Automatically managing remote sources with niv

Contents

The Nix language can be used to describe dependencies between files managed by Nix. Nix expressions themselves can depend on remote sources, and there are multiple ways to specify their origin, as shown in Towards reproducibility: pinning Nixpkgs.

For more automation around handling remote sources, set up niv in your project:

```
$ nix-shell -p niv --run "niv init --nixpkgs nixos/nixpkgs --nixpkgs-branch
```

This command will fetch the latest revision of the Nixpkgs 23.05 release branch. In the current directory it will generate `nix/sources.json`, which will contain a pinned reference to the obtained revision. It will also create `nix/sources.nix`, which exposes those dependencies as an attribute set.

Import the generated `nix/sources.nix` as the default value for the argument to the function in `default.nix` and use it to refer to the Nixpkgs source directory:

```
1 { sources ? import ./nix/sources.nix }:
2 let
3   pkgs = import sources.nixpkgs {};
4   build = pkgs.hello;
5 in {
6   inherit build;
7 }
```

`nix-build` will call the top-level function with the empty attribute set `{}`, or with the

attributes passed via `--arg` or `--argstr`. This pattern allows overriding remote sources programmatically.

Add niv to the development environment for your project to have it readily available:

```
 { sources ? import ./nix/sources.nix }:
 let
   pkgs = import sources.nixpkgs {};
   build = pkgs.hello;
 in {
   inherit build;
+  shell = pkgs.mkShell {
+    inputsFrom = [ build ];
+    packages = with pkgs; [
+      niv
+    ];
+  };
 }
```

Also add a `shell.nix` to enter that environment more conveniently:

```
1 (import ./. {}).shell
```

See Dependencies in the development shell for details, and note that here you have to pass an empty attribute set to the imported expression, since `default.nix` now contains a function.

Overriding sources

As an example, we will use the previously created expression with an older version of Nixpkgs.

Enter the development environment, create a new directory, and set up niv with a different version of Nixpkgs:

```
$ nix-shell
[nix-shell]$ mkdir old
[nix-shell]$ cd old
[nix-shell]$ niv init --nixpkgs nixos/nixpkgs --nixpkgs-branch 18.09
```

Create a file `default.nix` in the new directory, and import the original one with the `sources` just created.

```
1 import ../default.nix { sources = import ./nix/sources.nix; }
```

This will result in a different version being built:

```
$ nix-build -A build
$ ./result/bin/hello --version | head -1
hello (GNU Hello) 2.10
```

Sources can also be overridden on the command line:

```
nix-build .. -A build --arg sources 'import ./nix/sources.nix'
```

Check the built-in help for details:

```
niv --help
```

Next steps

- For more details and examples of the different ways to specify remote sources, see Towards reproducibility: pinning Nixpkgs.

Best practices

Contents

URLs

The Nix language syntax supports bare URLs, so one could write `https://example.com` instead of `"https://example.com"`

RFC 45 was accepted to deprecate unquoted URLs and provides a number of arguments how this feature does more harm than good.

> **Tip**
>
> Always quote URLs.

Recursive attribute set `rec { ... }`

`rec` allows you to reference names within the same attribute set.

Example:

Expression

```
rec {
  a = 1;
  b = a + 2;
}
```

Value

```
{ a = 1; b = 3; }
```

A common pitfall is to introduce a hard to debug error `infinite recursion` when shadowing a name. The simplest example for this is:

```
let a = 1; in rec { a = a; }
```

Tip

Avoid `rec`. Use `let ... in`.

Example:

Expression

```
let
  a = 1;
in {
  a = a;
  b = a + 2;
}
```

`with` scopes

It's still common to see the following expression in the wild:

Expression

```
with (import <nixpkgs> {});

# ... lots of code
```

This brings all attributes of the imported expression into scope of the current expression.

There are a number of problems with that approach:

- Static analysis can't reason about the code, because it would have to actually evaluate this file to see which names are in scope.
- When more than one `with` used, it's not clear anymore where the names are coming from.
- Scoping rules for `with` are not intuitive, see this Nix issue for details.

> **Tip**
>
> Do not use `with` at the top of a Nix file. Explicitly assign names in a `let` expression.
>
> Example:
>
> Expression
> ```
> let
>   pkgs = import <nixpkgs> {};
>   inherit (pkgs) curl jq;
> in
>
> # ...
> ```

Smaller scopes are usually less problematic, but can still lead to surprises due to scoping rules.

> **Tip**
>
> If you want to avoid `with` altogether, try replacing expressions of this form
>
> Expression
> ```
> buildInputs = with pkgs; [ curl jq ];
> ```
>
> with the following:
>
> Expression
> ```
> buildInputs = builtins.attrValues {
>   inherit (pkgs) curl jq;
> };
> ```

`<...>` lookup paths

You will often encounter Nix language code samples that refer to `<nixpkgs>`.

`<...>` is special syntax that was introduced in 2011 to conveniently access values from the environment variable `$NIX_PATH`.

This means, the value of a lookup path depends on external system state. When using lookup paths, the same Nix expression can produce different results.

In most cases, `$NIX_PATH` is set to the latest channel when Nix is installed, and is therefore likely to differ from machine to machine.

> **Note**
>
> Channels are a mechanism for referencing remote Nix expressions and retrieving their latest version.

The state of a subscribed channel is external to the Nix expressions relying on it. It is not easily portable across machines. This may limit reproducibility.

For example, two developers on different machines are likely to have `<nixpkgs>` point to different revisions of the Nixpkgs repository. Builds may work for one and fail for the other, causing confusion.

> **Tip**
>
> Declare dependencies explicitly using the techniques shown in Towards reproducibility: pinning Nixpkgs.
>
> Do not use lookup paths, except in minimal examples.

Some tools expect the lookup path to be set. In that case:

> **Tip**
>
> Set `$NIX_PATH` to a known value in a central location under version control.
>
> > **NixOS**
> >
> > On NixOS, `$NIX_PATH` can be set permanently with the `nix.nixPath` option.

Reproducible Nixpkgs configuration

To quickly obtain packages for demonstration, we use the following concise pattern:

```
1 import <nixpkgs> {}
```

However, even when `<nixpkgs>` is replaced as shown in Towards reproducibility: pinning Nixpkgs, the result may still not be fully reproducible. This is because, for historical reasons, the Nixpkgs top-level expression by default impurely reads from the file system to obtain configuration parameters. Systems that have the appropriate files populated may end up with different results.

It is a well-known problem that can't be resolved without breaking existing setups.

> **Tip**
>
> Explicitly set `config` and `overlays` when importing Nixpkgs:
>
> ```
> 1 import <nixpkgs> { config = {}; overlays = []; }
> ```

This is what we do in our tutorials to ensure that the examples will behave exactly as expected. We skip it in minimal examples reduce distractions.

Updating nested attribute sets

The attribute set update operator merges two attribute sets.

Example:

Expression
```
{ a = 1; b = 2; } // { b = 3; c = 4; }
```

Value
```
{ a = 1; b = 3; c = 4; }
```

However, names on the right take precedence, and updates are shallow.

Example:

Expression
```
{ a = { b = 1; }; } // { a = { c = 3; }; }
```

Value
```
{ a = { c = 3; }; }
```

Here, key `b` was completely removed, because the whole `a` value was replaced.

Tip

Use the `pkgs.lib.recursiveUpdate` Nixpkgs function:

Expression
```
let pkgs = import <nixpkgs> {}; in
pkgs.lib.recursiveUpdate { a = { b = 1; }; } { a = { c = 3;}; }
```

Value
```
{ a = { b = 1; c = 3; }; }
```

Reproducible source paths

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

pkgs.stdenv.mkDerivation {
  name = "foo";
  src = ./.;
}
```

If the Nix file containing this expression is in `/home/myuser/myproject`, then the store path of `src` will be `/nix/store/<hash>-myproject`.

The problem is that now your build is no longer reproducible, as it depends on the parent directory name. That cannot declared in the source code, and results in an impurity.

If someone builds the project in a directory with a different name, they will get a different store path for `src` and everything that depends on it. This can be the cause of needless rebuilds.

Tip

Use `builtins.path` with the `name` attribute set to something fixed.

This will derive the symbolic name of the store path from `name` instead of the working directory:

Expression

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

pkgs.stdenv.mkDerivation {
  name = "foo";
  src = builtins.path { path = ./.; name = "myproject"; };
}
```

Building a bootable ISO image

Contents

> **Note**
>
> If you need to build images for a different platform, see Cross compiling.

You may find that an official installation image lacks some hardware support.

The solution is to create `myimage.nix` to point to the latest kernel using the minimal installation ISO:

```
{ pkgs, modulesPath, lib, ... }: {
  imports = [
    "${modulesPath}/installer/cd-dvd/installation-cd-minimal.nix"
  ];

  # use the latest Linux kernel
  boot.kernelPackages = pkgs.linuxPackages_latest;

  # Needed for https://github.com/NixOS/nixpkgs/issues/58959
  boot.supportedFilesystems = lib.mkForce [ "btrfs" "reiserfs" "vfat" "f2
}
```

Generate an ISO with the above configuration:

```
$ NIX_PATH=nixpkgs=https://github.com/NixOS/nixpkgs/archive/74e2faf5965a12e8
```

Copy the new image to your USB stick by replacing `sdX` with the name of your device:

```
$ dd if=result/iso/*.iso of=/dev/sdX status=progress
$ sync
```

Next steps

- Take a look at this list of formats that generators support to find your cloud provider or virtualization technology.

Building and running Docker images

Contents

Docker is a set of tools and services used to build, manage and deploy containers.

As many cloud platforms offer Docker-based container hosting services, creating Docker containers for a given service is a common task when building reproducible software. In this tutorial, you will learn how to build Docker containers using Nix.

Prerequisites

You will need both Nix and Docker installed. Docker is available in `nixpkgs`, which is the preferred way to install it on NixOS. However, you can also use the native Docker installation of your OS, if you are on another Linux distribution or macOS.

Build your first container

Nixpkgs provides `dockerTools` to create Docker images:

```
1 { pkgs ? import <nixpkgs> { }
2 , pkgsLinux ? import <nixpkgs> { system = "x86_64-linux"; }
3 }:
```

```
4
5 pkgs.dockerTools.buildImage {
6   name = "hello-docker";
7   config = {
8     Cmd = [ "${pkgsLinux.hello}/bin/hello" ];
9   };
10 }
```

Note

If you're running **macOS** or any platform other than `x86_64-linux`, you'll need to either:

- Set up a remote builder to build on Linux
- Cross compile to Linux by replacing `pkgsLinux.hello` with `pkgs.pkgsCross.musl64.hello`

We call the `dockerTools.buildImage` and pass in some parameters:

- a `name` for our image
- the `config` including the command `Cmd` that should be run inside the container once the image is started. Here we reference the GNU hello package from `nixpkgs` and run its executable in the container.

Save this in `hello-docker.nix` and build it:

```
$ nix-build hello-docker.nix
these derivations will be built:
  /nix/store/qpgdp0qpd8ddi1ld72w02zkmm7n87b92-docker-layer-hello-docker.drv
  /nix/store/m4xyfyviwbi38sfplq3xx54j6k7mccfb-runtime-deps.drv
  /nix/store/v0bvy9qxa79izc7s03fhpq5nqs2h4sr5-docker-image-hello-docker.tar.
warning: unknown setting 'experimental-features'
building '/nix/store/qpgdp0qpd8ddi1ld72w02zkmm7n87b92-docker-layer-hello-doc
No contents to add to layer.
Packing layer...
Computing layer checksum...
Finished building layer 'hello-docker'
building '/nix/store/m4xyfyviwbi38sfplq3xx54j6k7mccfb-runtime-deps.drv'...
building '/nix/store/v0bvy9qxa79izc7s03fhpq5nqs2h4sr5-docker-image-hello-doc
Adding layer...
tar: Removing leading `/' from member names
Adding meta...
Cooking the image...
Finished.
/nix/store/y74sb4nrhxr975xs7h83izgm8z75x5fc-docker-image-hello-docker.tar.gz
```

The image tag (`y74sb4nrhxr975xs7h83izgm8z75x5fc`) refers to the Nix build hash and makes sure that the Docker image corresponds to our Nix build. The store path in the last line of the output references the Docker image.

Run the container

To work with the container, load this image into Docker's image registry from the default `result` symlink created by `nix-build`:

```
$ docker load < result
Loaded image: hello-docker:y74sb4nrhxr975xs7h83izgm8z75x5fc
```

You can also use the store path to load the image in order to avoid depending on the presence of `result`:

```
$ docker load < /nix/store/y74sb4nrhxr975xs7h83izgm8z75x5fc-docker-image-hel
Loaded image: hello-docker:y74sb4nrhxr975xs7h83izgm8z75x5fc
```

Even more conveniently, you can do everything in one command. The advantage of this approach is that `nix-build` will rebuild the image if there are any changes and pass the new store path to `docker load`:

```
$ docker load < $(nix-build hello-docker.nix)
Loaded image: hello-docker:y74sb4nrhxr975xs7h83izgm8z75x5fc
```

Now that you have loaded the image into Docker, you can run it:

```
$ docker run -t hello-docker:y74sb4nrhxr975xs7h83izgm8z75x5fc
Hello, world!
```

Working with Docker images

A general introduction to working with Docker images is not part of this tutorial. The official Docker documentation is a much better place for that.

Note that when you build your Docker images with Nix, you will probably not write a `Dockerfile` as Nix replaces the Dockerfile functionality within the Docker ecosystem. Nonetheless, understanding the anatomy of a Dockerfile may still be useful to understand how Nix replaces each of its functions. Using the Docker CLI, Docker Compose, Docker Swarm or Docker Hub on the other hand may still be relevant, depending on your use case.

Next steps

- More details on how to use `dockerTools` can be found in the reference documentation.
- You might like to browse through more examples of Docker images built with Nix.
- Take a look at Arion, a `docker-compose` wrapper with first-class support for Nix.
- Build docker images on a CI with GitHub Actions.

Continuous integration with GitHub Actions

Contents

In this tutorial, we'll show you **a few short steps** to get started using GitHub Actions as your continuous integration (CI) workflow for commits and pull requests.

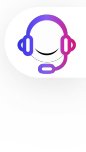
One benefit of Nix is that **CI can build and cache developer environments for every project** on every branch using binary caches.

An important aspect of CI is the feedback loop of, **how many minutes does the build take to finish?**

There are a several good options, but Cachix (below) and integrating with GitHub's built-in cache (at the end) are the most straightforward.

Caching builds using Cachix

Using Cachix you'll never have to waste time building a derivation twice, and you'll share built derivations with all your developers.



After each job, just-built derivations are pushed to your binary cache.

Before each job, derivations to be built are first substituted (if they exist) from your binary cache.

1. Creating your first binary cache

It's recommended to have different binary caches per team, depending who will have write/read access to it.

Fill out the form on the create binary cache page.

On your freshly created binary cache, follow the **Push binaries** tab instructions.

2. Setting up secrets

On your GitHub repository or organization (for use across all repositories):

1. Click on `Settings`.
2. Click on `Secrets`.
3. Add your previously generated secrets (`CACHIX_SIGNING_KEY` and/or `CACHIX_AUTH_TOKEN`).

3. Setting up GitHub Actions

Create `.github/workflows/test.yml` with:

```yaml
name: "Test"
on:
  pull_request:
  push:
jobs:
  tests:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - uses: cachix/install-nix-action@v20
      with:
        nix_path: nixpkgs=channel:nixos-unstable
    - uses: cachix/cachix-action@v12
      with:
        name: mycache
        # If you chose signing key for write access
        signingKey: '${{ secrets.CACHIX_SIGNING_KEY }}'
        # If you chose API tokens for write access OR if you have a private
        authToken: '${{ secrets.CACHIX_AUTH_TOKEN }}'
    - run: nix-build
```

```
    - run: nix-shell --run "echo OK"
```

Once you commit and push to your GitHub repository, you should see status checks appearing on commits and PRs.

Caching builds using GitHub Actions Cache

A quick and easy way to speed up CI on any GitHub repository is to use the Magic Nix Cache. The Magic Nix Cache doesn't require any configuration, secrets, or credentials. This means the caching benefits automatically work for anyone who forks the repository.

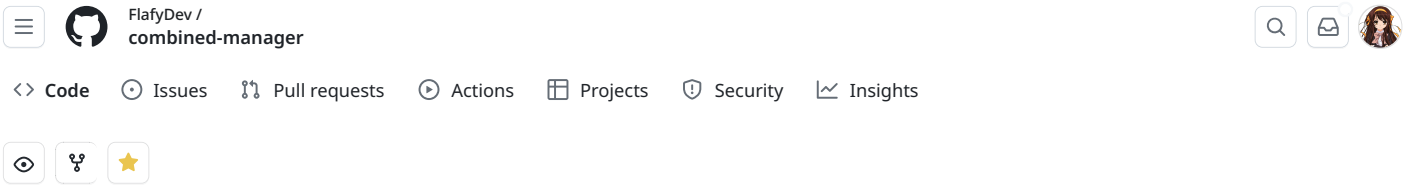
One downside to the Magic Nix Cache is it only works inside GitHub Actions. For more details, check out the readme and the limits of GitHub Actions caching.

Create `.github/workflows/test.yml` with:

```yaml
name: "Test"
on:
  pull_request:
  push:
jobs:
  tests:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - uses: cachix/install-nix-action@v20
      with:
        nix_path: nixpkgs=channel:nixos-unstable
    - uses: DeterminateSystems/magic-nix-cache-action@v2
    - run: nix-build
    - run: nix-shell --run "echo OK"
```

Next steps

- See GitHub Actions workflow syntax
- To quickly setup a Nix project read through Getting started Nix template.



NixOS personal configuration structure library

MIT license

38 stars **0** forks **2** watching **4** Branches **0** Tags Activity

Public repository

main 4 Branches 0 Tags Go to file t Go to file Add file Code

FlafyDev fix: use fixed version of lib when nixpkgs isn't found yesterday

nix-patches	feat: add 2.19.2 patch	yesterday
templates	templates: update combined-manager in e...	8 months ago
LICENSE	Initial commit	9 months ago
README.md	feat: allow reading other NixOS/Combined...	yesterday
default.nix	fix: use fixed version of lib when nixpkgs is...	yesterday
eval-modules.nix	feat: add inputOverriddes, useHomeMana...	2 days ago
flake.nix	move to setup without submodules	8 months ago

README MIT license

Combined Manager

Combined Manager provides a new structure for personal NixOS configurations.

Note: Requires patching `nix` to solve this issue. See more in the Nix Patches section.

Introduction: No separation

Combined Manager's main feature is to break separation. If you want, you should be able to keep everything in a single module. Most NixOS configuration structures are designed to separate related things into multiple files.

Most prominent separations:

- Dividing modules into system and home categories. These categories are then further maintained in separate files.
- All flake inputs must be in the same file in flake.nix.

Combined Manager breaks this pattern by allowing modules to add inputs, overlays and Home Manager and Nixpkgs options as if they are simple options.

Module Options

```
{
  lib,
  pkgs,
  config
```

```
    config,
    osConfig,
    hmConfig,
    inputs,
    combinedManager, # Path to the root of combinedManager
    configs, # The results of all NixOS/CombinedManager configurations
    ...
  }: {
    imports = [ ];

    options = { };

    config = {
      # Adding inputs.
      inputs = { name.url = "..."; };

      # Importing system modules.
      osModules = [ ];

      # Importing Home Manager modules.
      hmModules = [ ];

      # Setting overlays.
      os.nixpkgs.overlays = [ ];

      # Using `os` to set Nixpkgs options.
      os = { };

      # Set Home Manager username (Required to be set in at least one of the modules).
      hmUsername = "myname";

      # Using `hm` to set Home Manager options.
      hm = { };
    };
  }
```

Examples

Full configurations

- https://github.com/FlafyDev/nixos-config

Modules

- https://github.com/FlafyDev/nixos-config/blob/main/modules/display/hyprland/default.nix

Current limitations

- Home Manager required and only a single user with Home Manager.
- Nix must be patched.
- Only for NixOS.

Stability

As of the time of writing, stable *enough*.
While I'll use it for my configuraiton, I have not tested everything and cannot guarantee stability.
There might be breaking changes.

Setup

1. Patch Nix with the patches in the `nix-patches` directory. See more in the Nix Patches section.
2. Generate a template with `nix flake init -t github:FlafyDev/combined-manager#example .`
3. Run `nix flake metadata .`. You might need to run it twice if there is no `flake.lock` file(A message will appear).

Running

To bulid a VM: `nixos-rebuild build-vm --flake .#default`

To swtich: `sudo nixos-rebuild switch --flake .#default .`

Nix Patches

Combined Manager requires applying certain patches to Nix in order to work.
Alternatively, you can use Nix Super.

Required: evaluable-flake.patch (2 line diff)

This patch enables inputs(and the entire flake) to be evaluable. Solves issue #3966.
Combined Manager requires this since it evaluates `inputs` from all the modules.

See line 9 of the example flake.

FAQ

I want to get started, but don't know how to patch Nix.

You can add the following to your config:

```
nix = {
  enable = true;
  package = let
    combinedManager = pkgs.fetchFromGitHub {
      owner = "flafydev";
      repo = "combined-manager";
      rev = "9474a2432b47c0e6fa0435eb612a32e28cbd99ea";
      sha256 = "";
    };
  in
    pkgs.nix.overrideAttrs (old: {
      patches =
        (old.patches or [])
        ++ (
          map
          (file: "${combinedManager}/nix-patches/${file}")
          (lib.attrNames (lib.filterAttrs (_: type: type == "regular") (builtins.readDir "${combinedManager}/nix-patc
        );
    });
};
```

Once you start using Combined Manager, you'll be able to source the patches directly from your `combinedManager` module arg.

Why does Combined Manager need to evaluate inputs?

Each Combined Manager module has an `inputs` option. That option will eventually be merged and set as the inputs of the NixOS configuration.

Releases

No releases published

Packages

No packages published

Languages

- **Nix** 100.0%

Frequently Asked Questions

Contents

Nix

How do I add a new binary cache?

Using NixOS (≥ 22.05):

```
nix.settings = {
  trusted-substituters = [ "https://cache.nixos.org" ];
  substituters = [ "https://cache.nixos.org" ];
};
```

Using NixOS (≤ 21.11):

```
nix = {
  trustedBinaryCaches = [ "https://cache.nixos.org" ];
  binaryCaches = [ "https://cache.nixos.org" ];
};
```

Using `Nix`:

```
$ echo "trusted-binary-caches = https://cache.nixos.org" >> /etc/nix/nix.con
$ nix-build helpers/bench.nix --option extra-binary-caches https://cache.nix
```

How to operate between Nix paths and strings?

See http://stackoverflow.com/a/43850372

How to build reverse dependencies of a package?

```
$ nix-shell -p nixpkgs-review --run "nixpkgs-review wip"
```

How can I manage dotfiles in $HOME with Nix?

See nix-community/home-manager

What's the recommended process for building custom packages?

Please read Packaging existing software with Nix.

How to use a clone of the Nixpkgs repository to update or write new packages?

Please read Packaging existing software with Nix and the Nixpkgs contributing guide.

NixOS

How to run non-nix executables?

NixOS cannot run dynamically linked executables intended for generic Linux environments out of the box. This is because, by design, it does not have a global library path, nor does it follow the Filesystem Hierarchy Standard (FHS).

There are a few ways to resolve this mismatch in environment expectations:

- Use the version packaged in Nixpkgs, if there is one. You can search available packages at https://search.nixos.org/packages.
- Write a Nix expression for the program to package it in your own configuration.

  There are multiple approaches to this:

  - Build from source.

    Many open-source programs are highly flexible at compile time in terms of where their files go. For an introduction to this, see Packaging existing software with Nix.

  - Modify the program's ELF header to include paths to libraries using `autoPatchelfHook`.

    Do this if building from source isn't feasible.

  - Wrap the program to run in an FHS-like environment using `buildFHSEnv`.

    This is a last resort, but sometimes necessary, for example if the program downloads and runs other executables.

- Create a library path that only applies to unpackaged programs by using `nix-ld`. Add this to your `configuration.nix`:

```
programs.nix-ld.enable = true;
programs.nix-ld.libraries = with pkgs; [
  # Add any missing dynamic libraries for unpackaged programs
  # here, NOT in environment.systemPackages
];
```

  Then run `nixos-rebuild switch`, and log out and back in again to propagate the new environment variables. (This is only necessary when enabling `nix-ld`; changes in included libraries take effect immediately on rebuild.)

> **Note**
>
> `nix-ld` does not work for 32-bit executables on `x86_64` machines.

- Run your program in the FHS-like environment made for the Steam package using `steam-run`:

```
$ nix-shell -p steam-run --run "steam-run <command>"
```

How to build my own ISO?

See http://nixos.org/nixos/manual/index.html#sec-building-image

How do I connect to any of the machines in NixOS tests?

Apply following patch:

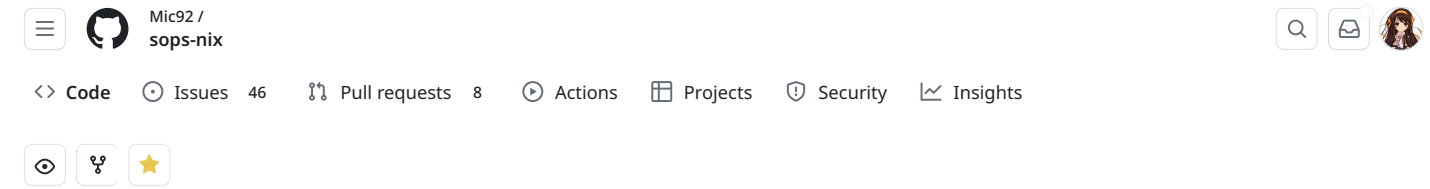
```
diff --git a/nixos/lib/test-driver/test-driver.pl b/nixos/lib/test-driver/te
index 8ad0d67..838fbdd 100644
--- a/nixos/lib/test-driver/test-driver.pl
+++ b/nixos/lib/test-driver/test-driver.pl
@@ -34,7 +34,7 @@ foreach my $vlan (split / /, $ENV{VLANS} || "") {
     if ($pid == 0) {
         dup2(fileno($pty->slave), 0);
         dup2(fileno($stdoutW), 1);
-        exec "vde_switch -s $socket" or _exit(1);
+        exec "vde_switch -tap tap0 -s $socket" or _exit(1);
     }
     close $stdoutW;
     print $pty "version\n";
```

And then the vde_switch network should be accessible locally.

How to bootstrap NixOS inside an existing Linux installation?

There are a couple of tools:

- nix-community/nixos-anywhere
- jeaye/nixos-in-place
- elitak/nixos-infect
- cleverca22/nix-tests



Atomic secret provisioning for NixOS based on sops

MIT license

1k stars **112** forks **10** watching **10** Branches **1** Tags Activity

Public repository

master 10 Branches 1 Tags Go to file t Go to file Add file Code

github-actions[bot] and mergify[bot] flake.lock: Update		last week
.github	build(deps): bump cachix/install-nix-action ...	3 weeks ago
modules	don't substitute binaries	4 months ago
pkgs	fix wrong error message in ssh key import	3 weeks ago
scripts	drop git commit from update-vendor-hash....	7 months ago
.gitignore	add tests + ssh key support	4 years ago
.mergify.yml	mergify: rebase prs	5 months ago
LICENSE	add MIT license	4 years ago
README.md	Add info about hash passwords	last week
default.nix	update vendorHash	last month
flake.lock	flake.lock: Update	last week
flake.nix	sops-nix: upgrade to 23.05	8 months ago
go.mod	build(deps): bump golang.org/x/crypto fro...	last month
go.sum	build(deps): bump golang.org/x/crypto fro...	last month
shell.nix	shell.nix: no longer use out-dated nixFlakes...	2 years ago

README MIT license

sops-nix

Atomic, declarative, and reproducible secret provisioning for NixOS based on sops.

How it works

Secrets are decrypted from sops files during activation time. The secrets are stored as one secret per file and access-controlled by fu

declarative configuration of their users, permissions, and groups. GPG keys or `age` keys can be used for decryption, and compatibilit shims are supported to enable the use of SSH RSA or SSH Ed25519 keys. Sops also supports cloud key management APIs such as AW KMS, GCP KMS, Azure Key Vault and Hashicorp Vault. While not officially supported by sops-nix yet, these can be controlled using environment variables that can be passed to sops.

Features

- Compatible with all NixOS deployment frameworks: NixOps, nixos-rebuild, krops, morph, nixus, etc.
- Version-control friendly: Since all files are encrypted they can be directly committed to version control without worry. Diffs of the secrets are readable, and can be shown in cleartext.
- CI friendly: Since sops files can be added to the Nix store without leaking secrets, a machine definition can be built as a whole fro a repository, without needing to rely on external secrets or services.
- Home-manager friendly: Provides a home-manager module
- Works well in teams: sops-nix comes with `nix-shell` hooks that allows multiple people to quickly import all GPG keys. The cryptography used in sops is designed to be scalable: Secrets are only encrypted once with a master key instead of encrypted pe machine/developer key.
- Atomic upgrades: New secrets are written to a new directory which replaces the old directory atomically.
- Rollback support: If sops files are added to the Nix store, old secrets can be rolled back. This is optional.
- Fast time-to-deploy: Unlike solutions implemented by NixOps, krops and morph, no extra steps are required to upload secrets.
- A variety of storage formats: Secrets can be stored in YAML, dotenv, INI, JSON or binary.
- Minimizes configuration errors: sops files are checked against the configuration at evaluation time.

Demo

There is a `configuration.nix` example in the deployment step of our usage example.

Supported encryption methods

sops-nix supports two basic ways of encryption, GPG and `age` .

GPG is based on GnuPG and encrypts against GPG public keys. Private GPG keys may be used to decrypt the secrets on the target machine. The tool `ssh-to-pgp` can be used to derive a GPG key from a SSH (host) key in RSA format.

The other method is `age` which is based on `age` . The tool (`ssh-to-age`) can convert SSH host or user keys in Ed25519 format to `age` keys.

Usage example

If you prefer video over the textual description below, you can also checkout this 6min tutorial by @vimjoyer.

▸ **1. Install sops-nix**

▸ **2. Generate a key for yourself**

▸ **3. Get a public key for your target machine**

▸ **4. Create a sops file**

▸ **5. Deploy**

Set secret permission/owner and allow services to access it

By default secrets are owned by `root:root` . Furthermore the parent directory `/run/secrets.d` is only owned by `root` and the group has read access to it:

```
$ ls -la /run/secrets.d/1
total 24
drwxr-x--- 2 root keys   0 Jul 12  6:23 .
drwxr-x--- 3 root keys   0 Jul 12  6:23 ..
-r-------- 1 root root  20 Jul 12  6:23 example-secret
```

The secrets option has further parameter to change secret permission. Consider the following nixos configuration example:

```
{
  # Permission modes are in octal representation (same as chmod),
  # the digits represent: user|group|others
  # 7 - full (rwx)
  # 6 - read and write (rw-)
  # 5 - read and execute (r-x)
  # 4 - read only (r--)
  # 3 - write and execute (-wx)
  # 2 - write only (-w-)
  # 1 - execute only (--x)
  # 0 - none (---)
  sops.secrets.example-secret.mode = "0440";
  # Either a user id or group name representation of the secret owner
  # It is recommended to get the user name from `config.users.users.<?name>.name` to avoid misconfiguration
  sops.secrets.example-secret.owner = config.users.users.nobody.name;
  # Either the group id or group name representation of the secret group
  # It is recommended to get the group name from `config.users.users.<?name>.group` to avoid misconfiguration
  sops.secrets.example-secret.group = config.users.users.nobody.group;
}
```

▸ This example configures secrets for buildkite, a CI agent; the service needs a token and a SSH private key to function.

Restarting/reloading systemd units on secret change

It is possible to restart or reload units when a secret changes or is newly initialized.

This behavior can be configured per-secret:

```
{
  sops.secrets."home-assistant-secrets.yaml" = {
    restartUnits = [ "home-assistant.service" ];
    # there is also `reloadUnits` which acts like a `reloadTrigger` in a NixOS systemd service
  };
}
```

Symlinks to other directories

Some services might expect files in certain locations. Using the `path` option a symlink to this directory can be created:

```
{
  sops.secrets."home-assistant-secrets.yaml" = {
    owner = "hass";
    path = "/var/lib/hass/secrets.yaml";
  };
}
```

```
$ ls -la /var/lib/hass/secrets.yaml
lrwxrwxrwx 1 root root 40 Jul 19 22:36 /var/lib/hass/secrets.yaml -> /run/secrets/home-assistant-secrets.yaml
```

Setting a user's password

sops-nix has to run after NixOS creates users (in order to specify what users own a secret.) This means that it's not possible to set `users.users.<name>.hashedPasswordFile` to any secrets managed by sops-nix. To work around this issue, it's possible to set `neededForUsers = true` in a secret. This will cause the secret to be decrypted to `/run/secrets-for-users` instead of `/run/secrets` before NixOS creates users. As users are not created yet, it's not possible to set an owner for these secrets.

The password must be stored as a hash for this to work, which can be created with the command `mkpasswd`

```
$ echo "password" | mkpasswd -s
$y$j9T$WFoiErKnEnMcGq0ruQK4K.$4nJAY3LBeBsZBTYSkdTOejKU6KlDmhnfUV3Ll1K/1b.
```

```
{ config, ... }: {
  sops.secrets.my-password.neededForUsers = true;

  users.users.mic92 = {
    isNormalUser = true;
    hashedPasswordFile = config.sops.secrets.my-password.path;
  };
}
```

Different file formats

At the moment we support the following file formats: YAML, JSON, INI, dotenv and binary.

sops-nix allows specifying multiple sops files in different file formats:

```
{
  imports = [ <sops-nix/modules/sops> ];
  # The default sops file used for all secrets can be controlled using `sops.defaultSopsFile`
  sops.defaultSopsFile = ./secrets.yaml;
  # If you use something different from YAML, you can also specify it here:
  #sops.defaultSopsFormat = "yaml";
  sops.secrets.github_token = {
    # The sops file can be also overwritten per secret...
    sopsFile = ./other-secrets.json;
    # ... as well as the format
    format = "json";
  };
}
```

YAML

Open a new file with sops ending in `.yaml`:

```
$ sops secrets.yaml
```

Then, put in the following content:

```
github_token: 4a6c73f74928a9c4c4bc47379256b72e598e2bd3
ssh_key: |
  -----BEGIN OPENSSH PRIVATE KEY-----
  b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAMwAAAAtzc2gtZW
  QyNTUxOQAAACDENhLwQI4v/Ecv65iCMZ7aZAL+Sdc0Cqyjkd012XwJzQAAAJht4at6beGr
  egAAAAtzc2gtZWQyNTUxOQAAACDENhLwQI4v/Ecv65iCMZ7aZAL+Sdc0Cqyjkd012XwJzQ
  AAAEBizgX7v+VMZeiCtWRjpl95dxqBWUkbrPsUSYF3DGV0rsQ2EvBAji/8Ry/rmIIxntpk
  Av5J1zQKrKOR3TXZfAnNAAAAE2pvZXJnQHR1cmluZ21hY2hpbmUBAg==
  -----END OPENSSH PRIVATE KEY-----
```

You can include it like this in your `configuration.nix`:

```
{
  sops.defaultSopsFile = ./secrets.yaml;
  # YAML is the default
  #sops.defaultSopsFormat = "yaml";
  sops.secrets.github_token = {
    format = "yaml";
    # can be also set per secret
    sopsFile = ./secrets.yaml;
  };
}
```

JSON

Open a new file with sops ending in `.json`:

```
$ sops secrets.json
```

Then, put in the following content:

```
{
  "github_token": "4a6c73f74928a9c4c4bc47379256b72e598e2bd3",
  "ssh_key": "-----BEGIN OPENSSH PRIVATE KEY-----\\nb3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAMwAAAAtzc
}
```

You can include it like this in your `configuration.nix`:

```
{
  sops.defaultSopsFile = ./secrets.json;
  # YAML is the default
  sops.defaultSopsFormat = "json";
  sops.secrets.github_token = {
    format = "json";
    # can be also set per secret
    sopsFile = ./secrets.json;
  };
}
```

Binary

This format allows to encrypt an arbitrary binary format that can't be put into JSON/YAML files. Unlike the other two formats, for bina files, one file corresponds to one secret.

To encrypt an binary file use the following command:

```
$ cp /etc/krb5/krb5.keytab > krb5.keytab
$ sops -e -o krb5.keytab
# an example of what this might result in:
$ head krb5.keytab
{
        "data": "ENC[AES256_GCM,data:bIsPHrjrl9wxvKMcQzaAbS3RXCI2h8spw2Ee+KYUTsuousUBU6OMIdyY0wqrX3eh/1BUtl8H9EZciCTw
        "sops": {
                "kms": null,
                "gcp_kms": null,
                "azure_kv": null,
                "lastmodified": "2020-07-06T06:21:06Z",
                "mac": "ENC[AES256_GCM,data:ISjUzaw/5mNiwypmUrOk2DAZnlkbnhURHmTTYA3705NmRsSyUh1PyQvCuwglmaHscwl4GrsnI
                "pgp": [
                        {
```

It can be decrypted again like this:

```
$ sops -d krb5.keytab > /tmp/krb5.keytab
```

This is how it can be included in your `configuration.nix`:

```
{
  sops.secrets.krb5-keytab = {
    format = "binary";
    sopsFile = ./krb5.keytab;
  };
}
```

Use with home manager

sops-nix also provides a home-manager module. This module provides a subset of features provided by the system-wide sops-nix since features like the creation of the ramfs and changing the owner of the secrets are not available for non-root users.

Instead of running as an activation script, sops-nix runs as a systemd user service called `sops-nix.service`. And instead of decryptin to `/run/secrets`, the secrets are decrypted to `$XDG_RUNTIME_DIR/secrets` that is located on a tmpfs or similar non-persistent

to `/run/secrets`, the secrets are decrypted to `$XDG_RUNTIME_DIR/secrets` that is located on a tmpfs or similar non-persistent filesystem.

Depending on whether you use home-manager system-wide or using a home.nix, you have to import it in a different way. This example shows the `flake` approach from the recommended example Install: Flakes (current recommendation)

```
{
  # NixOS system-wide home-manager configuration
  home-manager.sharedModules = [
    inputs.sops-nix.homeManagerModules.sops
  ];
}
```

```
{
  # Configuration via home.nix
  imports = [
    inputs.sops-nix.homeManagerModules.sops
  ];
}
```

This example show the `channel` approach from the example Install: nix-channel. All other methods work as well.

```
{
  # NixOS system-wide home-manager configuration
  home-manager.sharedModules = [
    <sops-nix/modules/home-manager/sops.nix>
  ];
}
```

```
{
  # Configuration via home.nix
  imports = [
    <sops-nix/modules/home-manager/sops.nix>
  ];
}
```

The actual sops configuration is in the `sops` namespace in your home.nix (or in the `home-manager.users.<name>` namespace when using home-manager system-wide):

```
{
  sops = {
    age.keyFile = "/home/user/.age-key.txt"; # must have no password!
    # It's also possible to use a ssh key, but only when it has no password:
    #age.sshKeyPaths = [ "/home/user/path-to-ssh-key" ];
    defaultSopsFile = ./secrets.yaml;
    secrets.test = {
      # sopsFile = ./secrets.yml.enc; # optionally define per-secret files

      # %r gets replaced with a runtime directory, use %% to specify a '%'
      # sign. Runtime dir is $XDG_RUNTIME_DIR on linux and $(getconf
      # DARWIN_USER_TEMP_DIR) on darwin.
      path = "%r/test.txt";
    };
  };
}
```

The secrets are decrypted in a systemd user service called `sops-nix`, so other services needing secrets must order after it:

```
{
  systemd.user.services.mbsync.Unit.After = [ "sops-nix.service" ];
}
```

Use with GPG instead of SSH keys

If you prefer having a separate GPG key, sops-nix also comes with a helper tool, `sops-init-gpg-key`:

```
$ nix run github:Mic92/sops-nix#sops-init-gpg-key -- --hostname server01 --gpghome /tmp/newkey
# You can use the following command to save it to a file:
$ cat > server01.asc <<EOF
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQENBF8L/iQBCACroEaUfvPBMMorNepNQmideOtNztALejgEJ5wZmxabck+qC1Gb
NWe3tmvChXVHgL7DzodSUfX1PuIjTTeRr2clMXtISPFIsBlRQb4MiErZfsardITM
n4WScg8sTb4nnqEOJiRknwAhBryIjH8kkCXxKlYK67re281dIK4dKBMIolFADlyv
wyHurJ7NPpHxR2WXHcIqXX1DaT6RvGQvZHMpfctob8k/QD4CyV6QwG5IVACQ/tuC
bEUggrkGw+g+XdeieUfWbRsHM4C4pv8BNwA/EYD5d0eKI+rshSPoTT+hcGn8Uh8w
MVQ8PVs6jWMMOAF1JH/stoPr9Yha+TGbMRi5ABEBAAG0GHNlcnZlcjAxIDxyb290
QHNlcnZlcjAxPokBTgQTAQgAOBYhBOTKhnaPF2rrbAFVQVOvjX8UlhOxBQJfC/4k
AhsvBQsJCAcCBhUKCQgLAgQWAgMBAh4BAheAAAoJEFOvjX8UlhOx1XIH/jUOrSR2
wuoqFiHcqaDPgXmTVJk8QanVkmiP3tk0mz5rRKrDX2eX5GnHqYR4PfpjUYNzedQE
sGyTjl7+DvglWJ2Q8m3yD/9+1agBmeqEVQlKqwL6Sc3bI4WBwHaxwVDo/bNwMs0w
o8ngOs1jPd3LfQdfG/rE1NolpHm4LWqYj0D2zEGqozLXVBx2wiuwmm6OKX4U4EHR
UwKax+VZYA+J9oFDN+kOy/yR+bKnOvg5eyOv2ZrK5BKceSBhDTOclMIWTL2cGxcL
jsq4N7fobs4TbwFPxRUi/T9ldXi0LXeGhTl9stImTtj3bL+4Y734TipvB5UvzCDK
CkjjwEvD5MYdGDE=
=uvIf
-----END PGP PUBLIC KEY BLOCK-----
EOF
# fingerprint: E4CA86768F176AEB6C01554153AF8D7F149613B1
```

You can choose between a RSA GPG key (default, like in the example above) or a Curve25519 based one by adding `--keytype Curve25519` like so:

```
$ nix run github:Mic92/sops-nix#sops-init-gpg-key -- --hostname server01 --gpghome /tmp/newkey --keytype Curve25519
You can use the following command to save it to a file:
cat > server01.asc <<EOF
-----BEGIN PGP PUBLIC KEY BLOCK-----

mDMEY7dJExYJKwYBBAHaRw8BAQdAloRZFyqNh3nIDtyUQKaBSMJOtLkbNeg+4TPg
BG5TduG0OG5peC1hLmhvbWUua3VldGVtZWllci5kZSA8cm9vdEBuaXgtYS5ob21l
Lmt1ZXRlbWVpZXIuZGU+iJMEExYKADsWIQREE2hPxiNijOo+CSmrLxbGte+J7wUC
Y7dJEwIbAwULCQgHAgIiAgYVCgkICwIEFgIDAQIeBwIXgAAKCRCrLxbGte+J79LX
AQDtLfQFDKm04ORIk28DrzTBbMTFQEW21dGBXk7ykBx4jQD/ZOnt1RPnB9mzMc8L
wIS3oI8D9719DjoS9hrHnJ4xvge4OARjt0kTEgorBgEEAZdVAQUBAQdA0t1X35pN
ic+etscIIkHjKUwrXhbTgWrARgXUuEMwwz8DAQgHiHgEGBYKACAWIQREE2hPxiNi
jOo+CSmrLxbGte+J7wUCY7dJEwIbDAAKCRCrLxbGte+J7+0NAQCfj95TSyPEFKz3
eLJ1aCA1bZZV/rkhHd+OwX1MFL3mKQD9GMPgvMzDIoofycDzMY2ttJgkRJfq+zOZ
juXFQdUkMgY=
=pf3V
-----END PGP PUBLIC KEY BLOCK-----
EOF
fingerprint: 4413684FC623628CEA3E0929AB2F16C6B5EF89EF
F0477297E369CD1D189DD901278D1535AB473B9E
```

In both cases, you must upload the GPG key directory `/tmp/newkey` onto the server. If you uploaded it to `/var/lib/sops` than your sops configuration will look like this:

```
{
  # Make sure that `/var/lib/sops` is owned by root and is not world-readable/writable
  sops.gnupg.home = "/var/lib/sops";
  # disable importing host ssh keys
  sops.gnupg.sshKeyPaths = [];
}
```

However be aware that this will also run GnuPG on your server including the GnuPG daemon. GnuPG is in general not great sof and might break in hilarious ways. If you experience problems, you are on your own. If you want a more stable and predictable solution go with SSH keys or one of the KMS services.

Share secrets between different users

Secrets can be shared between different users by creating different files pointing to the same sops key but with different permissions In the following example the `drone` secret is exposed as `/run/secrets/drone-server` for `drone-server` and as `/run/secrets/drone-`

agent for drone-agent:

```
{
  sops.secrets.drone-server = {
    owner = config.systemd.services.drone-server.serviceConfig.User;
    key = "drone";
  };
  sops.secrets.drone-agent = {
    owner = config.systemd.services.drone-agent.serviceConfig.User;
    key = "drone";
  };
}
```

Migrate from pass/krops

If you have used pass before (e.g. in krops) than you can use the following one-liner to convert all your secrets to a YAML structure:

```
$ for i in *.gpg; do echo "$(basename $i .gpg): |\n$(pass $(dirname $i)/$(basename $i .gpg)| sed 's/^/  /')"; done
```

Copy the output to the editor you have opened with sops.

Real-world examples

The nix-community infra makes extensive usage of sops-nix. Each host has a secrets.yaml containing secrets for the host. Also Samue Leathers explains his personal setup in this blog article.

Known limitations

Initrd secrets

sops-nix does not fully support initrd secrets. This is because `nixos-rebuild switch` installs the bootloader before running sops-nix's activation hook.
As a workaround, it is possible to run `nixos-rebuild test` before `nixos-rebuild switch` to provision initrd secrets before actually using them in the initrd. In the future, we hope to extend NixOS to allow keys to be provisioned in the bootloader install phase.

Using secrets at evaluation time

It is not possible to use secrets at evaluation time of nix code. This is because sops-nix decrypts secrets only in the activation phase o nixos i.e. in `nixos-rebuild switch` on the target machine. If you rely on this feature for some secrets, you should also include solutions that allow secrets to be stored securely in your version control, e.g. git-agecrypt. These types of solutions can be used together with sops-nix.

Templates

If your setup requires embedding secrets within a configuration file, the `template` feature of `sops-nix` provides a seamless way to c this.

Here's how to use it:

1. **Define Your Secret**

   Specify the secrets you intend to use. This will be encrypted and managed securely by `sops-nix`.

   ```
   {
     sops.secrets.your-secret = { };
   }
   ```

2. **Use Templates for Configuration with Secrets**

   Create a template for your configuration file and utilize the placeholder where you'd like the secret to be inserted. During the activation phase, `sops-nix` will substitute the placeholder with the actual secret content.

```
{
  sops.templates."your-config-with-secrets.toml".content = ''
    password = "${config.sops.placeholder.your-secret}"
  '';
}
```

You can also define ownership properties for the configuration file:

```
{
  sops.templates."your-config-with-secrets.toml".owner = "serviceuser";
}
```

3. **Reference the Rendered Configuration in Services**

   When defining a service (e.g., using `systemd`), refer to the rendered configuration (with secrets in place) by leveraging the `.path` attribute.

```
{
  systemd.services.myservice = {
    # ... (any other service attributes)

    serviceConfig = {
      ExecStart = "${pkgs.myservice}/bin/myservice --config ${config.sops.templates."your-config-with-secrets.tom
      User = "serviceuser";
    };
  };
}
```

Related projects

- agenix: Similar features as sops-nix but uses age.
- scalpel: Provides a simple template mechanism to inject secrets into configuration files in the nixos activation phase

Need more commercial support?

We are building sops-nix very much as contributors to the community and are committed to keeping it open source.

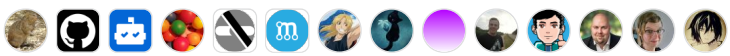
That said, many of us that are contributing to sops-nix also work for consultancies. If you want to contact one of those for paid-for support setting up sops-nix in your infrastructure you can do so here:

- Numtide
- Helsinki Systems

Releases

1 tags

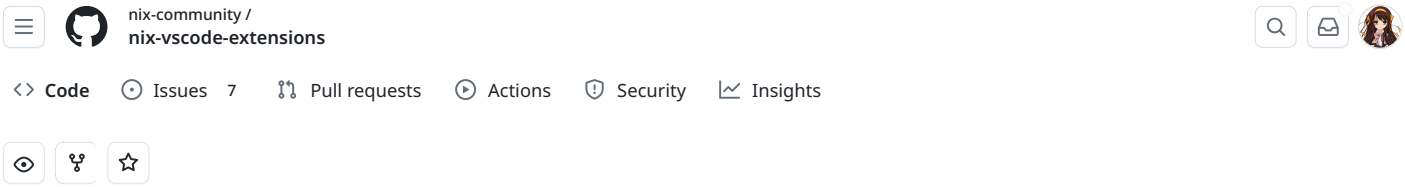
Contributors 57



+ 43 contributors

Languages

Go 48.9% Nix 46.6% Shell 4.0% Python 0.5%



Nix expressions for VSCode and OpenVSX extensions [maintainers: **@deemp**, **@AmeerTaweel**]

MIT license
Code of conduct
Security policy

95 stars **7** forks **5** watching **2** Branches **0** Tags Activity Custom properties

Public repository

master · 2 Branches · 0 Tags · Go to file t · Go to file · Add file · Code

github-actions action		1 hour ago
.github	upd ci	6 months ago
data	action	1 hour ago
haskell	nix-dev, haskell: refactor default.nix	7 months ago
nix-dev	upd ci	6 months ago
template	action	1 hour ago
.env	fix paths	7 months ago
.envrc	update descriptions	9 months ago
.gitignore	add extra extensions	7 months ago
.markdownlint.jsonc	add template	2 years ago
LICENSE	chore: add license	2 years ago
README.md	Update README.md	3 months ago
default.nix	add default.nix	last year
extra-extensions.toml	add extra extensions	7 months ago
flake.lock	add platform-specific exts, improve cache e...	9 months ago
flake.nix	add overrides	7 months ago
overrides.nix	add overrides	7 months ago

README · Code of conduct · MIT license · Security

Nix expressions for VS Code Extensions

At the time of writing this, `nixpkgs` contains **271** `VS Code` extensions. This is a small fraction of the more than **40,000** extensions in the `VS Code Marketplace`! In addition, many of the extensions in `nixpkgs` are significantly out-of-date.

This flake provides Nix expressions for the majority of available extensions from Open VSX and VS Code Marketplace. A `GitHub Actio` updates the extensions daily.

That said, you can now use a different set of extensions for `VS Code` / `VSCodium` in each of your projects. Moreover, you can share you flakes and cache them so that other people don't need to install these extensions manually!

Note

- Check nix4vscode (and contribute!) if you need a more individual approach to extensions.
- NixOS wiki has a page about VS Code.
- Extension publishers and names are lowercased.
- Access an extension in the format `<attrset>.<publisher>.<name>`, where `<attrset>` is `vscode-marketplace`, `open-vsx`, et Explore).
- If an extension publisher or name aren't valid Nix identifiers, quote them like `<attrset>."4"."2"`.
- We have a permission from MS to use a crawler on their API in this case (see the discussion). Please, don't abuse this flake!

Template

This repository has a flake template. This template provides a VSCodium with a couple of extensions.

1. Create a flake from the template (see nix flake new).

```
nix flake new vscodium-project -t github:nix-community/nix-vscode-extensions
cd vscodium-project
git init && git add .
```

2. Run `VSCodium`.

```
nix run .# .
```

3. Alternatively, start a devShell and run `VSCodium`. A `shellHook` will print extensions available in the `VSCodium`.

```
nix develop
codium .
```

In case of problems see Troubleshooting.

Example

flake.nix provides a default package. This package is `VSCodium` with a couple of extensions.

Run `VSCodium` and list installed extensions.

```
nix run github:nix-community/nix-vscode-extensions# -- --list-extensions
```

Usage

Extensions

We provide extensions attrsets that contain both universal and platform-specific extensions. We use a reasonable mapping between the sites target platforms and Nix-supported platforms (see the issue and `systemPlatform` in flake.nix).

There are several attrsets:

- `vscode-marketplace` and `open-vsx` contain the latest versions of extensions, including pre-release ones. Such pre-release versions expire in some time. That's why, there are `-release` attrsets.
- `vscode-marketplace-release` and `open-vsx-release` contain the release versions of extensions (see Release extensions).
- `forVSCodeVersion "4.228.1"` allows to leave only the extensions compatible with the `"4.228.1"` version of `VS Code`.
  - You may supply the actual version of your `VS Code` instead of `"4.228.1"`.

With flakes

See Template.

Add the following to your `flake.nix` (see Flakes).

```
inputs.nix-vscode-extensions.url = "github:nix-community/nix-vscode-extensions";
```

Without flakes

```
let
  system = builtins.currentSystem;
  extensions =
    (import (builtins.fetchGit {
      url = "https://github.com/nix-community/nix-vscode-extensions";
      ref = "refs/heads/master";
      rev = "c43d9089df96cf8aca157762ed0e2ddca9fcd71e";
    })).extensions.${system};
```

```
  extensionsList = with extensions.vscode-marketplace; [
      rust-lang.rust-analyzer
  ];
in ...
```

History

You can search for an extension in the repo history:

- get commits containing the extension: `git log -S '"copilot"' --oneline data/cache/vscode-marketplace-latest.json`
- select a commit: `0910d1e`
- search in that commit: `git grep '"copilot"' 0910d1e -- data/cache/vscode-marketplace-latest.json`

Explore

Explore extensions via `nix repl`.

Use your system instead of `x86_64-linux`.

Press the `Tab` button (denoted as `<TAB>` here) to see attrset attributes.

Get the `extensions` attrset

Get `extensions` with flakes

```
$ nix repl

nix-repl> :lf github:nix-community/nix-vscode-extensions/c43d9089df96cf8aca157762ed0e2ddca9fcd71e
Added 10 variables.

nix-repl> t = extensions.<TAB>
extensions.aarch64-darwin  extensions.aarch64-linux   extensions.x86_64-darwin   extensions.x86_64-linux

nix-repl> t = extensions.x86_64-linux

nix-repl> t.<TAB>
t.forVSCodeVersion           t.open-vsx-release           t.vscode-marketplace-release
t.open-vsx                   t.vscode-marketplace
```

Get `extensions` without flakes

```
$ nix repl

nix-repl> t1 = (import (builtins.fetchGit {
                url = "https://github.com/nix-community/nix-vscode-extensions";
                ref = "refs/heads/master";
                rev = "c43d9089df96cf8aca157762ed0e2ddca9fcd71e";
              }))

nix-repl> t = t1.extensions.<TAB>
t1.extensions.aarch64-darwin  t1.extensions.aarch64-linux   t1.extensions.x86_64-darwin   t1.extensions.x86_64-linux

nix-repl> t = t1.extensions.x86_64-linux

nix-repl> t.<TAB>
t.forVSCodeVersion           t.open-vsx-release           t.vscode-marketplace-release
t.open-vsx                   t.vscode-marketplace
```

Pre-release versions

```
nix-repl> t.vscode-marketplace.rust-lang.rust-analyzer
«derivation /nix/store/jyzab0pdcgj4q9l73zsnyvc1k7qpb381-vscode-extension-rust-lang-rust-analyzer-0.4.1582.drv»
```

Release versions

```
nix-repl> t.vscode-marketplace-release.rust-lang.rust-analyzer
«derivation /nix/store/qjlr7iqgqrf2hd2z21xz96nmblxy680m-vscode-extension-rust-lang-rust-analyzer-0.3.1583.drv»
```

Pre-release versions compatible with a given version of VS Code

```
nix-repl> (t.forVSCodeVersion "1.78.2").vscode-marketplace.rust-lang.rust-analyzer
«derivation /nix/store/jyzab0pdcgj4q9l73zsnyvc1k7qpb381-vscode-extension-rust-lang-rust-analyzer-0.4.1582.drv»
```

Overlay

See Overlays.

Get an overlay with flakes

```
nix-repl> :lf github:nix-community/nix-vscode-extensions/c43d9089df96cf8aca157762ed0e2ddca9fcd71e
Added 14 variables.

nix-repl> :lf github:nix-community/nix-vscode-extensions/c43d9089df96cf8aca157762ed0e2ddca9fcd71e
Added 14 variables.

nix-repl> pkgs = (legacyPackages.x86_64-linux.extend overlays.default)

nix-repl> pkgs.vscode-marketplace-release.rust-lang.rust-analyzer
«derivation /nix/store/midv6wrnpxfm3in3miilyx914zzck4d7-vscode-extension-rust-lang-rust-analyzer-0.3.1575.drv»
```

Get an overlay without flakes

```
nix-repl> t1 = (import (builtins.fetchGit {
                 url = "https://github.com/nix-community/nix-vscode-extensions";
                 ref = "refs/heads/master";
                 rev = "c43d9089df96cf8aca157762ed0e2ddca9fcd71e";
               }))

nix-repl> pkgs = import <nixpkgs> { overlays = [ t1.overlays.default ]; system = builtins.currentSystem; }

nix-repl> pkgs.vscode-marketplace-release.rust-lang.rust-analyzer
«derivation /nix/store/a701wlb8ckidpikr57bff16mmvsf3jir-vscode-extension-rust-lang-rust-analyzer-0.3.1575.drv»
```

Contribute

Issues

Resolve issues.

README

- Fix links.
- Write new sections.
- Update commit SHA used in examples if they're too old.
- Enhance text.

Release extensions

The config contains several extensions. We cache the information about the latest **release** versions of these extensions (see Extensions). If you'd like to use release versions of an extension, please, add that extension to the config and make a Pull Request.

Extra extensions

The extra-extensions.toml file contains a list of extensions to be fetched from sites other than `VS Code Marketplace` and `Open VSX`. These extensions replace ones fetched from `VS Code Marketplace` and `Open VSX`. Add necessary extensions there, preferrably, for a supported platforms (see Extensions). nvfetcher will fetch the latest release versions of these extensions and write configs to

generated.nix.

Special extensions

Certain extensions require special treatment. Provide functions to modify such extensions derivations in overrides.nix. Optionally, create and link there issues explaining chosen override functions. The overrides apply to a whole attrset of extensions, including Extra extensions.

Build problems

- Extension with multiple extensions in a zipfile (issue)
- Platform-specific extensions (comment)

Main flake

1. (Optionally) Install direnv, e.g., via `nix profile install nixpkgs#direnv`.
2. Run a devshell. When prompted about `extra-trusted-substituters` answer `y`. This is to use binary caches.

```
nix develop nix-dev/
```

3. (Optionally) Start `VSCodium` with necessary extensions and tools.

```
nix run nix-dev/#writeSettings
nix run nix-dev/#codium .
```

Haskell script

1. See the README.
2. Set the environment.

```
set -a
source .env
```

3. Run the script.

```
nix run haskell/#updateExtensions
```

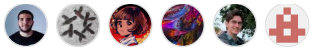
Pull requests

Pull requests are welcome!

Troubleshooting

- If `Nix`-provided `VSCodium` doesn't pick up the extensions:
  - Close other instance of `Nix`-provided `VSCodium`.
  - Try to reboot your computer and start `VSCodium` again.
- See troubleshooting.

Contributors 7



Languages

NixOps

From NixOS Wiki

NixOps is a tool for deploying NixOS machines in a network or cloud. It takes as input a declarative specification of a set of “logical” machines and then performs any necessary steps or actions to realise that specification: instantiate cloud machines, build and download dependencies, stop and start services, and so on. NixOps has several nice properties:
It’s declarative: NixOps specifications state the desired configuration of the machines, and NixOps then figures out the actions necessary to realise that configuration. So there is no difference between doing a new deployment or doing a redeployment: the resulting machine configurations will be the same.

It performs fully automated deployment. This is a good thing because it ensures that deployments are reproducible.

It performs provisioning. Based on the given deployment specification, it will start missing virtual machines, create disk volumes, and so on.

It’s based on the Nix package manager, which has a purely functional model that sets it apart from other package managers. Concretely this means that multiple versions of packages can coexist on a system, that packages can be upgraded or rolled back atomically, that dependency specifications can be guaranteed to be complete, and so on.

It’s based on NixOS, which has a declarative approach to describing the desired configuration of a machine. This makes it an ideal basis for automated configuration management of sets of machines. NixOS also has desirable properties such as (nearly) atomic upgrades, the ability to roll back to previous configurations, and more.

It’s multi-cloud. Machines in a single NixOps deployment can be deployed to different target environments. For instance, one logical machine can be deployed to a local “physical” machine, another to an automatically instantiated Amazon EC2 instance in the eu-west-1 region, another in the us-east-1 region, and so on. NixOps arranges the necessary network configuration to ensure that these machines can communicate securely with each other (e.g. by setting up encrypted tunnels).

It supports separation of “logical” and “physical” aspects of a deployment. NixOps specifications are modular, and this makes it easy to separate the parts that say what logical machines should do from where they should do it. For instance, the former might say that machine X should run a PostgreSQL database and machine Y should run an Apache web server, while the latter might state that X should be instantiated as an EC2 m1.large machine while Y should be instantiated as an m1.small. We could also have a second physical specification that says that X and Y should both be instantiated as VirtualBox VMs on the developer’s workstation. So the same logical specification can easily be deployed to different environments.

It uses a single formalism (the Nix expression language) for package management and system configuration management. This makes it very easy to add ad hoc packages to a deployment.

It combines system configuration management and provisioning. Provisioning affects configuration management: for instance, if we instantiate an EC2 machine as part of a larger deployment, it may be necessary to put the IP address or hostname of that machine in a configuration file on another machine. NixOps takes care of this automatically.
It can provision non-machine cloud resources such as Amazon S3 buckets and EC2 keypairs.

— From the NixOps User's Guide (manual) (https://releases.nixos.org/nixops/latest/manual/manual.html#chap-introduction)

The NixOps User's Guide (https://releases.nixos.org/nixops/latest/manual/manual.html) provides an overview of the functionality and features of NixOps, as well as an up-to-date installation guide. Some other topics covered:

- Deploying to local targets:
  - VirtualBox machine(s)
  - Libvirtd (Qemu) machine(s)
  - Existing NixOS machine(s)
- Deploying to clouds:
  - Amazon EC2, Google Compute Engine, Microsoft Azure (Azure disabled since 2018)
  - Hetzner, DigitalOcean
- Setting up DataDog (https://www.datadoghq.com/) cloud monitoring.

Usage

Internals

See also

- A presentation on NixOps by Kim Lindberger (talyz) - Oslo NixOS MiniCon, March 2020 (https://www.youtube.com/watch?v=SoHtccHNOJ8)
- krops (/wiki/Krops)
- morph (/wiki/Morph)
- nix-deploy (/wiki/Nix-deploy)
- nixus (/index.php?title=Nixus&action=edit&redlink=1)

This article or section needs expansion.
Reason: This article is a stub. (Discuss in Talk:NixOps# (https://nixos.wiki/wiki/Talk:NixOps))
Please consult the pedia article metapage (/wiki/Category:Pedias) for guidelines on contributing.

Categories (/wiki/Special:Categories): Pedias (/wiki/Category:Pedias) | NixOps (/wiki/Category:NixOps) | DevOps (/index.php?title=Category:DevOps&action=edit&redlink=1) | Deployment (/index.php?title=Category:Deployment&action=edit&redlink=1) | Nix (/wiki/Category:Nix) | Server (/wiki/Category:Server) | Incomplete (/index.php?title=Category:Incomplete&action=edit&redlink=1) | Software (/wiki/Category:Software) | Pages or sections flagged with Template:Expansion (/wiki/Category:Pages_or_sections_flagged_with_Template:Expansion)

NixOS virtual machines

Contents

One of the most important features of NixOS is the ability to configure the entire system declaratively, including packages to be installed, services to be run, as well as other settings and options.

NixOS configurations can be used to test and use NixOS using a virtual machine, independent of an installation on a “bare metal” computer.

What will you learn?

This tutorial serves as an introduction creating NixOS virtual machines. Virtual machines are a practical tool for experimenting with or debugging NixOS configurations.

What do you need?

- A working Nix installation on Linux, or NixOS, with a graphical environment
- Basic knowledge of the Nix language

Important

A NixOS configuration is a Nix language function following the NixOS module convention. For a thorough treatment of the module system, check the Module system deep dive tutorial.

Starting from a default NixOS configuration

In this tutorial you will use a default configuration that is shipped with NixOS.

NixOS

On NixOS, use the `nixos-generate-config` command to create a configuration file that contains some useful defaults and configuration suggestions.

Beware that the result of this command depends on your current NixOS configuration. The output of 'nixos-generate-config' can be made reproducible in a `nix-shell` environment. Here we provide a configuration that is used for the NixOS GNOME graphical ISO image:

```
nix-shell -I nixpkgs=channel:nixos-23.11 -p 'let pkgs = import <nixp
```

Detailed explanation

By default, the generated configuration file is written to `/etc/nixos/configuration.nix`. To avoid overwriting this file you have to specify the output directory. Create a NixOS configuration in your working directory:

```
nixos-generate-config --dir ./
```

In the working directory you will then find two files:

1. `hardware-configuration.nix` is specific to the hardware `nix-generate-config` is being run on. You can ignore that file for this tutorial because it has no effect inside a virtual machine.
2. `configuration.nix` contains various suggestions and comments for the initial setup of a desktop computer.

The default NixOS configuration without comments is:

```
{ config, pkgs, ... }:
{
  imports =  [ ./hardware-configuration.nix ];

  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;

```

```
  services.xserver.enable = true;

  services.xserver.displayManager.gdm.enable = true;
  services.xserver.desktopManager.gnome.enable = true;

  system.stateVersion = "23.11";
}
```

To be able to log in, add the following lines to the returned attribute set:

```
  users.users.alice = {
    isNormalUser = true;
    extraGroups = [ "wheel" ];
    packages = with pkgs; [
      firefox
      tree
    ];
  };
```

> **NixOS**
>
> On NixOS your configuration generated with `nixos-generate-config` contains this user configuration commented out.

Additionally, you need to specify a password for this user. For the purpose of demonstration only, you specify an insecure, plain text password by adding the `initialPassword` option to the user configuration:

```
initialPassword = "testpw";
```

> **Warning**
>
> Do not use plain text passwords outside of this example unless you know what you are doing. See `initialHashedPassword` or `ssh.authorizedKeys` for more secure alternatives.

This tutorial focuses on testing NixOS configurations on a virtual machine. Therefore you will remove the reference to `hardware-configuration.nix`:

```
-  imports =  [ ./hardware-configuration.nix ];
```

The complete `configuration.nix` file now looks like this:

```nix
{ config, pkgs, ... }:
{
  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;

  services.xserver.enable = true;

  services.xserver.displayManager.gdm.enable = true;
  services.xserver.desktopManager.gnome.enable = true;

  users.users.alice = {
    isNormalUser = true;
    extraGroups = [ "wheel" ]; # Enable ‘sudo’ for the user.
    packages = with pkgs; [
      firefox
      tree
    ];
    initialPassword = "testpw";
  };

  system.stateVersion = "23.11";
}
```

Creating a QEMU based virtual machine from a NixOS configuration

A NixOS virtual machine is created with the `nix-build` command:

```
nix-build '<nixpkgs/nixos>' -A vm \
-I nixpkgs=channel:nixos-23.11 \
-I nixos-config=./configuration.nix
```

This command builds the attribute `vm` from the `nixos-23.11` release of NixOS, using the NixOS configuration as specified in the relative path.

▸ Detailed explanation

Running the virtual machine

The previous command created a link with the name `result` in the working directory. It links to the directory that contains the virtual machine.

```
ls -R ./result
```

```
    result:
    bin  system

    result/bin:
    run-nixos-vm
```

Run the virtual machine:

```
./result/bin/run-nixos-vm
```

This command opens a QEMU window that shows the boot process of the virtual machine which ends at the GDM login screen. Log in as `alice` with the password `testpw`.

Running the virtual machine will create a `nixos.qcow2` file in the current directory. This disk image file contains the dynamic state of the virtual machine. It can interfere with debugging as it keeps the state of previous runs, for example the user password.

Delete this file when you change the configuration:

```
rm nixos.qcow2
```

References

- NixOS Manual: NixOS Configuration.
- NixOS Manual: Modules.
- NixOS Manual Options reference.
- NixOS Manual: Changing the configuration.
- NixOS source code: `configuration template` in `tools.nix`.

- NixOS source code: `vm` attribute in `default.nix`.
- Nix manual: `nix-build`.
- Nix manual: common command-line options.
- Nix manual: `NIX_PATH` environment variable.

Next steps

- Module system deep dive
- Integration testing with NixOS virtual machines
- Building a bootable ISO image

Overlays

From NixOS Wiki

> Overlays are Nix functions which accept two arguments, conventionally called `final` and `prev` (formerly also `self` and `super`), and return a set of packages. ... Overlays are similar to other methods for customizing Nixpkgs, in particular the packageOverrides ... Indeed, packageOverrides acts as an overlay with only the `prev` (`super`) argument. It is therefore appropriate for basic use, but overlays are more powerful and easier to distribute.
>
> *— From the Nixpkgs manual (https://nixos.org/manual/nixpkgs/stable/#sec-overlays-definition)*

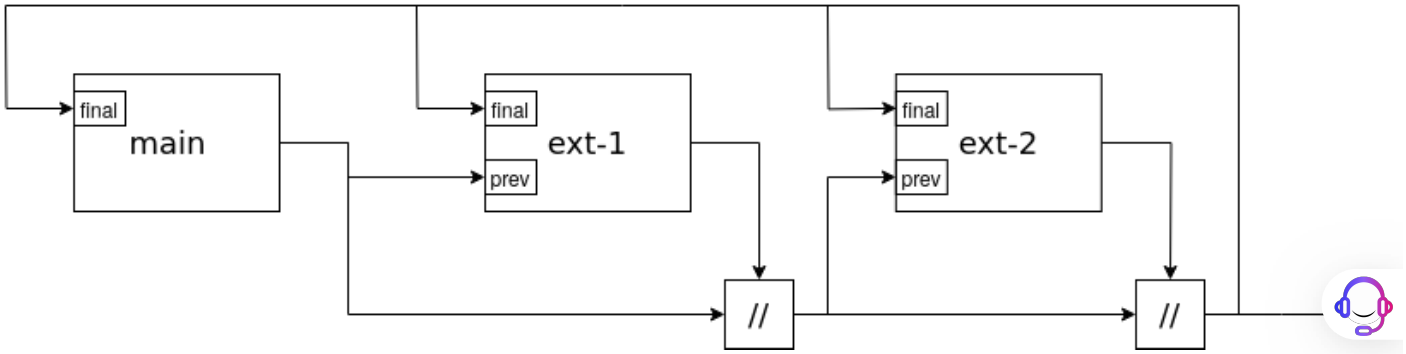
Overlays provide a method to extend and change nixpkgs. They replace constructs like `packageOverride` and `overridePackages`.

Consider a simple example of setting the default proxy in Google Chrome:

```
final: prev: {
   google-chrome = prev.google-chrome.override {
     commandLineArgs =
       "--proxy-server='https=127.0.0.1:3128;http=127.0.0.1:3128'";
   };
};
```

Data flow of overlays

The data flow of overlays, especially regarding `prev` and `final` arguments can be a bit confusing if you are not familiar with how overlays work. This graph shows the data flow:



(/wiki/File:Dram-overlay-final-prev.png)

Here the main package set is extended with two overlays, ext-1 and ext-2. `x // y` is represented by a `//` box with x coming in from the left and y from above.

As you can see, `final` is the same for every stage, but `prev` comes from only the stage before. So when you define an attribute `foo` in the set to override it, within that overlay `final.foo` will be its version, and `prev.foo` will be the non-overriden version. This is why you see patterns like `foo = prev.foo.override { ... }`.

The names `final` and `prev` might remind you of inheritance in object-oriented languages. In fact, overlays are exactly the same thing as subclasses, with regards to overriding and calling methods. This data flow is also how objects know which method to call. This is probably how the two arguments got their names, too.

Data flow of overlays (alternative explanation)

Source: https://discourse.nixos.org/t/how-to-exclude-packages/13039/4 (https://discourse.nixos.org/t/how-to-exclude-packages/13039/4)

I recommend final: prev. That's also easier to explain. The first argument is nixpkgs with your overlay applied, and the second argument is nixpkgs without your overlay. So the “final” nixpkgs and the “previous” nixpkgs. This allows you to access things you defined in your overlay along with things from nixpkgs itself.

```
final: prev: { f = final.firefox; }
```

would work, but

```
final: prev: { f = prev.firefox; }
```

would make more sense.
This could be useful:

```
final: prev: {
  firefox = prev.firefox.override { ... };
  myBrowser = final.firefox;
}
```

And

```
final: prev: firefox = final.firefox.override { ... };
```

would cause infinite recursion.

Using overlays

Applying overlays manually

In standalone nix code

In a shell.nix

When writing standalone nix code, for example a `shell.nix` for a project, one usually starts by importing nixpkgs: `let pkgs = import <nixpkgs> {}` . To use an overlay in this context, replace that by:

```
import <nixpkgs> { overlays = [ overlay1 overlay2 ]; }
```

In a Nix flake

In a Nix flake, nixpkgs will be coming from the inputs. It is common to write something like

```
let pkgs = nixpkgs.legacyPackages.${system}
```

where `system` is a variable containing eg. `"x86_64-linux"`. In order to apply overlays to this, one can do either of:

```
let pkgs = (nixpkgs.legacyPackages.${system}.extend overlay1).extend overlay2
```

or, using the `import` function:

```
let pkgs = import nixpkgs { inherit system; overlays = [ overlay1 overlay2 ]; }
```

In NixOS

In `/etc/nixos/configuration.nix`, use the `nixpkgs.overlays` option:

```
{ config, pkgs, lib, ... }:
{
   # [...]
   nixpkgs.overlays = [ (final: prev: /* overlay goes here */) ];
}
```

Tip

In order to **affect** your system by your nix-language-specific changes you have to **evaluate** it, run (as root):

```
# nixos-rebuild switch
```

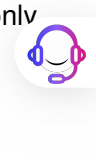
Note that this does not impact the usage of nix on the command line, only your NixOS configuration.

In (/wiki/Home_Manager)Home Manager (/wiki/Home_Manager)

In `~/.config/nixpkgs/home.conf`, use the `nixpkgs.overlays` option:

```
{ config, pkgs, lib, ... }:
{
   # [...]
   nixpkgs.overlays = [ (final: prev: /* overlay goes here */) ];
}
```

Note that this does not impact the usage of nix on the command line or in your your NixOS configuration, only your home-manager configuration.



Applying overlays automatically

On the user level

A list of overlays placed into `~/.config/nixpkgs/overlays.nix` will be automatically loaded by all nix tools run as your user (hence not nixos-rebuild).

Alternatively, you can put each overlay in its own .nix file under your `~/.config/nixpkgs/overlays` directory.

On the system level

If you want your overlays to be accessible by nix tools and also in the system-wide configuration, add `nixpkgs-overlays` to your `NIX_PATH`:

```
NIX_PATH="$NIX_PATH:nixpkgs-overlays=/etc/nixos/overlays"
```

Currently `nixos-rebuild` only works with a `<nixpkgs-overlays>` path that is a directory.

There is a configuration option `nixpkgs.overlays`. Overlays set here will **not** be automatically applied by nix tools.

Using `nixpkgs.overlays` from `configuration.nix` as `<nixpkgs-overlays>` in your NIX_PATH

Configuration below will allow all of the Nix tools to see the exact same overlay as is defined in your `configuration.nix` in the `nixpkgs.overlays (https://search.nixos.org/options?query=nixpkgs.overlays)` option.

The core of the idea here is to point the `nixpkgs-overlays` element of `NIX_PATH` to a "compatibility" overlay, which will load all of the overlays defined in your NixOS system configuration and apply them to its own input. Thus, when various Nix tools attempt to load the overlays from the `nixpkgs-overlays` element of `NIX_PATH`, they will get contents of overlays defined in your NixOS system config.

First, in the `configuration.nix` file, depending on whether your `configuration.nix` already defines `nix.nixPath`, add one of these definitions:

```
{ config, pkgs, options, ... }: {
  # With an existing `nix.nixPath` entry:
  nix.nixPath = [
    # Add the following to existing entries.
    "nixpkgs-overlays=/etc/nixos/overlays-compat/"
  ];

  # Without any `nix.nixPath` entry:
  nix.nixPath =
    # Prepend default nixPath values.
    options.nix.nixPath.default ++
    # Append our nixpkgs-overlays.
    [ "nixpkgs-overlays=/etc/nixos/overlays-compat/" ]
  ;
}
```

Then, add the following contents to `/etc/nixos/overlays-compat/overlays.nix`[1]:

```
final: prev:
with prev.lib;
let
  # Load the system config and get the `nixpkgs.overlays` option
  overlays = (import <nixpkgs/nixos> { }).config.nixpkgs.overlays;
in
  # Apply all overlays to the input of the current "main" overlay
  foldl' (flip extends) (_: prev) overlays final
```

The `/etc/nixos/overlays-compat` directory should contain a single `overlays.nix` file to be understood by the Nix tooling, but the location of this directory can be arbitrary, as long as it is set correctly in the `nix.nixPath` option.

Examples of overlays

Here are a few example usages of overlays.

Overriding a version

Assume you want the original version of sl, not the fork that nixpkgs ships. First, you have to choose the exact revision you want nix to build. Here we will build revision 923e7d7ebc5c1f009755bdeb789ac25658ccce03. The core of the method is to override the attribute `src` of the derivation with an updated value. Here we use `fetchFromGitHub` because sl is hosted on github, but other locations need other functions. To see the original derivation, run `nix edit -f "<nixpkgs>" sl`. This method will fail if the build system changed or new dependencies are required.

```
final: prev:
{
  sl = prev.sl.overrideAttrs (old: {
    src = prev.fetchFromGitHub {
      owner = "mtoyoda";
      repo = "sl";
      rev = "923e7d7ebc5c1f009755bdeb789ac25658ccce03";
      # If you don't know the hash, the first time, set:
      # hash = "";
      # then nix will fail the build with such an error message:
      # hash mismatch in fixed-output derivation '/nix/store/m1ga09c0z1a6n7rj8ky3s31dpg
      # specified: sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
      # got:    sha256-173gxk0ymiw94glyjzjizp8bv8g72gwkjhacigd1an09jshdrjb4
      hash = "173gxk0ymiw94glyjzjizp8bv8g72gwkjhacigd1an09jshdrjb4";
    };
  });
}
```

Adding patches

It is easy to add patches to a nix package:

```
final: prev:
{
  sl = prev.sl.overrideAttrs (old: {
    patches = (old.patches or []) ++ [
      (prev.fetchpatch {
        url = "https://github.com/charlieLehman/sl/commit/e20abbd7e1ee26af53f34451a8f7a
        hash = "07sx98d422589gxr8wflfpkdd0k44kbagxl3b51i56ky2wfix7rc";
      })
      # alternatively if you have a local patch,
      /path/to/file.patch
      # or a relative path (relative to the current nix file)
      ./relative.patch
    ];
  });
}
```

Compilation options

Some packages provide compilation options. Those are not easily disoverable; to find them you need to have a look at the source. For example, with `nix edit -f "<nixpkgs>" pass` one can see that pass can be compiled with or without dependencies on X11 with the `x11Support` argument. Here is how you can remove X11 dependencies:

```
final: prev:
{
  pass = prev.pass.override { x11Support = false; };
}
```

Overriding a package inside a scope

Some packages are not in the top level of nixpkgs but inside a *scope*. For example all GNOME (/wiki/GNOME) packages are in the `gnome` attribute set and Xfce (/wiki/Xfce) packages inside `xfce`. These attributes are often *scopes* and must be overriden specially. Here is an example of patching `gnome.mutter` and `gnome.gnome-control-center`.

```
# elements of nixpkgs must be taken from final and prev
final: prev: {
  # elements of pkgs.gnome must be taken from gfinal and gprev
  gnome = prev.gnome.overrideScope' (gfinal: gprev: {
    mutter = gprev.mutter.overrideAttrs (oldAttrs: {
      patches = oldAttrs.patches ++ [
        # https://salsa.debian.org/gnome-team/mutter/-/blob/ubuntu/master/debian/patche
        (prev.fetchpatch {
          url = "https://salsa.debian.org/gnome-team/mutter/-/raw/91d9bdafd5d624fe1f40f
          hash = "m6PKjVxhGVuzsMBVA82UyJ6Cb1s6SMI0eRooa+F2MY8=";
        })
    ];
    });
    gnome-control-center = gprev.gnome-control-center.overrideAttrs (oldAttrs: {
      patches = oldAttrs.patches ++ [
        # https://salsa.debian.org/gnome-team/gnome-control-center/-/blob/ubuntu/master
        (prev.fetchpatch {
          url = "https://salsa.debian.org/gnome-team/gnome-control-center/-/raw/f185f33
          hash = "XBMD0chaV6GGg3R9/rQnsBejXspomVZz/a4Bvv/AHCA=";
        })
        # https://salsa.debian.org/gnome-team/gnome-control-center/-/blob/ubuntu/master
        (prev.fetchpatch {
          url = "https://salsa.debian.org/gnome-team/gnome-control-center/-/raw/f185f33
          hash = "Pm6PTmsL2bW9JAHD1u0oUEqD1PCIErOlcuqlwvP593I=";
        })
      ];
    });
  });
}
```

Overriding a package inside an extensible attribute set

Here is an example of adding plugins to `vimPlugins`.

```
final: prev: {
  vimPlugins = prev.vimPlugins.extend (final': prev': {
    indent-blankline-nvim-lua = prev.callPackage ../packages/indent-blankline-nvim-lua
  });
}
```

Overrding a package inside a plain attribute set

Here's an example of overriding the source of `obs-studio-plugins.obs-backgroundremoval`.

```
final: prev: {
  obs-studio-plugins = prev.obs-studio-plugins // {
    obs-backgroundremoval =
      prev.obs-studio-plugins.obs-backgroundremoval.overrideAttrs (old: {
        version = "0.5.17";
        src = prev.fetchFromGitHub {
          owner = "royshil";
          repo = "obs-backgroundremoval";
          rev = "v0.5.17";
          hash = "";
        };
      });
  };
};
```

Python Packages Overlay

Here is an example of Python packages overlay. The trick is to also override python itself with `packageOverrides` .

Github issue with the snippet below: [[1] (https://github.com/NixOS/nixpkgs/issues/26487#issuecomment-307363295)]

```
final: prev:
# Within the overlay we use a recursive set, though I think we can use `final` as well.
rec {
  # nix-shell -p python.pkgs.my_stuff
  python = prev.python.override {
    # Careful, we're using a different final and prev here!
    packageOverrides = final: prev: {
      my_stuff = prev.buildPythonPackage rec {
        pname = "pyaes";
        version = "1.6.0";
        src = prev.fetchPypi {
          inherit pname version;
          hash = "0bp9bjqy1n6ij1zb86wz9lqa1dhla8qr1d7w2kxyn7jbj56sbmcw";
        };
      };
    };
  };
  # nix-shell -p pythonPackages.my_stuff
  pythonPackages = python.pkgs;

  # nix-shell -p my_stuff
  my_stuff = pythonPackages.buildPythonPackage rec {
    pname = "pyaes";
    version = "1.6.0";
    src = pythonPackages.fetchPypi {
      inherit pname version;
      hash = "0bp9bjqy1n6ij1zb86wz9lqa1dhla8qr1d7w2kxyn7jbj56sbmcw";
    };
  };
}
```

R Packages Overlay

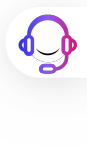
Here is an example of an R packages overlay, in which it can be seen how to provide different versions of packages then those available in the current R version. It should be noted that in the case of R and Python the argument to `override` is named differently. Names of these can be found using `nix repl` and evaluating e.g. `python.override.__functionArgs` .

```
final: prev:

{
  rPackages = prev.rPackages.override {
    overrides = {

      rprojroot = prev.rPackages.buildRPackage rec {
        name = "rprojroot-${version}";
        version = "2.0.2";
        src = prev.fetchurl {
          url =
            "https://github.com/r-lib/rprojroot/archive/refs/tags/v2.0.2.tar.gz";
          hash = "1i0s1f7hla91yw1fdx0rn7c18dp6jwmg2mlww8dix1kk7qbxfjww";
        };
        nativeBuildInputs = [ prev.R ];
      };

      here = prev.rPackages.buildRPackage rec {
        name = "here-${version}";
        version = "1.0.1";
        src = prev.fetchurl {
          url = "https://github.com/r-lib/here/archive/refs/tags/v1.0.1.tar.gz";
          hash = "0ky6sq6n8px3b70s10hy99sccf3vcjjpdhamql5dr7i9igsf8nqy";
        };
        nativeBuildInputs = [ prev.R final.rPackages.rprojroot ];
        propagatedBuildInputs = [ final.rPackages.rprojroot ];
      };
    };
  };
}
```

Rust packages

Due to https://github.com/NixOS/nixpkgs/issues/107070 (https://github.com/NixOS/nixpkgs/issues/107070)

it is not possible to just override `cargoHash` , instead cargoDeps has to be overriden



```
final: prev: {
  rnix-lsp = prev.rnix-lsp.overrideAttrs (oldAttrs: rec {
    version = "master";

    src = prev.fetchFromGitHub {
      owner = "nix-community";
      repo = "rnix-lsp";
      rev = "1fdd7cf9bf56b8ad2dddcfd27354dae8aef2b453";
      hash = "sha256-w0hpyFXxltmOpbBKNQ2tfKRWELQzStc/ho1EcNyYaWc=";
    };

    cargoDeps = oldAttrs.cargoDeps.overrideAttrs (lib.const {
      name = "rnix-lsp-vendor.tar.gz";
      inherit src;
      outputHash = "sha256-6ZaaWYajmgPXQ5sbeRQWzsbaf0Re3F7mTPOU3xqY02g=";
    });
  });
}
```

List of 3rd party overlays

This is an non-exhaustive list:

- Details in the Nixpkgs manual for using Rust overlays (https://nixos.org/manual/nixpkgs/unstable/#using-community-maintained-rust-toolchains)
- Overlay for Radeon Open-Compute packages (https://github.com/peter-sa/nixos-rocm)
- Overlay by Rok Garbas for a set of python packages built by pypi2nix (https://github.com/garbas/nixpkgs-python)

See also

- Overlays in nixpkgs manual (https://nixos.org/nixpkgs/manual/#chap-overlays)
- Blog post "The DOs and DON'Ts of nixpkgs overlays" (https://blog.flyingcircus.io/2017/11/07/nixos-the-dos-and-donts-of-nixpkgs-overlays/)

References

1. Based on (https://gitlab.com/samueldr/nixos-configuration /blob/3febd83b15210282d6435932944d426cd0a9e0ca/modules/overlays-compat /overlays.nix)@samueldr (/wiki/User:Samueldr)'s configuration: overlays-compat (https://gitlab.com /samueldr/nixos-configuration/blob/3febd83b15210282d6435932944d426cd0a9e0ca/modules/overlays-compat/overlays.nix)

Categories (/wiki/Special:Categories): Cookbook (/wiki/Category:Cookbook)
Nixpkgs (/wiki/Category:Nixpkgs)

freedesktop.org

www/ Software/ systemd/ TipsAndTricks

Edit | Page History | Repo Info

Back to systemd

Tips & Tricks

Also check out the Frequently Asked Questions!

Listing running services

```
$ systemctl
UNIT                         LOAD   ACTIVE SUB     JOB DESCRIPTION
accounts-daemon.service      loaded active running     Accounts Service
atd.service                  loaded active running     Job spooling tools
avahi-daemon.service         loaded active running     Avahi mDNS/DNS-SD Stack
bluetooth.service            loaded active running     Bluetooth Manager
colord-sane.service          loaded active running     Daemon for monitoring attached s
colord.service               loaded active running     Manage, Install and Generate Col
crond.service                loaded active running     Command Scheduler
cups.service                 loaded active running     CUPS Printing Service
dbus.service                 loaded active running     D-Bus System Message Bus
...
```

Showing runtime status

```
$ systemctl status udisks2.service
udisks2.service - Storage Daemon
          Loaded: loaded (/usr/lib/systemd/system/udisks2.service; static)
          Active: active (running) since Wed, 27 Jun 2012 20:49:25 +0200; 1 day and 1
        Main PID: 615 (udisksd)
          CGroup: name=systemd:/system/udisks2.service
                  └ 615 /usr/lib/udisks2/udisksd --no-debug

Jun 27 20:49:25 epsilon udisksd[615]: udisks daemon version 1.94.0 starting
Jun 27 20:49:25 epsilon udisksd[615]: Acquired the name org.freedesktop.UDisks2 on th
```

cgroup tree

```
$ systemd-cgls
└ system
  ├ 1 /usr/lib/systemd/systemd --system --deserialize 18
  ├ ntpd.service
  │ └ 8471 /usr/sbin/ntpd -u ntp:ntp -g
  ├ upower.service
  │ └ 798 /usr/libexec/upowerd
```

```
├ wpa_supplicant.service
│ └ 751 /usr/sbin/wpa_supplicant -u -f /var/log/wpa_supplicant.log -c /etc/wpa_supp
├ nfs-idmap.service
│ └ 731 /usr/sbin/rpc.idmapd
├ nfs-rquotad.service
│ └ 753 /usr/sbin/rpc.rquotad
├ nfs-mountd.service
│ └ 732 /usr/sbin/rpc.mountd
├ nfs-lock.service
│ └ 704 /sbin/rpc.statd
├ rpcbind.service
│ └ 680 /sbin/rpcbind -w
├ postfix.service
│ ├  859 /usr/libexec/postfix/master
│ ├  877 qmgr -l -t fifo -u
│ └ 32271 pickup -l -t fifo -u
├ colord-sane.service
│ └ 647 /usr/libexec/colord-sane
├ udisks2.service
│ └ 615 /usr/lib/udisks2/udisksd --no-debug
├ colord.service
│ └ 607 /usr/libexec/colord
├ prefdm.service
│ ├ 567 /usr/sbin/gdm-binary -nodaemon
│ ├ 602 /usr/libexec/gdm-simple-slave --display-id /org/gnome/DisplayManager/Displa
│ ├ 612 /usr/bin/Xorg :0 -br -verbose -auth /var/run/gdm/auth-for-gdm-O00GPA/databa
│ └ 905 gdm-session-worker [pam/gdm-password]
├ systemd-ask-password-wall.service
│ └ 645 /usr/bin/systemd-tty-ask-password-agent --wall
├ atd.service
│ └ 544 /usr/sbin/atd -f
├ ksmtuned.service
│ ├  548 /bin/bash /usr/sbin/ksmtuned
│ └ 1092 sleep 60
├ dbus.service
│ ├ 586 /bin/dbus-daemon --system --address=systemd: --nofork --systemd-activation
│ ├ 601 /usr/libexec/polkit-1/polkitd --no-debug
│ └ 657 /usr/sbin/modem-manager
├ cups.service
│ └ 508 /usr/sbin/cupsd -f
├ avahi-daemon.service
│ ├ 506 avahi-daemon: running [epsilon.local]
│ └ 516 avahi-daemon: chroot helper
├ system-setup-keyboard.service
│ └ 504 /usr/bin/system-setup-keyboard
├ accounts-daemon.service
│ └ 502 /usr/libexec/accounts-daemon
├ systemd-logind.service
│ └ 498 /usr/lib/systemd/systemd-logind
├ crond.service
│ └ 486 /usr/sbin/crond -n
├ NetworkManager.service
│ ├  484 /usr/sbin/NetworkManager --no-daemon
│ └ 8437 /sbin/dhclient -d -4 -sf /usr/libexec/nm-dhcp-client.action -pf /var/run/d
├ libvirtd.service
│ ├ 480 /usr/sbin/libvirtd
│ └ 571 /sbin/dnsmasq --strict-order --bind-interfaces --pid-file=/var/run/libvirt/
├ bluetooth.service
│ └ 479 /usr/sbin/bluetoothd -n
├ systemd-udev.service
│ └ 287 /usr/lib/systemd/systemd-udevd
└ systemd-journald.service
  └ 280 /usr/lib/systemd/systemd-journald
```

ps with cgroups

```
$ alias psc='ps xawf -eo pid,user,cgroup,args'
$ psc
  PID USER     CGROUP                              COMMAND
...
    1 root     name=systemd:/systemd-1             /bin/systemd systemd.log_target=km
  415 root     name=systemd:/systemd-1/sysinit.service /sbin/udevd -d
  928 root     name=systemd:/systemd-1/atd.service /usr/sbin/atd -f
  930 root     name=systemd:/systemd-1/ntpd.service /usr/sbin/ntpd -n
  932 root     name=systemd:/systemd-1/crond.service /usr/sbin/crond -n
  935 root     name=systemd:/systemd-1/auditd.service /sbin/auditd -n
  943 root     name=systemd:/systemd-1/auditd.service  \_ /sbin/audispd
  964 root     name=systemd:/systemd-1/auditd.service      \_ /usr/sbin/sedispatch
  937 root     name=systemd:/systemd-1/acpid.service /usr/sbin/acpid -f
  941 rpc      name=systemd:/systemd-1/rpcbind.service /sbin/rpcbind -f
  944 root     name=systemd:/systemd-1/rsyslog.service /sbin/rsyslogd -n -c 4
  947 root     name=systemd:/systemd-1/systemd-logger.service /lib/systemd/systemd-lo
  950 root     name=systemd:/systemd-1/cups.service /usr/sbin/cupsd -f
  955 dbus     name=systemd:/systemd-1/messagebus.service /bin/dbus-daemon --system -
  969 root     name=systemd:/systemd-1/getty@.service/tty6 /sbin/mingetty tty6
  970 root     name=systemd:/systemd-1/getty@.service/tty5 /sbin/mingetty tty5
  971 root     name=systemd:/systemd-1/getty@.service/tty1 /sbin/mingetty tty1
  973 root     name=systemd:/systemd-1/getty@.service/tty4 /sbin/mingetty tty4
  974 root     name=systemd:/user/lennart/2        login -- lennart
 1824 lennart  name=systemd:/user/lennart/2         \_ -bash
  975 root     name=systemd:/systemd-1/getty@.service/tty3 /sbin/mingetty tty3
  988 root     name=systemd:/systemd-1/polkitd.service /usr/libexec/polkit-1/polkitd
  994 rtkit    name=systemd:/systemd-1/rtkit-daemon.service /usr/libexec/rtkit-daemon
...
```

Changing the Default Boot Target

```
$ ln -sf /usr/lib/systemd/system/multi-user.target /etc/systemd/system/default.target
```

This line makes the multi user target (i.e. full system, but no graphical UI) the default target to boot into. This is kinda equivalent to setting runlevel 3 as the default runlevel on Fedora/sysvinit systems.

```
$ ln -sf /usr/lib/systemd/system/graphical.target /etc/systemd/system/default.target
```

This line makes the graphical target (i.e. full system, including graphical UI) the default target to boot into. Kinda equivalent to runlevel 5 on fedora/sysvinit systems. This is how things are shipped by default.

What other units does a unit depend on?

For example, if you want to figure out which services a target like multi-user.target pulls in, use something like this:

```
$ systemctl show -p "Wants" multi-user.target
Wants=rc-local.service avahi-daemon.service rpcbind.service NetworkManager.service ac
```

Instead of "Wants" you might also try "WantedBy", "Requires", "RequiredBy",

"Conflicts", "ConflictedBy", "Before", "After" for the respective types of dependencies and their inverse.

What would get started if I booted into a specific target?

If you want systemd to calculate the "initial" transaction it would execute on boot, try something like this:

```
$ systemd --test --system --unit=foobar.target
```

for a boot target foobar.target. Note that this is mostly a debugging tool that actually does a lot more than just calculate the initial transaction, so don't build scripts based on this.

Last edited Fri May 7 01:22:37 2021