worry about running Janus yourself, Casey will happily run it for you on changes that need it.



## Minimum Supported Rust Version

The minimum supported Rust version, or MSRV, is current stable Rust. It may build on older versions of Rust, but this is not guaranteed.



## New Releases

New releases of `just` are made frequently so that users quickly get access to new features.

Release commit messages use the following template:

`Release x.y.z`

- Bump version: `x.y.z → x.y.z`
- Update changelog
- Update changelog contributor credits
- Update dependencies
- Update man page
- Update version references in readme



# Frequently Asked Questions



## What are the idiosyncrasies of Make that Just avoids?

`make` has some behaviors which are confusing, complicated, or make it unsuitable for use as a general command runner.

One example is that under some circumstances, `make` won't actually run the commands in a recipe. For example, if you have a file called `test` and the following makefile:

```
test:
 ./test
```

`make` will refuse to run your tests:

```
$ make test
make: `test' is up to date.
```

`make` assumes that the `test` recipe produces a file called `test`. Since this file exists and the recipe has no other dependencies, `make` thinks that it doesn't have anything to do and exits.

To be fair, this behavior is desirable when using `make` as a build system, but not when using it as a command runner. You can disable this behavior for specific targets using `make`'s built-in `.PHONY target name`, but the syntax is verbose and can be hard to remember. The explicit list of phony targets, written separately from the recipe definitions, also introduces the risk of accidentally defining a new non-phony target. In `just`, all recipes are treated as if they were phony.

Other examples of `make`'s idiosyncrasies include the difference between `=` and `:=` in assignments, the confusing error messages that are produced if you mess up your makefile, needing `$$` to use environment variables in recipes, and incompatibilities between different flavors of `make`.



## What's the relationship between Just and Cargo build scripts?

`cargo build scripts` have a pretty specific use, which is to control how `cargo` builds your Rust project. This might include adding flags to `rustc` invocations, building an external dependency, or running some kind of codegen step.

`just`, on the other hand, is for all the other miscellaneous commands you might run as part of development. Things like running tests in different configurations, linting your code, pushing build artifacts to a server, removing temporary files, and the like.

Also, although `just` is written in Rust, it can be used regardless of the language or build system your project uses.



## Further Ramblings

I personally find it very useful to write a `justfile` for almost every project, big or small.

On a big project with multiple contributors, it's very useful to have a file with all the commands needed to work on the project close at hand.

There are probably different commands to test, build, lint, deploy, and the like, and having them all in one place is useful and cuts down on the time you have to spend telling people which commands to run and how to type them.

And, with an easy place to put commands, it's likely that you'll come up with other useful things which are part of the project's collective wisdom, but which aren't written down anywhere, like the arcane commands needed for some part of your revision control workflow, to install all your project's dependencies, or all the random flags you might need to pass to the build system.

Some ideas for recipes:

- Deploying/publishing the project
- Building in release mode vs debug mode
- Running in debug mode or with logging enabled
- Complex git workflows
- Updating dependencies
- Running different sets of tests, for example fast tests vs slow tests, or running them with verbose output
- Any complex set of commands that you really should write down somewhere, if only to be able to remember them

Even for small, personal projects it's nice to be able to remember commands by name instead of ^Reverse searching your shell history, and it's a huge boon to be able to go into old project written in a random language with a mysterious build system and know that all the commands you need to do whatever you need to do are in the `justfile`, and that if you type `just` something useful (or at least interesting!) will probably happen.

For ideas for recipes, check out [this project's justfile](#), or some of the [justfiles out in the wild](#).

Anyways, I think that's about it for this incredibly long-winded README.

I hope you enjoy using `just` and find great success and satisfaction in all your computational endeavors!



# Module system deep dive

## Contents

- Overview
- The empty module
- Module arguments
- Declaring options
- Evaluating modules
- Type checking
- Interlude: reproducible scripts
- Declaring more options
- Dependencies between options
- Conditional definitions
- Default values
- Wrapping shell commands
- Splitting modules
- The `submodule` type
- Defining options in other modules
- Nested submodules
- The `strMatching` type
- Functions as submodule arguments
- The `either` and `enum` types
- The `pathType` submodule
- The `between` constraint on integer values
- The `pathStyle` submodule
- Path styling: color
- Further styling

Or: *Wrapping the world in modules*

Much of the power in Nixpkgs and NixOS comes from the module system. It provides mechanisms for conveniently declaring and automatically merging interdependent attribute sets that follow dynamic type constraints, making it easy to express modular configurations.

## Overview

This tutorial follows [@infinisil's presentation on modules \(source\)](#) for participants of [Summer of Nix 2021](#).

It may help playing it alongside this tutorial to better keep track of changes to the code you will work on.

## What will you learn?

In this tutorial you'll learn

- what a module is
- how to define one
- what options are
- how to declare them
- how to express dependencies between modules

and follow an extensive demonstration of how to wrap an existing API with Nix modules.

Concretely, you'll write modules to interact with the [Google Maps API](#), declaring options which represent map geometry, location pins, and more.

During the tutorial, you will first write some *incorrect* configurations, creating opportunities to discuss the resulting error messages and how to resolve them, particularly when discussing type checking.

## What do you need?

- Familiarity with data types and general programming concepts
- A [Nix installation](#) to run the examples
- Intermediate proficiency in reading and writing the Nix language

You will use two helper scripts for this exercise. Download  [map](#) and  [geocode](#) to your working directory.

### Warning

To run the examples in this tutorial, you will need a [Google API key](#) in  
`$XDG_DATA_HOME/google-api/key`.

## How long will it take?

This is a very long tutorial. Prepare for at least 3 hours of work.

## The empty module

We have to start somewhere. The simplest module is just a function that takes any attributes and returns an empty attribute set.

Write the following into a file called `default.nix`:

```
default.nix
```

```
{ ... }:
{
}
```

## Module arguments

We will need some helper functions, which will come from the Nixpkgs library. Start by changing the first line in `default.nix`:

### default.nix

```
- { ... }:
+ { lib, ... }:
{

}
```

Now the module is a function which takes *at least* one argument, called `lib`, and may accept other arguments (expressed by the ellipsis `...`).

This will make Nixpkgs library functions available within the function body. The `lib` argument is passed automatically by the module system.

#### Note

The ellipsis `...` is necessary because arbitrary arguments can be passed to modules.

## Declaring options

To set any values, the module system first has to know which ones are allowed.

This is done by declaring *options* that specify which values can be set and used elsewhere. Options are declared by adding an attribute under the top-level `options` attribute, using `lib.mkOption`.

In this section, you will define the `scripts.output` option.

Change `default.nix` to include the following declaration:

### default.nix

```
{ lib, ... }:

+ options = {
+ scripts.output = lib.mkOption {
+ type = lib.types.lines;
+ };
+ };
```

}

While many attributes for customizing options are available, the most important one is `type`, which specifies which values are valid for an option. There are several types available under `lib.types` in the Nixpkgs library.

You have just declared `scripts.output` with the `lines` type, which specifies that the only valid values are strings, and that multiple definitions should be joined with newlines.

### Note

The name and attribute path of the option is arbitrary. Here we use `scripts`, because we will add another script later, and call this one `output`, because it will output the resulting map.

## Evaluating modules

Write a new file, `eval.nix`, which you will use to evaluate `default.nix`:

eval.nix

```
let
 nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-22.
 pkgs = import nixpkgs { config = {}; overlays = []; };
in
pkgs.lib.evalModules {
 modules = [
 ./default.nix
];
}
```

`evalModules` is the function that evaluates modules, applies type checking, and merges values into the final attribute set. It expects a `modules` attribute whose value is a list, where each element can be a path to a module or an expression that follows the `module schema`.

Run the following command:

### ⚠ Warning

This will result in an error.

```
nix-instantiate --eval eval.nix -A config.scripts.output
```

`nix-instantiate --eval` parses and evaluates the Nix file at the specified path, and prints the result. `evalModules` produces an attribute set where the final configuration values appear in the `config` attribute. Therefore we evaluate the Nix expression in `eval.nix` at the attribute path `config.scripts.output`.

The error message indicates that the `scripts.output` option is used but not defined: a value must be set for the option before accessing it. You will do this in the next steps.

## Type checking

As previously mentioned, the `lines` type only permits string values.

### ⚠ Warning

In this section, you will set an invalid value and encounter a type error.

What happens if you instead try to assign an integer to the option?

Add the following lines to `default.nix`:

default.nix

```
{ lib, ... }: {

 options = {
 scripts.output = lib.mkOption {
 type = lib.types.lines;
 };
 };

 + config = {
 + scripts.output = 42;
 + };
};
```

}

Now try to execute the previous command, and witness your first module error:

```
$ nix-instantiate --eval eval.nix -A config.scripts.output
error:
...
 error: A definition for option `scripts.output' is not of type `strin
- In `/home/nix-user/default.nix': 42
```

The definition `scripts.output = 42;` caused a type error: integers are not strings concatenated with the newline character.

To make this module pass the type checks and successfully evaluate the `scripts.output` option, you will now assign a string to `scripts.output`.

In this case, you will assign a shell command that runs the [map](#) script in the current directory. That in turn calls the Google Maps Static API to generate a world map. The output is passed on to display it with [feh](#), a minimalistic image viewer.

Update `default.nix` by changing the value of `scripts.output` to the following string:

default.nix

```
config = {
- scripts.output = 42;
+ scripts.output = ''
+ ./map size=640x640 scale=2 | feh -
+ '';
};
```

## Interlude: reproducible scripts

That simple command will likely not work as intended on your system, as it may lack the required dependencies (`curl` and `feh`). We can solve this by packaging the raw [map](#) script with `pkgs.writeShellApplication`.

First, make available a `pkgs` argument in your module evaluation by adding a module that sets `config._module.args`:

## eval.nix

```
pkgs.lib.evalModules {
 modules = [
+ ({ config, ... }: { config._module.args = { inherit pkgs; }; })
 ./test.nix
];
}
```

**i Note**

This mechanism is currently only [documented in the module system code](#), and that documentation is incomplete and out of date.

Then change `default.nix` to have the following contents:

## default.nix

```
{ pkgs, lib, ... }: {

 options = {
 scripts.output = lib.mkOption {
 type = lib.types.package;
 };
 };

 config = {
 scripts.output = pkgs.writeShellApplication {
 name = "map";
 runtimeInputs = with pkgs; [curl feh];
 text = ''
 ${./map} size=640x640 scale=2 | feh -
 '';
 };
 };
}
```

This will access the previously added `pkgs` argument so we can use dependencies, and copy the `map` file in the current directory into the Nix store so it's available to the wrapped script, which will also live in the Nix store.

Run the script with:

```
nix-build eval.nix -A config.scripts.output
./result/bin/map
```

To iterate more quickly, open a new terminal and set up `entr` to re-run the script whenever any source file in the current directory changes:

```
nix-shell -p entr findutils bash --run \
"ls *.nix | \
 entr -rs ' \
 nix-build eval.nix -A config.scripts.output --no-out-link \
 | xargs printf -- \"%s/bin/map\" \
 | xargs bash \
' \
"
```

This command does the following:

- List all `.nix` files
- Make `entr` watch them for changes. Terminate the invoked command on each change with `-r`.
- On each change:
  - Run the `nix-build` invocation as above, but without adding a `./result` symlink
  - Take the resulting store path and append `/bin/map` to it
  - Run the executable at the path constructed this way

## Declaring more options

Rather than setting all script parameters directly, we will do that through the module system. This will not just add some safety through type checking, but also allow to build abstractions to manage growing complexity and changing requirements.

Let's begin by introducing another option, `requestParams`, which will represent the parameters of the request made to the Google Maps API.

Its type will be `listOf <elementType>`, which is a list of elements of one type.

Instead of `lines`, in this case you will want the type of the list elements to be `str`, a generic string type.

The difference between `str` and `lines` is in their merging behavior: Module option types not only check for valid values, but also specify how multiple definitions of an option are to be combined into one.

- For `lines`, multiple definitions get merged by concatenation with newlines.
- For `str`, multiple definitions are not allowed. This is not a problem here, since one can't define a list element multiple times.

Make the following additions to your `default.nix` file:

### default.nix

```
scripts.output = lib.mkOption {
 type = lib.types.package;
};

+ requestParams = lib.mkOption {
+ type = lib.types.listOf lib.types.str;
+ };
};

config = {
 scripts.output = pkgs.writeShellApplication {
 name = "map";
 runtimeInputs = with pkgs; [curl feh];
 text = ''
 ${./map} size=640x640 scale=2 | feh -
 '';
 };
+
+ requestParams = [
+ "size=640x640"
+ "scale=2"
+];
};
}
```

## Dependencies between options

A given module generally declares one option that produces a result to be used elsewhere, in this case `scripts.output`.

Options can depend on other options, making it possible to build more useful abstractions.

Here, we want the `scripts.output` option to use the values of `requestParams` as arguments to the `./map` script.

## Accessing option values

To make option values available to a module, the arguments of the function declaring the module must include the `config` attribute.

Update `default.nix` to add the `config` attribute:

default.nix

```
-{ pkgs, lib, ... }: {
+{ pkgs, lib, config, ... }: {
```

When a module that sets options is evaluated, the resulting values can be accessed by their corresponding attribute names under `config`.

### Note

Option values can't be accessed directly from the same module.

The module system evaluates all modules it receives, and any of them can define a particular option's value. What happens when an option is set by multiple modules is determined by that option's type.

### Warning

The `config argument` is **not** the same as the `config attribute`:

- The `config argument` holds the result of the module system's lazy evaluation, which takes into account all modules passed to `evalModules` and their `imports`.
- The `config attribute` of a module exposes that particular module's option values to the module system for evaluation.

Now make the following changes to `default.nix`:

### default.nix

```
config = {
 scripts.output = pkgs.writeShellApplication {
 name = "map";
 runtimeInputs = with pkgs; [curl feh];
 text = ''
- ${./map} size=640x640 scale=2 | feh -
+ ${./map} ${lib.concatStringsSep " "
+ config.requestParams} | feh -
 '';
```

Here, the value of the `config.requestParams` attribute is populated by the module system based on the definitions in the same file.

#### Note

Lazy evaluation in the Nix language allows the module system to make a value available in the `config` argument passed to the module which defines that value.

`lib.concatStringsSep " "` is then used to join each list element from the value of `config.requestParams` into a single string, with the list elements of `requestParams` separated by a space character.

The result of this represents the list of command line arguments to pass to the `./map` script.

## Conditional definitions

Sometimes, you will want option values to be, well, optional. This can be useful when defining a value for an option is not required, as in the following case.

You will define a new option, `map.zoom`, to control the zoom level of the map. The Google Maps API will infer a zoom level if no corresponding argument is passed, a situation you can represent with the `nullOr <type>`, which represents values of type `<type>` or `null`. This means that when the option isn't defined, the value of such an option is `null`, a value that can be checked against in a conditional.

Add the `map` attribute set with the `zoom` option into the top-level `options` declaration, like so:

### default.nix

```
requestParams = lib.mkOption {
 type = lib.types.listOf lib.types.str;
};

+ map = {
+ zoom = lib.mkOption {
+ type = lib.types.nullOr lib.types.int;
+ };
+ };
};


```

To make use of this, use the `mkIf <condition> <definition>` function, which only adds the definition if the condition evaluates to `true`. Make the following additions to the `requestParams` list in the `config` block:

### default.nix

```
requestParams = [
 "size=640x640"
 "scale=2"
+ (lib.mkIf (config.map.zoom != null)
+ "zoom=${toString config.map.zoom}")
];
};


```

This will will only add a `zoom` parameter to the script invocation if the value of `config.map.zoom` is not `null`.

## Default values

Let's say that in our application we want to have a different default behavior that sets the zoom level to `2`, such that automatic zooming has to be enabled explicitly.

This can be done with the `default` argument to `mkOption`. Its value will be used if the value of the option declaring it is not specified otherwise.

Add the corresponding line:

```
default.nix

map = {
 zoom = lib.mkOption {
 type = lib.types.nullOr lib.types.int;
+ default = 2;
 };
};
```

## Wrapping shell commands

You have now declared options controlling the map dimensions and zoom level, but have not provided a way to specify where the map should be centered.

Add the `center` option now, possibly with your own location as default value:

```
default.nix

type = lib.types.nullOr lib.types.int;
default = 2;
};

+
+ center = lib.mkOption {
+ type = lib.types.nullOr lib.types.str;
+ default = "switzerland";
+ };
};
```

To implement this behavior, you will use the `↓ geocode` utility, which turns location names into coordinates. There are multiple ways of making a new package accessible, but as an exercise, you will add it as an option in the module system.

First, add a new option to accommodate the package:

```
default.nix
```

```
options = {
```

```
scripts.output = lib.mkOption {
 type = lib.types.package;
};

+ scripts.geocode = lib.mkOption {
+ type = lib.types.package;
+ };
```

Then define the value for that option where you make the raw script reproducible by wrapping a call to it in `writeShellApplication`:

### default.nix

```
config = {
+ scripts.geocode = pkgs.writeShellApplication {
+ name = "geocode";
+ runtimeInputs = with pkgs; [curl jq];
+ text = ''exec ${./geocode} "$@"''';
+ };
+
+ scripts.output = pkgs.writeShellApplication {
+ name = "map";
+ runtimeInputs = with pkgs; [curl feh];
```

Add another `mkIf` call to the list of `requestParams` now where you access the wrapped package through `config.scripts.geocode`, and run the executable `/bin/geocode` inside:

### default.nix

```
"scale=2"
 (lib.mkIf (config.map.zoom != null)
 "zoom=${toString config.map.zoom}")
+ (lib.mkIf (config.map.center != null)
+ "center=\"$(${config.scripts.geocode}/bin/geocode ${
+ lib.escapeShellArg config.map.center
+ })\"")
];
};
```

This time, you've used `escapeShellArg` to pass the `config.map.center` value as a command-line argument to `geocode`, string interpolating the result back into the `requestParams` string which sets the `center` value.

Wrapping shell command execution in Nix modules is a helpful technique for controlling system changes, as it uses the more ergonomic attributes and values interface rather than

dealing with the peculiarities of escaping manually.

# Splitting modules

The [module schema](#) includes the `imports` attribute, which allows incorporating further modules, for example to split a large configuration into multiple files.

In particular, this allows you to separate option declarations from where they are used in your configuration.

Create a new module, `marker.nix`, where you can declare options for defining location pins and other markers on the map:

marker.nix

```
{ lib, config, ... }: {
}
```

Reference this new file in `default.nix` using the `imports` attribute:

default.nix

```
{ pkgs, lib, config ... }: {
+ imports = [
+ ./marker.nix
+];
+
```

## The `submodule` type

We want to set multiple markers on the map. A marker is a complex type with multiple fields.

This is where one of the most useful types included in the module system's type system comes into play: `submodule`. This type allows you to define nested modules with their own options.

Here, you will define a new `map.markers` option whose type is a list of submodules, each with a nested `location` type, allowing you to define a list of markers on the map.

Each assignment of markers will be type-checked during evaluation of the top-level `config`.

Make the following changes to `marker.nix`:

### marker.nix

```
-{ pkgs, lib, config, ... }: {
+{ pkgs, lib, config, ... }:
+let
+ markerType = lib.types.submodule {
+ options = {
+ location = lib.mkOption {
+ type = lib.types.nullOr lib.types.str;
+ default = null;
+ };
+ };
+ };
+in {
+ options = {
+ map.markers = lib.mkOption {
+ type = lib.types.listOf markerType;
+ };
+ };
+}
```

## Defining options in other modules

Because of the way the module system composes option definitions, you can freely assign values to options defined in other modules.

In this case, you will use the `map.markers` option to produce and add new elements to the `requestParams` list, making your declared markers appear on the returned map – but from the module declared in `marker.nix`.

To implement this behavior, add the following `config` block to `marker.nix`:

### marker.nix

```
+ config = {
+}
```

```
+ map.markers = [
+ { location = "new york"; }
+];
+
+ requestParams = let
+ paramForMarker =
+ builtins.map (marker: "${${config.scripts.geocode}/bin/geocode ${lib.escapeShellArg marker.location}}") config.map.markers;
+ in ["markers=\"${lib.concatStringsSep "|" paramForMarker}\""];
+
```

## ⚠ Warning

To avoid confusion with the `map` option setting and the final `config.map` configuration value, here we use the `map` function explicitly as `builtins.map`.

Here, you again used `escapeShellArg` and string interpolation to generate a Nix string, this time producing a pipe-separated list of geocoded location attributes.

The `requestParams` value was also set to the resulting list of strings, which gets appended to the `requestParams` list defined in `default.nix`, thanks to the default merging behavior of the `list` type.

When defining multiple markers, determining an appropriate center or zoom level for the map may be challenging; it's easier to let the API do this for you.

To achieve this, make the following additions to `marker.nix`, above the `requestParams` declaration:

### marker.nix

```
+ map.center = lib.mkIf
+ (lib.length config.map.markers >= 1)
+ null;
+
+ map.zoom = lib.mkIf
+ (lib.length config.map.markers >= 2)
+ null;
+
requestParams = let
 paramForMarker = marker:
 let
```

In this case, the default behavior of the Google Maps API when not passed a center or zoom level is to pick the geometric center of all the given markers, and to set a zoom level appropriate for viewing all markers at once.

## Nested submodules

Next, we want to allow multiple named users to define a list of markers each.

For that you'll add a `users` option with type `lib.types.attrsOf <subtype>`, which will allow you to define `users` as an attribute set, whose values have type `<subtype>`.

Here, that subtype will be another submodule which allows declaring a departure marker, suitable for querying the API for the recommended route for a trip.

This will again make use of the `markerType` submodule, giving a nested structure of submodules.

To propagate marker definitions from `users` to the `map.markers` option, make the following changes.

In the `let` block:

marker.nix

```
+ userType = lib.types.submodule {
+ options = {
+ departure = lib.mkOption {
+ type = markerType;
+ default = {};
+ };
+ };
+ };
+
in {
```

This defines a submodule type for a user, with a `departure` option of type `markerType`.

In the `options` block, above `map.markers`:

marker.nix

```
+ users = lib.mkOption {
+ type = lib.types.attrsOf userType;
+ };
```

That allows adding a `users` attribute set to `config` in any submodule that imports `marker.nix`, where each attribute will be of type `userType` as declared in the previous step.

In the `config` block, above `map.center`:

marker.nix

```
config = {

- map.markers = [
- { location = "new york"; }
-];
+ map.markers = lib.filter
+ (marker: marker.location != null)
+ (lib.concatMap (user: [
+ user.departure
+]) (lib.attrValues config.users));

 map.center = lib.mkIf
 (lib.length config.map.markers >= 1)
```

This takes all the `departure` markers from all users in the `config` argument, and adds them to `map.markers` if their `location` attribute is not `null`.

The `config.users` attribute set is passed to `attrValues`, which returns a list of values of each of the attributes in the set (here, the set of `config.users` you've defined), sorted alphabetically (which is how attribute names are stored in the Nix language).

Back in `default.nix`, the resulting `map.markers` option value is still accessed by `requestParams`, which in turn is used to generate arguments to the script that ultimately calls the Google Maps API.

Defining the options in this way allows you to set multiple `users.<name>.departure.location` values and generate a map with the appropriate zoom and center, with pins corresponding to the set of `departure.location` values for all `users`.

In the 2021 Summer of Nix, this formed the basis of an interactive multi-person map demo.

## The `strMatching` type

Now that the map can be rendered with multiple markers, it's time to add some style customizations.

To tell the markers apart, add another option to the `markerType` submodule, to allow labeling each marker pin.

The API documentation states that [these labels](#) must be either an uppercase letter or a number.

You can implement this with the `strMatching "<regex>"` type, where `<regex>` is a regular expression that will accept any matching values, in this case an uppercase letter or number.

In the `let` block:

marker.nix

```
type = lib.types.nullOr lib.types.str;
default = null;
};

+ style.label = lib.mkOption {
+ type = lib.types.nullOr
+ (lib.types.strMatching "[A-Z0-9]");
+ default = null;
+ };
};

};
```

Again, `types.nullOr` allows for `null` values, and the default has been set to `null`.

In the `paramForMarker` function:

marker.nix

```
requestParams = let
+ paramForMarker = marker:
```

```
+ let
+ attributes =
+ lib.optional (marker.style.label != null)
+ "label:${marker.style.label}"
+ ++
+ "$(${config.scripts.geocode}/bin/geocode ${"
+ lib.escapeShellArg marker.location
+ })"
+];
+ in "markers=\\"${lib.concatStringsSep " | " attributes}\\"";
+ in
+ builtins.map paramForMarker config.map.markers;
```

Note how we now create a unique `marker` for each user by concatenating the `label` and `location` attributes together, and assigning them to the `requestParams`. The label for each `marker` is only propagated to the CLI parameters if `marker.style.label` is set.

## Functions as submodule arguments

Right now, if a label is not explicitly set, none will show up. But since every `users` attribute has a name, we could use that as an automatic value instead.

This `firstUpperAlnum` function allows you to retrieve the first character of the username, with the correct type for passing to `departure.style.label`:

marker.nix

```
{ lib, config, ... }:
let
+ # Returns the uppercased first letter
+ # or number of a string
+ firstUpperAlnum = str:
+ lib.mapNullable lib.head
+ (builtins.match "[^A-Z0-9]*([A-Z0-9]).*""
+ (lib.toUpperCase str));

markerType = lib.types.submodule {
 options = {
```

By transforming the argument to `lib.types.submodule` into a function, you can access arguments within it.

One special argument automatically available to submodules is `name`, which when used in

`attrsOf`, gives you the name of the attribute the submodule is defined under:

### marker.nix

```
- userType = lib.types.submodule {
+ userType = lib.types.submodule ({ name, ... }: {
 options = {
 departure = lib.mkOption {
 type = markerType;
 default = {};
 };
 };
- };
```

In this case, you don't easily have access to the name from the marker submodules `label` option, where you otherwise could set a `default` value.

Instead you can use the `config` section of the `user` submodule to set a default, like so:

### marker.nix

```
+
+ config = {
+ departure.style.label = lib.mkDefault
+ (firstUpperAlnum name);
+ };
+ }));

in {
```

#### Note

Module options have a *priority*, represented as an integer, which determines the precedence for setting the option to a particular value. When merging values, the priority with lowest numeric value wins.

The `lib.mkDefault` modifier sets the priority of its argument value to 1000, the lowest precedence.

This ensures that other values set for the same option will prevail.

# The `either` and `enum` types

For better visual contrast, it would be helpful to have a way to change the *color* of a marker.

Here you will use two new type-functions for this:

- `either <this> <that>`, which takes two types as arguments, and allows either of them
- `enum [ <allowed values> ]`, which takes a list of allowed values, and allows any of them

In the `let` block, add the following `colorType` option, which can hold strings containing either some given color names or an RGB value add the new compound type:

marker.nix

```
...
(builtins.match "[^A-Z0-9]*([A-Z0-9]).*"
 (lib.toUpperCase str));

+ # Either a color name or `0xRRGGBB`
+ colorType = lib.types.either
+ (lib.types.strMatching "0x[0-9A-F]{6}")
+ (lib.types.enum [
+ "black" "brown" "green" "purple" "yellow"
+ "blue" "gray" "orange" "red" "white"]);
+
+ markerType = lib.types.submodule {
 options = {
 location = lib.mkOption {
```

This allows either strings that match a 24-bit hexadecimal number or are equal to one of the specified color names.

At the bottom of the `let` block, add the `style.color` option and specify a default value:

marker.nix

```
 (lib.types.strMatching "[A-Z0-9]");
 default = null;
 };
+
+ style.color = lib.mkOption {
+ type = colorType;
```

```
+ default = "red";
+ };
};

};
```

Now add an entry to the `paramForMarker` list which makes use of the new option:

### marker.nix

```
(marker.style.label != null)
"label:${marker.style.label}"
++ [
+ "color:${marker.style.color}"
"$((${config.scripts.geocode}/bin/geocode ${lib.escapeShellArg marker.location
}))"
```

In case you set many different markers, it would be helpful to have the ability to change their size individually.

Add a new `style.size` option to `marker.nix`, allowing you to choose from the set of pre-defined sizes:

### marker.nix

```
type = colorType;
default = "red";
};

+
+ style.size = lib.mkOption {
+ type = lib.types.enum
+ ["tiny" "small" "medium" "large"];
+ default = "medium";
+ };
};

};
```

Now add a mapping for the size parameter in `paramForMarker`, which selects an appropriate string to pass to the API:

### marker.nix

```
requestParams = let
 paramForMarker = marker:
```

```
let
+ size = {
+ tiny = "tiny";
+ small = "small";
+ medium = "mid";
+ large = null;
+ }.${marker.style.size};
+
```

Finally, add another `lib.optional` call to the `attributes` string, making use of the selected size:

### marker.nix

```
attributes =
 lib.optional
 (marker.style.label != null)
 "label:${marker.style.label}"
+ ++ lib.optional
+ (size != null)
+ "size:${size}"
+ ++
+ [
+ "color:${marker.style.color}"
+ "${(${config.scripts.geocode})/bin/geocode ${}
```

## The `pathType` submodule

So far, you've created an option for declaring a *destination* marker, as well as several options for configuring the marker's visual representation.

Now we want to compute and display a route from the user's location to some destination.

The new option defined in the next section will allow you to set an *arrival* marker, which together with a destination allows you to draw *paths* on the map using the new module defined below.

To start, create a new `path.nix` file with the following contents:

### path.nix

```
{ lib, config, ... }:
let
```

```
pathType = lib.types.submodule {
 options = {
 locations = lib.mkOption {
 type = lib.types.listOf lib.types.str;
 };
 };
in
{
 options = {
 map.paths = lib.mkOption {
 type = lib.types.listOf pathType;
 };
 };
 config = {
 requestParams =
 let
 attrForLocation = loc:
 "${${config.scripts.geocode}/bin/geocode ${lib.escapeShellArg loc}}"
 paramForPath = path:
 let
 attributes =
 builtins.map attrForLocation path.locations;
 in
 "'path=\"${lib.concatStringsSep " | " attributes}\"'";
 in
 builtins.map paramForPath config.map.paths;
 };
}
```

The `path.nix` module declares an option for defining a list of paths on our `map`, where each path is a list of strings for geographic locations.

In the `config` attribute we augment the API call by setting the `requestParams` option value with the coordinates transformed appropriately, which will be concatenated with request parameters set elsewhere.

Now import this new `path.nix` module from your `marker.nix` module:

### marker.nix

```
in {

+ imports = [
+ ./path.nix
+];
+
 options = {

 users = lib.mkOption {
```

Copy the `departure` option declaration to a new `arrival` option in `marker.nix`, to complete the initial path implementation:

### marker.nix

```
 type = markerType;
 default = {};
};

+
+ arrival = lib.mkOption {
+ type = markerType;
+ default = {};
+ };
};
```

Next, add an `arrival.style.label` attribute to the `config` block, mirroring the `departure.style.label`:

### marker.nix

```
config = {
 departure.style.label = lib.mkDefault
 (firstUpperAlnum name);
+
+ arrival.style.label = lib.mkDefault
+ (firstUpperAlnum name);
};
});
```

Finally, update the return list in the function passed to `concatMap` in `map.markers` to also include the `arrival` marker for each user:

### marker.nix

```
map.markers = lib.filter
 (marker: marker.location != null)
 (lib.concatMap (user: [
-
+ user.departure
+ user.departure user.arrival
]) (lib.attrValues config.users));

map.center = lib.mkIf
```

Now you have the basesis to define paths on the map, connecting pairs of departure and arrival points.

In the path module, define a path connecting every user's departure and arrival locations:

### path.nix

```
config = {
+
+ map.paths = builtins.map (user: {
+ locations = [
+ user.departure.location
+ user.arrival.location
+];
+ }) (lib.filter (user:
+ user.departure.location != null
+ && user.arrival.location != null
+) (lib.attrValues config.users));
+
 requestParams = let
 attrForLocation = loc:
 "$(geocode ${lib.escapeShellArg loc})";
```

The new `map.paths` attribute contains a list of all valid paths defined for all users.

A path is valid only if the `departure` and `arrival` attributes are set for that user.

## The `between` constraint on integer values

Your users have spoken, and they demand the ability to customize the styles of their paths with a `weight` option.

As before, you'll now declare a new submodule for the path style.

While you could also directly declare the `style.weight` option, in this case you should use the submodule to be able reuse the path style type later.

Add the `pathStyleType` submodule option to the `let` block in `path.nix`:

### path.nix

```
{ lib, config, ... }:
```

```
let
+
+ pathStyleType = lib.types.submodule {
+ options = {
+ weight = lib.mkOption {
+ type = lib.types.ints.between 1 20;
+ default = 5;
+ };
+ };
+ };
+
 pathType = lib.types.submodule {
```

### Note

The `ints.between <lower> <upper>` type allows integers in the given (inclusive) range.

The path weight will default to 5, but can be set to any integer value in the 1 to 20 range, with higher weights producing thicker paths on the map.

Now add a `style` option to the `options` set further down the file:

path.nix

```
options = {
 locations = lib.mkOption {
 type = lib.types.listOf lib.types.str;
 };
+
+ style = lib.mkOption {
+ type = pathStyleType;
+ default = {};
+ };
};

};
```

Finally, update the `attributes` list in `paramForPath`:

path.nix

```
paramForPath = path:
 let
 attributes =
```

```
- builtins.map attrForLocation path.locations;
+ [
+ "weight:${toString path.style.weight}"
+]
+ ++ builtins.map attrForLocation path.locations;
in "path=\"$${lib.concatStringsSep "|" attributes}\"";
```

## The `pathStyle` submodule

Users still can't actually customize the path style yet. Introduce a new `pathStyle` option for each user.

The module system allows you to declare values for an option multiple times, and if the types permit doing so, takes care of merging each declaration's values together.

This makes it possible to have a definition for the `user` option in the `marker.nix` module, as well as a `user` definition in `path.nix`:

### path.nix

```
in {
 options = {
+
+ users = lib.mkOption {
+ type = lib.types.attrsOf (lib.types.submodule {
+ options.pathStyle = lib.mkOption {
+ type = pathStyleType;
+ default = {};
+ };
+ });
+);
+
+ map.paths = lib.mkOption {
+ type = lib.types.listOf pathType;
+ };
+ };
+}
```

Then add a line using the `user.pathStyle` option in `map.paths` where each user's paths are processed:

### path.nix

```
user.departure.location
user.arrival.location
```

```
];
+ style = user.pathStyle;
}) (lib.filter (user:
 user.departure.location != null
 && user.arrival.location != null
```

## Path styling: color

As with markers, paths should have customizable colors.

You can accomplish this using types you've already encountered by now.

Add a new `colorType` block to `path.nix`, specifying the allowed color names and RGB/RGBA hexadecimal values:

path.nix

```
{ lib, config, ... }:
let

+ # Either a color name, `0xRRGGBB` or `0xRRGGBBAA`
+ colorType = lib.types.either
+ (lib.types.strMatching "0x[0-9A-F]{6}[0-9A-F]{2}?"")
+ (lib.types.enum [
+ "black" "brown" "green" "purple" "yellow"
+ "blue" "gray" "orange" "red" "white"
+]);
+
 pathStyleType = lib.types.submodule {
```

Under the `weight` option, add a new `color` option to use the new `colorType` value:

path.nix

```
 type = lib.types.ints.between 1 20;
 default = 5;
 };
+
+ color = lib.mkOption {
+ type = colorType;
+ default = "blue";
+ };
};

};
```

Finally, add a line using the `color` option to the `attributes` list:

path.nix

```
 attributes =
 [
 "weight:${toString path.style.weight}"
+ "color:${path.style.color}"
]
 ++ map attrForLocation path.locations;
in "path=${
```

## Further styling

Now that you've got this far, to further improve the aesthetics of the rendered map, add another style option allowing paths to be drawn as *geodesics*, the shortest “as the crow flies” distance between two points on Earth.

Since this feature can be turned on or off, you can do this using the `bool` type, which can be `true` or `false`.

Make the following changes to `path.nix` now:

path.nix

```
 type = colorType;
 default = "blue";
};

+
+ geodesic = lib.mkOption {
+ type = lib.types.bool;
+ default = false;
+ };
};

};
```

Make sure to also add a line to use that value in `attributes` list, so the option value is included in the API call:

path.nix

```
[
 "weight:${toString path.style.weight}"
 "color:${path.style.color}"
+ "geodesic:${lib.boolToString path.style.geodesic}"
]
++ map attrForLocation path.locations;
in "path=${
```

# Wrapping up

In this tutorial, you've learned how to write custom Nix modules to bring external services under declarative control, with the help of several new utility functions from the `Nixpkgs.lib`.

You defined several modules in multiple files, each with separate submodules making use of the module system's type checking.

These modules exposed features of the external API in a declarative way.

You can now conquer the world with Nix.

# NixOS modules

From NixOS Wiki

NixOS produces a full system configuration by combining smaller, more isolated and reusable components: **Modules**. A module is a file containing a Nix expression with a specific structure. It *declares* options for other modules to *define* (give a value). It processes them and defines options declared in other modules.<sup>[1]</sup>

For example, `/etc/nixos/configuration.nix` is a module. Most other modules are in ◇  
(<https://github.com/NixOS/nixpkgs/blob/master/nixos/modules>)  
(<https://github.com/NixOS/nixpkgs/blob/master/nixos/modules>) .

## Structure

Modules have the following syntax:

```
{
 imports = [
 # Paths to other modules.
 # Compose this module out of smaller ones.
];

 options = {
 # Option declarations.
 # Declare what settings a user of this module module can set.
 # Usually this includes an "enable" option to let a user of this module choose.
 };

 config = {
 # Option definitions.
 # Define what other settings, services and resources should be active.
 # Usually these are depend on whether a user of this module chose to "enable" it
 # using the "option" above.
 # You also set options here for modules that you imported in "imports".
 };
}
```

There is a shorthand for modules without any declarations:

```
{
 imports = [
 # Paths to other modules.
 ./module.nix
 /path/to/absolute/module.nix
];

 # Config definitions.
 services.othermodule.enable = true;
 # ...
 # Notice that you can leave out the "config { }" wrapper.
}
```

Beginners often confuse the modules attribute `imports = [./module.nix]` here with the Nix builtins (<https://nixos.org/manual/nix/stable/language/builtins.html#builtins-import>) function `import module.nix`. The first expects a path to a file containing a NixOS module (having the same specific structure we're describing here), while the second loads whatever Nix expression is in that file (no expected structure). See this post (<https://discourse.nixos.org/t/import-list-in-configuration-nix-vs-import-function/11372/8>).

Note: `imports` provides the same behavior as the obsolete `require`. There is no reason to use `require` anymore, however it may still linger in some legacy code.

## Function

A module can be turned into a function accepting an attribute set.

```
{ config, pkgs, ... }:
{
 imports = [];
 # ...
}
```

It may require the attribute set to contain:

**config** ([/wiki/NixOS:config\\_argument](/wiki/NixOS:config_argument))

The configuration of the entire system.

**options**

All option declarations refined with all definition and declaration references.

**pkgs**

The attribute set extracted from the Nix package collection and enhanced with the `nixpkgs.config` option.

**modulesPath**

The location of the `module` directory of NixOS.

## modulesPath

Some modules use `modulesPath` to import nixos libraries

For example `nixos/modules/virtualisation/digital-ocean-config.nix`

```
{ config, pkgs, lib, modulesPath, ... }:
 imports = [
 (modulesPath + "/profiles/qemu-guest.nix")
 (modulesPath + "/virtualisation/digital-ocean-init.nix")
];
```

The Nix variable `modulesPath` is parsed from the environment variable `NIX_PATH`

When `NIX_PATH` is empty, Nix can throw the error `undefined variable 'modulesPath'`

`NIX_PATH` should look something like this:

```
echo $NIX_PATH | tr : '\n'
nixpkgs=/nix/var/nix/profiles/per-user/root/channels/nixos
nixos-config=/etc/nixos/configuration.nix
/nix/var/nix/profiles/per-user/root/channels
```

Here, the `modulesPath` is `/nix/var/nix/profiles/per-user/root/channels`

When a Nix expression calls `import <nixpkgs>`,  
then Nix will load `/nix/var/nix/profiles/per-user/root/channels/nixos`

## Imports

Imports are paths to other NixOS modules that should be included in the evaluation of the system configuration.  
A default set of modules is defined in ◇ (<https://github.com/NixOS/nixpkgs/blob/master/nixos/modules/module-list.nix>) (<https://github.com/NixOS/nixpkgs/blob/master/nixos/modules/module-list.nix>). These don't need to be added in the import list.

## Declarations

Declarations specify a module's external interfaces.

```
optionName = mkOption {
 # ...
}
```

They are created with `mkOption`, a function accepting a set with following attributes:[2][3]

### **type**

The type of the option. It may be omitted, but that's not advisable since it may lead to errors that are hard to diagnose.

### **default**

The default value used if no value is defined by any module. A default is not required; but if a default is not given, then users of the module will have to define the value of the option, otherwise an error will be thrown.

### **example**

An example value that will be shown in the NixOS manual.

### **description**

A textual description of the option, in DocBook format, that will be included in the NixOS manual.

## Rationale

Modules were introduced to allow extending NixOS without modifying its source code.<sup>[4]</sup> They also allow splitting up `configuration.nix`, making the system configuration easier to maintain and to reuse.

## Example

To see how modules are setup and reuse other modules in practice put `hello.nix` in the same folder as your `configuration.nix`:

```
{ lib, pkgs, config, ... }:
with lib;
let
 # Shorter name to access final settings a
 # user of hello.nix module HAS ACTUALLY SET.
 # cfg is a typical convention.
 cfg = config.services.hello;

in {
 # Declare what settings a user of this "hello.nix" module CAN SET.
 options.services.hello = {
 enable = mkEnableOption "hello service";
 greeter = mkOption {
 type = types.str;
 default = "world";
 };
 };

 # Define what other settings, services and resources should be active IF
 # a user of this "hello.nix" module ENABLED this module
 # by setting "services.hello.enable = true;".
 config = mkIf cfg.enable {
 systemd.services.hello = {
 wantedBy = ["multi-user.target"];
 serviceConfig.ExecStart = "${pkgs.hello}/bin/hello -g'Hello, ${escapeShellArg cfg
 };
 };
}
```

The other `configuration.nix` module can then import this `hello.nix` module and decide to enable it (and optionally set other allowed settings) as follows:

```
{
 imports = [./hello.nix];
 ...
 services.hello = {
 enable = true;
 greeter = "Bob";
 };
}
```

## Advanced Use Cases

### Compatibility Issues with Different Nixpkgs Versions

Module options between Nixpkgs revisions can sometimes change in incompatible ways.

For example, the option `services.nginx.virtualHosts.*.port` in nixpkgs-17.03 was replaced by `services.nginx.virtualHosts.*.listen` in nixpkgs-17.09. If configuration.nix has to accommodate both variants, options can be inspected:

```
{ options, ... }: {
 services.nginx.virtualHosts.somehost = { /* common configuration */ }
 // (if builtins.hasAttr "port" (builtins.head options.services.nginx.virtualHosts.t
 then { port = 8000; }
 else { listen = [{ addr = "0.0.0.0"; port = 8000; }]; });
}
```

## Abstract imports

To import a module that's stored *somewhere* (but for which you have neither an absolute nor a relative path), you can use `NIX_PATH` elements (<https://github.com/musnix/musnix#basic-usage>) or `specialArgs` from `nixos/lib/eval-config.nix`.

This is useful for e.g. pulling modules from a git repository without adding it as a channel, or if you just prefer using paths relative to a root you can change (as opposed to the current file, which could move in the future).

```
let
 inherit (import <nixpkgs> {}) writeShellScriptBin fetchgit;
 yourModules = fetchgit { ... };
in rec {
 nixos = import <nixpkgs/nixos/lib/eval-config.nix> {
 modules = [./configuration.nix];
 specialArgs.mod = name: "${yourModules}/${name}";
 };
 /* use nixos here, e.g. for deployment or building an image */
}
```

```
{ config, lib, pkgs, mod, ... }: {
 imports = [
 (mod "foo.nix")
];

 ...
}
```

## Using external NixOS modules

Some external modules provide extra functionality to the NixOS module system. You can include these modules, after making them available as a file system path (e.g. through `builtins.fetchTarball`), by using `imports = [ `path to module` ]` in your `configuration.nix`.

- [Nixsap](https://github.com/ip1981/nixsap) (<https://github.com/ip1981/nixsap>) - allows to run multiple instances of a service without containers.
- [musnix](https://github.com/musnix/musnix) (<https://github.com/musnix/musnix>) - real-time audio in NixOS.
- [nixos-mailserver](https://gitlab.com/simple-nixos-mailserver/nixos-mailserver) (<https://gitlab.com/simple-nixos-mailserver/nixos-mailserver>) - full-featured mail server module
- [X-Truder Nix-profiles](https://github.com/xtruder/nix-profiles) (<https://github.com/xtruder/nix-profiles>) - modules for Nix to quickly configure your system based on application profiles.

## Under the hood

The following was taken from a comment by Infinisil on reddit <sup>[5]</sup>.

A NixOS system is described by a single system derivation. `nixos-rebuild` builds this derivation with `nix-build '<nixpkgs/nixos>'` -A system and then switches to that system with `result/bin/switch-to-configuration`.

The entrypoint is the file at '`<nixpkgs/nixos>`' (`./default.nix`), which defines the `system` attribute to be the NixOS option `config.system.build.toplevel`. This `toplevel` option is the topmost level of the NixOS evaluation and it's what almost all options eventually end up influencing through potentially a number of intermediate options.

As an example:

- The high-level option `services.nginx.enable` uses the lower-level option `systemd.services.nginx`
- Which in turn uses the even-lower-level option `systemd.units."nginx.service"`
- Which in turn uses `environment.etc."systemd/system"`
- Which then ends up as `result/etc/systemd/system/nginx.service` in the `toplevel` derivation

So high-level options use lower-level ones, eventually ending up at `config.system.build.toplevel`.

How do these options get evaluated though? That's what the NixOS module system does, which lives in the `./lib` directory (in `modules.nix`, `options.nix` and `types.nix`). The module system can even be used without NixOS, allowing you to use it for your own option sets. Here's a simple example of this, whose `toplevel` option you can evaluate with `nix-instantiate --eval file.nix -A config.toplevel`:

```

let
 systemModule = { lib, config, ... }: {
 options.toplevel = lib.mkOption {
 type = lib.types.str;
 };

 options.enableFoo = lib.mkOption {
 type = lib.types.bool;
 default = false;
 };

 config.toplevel = ''
 Is foo enabled? ${lib.boolToString config.enableFoo}
 '';
 };

 userModule = {
 enableFoo = true;
 };

in (import <nixpkgs/lib>).evalModules {
 modules = [systemModule userModule];
}

```

The module system itself is rather complex, but here's a short overview. A module evaluation consists of a set of "modules", which can do three things:

- Import other modules (through `imports = [ ./other-module.nix ]`)
- Declare options (through `options = { ... }`)
- Define option values (through `|config = { ... }`, or without the `config` key as a shorthand if you don't have imports or options)

To do the actual evaluation, there's these rough steps:

- Recursively collect all modules by looking at all `imports` statements
- Collect all option declarations (with `options`) of all modules and merge them together if necessary
- For each option, evaluate it by collecting all its definitions (with `config`) from all modules and merging them together according to the options type.

Note that the last step is lazy (only the options you need are evaluated) and depends on other options itself (all the ones that influence it)

## More complex usages

The examples below contain:

- a child `mkOption` inherits their default from a parent `mkOption`
- reading default values from neighbouring `mkOption`'s for conditional defaults
- passing in the config, to read the `hostName` from a submodule (email system)
- setting default values from attrset (email system)
- generating documentation for custom modules (outside of nixpkgs). See here (<https://discourse.nixos.org>

/t/franken-script-to-generate-nixos-options-docs-with-custom-modules/1674)

Source:

- [\(https://github.com/nixcloud/nixcloud-webservices/blob/master/modules/services/reverse-proxy/default.nix\)](https://github.com/nixcloud/nixcloud-webservices/blob/master/modules/services/reverse-proxy/default.nix)
- [\(https://github.com/nixcloud/nixcloud-webservices/blob/master/modules/services/reverse-proxy/options.nix\)](https://github.com/nixcloud/nixcloud-webservices/blob/master/modules/services/reverse-proxy/options.nix)
- [\(https://github.com/nixcloud/nixcloud-webservices/blob/master/modules/services/TLS/default.nix\)](https://github.com/nixcloud/nixcloud-webservices/blob/master/modules/services/TLS/default.nix)
- [\(https://github.com/nixcloud/nixcloud-webservices/blob/master/modules/services/email/nixcloud-email.nix#L114\)](https://github.com/nixcloud/nixcloud-webservices/blob/master/modules/services/email/nixcloud-email.nix#L114)

(sorry, dont' have more time to make this into a nice little guide yet, but this links should be pretty good introductions into more advanced module system usages) qknight

## Developing modules

To test your module out, you can run the following from a local checkout of nixpkgs with a copy of a configuration.nix :

```
nixos-rebuild build-vm --fast -I nixos-config=./configuration.nix -I nixpkgs=.
```

If you're developing on top of master, this will potentially cause the compilation of lots of packages, since changes on master might not be cached on cache.nixos.org yet. To avoid that, you can develop your module on top of the nixos-unstable channel (/wiki/Channels), tracked by the eponymous branch in <https://github.com/NixOS/nixpkgs> (https://github.com/NixOS/nixpkgs):

```
git checkout -b mymodule upstream/nixos-unstable
```

## With Flakes

If you're developing a module from nixpkgs, you can try and follow the directions here: <https://github.com/Misterio77/nix-starter-configs/issues/28>.

If you want to develop a module from a git repo, you can use `--override-input`. For example, if you have an input in your flake called jovian, you can use

```
nixos-rebuild switch --override-input jovian <path-to-url>` --flake <uri>
```

Of course, it doesn't have to be {{|c|nixos-rebuild}} in particular.

## References

1. NixOS Manual, Chapter 42. Writing NixOS Modules (<https://nixos.org/nixos/manual/#sec-writing-modules>)
2. ◇ (<https://github.com/NixOS/nixpkgs/blob/master/lib/options.nix#L51-L84>) lib

/options.nix#L51-L84 (<https://github.com/NixOS/nixpkgs/blob/master/lib/options.nix#L51-L84>)  
3. NixOS Manual, 42.1. Option Declarations (<https://nixos.org/nixos/manual/#sec-option-declarations>)  
4. [Nix-dev] NixOS: New scheme (<https://nixos.org/nix-dev/2008-November/001467.html>)  
5. Infinisil, [https://www.reddit.com/r/NixOS/comments/gdnzhy/question\\_how\\_nixos\\_options\\_works\\_underthehood/](https://www.reddit.com/r/NixOS/comments/gdnzhy/question_how_nixos_options_works_underthehood/)

## See also

- [NixOS:extend\\_NixOS](/wiki/NixOS:extend_NixOS) (/wiki/NixOS:extend\_NixOS)
- [NixOS:Properties](/wiki/NixOS:Properties) (/wiki/NixOS:Properties)
- NixOS discourse, "Best resources for learning about the NixOS module system?" (<https://discourse.nixos.org/t/best-resources-for-learning-about-the-nixos-module-system>)
- [Debian Config::Model](http://wiki.debian.org/PackageConfigUpgrade) (<http://wiki.debian.org/PackageConfigUpgrade>): target configuration upgrades by abstracting the option of the configuration. Each file is a tree structure where leaves are values defined with an interpreted type. The interpreters are defined for each meta-configuration files name \*.conf. Configuration files does not seems to interact with each other to make consistent configuration. They provide an UI for editing their configuration file.

*Retrieved from "[https://nixos.wiki/index.php?title=NixOS\\_modules&oldid=10946](https://nixos.wiki/index.php?title=NixOS_modules&oldid=10946)"*

Categories (/wiki/Special:Categories): Configuration (/wiki/Category:Configuration)  
| Reference (/wiki/Category:Reference) | NixOS (/wiki/Category:NixOS)

NixOS /  
nixos-hardware

Code Issues 67 Pull requests 8 Actions Security Insights

Eye Heart Fork Star

A collection of NixOS modules covering hardware quirks.

CC0-1.0 license  
Code of conduct  
Security policy  
1.4k stars 473 forks 67 watching 20 Branches 2 Tags Activity Custom properties  
Public repository

---

master ▾ 20 Branches 2 Tags ⌂ Go to file t Go to file + Add file Code ⌂

|                                                                                                               |                                                                   |                                              |            |
|---------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|----------------------------------------------|------------|
|  Mic92                       | Merge pull request #851 from mexisme/microsoft/surface/kernel-6.6 | build(deps): bump cachix/install-nix-acti... | 2 days ago |
|  .github                     | treewide: mark things that have to be do...                       | 3 weeks ago                                  |            |
|  acer/aspire/4810t           | treewide: apply deadnix and statix                                | 7 years ago                                  |            |
|  airis/n990                  | apple-t2: avoid import-from-derivation                            | 2 years ago                                  |            |
|  apple                       | ga401: disable hardware.nvidia.powerM...                          | 2 months ago                                 |            |
|  asus                        | treewide: mark things that have to be do...                       | 2 days ago                                   |            |
|  audio-gd                    | beagleboard/pocketbeagle: init                                    | 7 years ago                                  |            |
|  beagleboard/pocketbeagle    | Added support for Lenovo Legion Slim 5 ...                        | last month                                   |            |
|  common                      | deciso/dec: init, tested with DEC2750                             | 7 months ago                                 |            |
|  dell                        | dell inspiron 5515: add early kms                                 | 2 days ago                                   |            |
|  focus/m2/gen1               | [FocusGen1M2] Disable TPM interrupt d...                          | 9 months ago                                 |            |
|  framework                   | framework amd: only apply suspend wor...                          | last month                                   |            |
|  friendlyarm                | remove unused variables with deadnix                              | 2 months ago                                 |            |
|  google/pixelbook          | google/pixelbook: init                                            | 4 years ago                                  |            |
|  gpd                       | remove unused variables with deadnix                              | 2 months ago                                 |            |
|  hardkernel                | remove unused variables with deadnix                              | 2 months ago                                 |            |
|  hp                        | Added file for HP EliteBook 845 G8 and u...                       | 2 months ago                                 |            |
|  intel/nuc/8i7beh          | treewide: apply deadnix and statix                                | 2 years ago                                  |            |
|  kobel/helios4             | treewide: apply deadnix and statix                                | 2 years ago                                  |            |
|  lenovo                    | Better default for amdgpuBusId                                    | 2 days ago                                   |            |
|  microchip                 | microchip icicle-kit: Fixes to kernel                             | last month                                   |            |
|  microsoft                 | Pick a better name than "versionsOfOpti..."                       | 3 days ago                                   |            |
|  morefine/m600             | remove unused variables with deadnix                              | 2 months ago                                 |            |
|  msi                       | treewide: apply deadnix and statix                                | 2 years ago                                  |            |
|  nxp                       | nxp-imx8: drop kernel overlay                                     | 10 months ago                                |            |
|  olimex/teres_i            | remove unused variables with deadnix                              | 2 months ago                                 |            |
|  omen                      | Add amd pstate to all omen laptops (#763)                         | 4 months ago                                 |            |
|  onenetbook/4              | remove unused variables with deadnix                              | 2 months ago                                 |            |
|  panasonic/letsnote/cf-lx4 | Added configuration for panasonic                                 | 2 years ago                                  |            |
|  ncengines/any             | ncengines/any expose GPU IR over serial                           | 5 years ago                                  |            |

|  purifynix/apu        | purifynix/apu. export dirfd over serial      | 3 years ago  |
|-------------------------------------------------------------------------------------------------------|----------------------------------------------|--------------|
|  pine64              | remove unused variables with deadnix         | 2 months ago |
|  purism/librem       | purism librem5r4: linuxPackages_librem...    | 2 months ago |
|  raspberry-pi        | add and use mkDisableOption                  | 3 months ago |
|  samsung/np900x3c    | samsung/np900x3c: drop deprecated sy...      | 9 months ago |
|  starfive/visionfive | starfive visionfive2: update kernel to 6.6.0 | 2 days ago   |
|  supermicro          | treewide: apply deadnix and statix           | 2 years ago  |
|  system76            | treewide: apply deadnix and statix           | 2 years ago  |
|  tests               | speed up ci using nix-eval-jobs              | 2 months ago |
|  toshiba/swanky      | treewide: apply deadnix and statix           | 2 years ago  |
|  tuxedo              | Add TUXEDO InfinityBook Pro 14 - Gen7 ...    | 2 months ago |
|  .editorconfig       | import editorconfig from nixpkgs             | 2 years ago  |
|  .gitignore          | Add .gitignore adapted from NixPkgs          | 8 years ago  |
|  .mergify.yml        | replace bors with mergify                    | 5 months ago |
|  CODEOWNERS          | docs(tuxedo/pulse/15/gen2): add code o...    | 4 months ago |
|  CONTRIBUTING.md   | CONTRIBUTING.md: mention bors                | 2 years ago  |
|  COPYING           | LICENSE -> COPYING                           | 6 years ago  |
|  README.md         | Add support for Yoga Slim 7 Gen8             | 3 weeks ago  |
|  default.nix       | add dummy default.nix                        | 6 years ago  |
|  flake.nix         | Add support for Yoga Slim 7 Gen8             | 3 weeks ago  |

 README  Code of conduct  CC0-1.0 license  Security



NixOS profiles to optimize settings for different hardware.

## Setup

### Using channels

Add and update nixos-hardware channel:

```
$ sudo nix-channel --add https://github.com/NixOS/nixos-hardware/archive/master.tar.gz nixos-hardware
$ sudo nix-channel --update
```



Then import an appropriate profile path from the table below. For example, to enable ThinkPad X220 profile, your imports in /etc/nixos/configuration.nix should look like:

```
imports = [
 <nixos-hardware/lenovo/thinkpad/x220>
 ./hardware-configuration.nix
];
```



New updates to the expressions here will be fetched when you update the channel.

## Using nix flakes support

There is also experimental flake support. In your `/etc/nixos/flake.nix` add the following:

```
{
 description = "NixOS configuration with flakes";
 inputs.nixos-hardware.url = "github:NixOS/nixos-hardware/master";

 outputs = { self, nixpkgs, nixos-hardware }: {
 # replace <your-hostname> with your actual hostname
 nixosConfigurations.<your-hostname> = nixpkgs.lib.nixosSystem {
 # ...
 modules = [
 # ...
 # add your model from this list: https://github.com/NixOS/nixos-hardware/blob/master/flake.nix
 nixos-hardware.nixosModules.dell-xps-13-9380
];
 };
 };
}
```



## Using fetchGit

You can fetch the git repository directly:

```
imports = [
 "${builtins.fetchGit { url = "https://github.com/NixOS/nixos-hardware.git"; }}/lenovo/thinkpad/x220"
];
```



Unlike the channel, this will update the git repository on a rebuild. However, you can easily pin to a particular revision if you desire more stability.

## How to contribute a new device profile

See [CONTRIBUTING.md](#).

## List of Profiles

See code for all available configurations.

| Model                                  | Path                                    |
|----------------------------------------|-----------------------------------------|
| <a href="#">Acer Aspire 4810T</a>      | <nixos-hardware/acer/aspire/4810t>      |
| <a href="#">Airis N990</a>             | <nixos-hardware/airis/n990>             |
| <a href="#">Apple MacBook Air 3,X</a>  | <nixos-hardware/apple/macbook-air/3>    |
| <a href="#">Apple MacBook Air 4,X</a>  | <nixos-hardware/apple/macbook-air/4>    |
| <a href="#">Apple MacBook Air 6,X</a>  | <nixos-hardware/apple/macbook-air/6>    |
| <a href="#">Apple MacBook Pro 10,1</a> | <nixos-hardware/apple/macbook-pro/10-1> |
| <a href="#">Apple MacBook Pro 11,5</a> | <nixos-hardware/apple/macbook-pro/11-5> |
| <a href="#">Apple MacBook Pro 12,1</a> | <nixos-hardware/apple/macbook-pro/12-1> |

|                                              |                                           |
|----------------------------------------------|-------------------------------------------|
| <a href="#">Apple MacBook Pro 14,1</a>       | <nixos-hardware/apple/macbook-pro/14-1>   |
| <a href="#">Apple Macs with a T2 Chip</a>    | <nixos-hardware/apple/t2>                 |
| <a href="#">Asus ROG Ally RC71L (2023)</a>   | <nixos-hardware/asus/ally/rc71l>          |
| <a href="#">Asus ROG Strix G513IM</a>        | <nixos-hardware/asus/rog-strix/g513im>    |
| <a href="#">Asus ROG Strix G733QS</a>        | <nixos-hardware/asus/rog-strix/g733qs>    |
| <a href="#">Asus ROG Zephyrus G14 GA401</a>  | <nixos-hardware/asus/zephyrus/ga401>      |
| <a href="#">Asus ROG Zephyrus G14 GA402</a>  | <nixos-hardware/asus/zephyrus/ga402>      |
| <a href="#">Asus ROG Zephyrus G15 GA502</a>  | <nixos-hardware/asus/zephyrus/ga502>      |
| <a href="#">Asus ROG Zephyrus G15 GA503</a>  | <nixos-hardware/asus/zephyrus/ga503>      |
| <a href="#">Asus ROG Zephyrus M16 GU603H</a> | <nixos-hardware/asus/zephyrus/gu603h>     |
| <a href="#">Asus TUF FX504GD</a>             | <nixos-hardware/asus/fx504gd>             |
| <a href="#">BeagleBoard PocketBeagle</a>     | <nixos-hardware/beagleboard/pocketbeagle> |
| <a href="#">Deciso DEC series</a>            | <nixos-hardware/deciso/dec>               |
| <a href="#">Dell G3 3779</a>                 | <nixos-hardware/dell/g3/3779>             |
| <a href="#">Dell Inspiron 14 5420</a>        | <nixos-hardware/dell/inspiron/14-5420>    |
| <a href="#">Dell Inspiron 5509</a>           | <nixos-hardware/dell/inspiron/5509>       |
| <a href="#">Dell Inspiron 5515</a>           | <nixos-hardware/dell/inspiron/5515>       |
| <a href="#">Dell Inspiron 7405</a>           | <nixos-hardware/dell/inspiron/7405>       |
| <a href="#">Dell Latitude 3340</a>           | <nixos-hardware/dell/latitude/3340>       |
| <a href="#">Dell Latitude 3480</a>           | <nixos-hardware/dell/latitude/3480>       |
| <a href="#">Dell Latitude 5520</a>           | <nixos-hardware/dell/latitude/5520>       |
| <a href="#">Dell Latitude 7390</a>           | <nixos-hardware/dell/latitude/7390>       |
| <a href="#">Dell Latitude 7430</a>           | <nixos-hardware/dell/latitude/7430>       |
| <a href="#">Dell Latitude 7490</a>           | <nixos-hardware/dell/latitude/7490>       |
| <a href="#">Dell Poweredge R7515</a>         | <nixos-hardware/dell/poweredge/r7515>     |
| <a href="#">Dell Precision 3541</a>          | <nixos-hardware/dell/precision/3541>      |
| <a href="#">Dell Precision 5530</a>          | <nixos-hardware/dell/precision/5530>      |
| <a href="#">Dell XPS 13 7390</a>             | <nixos-hardware/dell/xps/13-7390>         |
| <a href="#">Dell XPS 13 9300</a>             | <nixos-hardware/dell/xps/13-9300>         |
| <a href="#">Dell XPS 13 9310</a>             | <nixos-hardware/dell/xps/13-9310>         |
| <a href="#">Dell XPS 13 9333</a>             | <nixos-hardware/dell/xps/13-9333>         |
| <a href="#">Dell XPS 13 9343</a>             | <nixos-hardware/dell/xps/13-9343>         |
| <a href="#">Dell XPS 13 9350</a>             | <nixos-hardware/dell/xps/13-9350>         |
| <a href="#">Dell XPS 13 9360</a>             | <nixos-hardware/dell/xps/13-9360>         |
| <a href="#">Dell XPS 13 9370</a>             | <nixos-hardware/dell/xps/13-9370>         |

|                                                    |                                                   |
|----------------------------------------------------|---------------------------------------------------|
| <a href="#">Dell XPS 13 9380</a>                   | <nixos-hardware/dell/xps/13-9380>                 |
| <a href="#">Dell XPS 15 7590, nvidia</a>           | <nixos-hardware/dell/xps/15-7590/nvidia>          |
| <a href="#">Dell XPS 15 7590</a>                   | <nixos-hardware/dell/xps/15-7590>                 |
| <a href="#">Dell XPS 15 9500, nvidia</a>           | <nixos-hardware/dell/xps/15-9500/nvidia>          |
| <a href="#">Dell XPS 15 9500</a>                   | <nixos-hardware/dell/xps/15-9500>                 |
| <a href="#">Dell XPS 15 9510, nvidia</a>           | <nixos-hardware/dell/xps/15-9510/nvidia>          |
| <a href="#">Dell XPS 15 9510</a>                   | <nixos-hardware/dell/xps/15-9510>                 |
| <a href="#">Dell XPS 15 9520, nvidia</a>           | <nixos-hardware/dell/xps/15-9520/nvidia>          |
| <a href="#">Dell XPS 15 9520</a>                   | <nixos-hardware/dell/xps/15-9520>                 |
| <a href="#">Dell XPS 15 9550, nvidia</a>           | <nixos-hardware/dell/xps/15-9550/nvidia>          |
| <a href="#">Dell XPS 15 9550</a>                   | <nixos-hardware/dell/xps/15-9550>                 |
| <a href="#">Dell XPS 15 9560, intel only</a>       | <nixos-hardware/dell/xps/15-9560/intel>           |
| <a href="#">Dell XPS 15 9560, nvidia only</a>      | <nixos-hardware/dell/xps/15-9560/nvidia>          |
| <a href="#">Dell XPS 15 9560</a>                   | <nixos-hardware/dell/xps/15-9560>                 |
| <a href="#">Dell XPS 17 9700, intel</a>            | <nixos-hardware/dell/xps/17-9700/intel            |
| <a href="#">Dell XPS 17 9700, nvidia</a>           | <nixos-hardware/dell/xps/17-9700/nvidia>          |
| <a href="#">Dell XPS 17 9710, intel only</a>       | <nixos-hardware/dell/xps/17-9710/intel>           |
| <a href="#">Dell XPS E7240</a>                     | <nixos-hardware/dell/e7240>                       |
| <a href="#">Framework 11th Gen Intel Core</a>      | <nixos-hardware/framework/13-inch/11th-gen-intel> |
| <a href="#">Framework 12th Gen Intel Core</a>      | <nixos-hardware/framework/13-inch/12th-gen-intel> |
| <a href="#">Framework 13th Gen Intel Core</a>      | <nixos-hardware/framework/13-inch/13th-gen-intel> |
| <a href="#">Framework 13 AMD Ryzen 7040 Series</a> | <nixos-hardware/framework/13-inch/7040-amd>       |
| <a href="#">FriendlyARM NanoPC-T4</a>              | <nixos-hardware/friendlyarm/nanopc-t4>            |
| <a href="#">FriendlyARM NanoPi R5s</a>             | <nixos-hardware/friendlyarm/nanopi-r5s>           |
| <a href="#">Focus M2 Gen 1</a>                     | <nixos-hardware/focus/m2/gen1>                    |
| <a href="#">GPD MicroPC</a>                        | <nixos-hardware/gpd/micropc>                      |
| <a href="#">GPD P2 Max</a>                         | <nixos-hardware/gpd/p2-max>                       |
| <a href="#">GPD Pocket 3</a>                       | <nixos-hardware/gpd/pocket-3>                     |
| <a href="#">GPD WIN 2</a>                          | <nixos-hardware/gpd/win-2>                        |
| <a href="#">GPD WIN Max 2 2023</a>                 | <nixos-hardware/gpd/win-max-2/2023>               |
| <a href="#">Google Pixelbook</a>                   | <nixos-hardware/google/pixelbook>                 |
| <a href="#">HP Elitebook 2560p</a>                 | <nixos-hardware/hp/elitebook/2560p>               |
| <a href="#">HP Elitebook 845g7</a>                 | <nixos-hardware/hp/elitebook/845/g7>              |
| <a href="#">HP Elitebook 845g8</a>                 | <nixos-hardware/hp/elitebook/845/g8>              |
| <a href="#">HP Elitebook 845g9</a>                 | <nixos-hardware/hp/elitebook/845/g9>              |

|                                                          |                                                |
|----------------------------------------------------------|------------------------------------------------|
| <a href="#">HP Notebook 14-df0023</a>                    | <nixos-hardware/hp/notebook/14-df0023>         |
| <a href="#">i.MX8QuadMax Multisensory Enablement Kit</a> | <nixos-hardware/nxp/imx8qm-mek>                |
| <a href="#">Intel NUC 8i7BEH</a>                         | <nixos-hardware/intel/nuc/8i7beh>              |
| <a href="#">Lenovo IdeaPad Gaming 3 15arh05</a>          | <nixos-hardware/lenovo/ideapad/15arh05>        |
| <a href="#">Lenovo IdeaPad Z510</a>                      | <nixos-hardware/lenovo/ideapad/z510>           |
| <a href="#">Lenovo IdeaPad Slim 5</a>                    | <nixos-hardware/lenovo/ideapad/slim-5>         |
| <a href="#">Lenovo IdeaPad S145 15api</a>                | <nixos-hardware/lenovo/ideapad/s145-15api>     |
| <a href="#">Lenovo Legion 5 15arh05h</a>                 | <nixos-hardware/lenovo/legion/15arh05h>        |
| <a href="#">Lenovo Legion 7 Slim 15ach6</a>              | <nixos-hardware/lenovo/legion/15ach6>          |
| <a href="#">Lenovo Legion 5 Pro 16ach6h</a>              | <nixos-hardware/lenovo/legion/16ach6h>         |
| <a href="#">Lenovo Legion 5 Pro 16ach6h (Hybrid)</a>     | <nixos-hardware/lenovo/legion/16ach6h/hybrid>  |
| <a href="#">Lenovo Legion 5 Pro 16ach6h (Nvidia)</a>     | <nixos-hardware/lenovo/legion/16ach6h/nvidia>  |
| <a href="#">Lenovo Legion 7 16achg6 (Hybrid)</a>         | <nixos-hardware/lenovo/legion/16achg6/hybrid>  |
| <a href="#">Lenovo Legion 7 16achg6 (Nvidia)</a>         | <nixos-hardware/lenovo/legion/16achg6/nvidia>  |
| <a href="#">Lenovo Legion 7i Pro 16irx8h (Intel)</a>     | <nixos-hardware/lenovo/legion/16irx8h>         |
| <a href="#">Lenovo Legion Y530 15ICH</a>                 | <nixos-hardware/lenovo/legion/15ich>           |
| <a href="#">Lenovo ThinkPad E14 (AMD)</a>                | <nixos-hardware/lenovo/thinkpad/e14/amd>       |
| <a href="#">Lenovo ThinkPad E14 (Intel)</a>              | <nixos-hardware/lenovo/thinkpad/e14/intel>     |
| <a href="#">Lenovo ThinkPad E470</a>                     | <nixos-hardware/lenovo/thinkpad/e470>          |
| <a href="#">Lenovo ThinkPad E495</a>                     | <nixos-hardware/lenovo/thinkpad/e495>          |
| <a href="#">Lenovo ThinkPad L13 Yoga</a>                 | <nixos-hardware/lenovo/thinkpad/l13/yoga>      |
| <a href="#">Lenovo ThinkPad L13</a>                      | <nixos-hardware/lenovo/thinkpad/l13>           |
| <a href="#">Lenovo ThinkPad L14 (AMD)</a>                | <nixos-hardware/lenovo/thinkpad/l14/amd>       |
| <a href="#">Lenovo ThinkPad L14 (Intel)</a>              | <nixos-hardware/lenovo/thinkpad/l14/intel>     |
| <a href="#">Lenovo ThinkPad P1 Gen 3</a>                 | <nixos-hardware/lenovo/thinkpad/p1/3th-gen>    |
| <a href="#">Lenovo ThinkPad P14s AMD Gen 2</a>           | <nixos-hardware/lenovo/thinkpad/p14s/amd/gen2> |
| <a href="#">Lenovo ThinkPad P16s AMD Gen 1</a>           | <nixos-hardware/lenovo/thinkpad/p16s/amd/gen1> |
| <a href="#">Lenovo ThinkPad P1</a>                       | <nixos-hardware/lenovo/thinkpad/p1>            |
| <a href="#">Lenovo ThinkPad P50</a>                      | <nixos-hardware/lenovo/thinkpad/p50>           |
| <a href="#">Lenovo ThinkPad P51</a>                      | <nixos-hardware/lenovo/thinkpad/p51>           |
| <a href="#">Lenovo ThinkPad P52</a>                      | <nixos-hardware/lenovo/thinkpad/p52>           |
| <a href="#">Lenovo ThinkPad P53</a>                      | <nixos-hardware/lenovo/thinkpad/p53>           |
| <a href="#">Lenovo ThinkPad T14 AMD Gen 1</a>            | <nixos-hardware/lenovo/thinkpad/t14/amd/gen1>  |
| <a href="#">Lenovo ThinkPad T14 AMD Gen 2</a>            | <nixos-hardware/lenovo/thinkpad/t14/amd/gen2>  |
| <a href="#">Lenovo ThinkPad T14 AMD Gen 3</a>            | <nixos-hardware/lenovo/thinkpad/t14/amd/gen3>  |

|                                                    |                                                   |
|----------------------------------------------------|---------------------------------------------------|
| <a href="#">Lenovo ThinkPad T14</a>                | <nixos-hardware/lenovo/thinkpad/t14>              |
| <a href="#">Lenovo ThinkPad T14s AMD Gen 1</a>     | <nixos-hardware/lenovo/thinkpad/t14s/amd/gen1>    |
| <a href="#">Lenovo ThinkPad T14s</a>               | <nixos-hardware/lenovo/thinkpad/t14s>             |
| <a href="#">Lenovo ThinkPad T410</a>               | <nixos-hardware/lenovo/thinkpad/t410>             |
| <a href="#">Lenovo ThinkPad T420</a>               | <nixos-hardware/lenovo/thinkpad/t420>             |
| <a href="#">Lenovo ThinkPad T430</a>               | <nixos-hardware/lenovo/thinkpad/t430>             |
| <a href="#">Lenovo ThinkPad T440p</a>              | <nixos-hardware/lenovo/thinkpad/t440p>            |
| <a href="#">Lenovo ThinkPad T440s</a>              | <nixos-hardware/lenovo/thinkpad/t440s>            |
| <a href="#">Lenovo ThinkPad T450s</a>              | <nixos-hardware/lenovo/thinkpad/t450s>            |
| <a href="#">Lenovo ThinkPad T460</a>               | <nixos-hardware/lenovo/thinkpad/t460>             |
| <a href="#">Lenovo ThinkPad T460p</a>              | <nixos-hardware/lenovo/thinkpad/t460p>            |
| <a href="#">Lenovo ThinkPad T460s</a>              | <nixos-hardware/lenovo/thinkpad/t460s>            |
| <a href="#">Lenovo ThinkPad T470s</a>              | <nixos-hardware/lenovo/thinkpad/t470s>            |
| <a href="#">Lenovo ThinkPad T480</a>               | <nixos-hardware/lenovo/thinkpad/t480>             |
| <a href="#">Lenovo ThinkPad T480s</a>              | <nixos-hardware/lenovo/thinkpad/t480s>            |
| <a href="#">Lenovo ThinkPad T490</a>               | <nixos-hardware/lenovo/thinkpad/t490>             |
| <a href="#">Lenovo ThinkPad T495</a>               | <nixos-hardware/lenovo/thinkpad/t495>             |
| <a href="#">Lenovo ThinkPad T520</a>               | <nixos-hardware/lenovo/thinkpad/t520>             |
| <a href="#">Lenovo ThinkPad T550</a>               | <nixos-hardware/lenovo/thinkpad/t550>             |
| <a href="#">Lenovo ThinkPad T590</a>               | <nixos-hardware/lenovo/thinkpad/t590>             |
| <a href="#">Lenovo ThinkPad X1 Yoga</a>            | <nixos-hardware/lenovo/thinkpad/x1/yoga>          |
| <a href="#">Lenovo ThinkPad X1 Yoga Gen 7</a>      | <nixos-hardware/lenovo/thinkpad/x1/yoga/7th-gen>  |
| <a href="#">Lenovo ThinkPad X1 (6th Gen)</a>       | <nixos-hardware/lenovo/thinkpad/x1/6th-gen>       |
| <a href="#">Lenovo ThinkPad X1 (7th Gen)</a>       | <nixos-hardware/lenovo/thinkpad/x1/7th-gen>       |
| <a href="#">Lenovo ThinkPad X1 (9th Gen)</a>       | <nixos-hardware/lenovo/thinkpad/x1/9th-gen>       |
| <a href="#">Lenovo ThinkPad X1 (10th Gen)</a>      | <nixos-hardware/lenovo/thinkpad/x1/10th-gen>      |
| <a href="#">Lenovo ThinkPad X1 (11th Gen)</a>      | <nixos-hardware/lenovo/thinkpad/x1/11th-gen>      |
| <a href="#">Lenovo ThinkPad X1 Extreme Gen 2</a>   | <nixos-hardware/lenovo/thinkpad/x1-extreme/gen2>  |
| <a href="#">Lenovo ThinkPad X1 Extreme Gen 4</a>   | <nixos-hardware/lenovo/thinkpad/x1-extreme/gen4>  |
| <a href="#">Lenovo ThinkPad X1 Nano Gen 1</a>      | <nixos-hardware/lenovo/thinkpad/x1-nano/gen1>     |
| <a href="#">Lenovo ThinkPad X13 Yoga</a>           | <nixos-hardware/lenovo/thinkpad/x13/yoga>         |
| <a href="#">Lenovo ThinkPad X13 Yoga (3th Gen)</a> | <nixos-hardware/lenovo/thinkpad/x13/yoga/3th-gen> |
| <a href="#">Lenovo ThinkPad X13</a>                | <nixos-hardware/lenovo/thinkpad/x13>              |
| <a href="#">Lenovo ThinkPad X140e</a>              | <nixos-hardware/lenovo/thinkpad/x140e>            |
| <a href="#">Lenovo ThinkPad X200s</a>              | <nixos-hardware/lenovo/thinkpad/x200s>            |

|                                                          |                                                       |
|----------------------------------------------------------|-------------------------------------------------------|
| <a href="#">Lenovo ThinkPad X220</a>                     | <nixos-hardware/lenovo/thinkpad/x220>                 |
| <a href="#">Lenovo ThinkPad X230</a>                     | <nixos-hardware/lenovo/thinkpad/x230>                 |
| <a href="#">Lenovo ThinkPad X250</a>                     | <nixos-hardware/lenovo/thinkpad/x250>                 |
| <a href="#">Lenovo ThinkPad X260</a>                     | <nixos-hardware/lenovo/thinkpad/x260>                 |
| <a href="#">Lenovo ThinkPad X270</a>                     | <nixos-hardware/lenovo/thinkpad/x270>                 |
| <a href="#">Lenovo ThinkPad X280</a>                     | <nixos-hardware/lenovo/thinkpad/x280>                 |
| <a href="#">Lenovo ThinkPad X390</a>                     | <nixos-hardware/lenovo/thinkpad/x390>                 |
| <a href="#">Lenovo ThinkPad Z Series</a>                 | <nixos-hardware/lenovo/thinkpad/z>                    |
| <a href="#">Lenovo ThinkPad Z13</a>                      | <nixos-hardware/lenovo/thinkpad/z/z13>                |
| <a href="#">LENOVO Yoga 6 13ALC6 82ND</a>                | <nixos-hardware/lenovo/yoga/6/13ALC6>                 |
| <a href="#">LENOVO Yoga 7 Slim Gen8</a>                  | <nixos-hardware/lenovo/yoga/7/slim/gen8>              |
| <a href="#">MSI GS60 2QE</a>                             | <nixos-hardware/msi/gs60>                             |
| <a href="#">MSI GL62/CX62</a>                            | <nixos-hardware/msi/gl62>                             |
| <a href="#">Microchip Icicle Kit</a>                     | <nixos-hardware/microchip/icicle-kit>                 |
| <a href="#">Microsoft Surface Go</a>                     | <nixos-hardware/microsoft/surface/surface-go>         |
| <a href="#">Microsoft Surface Pro (Intel)</a>            | <nixos-hardware/microsoft/surface/surface-pro-intel>  |
| <a href="#">Microsoft Surface Laptop (AMD)</a>           | <nixos-hardware/microsoft/surface/surface-laptop-amd> |
| <a href="#">Microsoft Surface Range (Common Modules)</a> | <nixos-hardware/microsoft/surface/common>             |
| <a href="#">Microsoft Surface Pro 3</a>                  | <nixos-hardware/microsoft/surface-pro/3>              |
| <a href="#">Morefine M600</a>                            | <nixos-hardware/morefine/m600>                        |
| <a href="#">Hardkernel Odroid HC4</a>                    | <nixos-hardware/hardkernel/odroid-hc4>                |
| <a href="#">Hardkernel Odroid H3</a>                     | <nixos-hardware/hardkernel/odroid-h3>                 |
| <a href="#">Omen 15-en0010ca</a>                         | <nixos-hardware/omen/15-en0010ca>                     |
| <a href="#">Omen 16-n0005ne</a>                          | <nixos-hardware/omen/16-n0005ne>                      |
| <a href="#">Omen 15-en1007sa</a>                         | <nixos-hardware/omen/15-en1007sa>                     |
| <a href="#">Omen en00015p</a>                            | <nixos-hardware/omen/en00015p>                        |
| <a href="#">One-Netbook OneNetbook 4</a>                 | <nixos-hardware/onenetbook/4>                         |
| <a href="#">Panasonic Let's Note CF-LX4</a>              | <nixos-hardware/panasonic/letsnote/cf-lx4>            |
| <a href="#">PC Engines APU</a>                           | <nixos-hardware/pcengines/apu>                        |
| <a href="#">PINE64 Pinebook Pro</a>                      | <nixos-hardware/pine64/pinebook-pro>                  |
| <a href="#">PINE64 RockPro64</a>                         | <nixos-hardware/pine64/rockpro64>                     |
| <a href="#">PINE64 STAR64</a>                            | <nixos-hardware/pine64/star64>                        |
| <a href="#">Purism Librem 13v3</a>                       | <nixos-hardware/purism/librem/13v3>                   |
| <a href="#">Purism Librem 15v3</a>                       | <nixos-hardware/purism/librem/15v3>                   |
| <a href="#">Purism Librem 5r4</a>                        | <nixos-hardware/purism/librem/5r4>                    |
| <a href="#">Rasberry Pi 2</a>                            | <nixos-hardware/rasberry-pi/2>                        |

| <a href="#">Raspberry Pi 2</a>                    | NixOS hardware/raspberry-pi/2                   |
|---------------------------------------------------|-------------------------------------------------|
| <a href="#">Raspberry Pi 4</a>                    | <nixos-hardware/raspberry-pi/4>                 |
| <a href="#">Samsung Series 9 NP900X3C</a>         | <nixos-hardware/samsung/np900x3c>               |
| <a href="#">StarFive VisionFive v1</a>            | <nixos-hardware/starfive/visionfive/v1>         |
| <a href="#">StarFive VisionFive 2</a>             | <nixos-hardware/starfive/visionfive/v2>         |
| <a href="#">Supermicro A1SRI-2758F</a>            | <nixos-hardware/supermicro/a1sri-2758f>         |
| <a href="#">Supermicro M11SDV-8C-LN4F</a>         | <nixos-hardware/supermicro/m11sdv-8c-ln4f>      |
| <a href="#">Supermicro X10SLL-F</a>               | <nixos-hardware/supermicro/x10sll-f>            |
| <a href="#">Supermicro X12SCZ-TLN4F</a>           | <nixos-hardware/supermicro/x12scz-tln4f>        |
| <a href="#">System76 (generic)</a>                | <nixos-hardware/system76>                       |
| <a href="#">System76 Darter Pro 6</a>             | <nixos-hardware/system76/darp6>                 |
| <a href="#">Toshiba Chromebook 2 swanky</a>       | <nixos-hardware/toshiba/swanky>                 |
| <a href="#">Tuxedo InfinityBook v4</a>            | <nixos-hardware/tuxedo/infinitybook/v4>         |
| <a href="#">TUXEDO InfinityBook Pro 14 - Gen7</a> | <nixos-hardware/tuxedo/infinitybook/pro14/gen7> |
| <a href="#">TUXEDO Pulse 15 - Gen2</a>            | <nixos-hardware/tuxedo/pulse/15/gen2>           |

## Releases 2

 Kernel config and sound state for mnt-reform2-nitrogen8m [\[Latest\]](#)  
on May 29, 2021

[+ 1 release](#)

## Sponsor this project

 [opencollective.com/nixos](https://opencollective.com/nixos)

## Packages

No packages published

## Contributors 285



[+ 271 contributors](#)

## Languages

● Nix 94.4%  
 ● Ruby 3.1%  
 ● Python 1.5%  
 ● Other 1.0%