nix.dev

nix.dev contributors

Jan 01, 2025

CONTENTS

CHAPTER

ONE

INSTALL NIX

Requirements:

- Prior to installation, you might need to first install `xz-utils` or similar for uncompressing the Nix binary tarball (`.tar.xz`) that will be downloaded via the scripts below.

Linux

Install Nix via the recommended multi-user installation[3]:

```
$ curl -L https://nixos.org/nix/install | sh -s -- --daemon
```

On Arch Linux, you can alternatively install Nix through `pacman`[4].

macOS

Install Nix via the recommended multi-user installation[5]:

```
$ curl -L https://nixos.org/nix/install | sh
```

Important

Updating to macOS 15 Sequoia

If you recently updated to macOS 15 Sequoia and are getting

```
error: the user '_nixbld1' in the group 'nixbld' does not exist
```

when running Nix commands, refer to GitHub issue NixOS/nix#10892[6] for instructions to fix your installation without reinstalling.

Windows (WSL2)

Install Nix via the recommended single-user installation[7]:

```
$ curl -L https://nixos.org/nix/install | sh -s -- --no-daemon
```

However, if you have systemd support[8] enabled, install Nix via the recommended multi-user installation[9]:

```
$ curl -L https://nixos.org/nix/install | sh -s -- --daemon
```

[3] https://nix.dev/manual/nix/stable/installation/multi-user.html
[4] https://wiki.archlinux.org/title/Nix#Installation
[5] https://nix.dev/manual/nix/stable/installation/multi-user.html
[6] https://github.com/NixOS/nix/issues/10892
[7] https://nix.dev/manual/nix/stable/installation/single-user.html
[8] https://learn.microsoft.com/en-us/windows/wsl/wsl-config#systemd-support
[9] https://nix.dev/manual/nix/stable/installation/multi-user.html

Docker

Start a Docker shell with Nix:

```
$ docker run -it nixos/nix
```

Or start a Docker shell with Nix exposing a `workdir` directory:

```
$ mkdir workdir
$ docker run -it -v $(pwd)/workdir:/workdir nixos/nix
```

The `workdir` example from above can be also used to start hacking on Nixpkgs:

```
$ git clone git@github.com:NixOS/nixpkgs
$ docker run -it -v $(pwd)/nixpkgs:/nixpkgs nixos/nix
bash-5.1# nix-build -I nixpkgs=/nixpkgs -A hello
bash-5.1# find ./result # this symlink points to the build package
```

1.1 Verify installation

Check the installation by opening **a new terminal** and typing:

```
$ nix --version
nix (Nix) 2.11.0
```

CHAPTER

TWO

TUTORIALS

These sections contains series of lessons to get started.

2.1 First steps

This tutorial series is where you should start learning Nix.

In these lessons, you will use basic Nix commands to obtain almost any piece of software, create development environments on the fly, and learn how to make reproducible scripts. You will also learn reading the Nix language, and later use it to build portable, reproducible development environments.

2.1.1 Ad hoc shell environments

In a Nix shell environment, you can immediately use any program packaged with Nix, without installing it permanently.

You can also share the command invoking such a shell with others, and it will work on all Linux distributions, WSL, and macOS[1].

Create a shell environment

Once you *install Nix* (page 1), you can use it to create new *shell environments* with programs that you want to use.

In this section you will run two exotic programs called `cowsay` and `lolcat` that you will probably not have installed on your machine:

```
$ cowsay no can do
The program 'cowsay' is currently not installed.

$ echo no chance | lolcat
The program 'lolcat' is currently not installed.
```

Use `nix-shell` with the `-p` (`--packages`) option to specify that we need the `cowsay` and `lolcat` packages:

Note

The first invocation of `nix-shell` for these packages may take a while to download all dependencies.

```
$ nix-shell -p cowsay lolcat
these 3 derivations will be built:
  /nix/store/zx1j8gchgwzfjn7sr4r8yxb7a0afkjdg-builder.pl.drv
  /nix/store/h9sbaa2k8ivnihw2czhl5b58k0f7fsfh-lolcat-100.0.1.drv
  ...
```

(continues on next page)

[1] Not all packages are supported for both Linux and macOS. Especially support for graphical programs may vary.

(continued from previous page)

```
[nix-shell:~]$
```

Within the Nix shell, you can use the programs provided by these packages:

```
[nix-shell:~]$ cowsay Hello, Nix! | lolcat
```

Type `exit` or press `CTRL-D` to exit the shell, and the programs won't be available anymore.

```
[nix-shell:~]$ exit
exit

$ cowsay no more
The program 'cowsay' is currently not installed.

$ echo all gone | lolcat
The program 'lolcat' is currently not installed.
```

Running programs once

You can go even faster, by running any program directly:

```
$ nix-shell -p cowsay --run "cowsay Nix"
```

If the command consists only of the program name, no quotes are needed:

```
$ nix-shell -p hello --run hello
```

Search for packages

What can you put in a shell environment? If you can think of it, there's probably a Nix package of it.

Tip

Enter the program name you want to run in search.nixos.org[10] to find packages that provide it.

For the following example, find the package names for these programs:

- `git`
- `nvim`
- `npm`

In the search results, each item shows the package name, and the details list the available programs.[2]

Run any combination of programs

Once you have the package name, you can start a shell with that package. The `-p` (`--packages`) argument can take multiple package names.

Start a Nix shell with the packages providing `git`, `nvim`, and `npm`. Again, the first invocation may take a while to download all dependencies.

[10] https://search.nixos.org/packages

[2] A package name is not the same as a program name. Many packages provide multiple programs, or no programs at all if they are libraries. Even for packages that provide exactly one program, the package and program name are not necessarily the same.

```
$ nix-shell -p git neovim nodejs
these 9 derivations will be built:
  /nix/store/7gz8jyn99kw4k74bgm4qp6z487l5ap06-packdir-start.drv
  /nix/store/d6fkgxc3b04m85wrhg6j0l5y0ray82l7-packdir-opt.drv
  /nix/store/da6njv7r0zzc2n54n2j54g2a5sbi4a5i-manifest.vim.drv
  /nix/store/zs4jb2ybr4rcyzwq0dagg9rlhlc368h6-builder.pl.drv
  /nix/store/g8sl2xnsshfrz9f39ki94k8p15vp3xd7-vim-pack-dir.drv
  /nix/store/jmxkg8b1psk52awsvfziy9nq6dwmxmjp-luajit-2.1.0-2022-10-04-env.drv
  /nix/store/kn83q8yk6ds74zgyklrjhvv5wkv5wmch-python3-3.10.9-env.drv
  /nix/store/m445wn3vizcgg7syna2cdkkws3kk1gq8-neovim-ruby-env.drv
  /nix/store/r2wa882mw99c311a4my7hcis9lq3kp3v-neovim-0.8.1.drv
these 151 paths will be fetched (186.43 MiB download, 1018.20 MiB unpacked):
  /nix/store/046zxlxhq4srm3ggafkymx794bn1jksc-bzip2-1.0.8
  /nix/store/0p1jxcb7b4p8jhhlf8qnjc4cqwy89460-unibilium-2.1.1
  /nix/store/0q4fpnqmg8liqraj7zidylcyd062f6z0-perl5.36.0-URI-5.05
  ...

[nix-shell:~]$
```

Check package versions

Check that you have indeed the specific version of these programs provided by Nix, even if you had any of them already installed on your machine.

```
[nix-shell:~]$ which git
/nix/store/3cdi52xh6lk3h1fb51jkxs3p561p37wg-git-2.38.3/bin/git

[nix-shell:~]$ git --version
git version 2.38.3

[nix-shell:~]$ which nvim
/nix/store/ynskzgkf07lmrrs3cl2kzr9ah487lwab-neovim-0.8.1/bin/nvim

[nix-shell:~]$ nvim --version | head -1
NVIM v0.8.1

[nix-shell:~]$ which npm
/nix/store/q12w83z0i5pi1y0m6am7qmw1r73228sh-nodejs-18.12.1/bin/npm

[nix-shell:~]$ npm --version
8.19.2
```

Nested shell sessions

If you need an additional program temporarily, you can run a nested Nix shell. The programs provided by the specified packages will be added to the current environment.

```
[nix-shell:~]$ nix-shell -p python3
this path will be fetched (11.42 MiB download, 62.64 MiB unpacked):
  /nix/store/pwy30a7siqrkki9r7xd1lksyv9fg4l1r-python3-3.10.11
copying path '/nix/store/pwy30a7siqrkki9r7xd1lksyv9fg4l1r-python3-3.10.11' from
↪'https://cache.nixos.org'...

[nix-shell:~]$ python --version
Python 3.10.11
```

Exit the shell as usual to return to the previous environment.

Towards reproducibility

These shell environments are very convenient, but the examples so far are not reproducible yet. Running these commands on another machine may fetch different versions of packages, depending on when Nix was installed there.

What do we mean by reproducible? A fully reproducible example would give exactly the same results no matter when or where you run the command. The environment provided would be identical each time.

The following example creates a fully reproducible environment. You can run it anywhere, anytime to obtain the exact same version of the `git`.

```
$ nix-shell -p git --run "git --version" --pure -I nixpkgs=https://github.com/
↪NixOS/nixpkgs/tarball/2a601aafdc5605a5133a2ca506a34a3a73377247
...
git version 2.33.1
```

There are three things going on here:

1. `--run` executes the given Bash command[11] within the environment created by Nix, and exits when it's done.

   You can use this with `nix-shell` whenever you want to quickly run a program you don't have installed on your machine.

2. `--pure` discards most environment variables set on your system when running the shell.

   It means that only the `git` provided by Nix is available inside that shell. This is useful for simple one-liners such as in the example. While developing, however, you will usually want to have your editor and other tools around. Therefore we recommend to omit `--pure` for development environments, and to add it only when the extra isolation is needed.

3. `-I` determines what to use as a source of package declarations.

   Here we provided a specific Git revision of `nixpkgs`[12], leaving no doubt about which version of the packages in that collection will be used.

References

- Nix manual: `nix-shell`[13] (or run `man nix-shell`)
- Nix manual: `-I` option[14]

Next steps

- *Reproducible interpreted scripts* (page 7) – use Nix for reproducible scripts
- *Nix language basics* (page 12) – learn reading the Nix language, which is used to declare packages and configurations
- *Declarative shell environments with shell.nix* (page 8) – create reproducible shell environments with a declarative configuration file
- *Towards reproducibility: pinning Nixpkgs* (page 11) – learn different ways of specifying exact versions of package sources

If you're done trying out Nix for now, you may want to free up some disk space occupied by the different versions of programs you downloaded by running the examples:

```
$ nix-collect-garbage
```

[11] https://www.gnu.org/software/bash/manual/bash.html#Shell-Commands
[12] https://github.com/NixOS/nixpkgs/tree/2a601aafdc5605a5133a2ca506a34a3a73377247
[13] https://nix.dev/manual/nix/stable/command-ref/nix-shell
[14] https://nix.dev/manual/nix/stable/command-ref/opt-common.html#opt-I

2.1.2 Reproducible interpreted scripts

In this tutorial, you will learn how to use Nix to create and run reproducible interpreted scripts, also known as shebang[15] scripts.

Requirements

- A working *Nix installation* (page 1)
- Familiarity with Bash[16]

A trivial script with non-trivial dependencies

Take the following script, which fetches the content XML of a URL, converts it to JSON, and formats it for better readability:

```
#! /bin/bash

curl https://github.com/NixOS/nixpkgs/releases.atom | xml2json | jq .
```

It requires the programs `curl`, `xml2json`, and `jq`. It also requires the `bash` interpreter. If any of these dependencies are not present on the system running the script, it will fail partially or altogether.

With Nix, we can declare all dependencies explicitly, and produce a script that will always run on any machine that supports Nix and the required packages taken from Nixpkgs.

The script

A shebang[17] determines which program to use for running an interpreted script.

We will use the shebang line `#!/usr/bin/env nix-shell`.

`env`[18] is a program available on most modern Unix-like operating systems at the file system path `/usr/bin/env`. It takes a command name as argument and will run the first executable by that name it finds in the directories listed in the environment variable `$PATH`.

We use `nix-shell` as a shebang interpreter[19]. It takes the following parameters relevant for our use case:

- `-i` tells which program to use for interpreting the rest of the file
- `--pure` excludes most environment variables when the script is run
- `-p` lists packages that should be present in the interpreter's environment
- `-I` explicitly sets the search path[20] for packages

More details on the options can be found in the `nix-shell` reference documentation[21].

Create a file named `nixpkgs-releases.sh` with the following content:

```
#!/usr/bin/env nix-shell
#! nix-shell -i bash --pure
#! nix-shell -p bash cacert curl jq python3Packages.xmljson
#! nix-shell -I nixpkgs=https://github.com/NixOS/nixpkgs/archive/
↪2a601aafdc5605a5133a2ca506a34a3a73377247.tar.gz

curl https://github.com/NixOS/nixpkgs/releases.atom | xml2json | jq .
```

The first line is a standard shebang. The additional shebang lines are a Nix-specific construct:

[15] https://en.wikipedia.org/wiki/Shebang_(Unix)
[16] https://www.gnu.org/software/bash/
[17] https://en.wikipedia.org/wiki/Shebang_(Unix)
[18] https://pubs.opengroup.org/onlinepubs/9699919799/utilities/env.html
[19] https://nix.dev/manual/nix/stable/command-ref/nix-shell.html#use-as-a--interpreter
[20] https://nix.dev/manual/nix/stable/command-ref/opt-common.html#opt-I
[21] https://nix.dev/manual/nix/stable/command-ref/nix-shell.html#options

- With the `-i` option, `bash` is specified as the interpreter for the rest of the file.
- In this case, the `--pure` option is enabled to prevent the script from implicitly using programs that may already exist on the system on which the script is run.
- The `-p` option lists the packages required for the script to run.

  The command `xml2json` is provided by the package `python3Packages.xmljson`, while `bash`, `jq`, and `curl` are provided by packages of the same name. `cacert` must be present for SSL authentication to work.

  Tip

  Use search.nixos.org[22] to find packages providing the program you need.

- The parameter of `-I` refers to a specific Git commit of the Nixpkgs repository.

  This ensures that the script will always run with the exact same packages versions, everywhere.

Make the script executable:

```
chmod +x nixpkgs-releases.sh
```

Run the script:

```
./nixpkgs-releases.sh
```

Next steps

- *Nix language basics* (page 12) to learn about the Nix language, which is used to declare packages and configurations.
- *Declarative shell environments with shell.nix* (page 8) to create reproducible shell environments with a declarative configuration file.
- Garbage Collection[23] – free up storage used by the programs made available through Nix

2.1.3 Declarative shell environments with `shell.nix`

Overview

Declarative shell environments allow you to:

- Automatically run bash commands during environment activation
- Automatically set environment variables
- Put the environment definition under version control and reproduce it on other machines

What will you learn?

In the *Ad hoc shell environments* (page 3) tutorial, you learned how to imperatively create shell environments using `nix-shell -p`. This is great when you want to quickly access tools without installing them permanently. You also learned how to execute that command with a specific Nixpkgs revision using a Git commit as an argument, to recreate the same environment used previously.

In this tutorial we'll take a look at how to create reproducible shell environments with a declarative configuration in a *Nix file*. This file can be shared with anyone to recreate the same environment on a different machine.

[22] https://search.nixos.org/packages
[23] https://nix.dev/manual/nix/stable/package-management/garbage-collection.html

How long will it take?

30 minutes

What do you need?

- Familiarity with the Unix shell
- A rudimentary understanding of the *Nix language* (page 12)

Entering a temporary shell

Suppose we want an environment where `cowsay` and `lolcat` are available. The simplest possible way to accomplish this is via the `nix-shell -p` command:

```
$ nix-shell -p cowsay lolcat
```

This command works, but there's a number of drawbacks:

- You have to type out `-p cowsay lolcat` every time you enter the shell.
- It doesn't (ergonomically) allow you any further customization of your shell environment.

A better solution is to create our shell environment from a `shell.nix` file.

A basic `shell.nix` file

Create a file called `shell.nix` with these contents:

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in

pkgs.mkShellNoCC {
  packages = with pkgs; [
    cowsay
    lolcat
  ];
}
```

Detailed explanation

We use a version of *Nixpkgs pinned to a release branch* (page 156). If you followed the *Ad hoc shell environments* (page 3) tutorial and don't want to download all dependencies again, specify the exact same revision as in the section *Towards reproducibility* (page 6):

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/
↪2a601aafdc5605a5133a2ca506a34a3a73377247";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
```

We explicitly set `config` and `overlays` to avoid them being inadvertently overridden by global configuration[24].

`mkShellNoCC` is a function that takes as argument an attribute set. Here we give it an attribute `packages` with a list containing two items from the `pkgs` attribute set.

[24] https://nixos.org/manual/nixpkgs/stable/#chap-packageconfig

Side note on `mkShell`

`nix-shell` and `mkShell` were originally conceived as a way to construct a shell environment containing the tools needed to debug package builds[25], such as Make or GCC. Only later it became widely used as a general way to make temporary environments for other purposes. `mkShellNoCC` is a function that produces such an environment, but without a compiler toolchain.

You may encounter examples of `mkShell` or `mkShellNoCC` that add packages to the `buildInputs` or `nativeBuildInputs` attributes instead. `mkShellNoCC` is a wrapper around `mkDerivation`[26], so it takes the same arguments as `mkDerivation`, such as `buildInputs` or `nativeBuildInputs`. The `packages` attribute argument to `mkShellNoCC` is simply an alias for `nativeBuildInputs`.

Enter the environment by running `nix-shell` in the same directory as `shell.nix`:

Note

The first invocation of `nix-shell` on this file may take a while to download all dependencies.

```
$ nix-shell
[nix-shell]$ cowsay hello | lolcat
```

`nix-shell` by default looks for a file called `shell.nix` in the current directory and builds a shell environment from the Nix expression in this file. Packages defined in the `packages` attribute will be available in `$PATH`.

Environment variables

You may want to automatically export certain environment variables when you enter a shell environment.

Set `GREETING` so it can be used in the shell environment:

```
 let
   nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
   pkgs = import nixpkgs { config = {}; overlays = []; };
 in

 pkgs.mkShellNoCC {
   packages = with pkgs; [
     cowsay
     lolcat
   ];

+  GREETING = "Hello, Nix!";
 }
```

Any attribute name passed to `mkShellNoCC` that is not reserved otherwise and has a value which can be coerced to a string will end up as an environment variable.

Try it out! Exit the shell by typing `exit` or pressing `Ctrl+D`, then start it again with `nix-shell`.

```
[nix-shell]$ echo $GREETING
```

Warning

Some variables are protected from being set as described above.

For example, the shell prompt format for most shells is set by the `PS1` environment variable, but `nix-shell` already sets this by default, and will ignore a `PS1` attribute set in the argument.

[25] https://nixos.org/manual/nixpkgs/stable/#sec-tools-of-stdenv
[26] https://nixos.org/manual/nixpkgs/stable/#sec-pkgs-mkShell

If you need to override these protected environment variables, use the `shellHook` attribute as described in the next section.

Startup commands

You may want to run some commands before entering the shell environment. These commands can be placed in the `shellHook` attribute provided to `mkShellNoCC`.

Set `shellHook` to output a colorful greeting:

```
 let
   nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
   pkgs = import nixpkgs { config = {}; overlays = []; };
 in

 pkgs.mkShellNoCC {
   packages = with pkgs; [
     cowsay
     lolcat
   ];

   GREETING = "Hello, Nix!";
+
+  shellHook = ''
+    echo $GREETING | cowsay | lolcat
+  '';
 }
```

Try it again! Exit the shell by typing `exit` or pressing `Ctrl+D`, then start it again with `nix-shell` to observe the effect.

References

- `mkShell` documentation[27]
- Nixpkgs shell functions and utilities[28] documentation
- `nix-shell` documentation[29]

Next steps

- *Nix language basics* (page 12)
- *Automatic environment activation with direnv* (page 134)
- *Dependencies in the development shell* (page 135)
- *Automatically managing remote sources with npins* (page 136)

2.1.4 Towards reproducibility: pinning Nixpkgs

In various Nix examples, you'll often see the following:

```
1 { pkgs ? import <nixpkgs> {} }:
2
3 ...
```

[27] https://nixos.org/manual/nixpkgs/stable/#sec-pkgs-mkShell
[28] https://nixos.org/manual/nixpkgs/stable/#ssec-stdenv-functions
[29] https://nix.dev/manual/nix/stable/command-ref/nix-shell

Note

`<nixpkgs>` points to the file system path of some revision of *Nixpkgs*. Find more information on *lookup paths* (page 25) in *Nix language basics* (page 12).

This is a **convenient** way to quickly demonstrate a Nix expression and get it working by importing Nix packages.

However, **the resulting Nix expression is not fully reproducible**.

Pinning packages with URLs inside a Nix expression

To create **fully reproducible** Nix expressions, we can pin an exact version of Nixpkgs.

The simplest way to do this is to fetch the required Nixpkgs version as a tarball specified via the relevant Git commit hash:

```
1 { pkgs ? import (fetchTarball "https://github.com/NixOS/nixpkgs/archive/
  ↪06278c77b5d162e62df170fec307e83f1812d94b.tar.gz") {}
2 }:
3
4 ...
```

Picking the commit can be done via status.nixos.org[30], which lists all the releases and the latest commit that has passed all tests.

When choosing a commit, it is recommended to follow either

- the **latest stable NixOS** release by using a specific version, such as `nixos-21.05`, **or**
- the latest **unstable release** via `nixos-unstable`.

Next steps

- For more examples and details of the different ways to pin `nixpkgs`, see *Pinning Nixpkgs* (page 156).
- *Automatically managing remote sources with npins* (page 136)

2.2 Nix language basics

The Nix language is designed for conveniently creating and composing *derivations* – precise descriptions of how contents of existing files are used to derive new files. It is a domain-specific, purely functional, lazily evaluated, dynamically typed programming language.

Notable uses of the Nix language

- *Nixpkgs*

  The largest, most up-to-date software distribution in the world, and written in the Nix language.

- *NixOS*

  A Linux distribution that can be configured fully declaratively and is based on Nix and Nixpkgs.

  Its underlying modular configuration system is written in the Nix language, and uses packages from Nixpkgs. The operating system environment and services it provides are configured with the Nix language.

You may quickly encounter Nix language expressions that look very complicated. As with any programming language, the required amount of Nix language code closely matches the complexity of the problem it is supposed to solve, and reflects how well the problem – and its solution – is understood. Building software is a complex undertaking, and Nix both *exposes* and *allows managing* this complexity with the Nix language.

[30] https://status.nixos.org/

Yet, the Nix language itself has only few basic concepts that will be introduced in this tutorial, and which can be combined arbitrarily. What may look complicated comes not from the language, but from how it is used.

2.2.1 Overview

This is an introduction to **reading the Nix language**, for the purpose of following other tutorials and examples.

Using the Nix language in practice entails multiple things:

- Language: syntax and semantics
- Libraries: `builtins` and `pkgs.lib`
- Developer tools: testing, debugging, linting, formatting, …
- Generic build mechanisms: `stdenv.mkDerivation`, trivial builders, …
- Composition and configuration mechanisms: `override`, `overrideAttrs`, overlays, `callPackage`, …
- Ecosystem-specific packaging mechanisms: `buildGoModule`, `buildPythonApplication`, …
- NixOS module system: `config`, `option`, …

This tutorial only covers the most important language features, briefly discusses libraries, and at the end will direct you to reference material and resources on the other components.

What will you learn?

This tutorial should enable you to read typical Nix language code and understand its structure. Its goal is to highlight where the Nix language may differ from languages you are used to.

It therefore shows the most common and distinguishing patterns in the Nix language:

- *Assigning names and accessing values* (page 16)
- Declaring and calling *functions* (page 26)
- *Built-in and library functions* (page 31)
- *Impurities* (page 34) to obtain build inputs
- *Derivations* (page 36) that describe build tasks

Important

This tutorial *does not* explain all Nix language features in detail and *does not* go into specifics of syntactical rules. For instance, we skip over commonplace constructs such as `if ... then ... else ...`.

See the Nix manual[31] for a full language reference.

What do you need?

- Familiarity with software development
- Familiarity with Unix shell, to read command line examples
- A *Nix installation* (page 1) to run the examples

How long does it take?

- No experience with functional programming: 2 hours
- Familiar with functional programming: 1 hour
- Proficient with functional programming: 30 minutes

[31] https://nix.dev/manual/nix/stable/language/index.html

We recommend to run all examples. Play with them to validate your assumptions and test what you have learned. Read detailed explanations if you want to make sure you fully understand the examples.

How to run the examples?

- A piece of Nix language code is a *Nix expression*.
- Evaluating a Nix expression produces a *Nix value*.
- The content of a *Nix file* (file extension `.nix`) is a Nix expression.

Note

To *evaluate* means to transform an expression into a value according to the language rules.

This tutorial contains many examples of Nix expressions. Each one is followed by the expected evaluation result.

The following example is a Nix expression adding two numbers:

```
1 + 2
```

```
3
```

Interactive evaluation

Use `nix repl`[32] to evaluate Nix expressions interactively (by typing them on the command line):

```
$ nix repl
Welcome to Nix 2.13.3. Type :? for help.

nix-repl> 1 + 2
3
```

Note

The Nix language uses lazy evaluation, and `nix repl` by default only computes values when needed.

Some examples show a fully evaluated data structure for clarity. If your output does not match the example, try prepending `:p` to the input expression.

Example:

```
nix-repl> { a.b.c = 1; }
{ a = { ... }; }

nix-repl> :p { a.b.c = 1; }
{ a = { b = { c = 1; }; }; }
```

Type `:q` to exit `nix repl`[33].

Evaluating Nix files

Use `nix-instantiate --eval`[34] to evaluate the expression in a Nix file.

[32] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-repl.html
[33] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-repl.html
[34] https://nix.dev/manual/nix/stable/command-ref/nix-instantiate.html

```
$ echo 1 + 2 > file.nix
$ nix-instantiate --eval file.nix
3
```

Detailed explanation

The first command writes `1 + 2` to a file `file.nix` in the current directory. The contents of `file.nix` are now `1 + 2`, which you can check with

```
$ cat file.nix
1 + 2
```

The second command runs `nix-instantiate` with the `--eval` option on `file.nix`, which reads the file and evaluates the contained Nix expression. The resulting value is printed as output.

`--eval` is required to evaluate the file and do nothing else. If `--eval` is omitted, `nix-instantiate` expects the expression in the given file to evaluate to a special value called a *derivation*, which we will come back to at the end of this tutorial in *Derivations* (page 36).

Note

`nix-instantiate --eval` will try to read from `default.nix` if no file name is specified.

```
$ echo 1 + 2 > default.nix
$ nix-instantiate --eval
3
```

Note

The Nix language uses lazy evaluation, and `nix-instantiate` by default only computes values when needed.

Some examples show a fully evaluated data structure for clarity. If your output does not match the example, try adding the `--strict` option to `nix-instantiate`.

Example:

```
$ echo "{ a.b.c = 1; }" > file.nix
$ nix-instantiate --eval file.nix
{ a = <CODE>; }
```

```
$ echo "{ a.b.c = 1; }" > file.nix
$ nix-instantiate --eval --strict file.nix
{ a = { b = { c = 1; }; }; }
```

Notes on whitespace

White space is used to delimit lexical tokens[35], where required. It is otherwise insignificant.

Line breaks, indentation, and additional spaces are for readers' convenience.

The following are equivalent:

```
let
  x = 1;
  y = 2;
in x + y
```

[35] https://en.wikipedia.org/wiki/Lexical_analysis#Lexical_token_and_lexical_tokenization

```
3
```

```
let x=1;y=2;in x+y
```

```
3
```

2.2.2 Names and values

Values in the Nix language can be primitive data types, lists, attribute sets, and functions.

We show examples of primitive data types and lists in the context of *attribute sets* (page 16). Later in this section we cover special features of character strings: *string interpolation* (page 22), *file system paths* (page 24), and *indented strings* (page 24). We deal with *functions* (page 26) separately.

Attribute sets (page 16) and `let` *expressions* (page 18) are used to assign names to values. Assignments are denoted by a single equal sign (=).

Whenever you encounter an equal sign (=) in Nix language code:

- On its left is the assigned name.
- On its right is the value, delimited by a semicolon (`;`).

Attribute set `{ ... }`

An attribute set is a collection of name-value-pairs, where names must be unique.

The following example shows all primitive data types, lists, and attribute sets.

> **Note**
>
> If you are familiar with JSON, imagine the Nix language as *JSON with functions*.
>
> Nix language data types *without functions* work just like their counterparts in JSON and look very similar.

Nix

```
{
  string = "hello";
  integer = 1;
  float = 3.141;
  bool = true;
  null = null;
  list = [ 1 "two" false ];
  attribute-set = {
    a = "hello";
    b = 2;
    c = 2.718;
    d = false;
  }; # comments are supported
}
```

JSON

```
{
  "string": "hello",
  "integer": 1,
  "float": 3.141,
  "bool": true,
```

(continues on next page)

(continued from previous page)

```
  "null": null,
  "list": [1, "two", false],
  "object": {
    "a": "hello",
    "b": 1,
    "c": 2.718,
    "d": false
  }
}
```

Note

- Attribute names usually do not need quotes.[66]
- List elements are separated by white space.[68]

Recursive attribute set `rec { ... }`

You will sometimes see attribute sets declared with `rec` prepended. This allows access to attributes from within the set.

Example:

```
rec {
  one = 1;
  two = one + 1;
  three = two + 1;
}
```

```
{ one = 1; three = 3; two = 2; }
```

Note

Elements in an attribute set can be declared in any order, and are ordered on evaluation.

Counter-example:

```
{
  one = 1;
  two = one + 1;
  three = two + 1;
}
```

```
error: undefined variable 'one'

       at «string»:3:9:

            2|   one = 1;
            3|   two = one + 1;
```

(continues on next page)

[66] Details: Nix manual - attribute set[67]
[67] https://nix.dev/manual/nix/stable/language/syntax#attrs-literal
[68] Details: Nix manual - list[69]
[69] https://nix.dev/manual/nix/stable/language/syntax#list-literal

(continued from previous page)

```
             |          ^
            4|   three = two + 1;
```

`let ... in ...`

Also known as "`let` expression" or "`let` binding"

`let` expressions allow assigning names to values for repeated use.

Example:

```
let
  a = 1;
in
a + a
```

```
2
```

Detailed explanation

Assignments are placed between the keywords `let` and `in`. In this example we assign `a = 1`.

After `in` comes the expression in which the assignments are valid, i.e., where assigned names can be used. In this example the expression is `a + a`, where `a` refers to `a = 1`.

By replacing the names with their assigned values, `a + a` evaluates to `2`.

Names can be assigned in any order, and expressions on the right of the assignment (=) can refer to other assigned names.

Example:

```
let
  b = a + 1;
  a = 1;
in
a + b
```

```
3
```

Detailed explanation

Assignments are placed between the keywords `let` and `in`. In this example we assign `a = 1` and `b = a + 1`.

The order of assignments does not matter. Therefore the following example, where the assignments are in reverse order, is equivalent:

```
let
  a = 1;
  b = a + 1;
in
a + b
```

```
3
```

Note that the `a` in `b = a + 1` refers to `a = 1`.

After `in` comes the expression in which the assignments are valid. In this example the expression is `a + b`, where `a` refers to `a = 1`, and `b` refers to `b = a + 1`.

By replacing the names with their assigned values, `a + b` evaluates to `3`.

This is similar to *recursive attribute sets* (page 17): in both, the order of assignments does not matter, and names on the left can be used in expressions on the right of the assignment (=).

Example:

`let ... in ...`

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

```
{ a = 1; b = 2; c = 3; }
```

`rec { ... }`

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

```
{ a = 1; b = 2; c = 3; }
```

The difference is that while a recursive attribute set evaluates to an *attribute set* (page 16), any expression can follow after the `in` keyword.

In the following example we use the `let` expression to form a list:

```
let
  b = a + 1;
  c = a + b;
  a = 1;
in [ a b c ]
```

```
[ 1 2 3 ]
```

Only expressions within the `let` expression itself can access the newly declared names. We say: the bindings have local scope.

Counter-example:

```
{
  a = let x = 1; in x;
  b = x;
}
```

```
error: undefined variable 'x'

       at «string»:3:7:

            2|   a = let x = 1; in x;
            3|   b = x;
             |       ^
            4| }
```

Attribute access

Attributes in a set are accessed with a dot (`.`) and the attribute name.

Example:

```
let
  attrset = { x = 1; };
in
attrset.x
```

```
1
```

Accessing nested attributes works the same way.

Example:

```
let
  attrset = { a = { b = { c = 1; }; }; };
in
attrset.a.b.c
```

```
1
```

The dot (`.`) notation can also be used for assigning attributes.

Example:

```
{ a.b.c = 1; }
```

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

`with ...; ...`

The `with` expression allows access to attributes without repeatedly referencing their attribute set.

Example:

```
let
  a = {
    x = 1;
    y = 2;
    z = 3;
  };
in
with a; [ x y z ]
```

```
[ 1 2 3 ]
```

The expression

```
with a; [ x y z ]
```

is equivalent to

```
[ a.x a.y a.z ]
```

Attributes made available through `with` are only in scope of the expression following the semicolon (`;`).

Counter-example:

```
let
  a = {
    x = 1;
    y = 2;
    z = 3;
  };
in
{
  b = with a; [ x y z ];
  c = x;
}
```

```
error: undefined variable 'x'

       at «string»:10:7:

            9|   b = with a; [ x y z ];
           10|   c = x;
             |       ^
           11| }
```

`inherit ...`

`inherit` is shorthand for assigning the value of a name from an existing scope to the same name in a nested scope. It is for convenience to avoid repeating the same name multiple times.

Example:

```
let
  x = 1;
  y = 2;
in
{
  inherit x y;
}
```

```
{ x = 1; y = 2; }
```

The fragment

```
inherit x y;
```

is equivalent to

```
x = x; y = y;
```

It is also possible to `inherit` names from a specific attribute set with parentheses (`inherit (...) ...`).

Example:

```
let
  a = { x = 1; y = 2; };
in
{
  inherit (a) x y;
}
```

```
{ x = 1; y = 2; }
```

The fragment

```
inherit (a) x y;
```

is equivalent to

```
x = a.x; y = a.y;
```

`inherit` also works inside `let` expressions.

Example:

```
let
  inherit ({ x = 1; y = 2; }) x y;
in [ x y ]
```

```
[ 1 2 ]
```

Detailed explanation

While this example is contrived, in more complex code you will regularly see nested `let` expressions that re-use names from their outer scope.

Here we use the attribute set `{ x = 1; y = 2; }` to have something non-trivial to inherit from. The `let` expression inherits `x` and `y` from that attribute set using `( )`, which is equivalent to writing:

```
let
  x = { x = 1; y = 2; }.x;
  y = { x = 1; y = 2; }.y;
in
```

The new inner scope now contains `x` and `y`, which are used in the list `[ x y ]`.

String interpolation `${ ... }`

Previously known as “antiquotation”.

The value of a Nix expression can be inserted into a character string with the dollar-sign and braces (`${ }`).

Example:

```
let
  name = "Nix";
in
"hello ${name}"
```

```
"hello Nix"
```

Only character strings or values that can be represented as a character string are allowed.

Counter-example:

```
let
  x = 1;
in
"${x} + ${x} = ${x + x}"
```

```
error: cannot coerce an integer to a string

       at «string»:4:2:

            3| in
            4| "${x} + ${x} = ${x + x}"
             |  ^
            5|
```

Interpolated expressions can be arbitrarily nested.

(This can become hard to read, and we recommend to avoid it in practice.)

Example:

```
let
  a = "no";
in
"${a + " ${a + " ${a}"}"}"
```

```
"no no no"
```

Detailed explanation

Any Nix expression where the value can be represented as a string can be used within `${ }`.

The + sign in the above expression is the string concatenation operator[36], which takes two strings and produces a new string.

The expression in the example is deliberately confusing in order to demonstrate that arbitrarily nested string interpolations are possible, but tend to be hard to read.

It denotes a string that contains the interpolation of concatenating the value of `a` with a string that starts with a space and is followed by another interpolated string. That second interpolated string is again the result of concatenating the value of `a` and yet another string that starts with a space and is followed by an interpolation of `a`.

Example:

```
let
  a = "one";
  b = "two";
in
"${a + b}"
```

```
"onetwo"
```

Built-in functions are discussed in a *later section* (page 31).

Warning

You may encounter strings that use the dollar sign (`$`) before an assigned name, but no braces (`{ }`):

These are *not* interpolated strings, but usually denote variables in a shell script.

In such cases, the use of names from the surrounding Nix expression is a coincidence.

Example:

[36] https://nix.dev/manual/nix/latest/language/operators#string-concatenation

```
let
  out = "Nix";
in
"echo ${out} > $out"
```

```
"echo Nix > $out"
```

Indented strings

Also known as “multi-line strings”.

The Nix language offers convenience syntax for character strings which span multiple lines that have common indentation.

Indented strings are denoted by *double single quotes* (`'' ''`).

Example:

```
''
multi
line
string
''
```

```
"multi\nline\nstring\n"
```

Equal amounts of prepended white space are trimmed from the result.

Example:

```
''
  one
   two
    three
''
```

```
"one\n two\n  three\n"
```

Note

Indented strings also support *string interpolation* (page 22). For details check the documentation on string literals in the Nix language[37].

File system paths

The Nix language offers convenience syntax for file system paths.

Absolute paths always start with a slash (`/`).

Example:

```
/absolute/path
```

```
/absolute/path
```

[37] https://nix.dev/manual/nix/2.24/language/syntax#string-literal

Paths are relative when they contain at least one slash (`/`) but do not start with one. They evaluate to the path relative to the file containing the expression.

The following examples assume the containing Nix file is in `/current/directory` (or `nix repl` is run in `/current/directory`).

Example:

```
./relative
```

```
/current/directory/relative
```

Example:

```
relative/path
```

```
/current/directory/relative/path
```

One dot (`.`) denotes the current directory within the given path.

You will often see the following expression, which specifies a Nix file's directory.

Example:

```
./.
```

```
/current/directory
```

Detailed explanation

Since relative paths must contain a slash (`/`) but must not start with one, and the dot (`.`) denotes no change of directory, the combination `./.` specifies the current directory as a relative path.

Two dots (`..`) denote the parent directory.

Example:

```
../.
```

```
/current
```

Note

Paths can be used in interpolated expressions – an *impure operation* (page 34) we will cover in detail in a *later section* (page 35).

Lookup paths

Also known as "angle bracket syntax".

Example:

```
<nixpkgs>
```

```
/nix/var/nix/profiles/per-user/root/channels/nixpkgs
```

The value of a lookup path[38] is a file system path that depends on the value of `builtins.nixPath`[39].

[38] https://nix.dev/manual/nix/2.22/language/constructs/lookup-path
[39] https://nix.dev/manual/nix/2.22/language/builtin-constants#builtins-nixPath

In practice, `<nixpkgs>` points to the file system path of some revision of *Nixpkgs*.

For example, `<nixpkgs/lib>` points to the subdirectory `lib` of that file system path:

```
<nixpkgs/lib>
```

```
/nix/var/nix/profiles/per-user/root/channels/nixpkgs/lib
```

While you will encounter many such examples, we recommend to *avoid lookup paths* (page 145) in production code, as they are *impurities* (page 34) which are not reproducible.

2.2.3 Functions

Functions are everywhere in the Nix language and deserve particular attention.

A function always takes exactly one argument. Argument and function body are separated by a colon (`:`).

Wherever you find a colon (`:`) in Nix language code:

- On its left is the function argument
- On its right is the function body.

Function arguments are the third way, apart from *attribute sets* (page 16) and *`let` expressions* (page 18), to assign names to values. Notably, values are not known in advance: the names are placeholders that are filled when *calling a function* (page 27).

Function declarations in the Nix language can appear in different forms. Each of them is explained in the following, and here is an overview:

- Single argument

  ```
  x: x + 1
  ```

  - Multiple arguments via nesting

    ```
    x: y: x + y
    ```

- Attribute set argument

  ```
  { a, b }: a + b
  ```

  - With default attributes

    ```
    { a, b ? 0 }: a + b
    ```

  - With additional attributes allowed

    ```
    { a, b, ...}: a + b
    ```

- Named attribute set argument

  ```
  args@{ a, b, ... }: a + b + args.c
  ```

  or

  ```
  { a, b, ... }@args: a + b + args.c
  ```

Functions in the Nix language have no names. We say they are anonymous, and call such a function a *lambda*.[70]

Example:

[70] The term *lambda* is a shorthand for lambda abstraction[71] in the lambda calculus[72].
[71] https://en.wikipedia.org/wiki/Lambda_calculus#lambdaAbstr
[72] https://en.wikipedia.org/wiki/Lambda_calculus

```
x: x + 1
```

```
<LAMBDA>
```

The `<LAMBDA>` indicates the resulting value is an anonymous function.

As with any other value, functions can be assigned to a name.

Example:

```
let
  f = x: x + 1;
in f
```

```
<LAMBDA>
```

Calling functions

Also known as "function application".

Calling a function with an argument means writing the argument after the function.

Example:

```
let
  f = x: x + 1;
in f 1
```

```
2
```

Example:

```
let
  f = x: x.a;
in
f { a = 1; }
```

```
1
```

The above example calls `f` on a literal attribute set. One can also pass arguments by name.

Example:

```
let
  f = x: x.a;
  v = { a = 1; };
in
f v
```

```
1
```

Since function and argument are separated by white space, sometimes parentheses (`( )`) are required to achieve the desired result.

Example:

```
(x: x + 1) 1
```

```
2
```

Detailed explanation

This expression applies an anonymous function `x: x + 1` to the argument `1`. The function has to be written in parentheses to distinguish it from the argument.

Example:

List elements are also separated by white space, therefore the following are different:

```
let
 f = x: x + 1;
 a = 1;
in [ (f a) ]
```

```
[ 2 ]
```

```
let
 f = x: x + 1;
 a = 1;
in [ f a ]
```

```
[ <LAMBDA> 1 ]
```

The first example reads: apply `f` to `a`, and put the result in a list. The resulting list has one element.

The second example reads: put `f` and `a` in a list. The resulting list has two elements.

Multiple arguments

Also known as "curried[40] functions".

Nix functions take exactly one argument. Multiple arguments can be handled by nesting functions.

Such a nested function can be used like a function that takes multiple arguments, but offers additional flexibility.

Example:

```
x: y: x + y
```

```
<LAMBDA>
```

The above function is equivalent to

```
x: (y: x + y)
```

```
<LAMBDA>
```

This function takes one argument and returns another function `y: x + y` with `x` set to the value of that argument.

Example:

```
let
  f = x: y: x + y;
in
f 1
```

[40] https://en.wikipedia.org/wiki/Currying

```
<LAMBDA>
```

Applying the function which results from `f 1` to another argument yields the inner body `x + y` (with `x` set to `1` and `y` set to the other argument), which can now be fully evaluated.

```
let
  f = x: y: x + y;
in
f 1 2
```

```
3
```

Attribute set argument

Also known as “keyword arguments” or “destructuring” .

Nix functions can be declared to require an attribute set with specific structure as argument.

This is denoted by listing the expected attribute names separated by commas (`,`) and enclosed in braces (`{ }`).

Example:

```
{a, b}: a + b
```

```
<LAMBDA>
```

The argument defines the exact attributes that have to be in that set. Leaving out or passing additional attributes is an error.

Example:

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

```
3
```

Counter-example:

```
let
  f = {a, b}: a + b;
in
f { a = 1; b = 2; c = 3; }
```

```
error: 'f' at (string):2:7 called with unexpected argument 'c'

       at «string»:4:1:

            3| in
            4| f { a = 1; b = 2; c = 3; }
             | ^
            5|
```

Default values

Also known as “default arguments”.

Destructured arguments can have default values for attributes.

This is denoted by separating the attribute name and its default value with a question mark (`?`).

Attributes in the argument are not required if they have a default value.

Example:

```
let
  f = {a, b ? 0}: a + b;
in
f { a = 1; }
```

```
1
```

Example:

```
let
  f = {a ? 0, b ? 0}: a + b;
in
f { } # empty attribute set
```

```
0
```

Additional attributes

Additional attributes are allowed with an ellipsis (`...`):

```
{a, b, ...}: a + b
```

Unlike in the previous counter-example, passing an argument that contains additional attributes is not an error.

Example:

```
let
  f = {a, b, ...}: a + b;
in
f { a = 1; b = 2; c = 3; }
```

```
3
```

Named attribute set argument

Also known as “@ pattern”, “@ syntax”, or “‘at’ syntax”.

An attribute set argument can be given a name to be accessible as a whole.

This is denoted by prepending or appending the name to the attribute set argument, separated by the at sign (`@`).

Example:

```
{a, b, ...}@args: a + b + args.c
```

```
<LAMBDA>
```

or

```
args@{a, b, ...}: a + b + args.c
```

```
<LAMBDA>
```

Example:

```
let
  f = {a, b, ...}@args: a + b + args.c;
in
f { a = 1; b = 2; c = 3; }
```

```
6
```

2.2.4 Function libraries

In addition to the built-in operators[41] (`+`, `==`, `&&`, etc.), there are two widely used libraries that *together* can be considered standard for the Nix language. You need to know about both to understand and navigate Nix language code.

We recommend to at least skim them to familiarise yourself with what is available.

`builtins`

Also known as "primitive operations" or "primops".

Nix comes with many functions that are built into the language. They are implemented in C++ as part of the Nix language interpreter.

Note

The Nix manual lists all Built-in Functions[42], and shows how to use them.

These functions are available under the `builtins` constant.

Example:

```
builtins.toString
```

```
<PRIMOP>
```

`import`

Most built-in functions are only accessible through `builtins`. A notable exception is `import`, which is also available at the top level.

`import` takes a path to a Nix file, reads it to evaluate the contained Nix expression, and returns the resulting value. If the path points to a directory, the file `default.nix` in that directory is used instead.

Example:

```
$ echo 1 + 2 > file.nix
```

```
import ./file.nix
```

[41] https://nix.dev/manual/nix/stable/language/operators.html
[42] https://nix.dev/manual/nix/stable/language/builtins.html

```
3
```

Detailed explanation

The preceding shell command writes the contents `1 + 2` to the file `file.nix` in the current directory.

The above Nix expression refers to this file as `./file.nix`. `import` reads the file and evaluates to the contained Nix expression.

It is an error if the file system path does not exist.

After reading `file.nix` the Nix expression is equivalent to the file contents:

```
1 + 2
```

```
3
```

Since a Nix file can contain any Nix expression, `import`ed functions can be applied to arguments immediately.

That is, whenever you find additional tokens after a call to `import`, the value it returns should be a function, and anything that follows are arguments to that function.

Example:

```
$ echo "x: x + 1" > file.nix
```

```
import ./file.nix 1
```

```
2
```

Detailed explanation

The preceding shell command writes the contents `x: x + 1` to the file `file.nix` in the current directory.

The above Nix expression refers to this file as `./file.nix`. `import ./file.nix` reads the file and evaluates to the contained Nix expression.

It is an error if the file system path does not exist.

After reading the file, the Nix expression `import ./file.nix` is equivalent to the file contents:

```
(x: x + 1) 1
```

```
2
```

This applies the function `x: x + 1` to the argument `1`, and therefore evaluates to `2`.

Note

Parentheses are required to separate function declaration from function application.

`pkgs.lib`

The `nixpkgs`[43] repository contains an attribute set called `lib`[44], which provides a large number of useful functions. They are implemented in the Nix language, as opposed to *`builtins`* (page 31), which are part of the language itself.

[43] https://github.com/NixOS/nixpkgs
[44] https://github.com/NixOS/nixpkgs/blob/master/lib/default.nix

Note

The Nixpkgs manual lists all Nixpkgs library functions[45].

These functions are usually accessed through `pkgs.lib`, as the Nixpkgs attribute set is given the name `pkgs` by convention.

Example:

```
let
  pkgs = import <nixpkgs> {};
in
pkgs.lib.strings.toUpper "lookup paths considered harmful"
```

```
LOOKUP PATHS CONSIDERED HARMFUL
```

Detailed explanation

This is a more complex example, but by now you should be familiar with all its components.

The name `pkgs` is declared to be the expression `import`ed from some file. That file's path is determined by the value of the lookup path `<nixpkgs>`, which in turn is determined by the `$NIX_PATH` environment variable at the time this expression is evaluated. As this expression happens to be a function, it requires an argument to evaluate, and in this case passing an empty attribute set `{}` is sufficient.

Now that `pkgs` is in scope of `let ... in ...`, its attributes can be accessed. From the Nixpkgs manual one can determine that there exists a function under `lib.strings.toUpper`[46].

For brevity, this example uses a lookup path to obtain *some version* of Nixpkgs. The function `toUpper` is trivial enough that we can expect it not to produce different results for different versions of Nixpkgs. Yet, more sophisticated software is likely to suffer from such problems. A fully reproducible example would therefore look like this:

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/archive/
↪06278c77b5d162e62df170fec307e83f1812d94b.tar.gz";
  pkgs = import nixpkgs {};
in
pkgs.lib.strings.toUpper "always pin your sources"
```

```
ALWAYS PIN YOUR SOURCES
```

See *Towards reproducibility: pinning Nixpkgs* (page 11) for details.

What you will also often see is that `pkgs` is passed as an argument to a function. By convention one can assume that it refers to the Nixpkgs attribute set, which has a `lib` attribute:

```
{ pkgs, ... }:
pkgs.lib.strings.removePrefix "no " "no true scotsman"
```

```
<LAMBDA>
```

To make this function produce a result, you can write it to a file (e.g. `file.nix`) and pass it an argument through `nix-instantiate`:

```
$ nix-instantiate --eval file.nix --arg pkgs 'import <nixpkgs> {}'
"true scotsman"
```

[45] https://nixos.org/manual/nixpkgs/stable/#sec-functions-library
[46] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.strings.toUpper

Oftentimes you will see in NixOS configurations, and also within Nixpkgs, that `lib` is passed directly. In that case one can assume that this `lib` is equivalent to `pkgs.lib` where only `pkgs` is available.

Example:

```
{ lib, ... }:
let
  to-be = true;
in
lib.trivial.or to-be (! to-be)
```

```
<LAMBDA>
```

To make this function produce a result, you can write it to a file (e.g. `file.nix`) and pass it an argument through `nix-instantiate`:

```
$ nix-instantiate --eval file.nix --arg lib '(import <nixpkgs> {}).lib'
true
```

Sometimes both `pkgs` and `lib` are passed as arguments. In that case, one can assume `pkgs.lib` and `lib` to be equivalent. This is done to improve readability by avoiding repeated use of `pkgs.lib`.

Example:

```
{ pkgs, lib, ... }:
# ... multiple uses of `pkgs`
# ... multiple uses of `lib`
```

For historical reasons, some of the functions in `pkgs.lib` are equivalent to *builtins* (page 31) of the same name.

2.2.5 Impurities

So far we have only covered what we call *pure expressions*: declaring data and transforming it with functions.

In practice, describing derivations – the Nix language's defining feature, which enables functional programming with the file system – requires observing the outside world. We will discuss *derivations* (page 36) later in the tutorial.

There is only one impurity in the Nix language that is relevant here: reading files from the file system as *build inputs*.

Build inputs are files that derivations refer to in order to describe how to derive new files. When run, a derivation will only have access to explicitly declared build inputs.

The only way to specify build inputs in the Nix language is explicitly with:

- File system paths
- Dedicated functions

Nix and the Nix language refer to files by their content hash. If file contents are not known in advance, it's unavoidable to read files during expression evaluation.

Note

Nix supports other types of impure expressions, such as *lookup paths* (page 145) or the constant `builtins.currentSystem`[47]. We do not cover those here in more detail, as they do not matter for how the Nix language works in principle, and because they are discouraged for the very reason of breaking reproducibility.

[47] https://nix.dev/manual/nix/stable/language/builtin-constants.html#builtins-currentSystem

Paths

Whenever a file system path is used in *string interpolation* (page 22), the contents of that file are copied to a special location in the file system, the *Nix store*, as a side effect.

The evaluated string then contains the Nix store path assigned to that file.

Example:

```
$ echo 123 > data
```

```
"${./data}"
```

```
"/nix/store/h1qj5h5n05b5dl5q4nldrqq8mdg7dhqk-data"
```

Detailed explanation

The preceding shell command writes the characters `123` to the file `data` in the current directory.

The above Nix expression refers to this file as `./data` and converts the file system path to an *interpolated string* (page 22) `${ ... }`.

Such interpolated expressions must evaluate to something that can be represented as a character string. A file system path is such a value, and its character string representation is the corresponding Nix store path:

```
/nix/store/<hash>-<name>
```

The Nix store path is obtained by taking the hash of the file's contents (`<hash>`) and combining it with the file name (`<name>`). The file is copied into the Nix store directory `/nix/store` as a side effect of evaluation. It is an error if the file system path does not exist.

For directories the same thing happens: The entire directory (including nested files and directories) is copied to the Nix store, and the evaluated string becomes the Nix store path of the directory.

Fetchers

Files to be used as build inputs do not have to come from the file system.

The Nix language provides built-in impure functions to fetch files over the network during evaluation:

- `builtins.fetchurl`[48]
- `builtins.fetchTarball`[49]
- `builtins.fetchGit`[50]
- `builtins.fetchClosure`[51]

These functions evaluate to a file system path in the Nix store.

Example:

```
builtins.fetchurl "https://github.com/NixOS/nix/archive/
↪7c3ab5751568a0bc63430b33a5169c5e4784a0ff.tar.gz"
```

```
"/nix/store/7dhgs330clj36384akg86140fqkgh8zf-
↪7c3ab5751568a0bc63430b33a5169c5e4784a0ff.tar.gz"
```

Some of them add extra convenience, such as automatically unpacking archives.

Example:

[48] https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchurl
[49] https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchTarball
[50] https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchGit
[51] https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchClosure

```
builtins.fetchTarball "https://github.com/NixOS/nix/archive/
↪7c3ab5751568a0bc63430b33a5169c5e4784a0ff.tar.gz"
```

```
"/nix/store/d59llm96vgis5fy231x6m7nrijs0ww36-source"
```

Note

The Nixpkgs manual on Fetchers[52] lists numerous additional library functions to fetch files over the network.

It is an error if the network request fails.

2.2.6 Derivations

Derivations are at the core of both Nix and the Nix language:

- The Nix language is used to describe derivations.
- Nix runs derivations to produce *build results*.
- Build results can in turn be used as inputs for other derivations.

The Nix language primitive to declare a derivation is the built-in impure function `derivation`.

It is usually wrapped by the Nixpkgs build mechanism `stdenv.mkDerivation`, which hides much of the complexity involved in non-trivial build procedures.

Note

You will probably never encounter `derivation` in practice.

Whenever you encounter `mkDerivation`, it denotes something that Nix will eventually *build*.

Example: *a package using `mkDerivation`* (page 38)

The evaluation result of `derivation` (and `mkDerivation`) is an *attribute set* (page 16) with a certain structure and a special property: It can be used in *string interpolation* (page 22), and in that case evaluates to the Nix store path of its build result.

Example:

```
let
  pkgs = import <nixpkgs> {};
in "${pkgs.nix}"
```

```
"/nix/store/sv2srrjddrp2isghmrla8s6lazbzmikd-nix-2.11.0"
```

Note

Your output may differ. It may produce a different hash or even a different package version.

A derivation's output path is fully determined by its inputs, which in this case come from *some* version of Nixpkgs.

This is why we recommend to *avoid lookup paths* (page 145) to ensure predictable outcomes, except in examples intended for illustration only.

[52] https://nixos.org/manual/nixpkgs/stable/#chap-pkgs-fetchers

Detailed explanation

The example imports the Nix expression from the lookup path `<nixpkgs>`, and applies the resulting function to an empty attribute set `{}`. Its output is assigned the name `pkgs`.

Converting the attribute `pkgs.nix` to a string with *string interpolation* (page 22) is allowed, as `pkgs.nix` is a derivation. That is, ultimately `pkgs.nix` boils down to a call to `derivation`.

The resulting string is the file system path where the build result of that derivation will end up.

There is more depth to the inner workings of derivations, but at this point it should be enough to know that such expressions evaluate to Nix store paths.

String interpolation on derivations is used to refer to their build results as file system paths when declaring new derivations.

This allows constructing arbitrarily complex compositions of derivations with the Nix language.

2.2.7 Worked examples

So far we have seen artificial examples illustrating the various constructs in the Nix language.

You should now be able to read Nix language code for simple packages and configurations, and come up with similar explanations of the following practical examples.

> **Note**
>
> The goal of the following exercises is not to understand what the code means or how it works, but how it is structured in terms of functions, attribute sets, and other Nix language data types.

Shell environment

```
{ pkgs ? import <nixpkgs> {} }:
let
  message = "hello world";
in
pkgs.mkShellNoCC {
  packages = with pkgs; [ cowsay ];
  shellHook = ''
    cowsay ${message}
  '';
}
```

This example declares a shell environment (which runs the `shellHook` on initialization).

Explanation:

- This expression is a function that takes an attribute set as an argument.
- If the argument has the attribute `pkgs`, it will be used in the function body. Otherwise, by default, import the Nix expression in the file found on the lookup path `<nixpkgs>` (which is a function in this case), call the function with an empty attribute set, and use the resulting value.
- The name `message` is bound to the string value `"hello world"`.
- The attribute `mkShellNoCC` of the `pkgs` set is a function that is passed an attribute set as argument. Its return value is also the result of the outer function.
- The attribute set passed to `mkShellNoCC` has the attributes `packages` (set to a list with one element: the `cowsay` attribute from `pkgs`) and `shellHook` (set to an indented string).
- The indented string contains an interpolated expression, which will expand the value of `message` to yield `"hello world"`.

NixOS configuration

```
{ config, pkgs, ... }: {

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

  environment.systemPackages = with pkgs; [ git ];

  # ...

}
```

This example is (part of) a NixOS configuration.

Explanation:

- This expression is a function that takes an attribute set as an argument. It returns an attribute set.
- The argument must at least have the attributes `config` and `pkgs`, and may have more attributes.
- The returned attribute set contains the attributes `imports` and `environment`.
- `imports` is a list with one element: a path to a file next to this Nix file, called `hardware-configuration.nix`.

> **Note**
>
> `imports` is not the impure built-in `import`, but a regular attribute name!

- `environment` is itself an attribute set with one attribute `systemPackages`, which will evaluate to a list with one element: the `git` attribute from the `pkgs` set.
- The `config` argument is not (shown to be) used.

Package

```
{ lib, stdenv, fetchurl }:

stdenv.mkDerivation rec {

  pname = "hello";

  version = "2.12";

  src = fetchurl {
    url = "mirror://gnu/${pname}/${pname}-${version}.tar.gz";
    sha256 = "1ayhp9v4m4rdhjmnl2bq3cibrbqqkgjbl3s7yk2nhlh8vj3ay16g";
  };

  meta = with lib; {
    license = licenses.gpl3Plus;
  };

}
```

This example is a (simplified) package declaration from Nixpkgs.

Explanation:

- This expression is a function that takes an attribute set which must have exactly the attributes `lib`, `stdenv`, and `fetchurl`.

- It returns the result of evaluating the function `mkDerivation`, which is an attribute of `stdenv`, applied to a recursive set.
- The recursive set passed to `mkDerivation` uses its own `pname` and `version` attributes in the argument to the function `fetchurl`. `fetchurl` itself comes from the outer function's arguments.
- The `meta` attribute is itself an attribute set, where the `license` attribute has the value that was assigned to the nested attribute `lib.licenses.gpl3Plus`.

2.2.8 References

- Nix manual: Nix language[53]
- Nix manual: String interpolation[54]
- Nix manual: Built-in Functions[55]
- Nix manual: `nix repl`[56]
- Nixpkgs manual: Functions reference[57]
- Nixpkgs manual: Fetchers[58]

2.2.9 Next steps

Get things done

- *Declarative shell environments with shell.nix* (page 8) – create reproducible shell environments from a Nix file
- *Packaging existing software with Nix* (page 40) – make more software available through Nix

If you want to take a longer break from learning Nix, you can remove unused build results from the Nix store with:

```
$ nix-collect-garbage
```

Learn more

If you worked through the examples, you will have noticed that reading the Nix language reveals the structure of the code, but does not necessarily tell what the code actually means.

Often it is not possible to determine from the code at hand

- the data type of a named value or function argument.
- the data type a called function accepts for its argument.
- which attributes are present in a given attribute set.

Example:

```
{ x, y, z }: (x y) z.a
```

How do we know…

- that `x` will be a function that, given an argument, returns a function?
- that, given `x` is a function, `y` will be an appropriate argument to `x`?
- that, given `(x y)` is a function, `z.a` will be an appropriate argument to `(x y)`?
- that `z` will be an attribute set at all?

[53] https://nix.dev/manual/nix/stable/language/index.html
[54] https://nix.dev/manual/nix/stable/language/string-interpolation.html
[55] https://nix.dev/manual/nix/stable/language/builtins.html
[56] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-repl.html
[57] https://nixos.org/manual/nixpkgs/stable/#sec-functions-library
[58] https://nixos.org/manual/nixpkgs/stable/#chap-pkgs-fetchers

- that, given `z` is an attribute set, it will have an attribute `a`?
- which data type `y` and `z.a` will be?
- the data type of the end result?

And how does the caller of this function know that it requires an attribute set with attributes `x`, `y`, `z`?

Answering such questions requires knowing the context in which a given expression is supposed to be used.

The Nix ecosystem and code style is driven by conventions. Most names you will encounter in Nix language code come from Nixpkgs:

- Nix Pills[59] - a detailed explanation of derivations and how Nixpkgs is constructed from first principles

Nixpkgs provides generic build mechanisms that are widely used:

- `stdenv`[60] - most importantly `mkDerivation`
- Trivial Builders[61] - to create files and shell scripts

Packages from Nixpkgs can be modified through multiple mechanisms:

- overrides[62] – specifically `override` and `overrideAttrs` to modify single packages
- overlays[63] – to produce a custom variant of Nixpkgs with individually modified packages

Different language ecosystems and frameworks have different requirements to accommodating them into Nixpkgs:

- Languages and frameworks[64] lists tools provided by Nixpkgs to build language- or framework-specific packages with Nix.

The NixOS Linux distribution has a modular configuration system that imposes its own conventions:

- NixOS modules[65] shows how NixOS configurations are organized.

2.3 Packaging existing software with Nix

One of Nix's primary use-cases is in addressing common difficulties encountered with packaging software, such as specifying and obtaining dependencies.

In the long term, Nix helps tremendously with alleviating such problems. But when *first* packaging existing software with Nix, it's common to encounter errors that seem inscrutable.

2.3.1 Introduction

In this tutorial, you'll create your first Nix derivations[73] to package C/C++ software, taking advantage of the Nixpkgs Standard Environment[74] (`stdenv`), which automates much of the work involved.

What will you learn?

The tutorial begins with `hello`, an implementation of "hello world" which only requires dependencies already provided by `stdenv`. Next, you will build more complex packages with their own dependencies, leading you to use additional derivation features.

You'll encounter and address Nix error messages, build failures, and a host of other issues, developing your iterative debugging techniques along the way.

[59] https://nixos.org/guides/nix-pills/
[60] https://nixos.org/manual/nixpkgs/stable/#chap-stdenv
[61] https://nixos.org/manual/nixpkgs/stable/#chap-trivial-builders
[62] https://nixos.org/manual/nixpkgs/stable/#chap-overrides
[63] https://nixos.org/manual/nixpkgs/stable/#chap-overlays
[64] https://nixos.org/manual/nixpkgs/stable/#chap-language-support
[65] https://nixos.org/manual/nixos/stable/index.html#sec-writing-modules
[73] https://nix.dev/manual/nix/stable/language/derivations
[74] https://nixos.org/manual/nixpkgs/stable/#part-stdenv

What do you need?

- Familiarity with the Unix shell and plain text editors
- You should be confident with *reading the Nix language* (page 12). Feel free to go back and work through the tutorial first.

How long does it take?

Going through all the steps carefully will take around 60 minutes.

2.3.2 Your first package

Note

A *package* is a loosely defined concept that refers to either a collection of files and other data, or a *Nix expression* representing such a collection before it comes into being. Packages in Nixpkgs have a conventional structure, allowing them to be discovered in searches and composed in environments alongside other packages.

For the purposes of this tutorial, a "package" is a Nix language function that will evaluate to a derivation. It will enable you or others to produce an artifact for practical use, as a consequence of having "packaged existing software with Nix".

To start, consider this skeleton derivation:

```
{ stdenv }:

stdenv.mkDerivation {         }
```

This is a function which takes an attribute set containing `stdenv`, and produces a derivation (which currently does nothing).

A package function

GNU Hello is an implementation of the "hello world" program, with source code accessible from the GNU Project's FTP server[75].

To begin, add a `pname` attribute to the set passed to `mkDerivation`. Every package needs a name and a version, and Nix will throw `error: derivation name missing` without.

```
stdenv.mkDerivation {
+ pname = "hello";
+ version = "2.12.1";
```

Next, you will declare a dependency on the latest version of `hello`, and instruct Nix to use `fetchzip` to download the source code archive[76].

Note

`fetchzip` can fetch more archives[77] than just zip files!

The hash cannot be known until after the archive has been downloaded and unpacked. Nix will complain if the hash supplied to `fetchzip` is incorrect. Set the `hash` attribute to an empty string and then use the resulting error message to determine the correct hash:

[75] https://ftp.gnu.org/gnu/hello/
[76] https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz
[77] https://nixos.org/manual/nixpkgs/stable/#fetchurl

```nix
# hello.nix
{
  stdenv,
  fetchzip,
}:

stdenv.mkDerivation {
  pname = "hello";
  version = "2.12.1";

  src = fetchzip {
    url = "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz";
    sha256 = "";
  };
}
```

Save this file to `hello.nix` and run `nix-build` to observe your first build failure:

```
$ nix-build hello.nix
error: cannot evaluate a function that has an argument without a value ('stdenv')
       Nix attempted to evaluate a function as a top level expression; in
       this case it must have its arguments supplied either by default
       values, or passed explicitly with '--arg' or '--argstr'. See
       https://nix.dev/manual/nix/stable/language/constructs.html#functions.

       at /home/nix-user/hello.nix:3:3:

            2| {
            3|   stdenv,
             |   ^
            4|   fetchzip,
```

Problem: the expression in `hello.nix` is a *function*, which only produces its intended output if it is passed the correct *arguments*.

Building with `nix-build`

`stdenv` is available from `nixpkgs`[78], which must be imported with another Nix expression in order to pass it as an argument to this derivation.

The recommended way to do this is to create a `default.nix` file in the same directory as `hello.nix`, with the following contents:

```nix
# default.nix
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
{
  hello = pkgs.callPackage ./hello.nix { };
}
```

This allows you to run `nix-build -A hello` to realize the derivation in `hello.nix`, similar to the current convention used in Nixpkgs.

[78] https://github.com/NixOS/nixpkgs/

Note

`callPackage` automatically passes attributes from `pkgs` to the given function, if they match attributes required by that function's argument attribute set. In this case, `callPackage` will supply `stdenv`, and `fetchzip` to the function defined in `hello.nix`.

The tutorial *Package parameters and overrides with callPackage* (page 52) goes into detail on how this works.

Now run the `nix-build` command with the new argument:

```
$ nix-build -A hello
error: hash mismatch in fixed-output derivation '/nix/store/
↪pd2kiyfa0c06giparlhd1k31bvllypbb-source.drv':
         specified: sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
            got:    sha256-1kJjhtlsAkpNB7f6tZEs+dbKd8z7KoNHyDHEJ0tmhnc=
error: 1 dependencies of derivation '/nix/store/b4mjwlv73nmiqgkdabsdjc4zq9gnma1l-
↪hello-2.12.1.drv' failed to build
```

Finding the file hash

As expected, the incorrect file hash caused an error, and Nix helpfully provided the correct one. In `hello.nix`, replace the empty string with the correct hash:

```
# hello.nix
{
  stdenv,
  fetchzip,
}:

stdenv.mkDerivation {
  pname = "hello";
  version = "2.12.1";

  src = fetchzip {
    url = "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz";
    sha256 = "sha256-1kJjhtlsAkpNB7f6tZEs+dbKd8z7KoNHyDHEJ0tmhnc=";
  };
}
```

Now run the previous command again:

```
$ nix-build -A hello
this derivation will be built:
  /nix/store/rbq37s3r76rr77c7d8x8px7z04kw2mk7-hello.drv
building '/nix/store/rbq37s3r76rr77c7d8x8px7z04kw2mk7-hello.drv'...
...
configuring
...
configure: creating ./config.status
config.status: creating Makefile
...
building
... <many more lines omitted>
```

Great news: the derivation built successfully!

The console output shows that `configure` was called, which produced a `Makefile` that was then used to build the project. It wasn't necessary to write any build instructions in this case because the `stdenv` build system is based on

GNU Autoconf[79], which automatically detected the structure of the project directory.

Build result

Check your working directory for the result:

```
$ ls
default.nix hello.nix  result
```

This `result` is a symbolic link[80] to a Nix store location containing the built binary; you can call `./result/bin/hello` to execute this program:

```
$ ./result/bin/hello
Hello, world!
```

Congratulations, you have successfully packaged your first program with Nix!

Next, you'll package another piece of software with external-to-`stdenv` dependencies that present new challenges, requiring you to make use of more `mkDerivation` features.

2.3.3 A package with dependencies

Now you will package a somewhat more complicated program, `icat`[81], which allows you to render images in your terminal.

Change the `default.nix` from the previous section by adding a new attribute for `icat`:

```
# default.nix
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
{
  hello = pkgs.callPackage ./hello.nix { };
  icat = pkgs.callPackage ./icat.nix { };
}
```

Copy `hello.nix` to a new file `icat.nix`, and update the `pname` and `version` attributes in that file:

```
# icat.nix
{
  stdenv,
  fetchzip,
}:

stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

  src = fetchzip {
    # ...
  };
}
```

Now to download the source code. `icat`'s upstream repository is hosted on GitHub[82], so you should replace the

[79] https://www.gnu.org/software/autoconf/
[80] https://en.wikipedia.org/wiki/Symbolic_link
[81] https://github.com/atextor/icat
[82] https://github.com/atextor/icat

previous source fetcher[83]. This time you will use `fetchFromGitHub`[84] instead of `fetchzip`, by updating the argument attribute set to the function accordingly:

```
# icat.nix
{
  stdenv,
  fetchFromGitHub,
}:

stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

  src = fetchFromGitHub {
    # ...
  };
}
```

Fetching source from GitHub

While `fetchzip` required `url` and `sha256` arguments, more are needed for `fetchFromGitHub`[85].

The source URL is `https://github.com/atextor/icat`, which already gives the first two arguments:

- `owner`: the name of the account controlling the repository

  ```
  owner = "atextor";
  ```

- `repo`: the name of the repository to fetch

  ```
  repo = "icat";
  ```

Navigate to the project's Tags page[86] to find a suitable Git revision[87] (`rev`), such as the Git commit hash or tag (e.g. `v1.0`) corresponding to the release you want to fetch.

In this case, the latest release tag is `v0.5`.

As in the `hello` example, a hash must also be supplied. This time, instead of using the empty string and letting `nix-build` report the correct one in an error, you can fetch the correct hash in the first place with the `nix-prefetch-url` command.

You need the SHA256 hash of the *contents* of the tarball (as opposed to the hash of the tarball file itself). Therefore pass the `--unpack` and `--type sha256` arguments:

```
$ nix-prefetch-url --unpack https://github.com/atextor/icat/archive/refs/tags/v0.5.
↪tar.gz --type sha256
path is '/nix/store/p8jl1jlqxcsc7ryiazbpm7c1mqb6848b-v0.5.tar.gz'
0wyy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pzr99ljs9awy8ka
```

Set the correct hash for `fetchFromGitHub`:

```
# icat.nix
{
  stdenv,
  fetchFromGitHub,
}:
```

(continues on next page)

[83] https://nixos.org/manual/nixpkgs/stable/#chap-pkgs-fetchers
[84] https://nixos.org/manual/nixpkgs/stable/#fetchfromgithub
[85] https://nixos.org/manual/nixpkgs/stable/#fetchfromgithub
[86] https://github.com/atextor/icat/tags
[87] https://git-scm.com/docs/revisions

(continued from previous page)

```
6
7  stdenv.mkDerivation {
8    pname = "icat";
9    version = "v0.5";
10
11   src = fetchFromGitHub {
12     owner = "atextor";
13     repo = "icat";
14     rev = "v0.5";
15     sha256 = "0wyy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pzr99ljs9awy8ka";
16   };
17 }
```

Missing dependencies

Running `nix-build` with the new `icat` attribute, an entirely new issue is reported:

```
$ nix-build -A icat
these 2 derivations will be built:
  /nix/store/86q9x927hsyyzfr4lcqirmsbimysi6mb-source.drv
  /nix/store/l5wz9inkvkf0qhl8kpl39vpg2xfm2qpy-icat.drv
...
error: builder for '/nix/store/l5wz9inkvkf0qhl8kpl39vpg2xfm2qpy-icat.drv' failed␣
↪with exit code 2;
       last 10 log lines:
       >                  from /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp4l-glibc-2.
↪37-8-dev/include/stdio.h:27,
       >                  from icat.c:31:
       > /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp4l-glibc-2.37-8-dev/include/
↪features.h:195:3: warning: #warning "_BSD_SOURCE and _SVID_SOURCE are deprecated,
↪ use _DEFAULT_SOURCE" [8;;https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
↪#index-Wcpp-Wcpp8;;]
       >   195 | # warning "_BSD_SOURCE and _SVID_SOURCE are deprecated, use _
↪DEFAULT_SOURCE"
       >       |   ^~~~~~
       > icat.c:39:10: fatal error: Imlib2.h: No such file or directory
       >    39 | #include <Imlib2.h>
       >       |          ^~~~~~~~~~
       > compilation terminated.
       > make: *** [Makefile:16: icat.o] Error 1
       For full logs, run 'nix log /nix/store/l5wz9inkvkf0qhl8kpl39vpg2xfm2qpy-
↪icat.drv'.
```

A compiler error! The `icat` source was pulled from GitHub, and Nix tried to build what it found, but compilation failed due to a missing dependency: the `imlib2` header.

If you search for `imlib2` on search.nixos.org[88], you'll find that `imlib2` is already in Nixpkgs.

Add this package to your build environment by adding `imlib2` to the arguments of the function in `icat.nix`. Then add the argument's value `imlib2` to the list of `buildInputs` in `stdenv.mkDerivation`:

```
1  # icat.nix
2  {
3    stdenv,
4    fetchFromGitHub,
5    imlib2,
```

(continues on next page)

[88] https://search.nixos.org/packages?query=imlib2

(continued from previous page)

```
}:

stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

  src = fetchFromGitHub {
    owner = "atextor";
    repo = "icat";
    rev = "v0.5";
    sha256 = "0wyy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pzr99ljs9awy8ka";
  };

  buildInputs = [ imlib2 ];
}
```

Run `nix-build -A icat` again and you'll encounter another error, but compilation proceeds further this time:

```
$ nix-build -A icat
this derivation will be built:
  /nix/store/bw2d4rp2k1l5rg49hds199ma2mz36x47-icat.drv
...
error: builder for '/nix/store/bw2d4rp2k1l5rg49hds199ma2mz36x47-icat.drv' failed␣
↪with exit code 2;
       last 10 log lines:
       >                  from icat.c:31:
       > /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp4l-glibc-2.37-8-dev/include/
↪features.h:195:3: warning: #warning "_BSD_SOURCE and _SVID_SOURCE are deprecated,
↪ use _DEFAULT_SOURCE" [8;;https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
↪#index-Wcpp-Wcpp8;;]
       >   195 | # warning "_BSD_SOURCE and _SVID_SOURCE are deprecated, use _
↪DEFAULT_SOURCE"
       >       |   ^~~~~~~
       > In file included from icat.c:39:
       > /nix/store/4fvrh0sjc8sbkbqda7dfsh7q0gxmnh9p-imlib2-1.11.1-dev/include/
↪Imlib2.h:45:10: fatal error: X11/Xlib.h: No such file or directory
       >    45 | #include <X11/Xlib.h>
       >       |          ^~~~~~~~~~~
       > compilation terminated.
       > make: *** [Makefile:16: icat.o] Error 1
       For full logs, run 'nix log /nix/store/bw2d4rp2k1l5rg49hds199ma2mz36x47-
↪icat.drv'.
```

You can see a few warnings which should be corrected in the upstream code. But the important bit for this tutorial is `fatal error: X11/Xlib.h: No such file or directory`: another dependency is missing.

2.3.4 Finding packages

Determining from where to source a dependency is currently somewhat involved, because package names don't always correspond to library or program names.

You will need the `Xlib.h` headers from the `X11` C package, the Nixpkgs derivation for which is `libX11`, available in the `xorg` package set. There are multiple ways to figure this out:

search.nixos.org

Tip

The easiest way to find what you need is on search.nixos.org/packages[89].

Unfortunately in this case, searching for `x11`[90] produces too many irrelevant results because X11 is ubiquitous. On the left side bar there is a list package sets, and selecting `xorg`[91] shows something promising.

In case all else fails, it helps to become familiar with searching the Nixpkgs source code[92] for keywords.

Local code search

To find name assignments in the source, search for `"<keyword> ="`. For example, these are the search results for `"x11 = "`[93] or `"libx11 ="`[94] on Github.

Or fetch a clone of the Nixpkgs repository[95] and search the code locally.

Start a shell that makes the required tools available – `git` for version control, and `rg` for code search (provided by the `ripgrep` package[96]):

```
$ nix-shell -p git ripgrep
[nix-shell:~]$
```

The Nixpkgs repository is huge. Only clone the latest revision to avoid waiting a long time for a full clone:

```
[nix-shell:~]$ git clone https://github.com/NixOS/nixpkgs --depth 1
...
[nix-shell:~]$ cd nixpkgs/
```

To narrow down results, only search the `pkgs` subdirectory, which holds all the package recipes:

```
[nix-shell:~]$ rg "x11 =" pkgs
pkgs/tools/X11/primus/default.nix
21:  primus = if useNvidia then primusLib_ else primusLib_.override { nvidia_x11 =␣
↪null; };
22:  primus_i686 = if useNvidia then primusLib_i686_ else primusLib_i686_.override
↪{ nvidia_x11 = null; };

pkgs/applications/graphics/imv/default.nix
38:    x11 = [ libGLU xorg.libxcb xorg.libX11 ];

pkgs/tools/X11/primus/lib.nix
14:    if nvidia_x11 == null then libGL

pkgs/top-level/linux-kernels.nix
573:    ati_drivers_x11 = throw "ati drivers are no longer supported by any kernel␣
↪>=4.1"; # added 2021-05-18;
... <a lot more results>
```

Since `rg` is case sensitive by default, Add `-i` to make sure you don't miss anything:

[89] http://search.nixos.org/packages
[90] https://search.nixos.org/packages?query=x11
[91] https://search.nixos.org/packages?buckets=%7B%22package_attr_set%22%3A%5B%22xorg%22%5D%2C%22package_license_set%22%3A%5B%5D%2C%22package_maintainers_set%22%3A%5B%5D%2C%22package_platforms%22%3A%5B%5D%7D&query=x11
[92] https://github.com/nixos/nixpkgs
[93] https://github.com/search?q=repo%3ANixOS%2Fnixpkgs+%22x11+%3D%22&type=code
[94] https://github.com/search?q=repo%3ANixOS%2Fnixpkgs+%22libx11+%3D%22&type=code
[95] https://github.com/nixos/nixpkgs
[96] https://search.nixos.org/packages?show=ripgrep

```
[nix-shell:~]$ rg -i "libx11 =" pkgs
pkgs/applications/version-management/monotone-viz/graphviz-2.0.nix
55:    ++ lib.optional (libX11 == null) "--without-x";

pkgs/top-level/all-packages.nix
14191:    libX11 = xorg.libX11;

pkgs/servers/x11/xorg/default.nix
1119:  libX11 = callPackage ({ stdenv, pkg-config, fetchurl, xorgproto,␣
↪libpthreadstubs, libxcb, xtrans, testers }: stdenv.mkDerivation (finalAttrs: {

pkgs/servers/x11/xorg/overrides.nix
147:  libX11 = super.libX11.overrideAttrs (attrs: {
```

Local derivation search

To search derivations on the command line, use `nix-locate` from the `nix-index`[97].

Adding package sets as dependencies

Add `xorg` to your derivation's input attribute set and use `xorg.libX11` in `buildInputs`:

```
# icat.nix
{
  stdenv,
  fetchFromGitHub,
  imlib2,
  xorg,
}:

stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

  src = fetchFromGitHub {
    owner = "atextor";
    repo = "icat";
    rev = "v0.5";
    sha256 = "0wyy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pzr99ljs9awy8ka";
  };

  buildInputs = [ imlib2 xorg.libX11 ];
}
```

Note

Because the Nix language is lazily evaluated, accessing only `xorg.libX11` means that the remaining contents of the `xorg` attribute set are never processed.

2.3.5 Fixing build failures

Run the last command again:

[97] https://github.com/nix-community/nix-index

```
$ nix-build -A icat
this derivation will be built:
  /nix/store/x1d79ld8jxqdla5zw2b47d2sl87mf56k-icat.drv
...
error: builder for '/nix/store/x1d79ld8jxqdla5zw2b47d2sl87mf56k-icat.drv' failed␣
↪with exit code 2;
       last 10 log lines:
       >   195 | # warning "_BSD_SOURCE and _SVID_SOURCE are deprecated, use _
↪DEFAULT_SOURCE"
       >       |   ^~~~~~
       > icat.c: In function 'main':
       > icat.c:319:33: warning: ignoring return value of 'write' declared with␣
↪attribute 'warn_unused_result' [8;;https://gcc.gnu.org/onlinedocs/gcc/Warning-
↪Options.html#index-Wunused-result-Wunused-result8;;]
       >   319 |                                 write(tempfile, &buf, 1);
       >       |                                 ^~~~~~~~~~~~~~~~~~~~~~~~
       > gcc -o icat icat.o -lImlib2
       > installing
       > install flags: SHELL=/nix/store/8fv91097mbh5049i9rglc73dx6kjg3qk-bash-5.2-
↪p15/bin/bash install
       > make: *** No rule to make target 'install'.  Stop.
       For full logs, run 'nix log /nix/store/x1d79ld8jxqdla5zw2b47d2sl87mf56k-
↪icat.drv'.
```

The missing dependency error is solved, but there is now another problem: `make: *** No rule to make target 'install'. Stop.`

installPhase

`stdenv` is automatically working with the `Makefile` that comes with `icat`. The console output shows that `configure` and `make` are executed without issue, so the `icat` binary is compiling successfully.

The failure occurs when the `stdenv` attempts to run `make install`. The `Makefile` included in the project happens to lack an `install` target. The `README` in the `icat` repository only mentions using `make` to build the tool, leaving the installation step up to users.

To add this step to your derivation, use the `installPhase` attribute[98]. It contains a list of command strings that are executed to perform the installation.

Because `make` finishes successfully, the `icat` executable is available in the build directory. You only need to copy it from there to the output directory.

In Nix, the output directory is stored in the `$out` variable. That variable is accessible in the derivation's `builder` execution environment[99]. Create a `bin` directory within the `$out` directory and copy the `icat` binary there:

```
1  # icat.nix
2  {
3    stdenv,
4    fetchFromGitHub,
5    imlib2,
6    xorg,
7  }:
8
9  stdenv.mkDerivation {
10   pname = "icat";
11   version = "v0.5";
12
```

(continues on next page)

[98] https://nixos.org/manual/nixpkgs/stable/#ssec-install-phase
[99] https://nix.dev/manual/nix/2.19/language/derivations#builder-execution

(continued from previous page)

```
  src = fetchFromGitHub {
    owner = "atextor";
    repo = "icat";
    rev = "v0.5";
    sha256 = "0wyy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pzr99ljs9awy8ka";
  };

  buildInputs = [ imlib2 xorg.libX11 ];

  installPhase = ''
    mkdir -p $out/bin
    cp icat $out/bin
  '';
}
```

Phases and hooks

Nixpkgs `stdenv.mkDerivation` derivations are separated into phases[100]. Each is intended to control some aspect of the build process.

Earlier you observed how `stdenv.mkDerivation` expected the project's `Makefile` to have an `install` target, and failed when it didn't. To fix this, you defined a custom `installPhase` containing instructions for copying the `icat` binary to the correct output location, in effect installing it. Up to that point, the `stdenv.mkDerivation` automatically determined the `buildPhase` information for the `icat` package.

During derivation realisation, there are a number of shell functions ("hooks", in Nixpkgs) which may execute in each derivation phase. Hooks do things like set variables, source files, create directories, and so on.

These are specific to each phase, and run both before and after that phase's execution. They modify the build environment for common operations during the build.

It's good practice when packaging software with Nix to include calls to these hooks in the derivation phases you define, even when you don't make direct use of them. This facilitates easy overriding[101] of specific parts of the derivation later. And it keeps the code tidy and makes it easier to read.

Adjust your `installPhase` to call the appropriate hooks:

```
# icat.nix

# ...

  installPhase = ''
    runHook preInstall
    mkdir -p $out/bin
    cp icat $out/bin
    runHook postInstall
  '';

# ...
```

2.3.6 A successful build

Running the `nix-build` command once more will finally do what you want, repeatably. Call `ls` in the local directory to find a `result` symlink to a location in the Nix store:

[100] https://nixos.org/manual/nixpkgs/stable/#sec-stdenv-phases
[101] https://nixos.org/manual/nixpkgs/stable/#chap-overrides

```
$ ls
default.nix hello.nix icat.nix result
```

`result/bin/icat` is the executable built previously. Success!

2.3.7 References

- Nixpkgs Manual - Standard Environment[102]

2.3.8 Next steps

- *Package parameters and overrides with callPackage* (page 52)
- *Dependencies in the development shell* (page 135)
- *Automatic environment activation with direnv* (page 134)
- *Setting up a Python development environment* (page 138)
- Add your own new packages to Nixpkgs[103]
  - *How to contribute* (page 165)
  - *How to get help* (page 167)

2.4 Package parameters and overrides with `callPackage`

Nix ships with a special-purpose programming language for creating packages and configurations: the Nix language. It is used to build the Nix package collection, known as *Nixpkgs*.

Being purely functional, the Nix language allows declaring custom functions to abstract over common patterns. One of the most prominent patterns in Nixpkgs is parametrisation of package recipes.

2.4.1 Overview

Nixpkgs is a sizeable software project on its own, with coding conventions and idioms that have emerged over the years. It has established a convention[104] of composing parameterised packages with automatic settings through a function named `callPackage`[105]. This tutorial shows how to use it and why it's beneficial.

What will you learn?

- Using `callPackage` to invoke package recipes that follow Nixpkgs conventions
- Overriding package parameters
- Creating interdependent package sets

What do you need?

- Familiarity with the *Nix language* (page 12)
- First experience with *packaging existing software* (page 40)

How long does it take?

- 45 minutes

[102] https://nixos.org/manual/nixpkgs/unstable/#part-stdenv
[103] https://github.com/NixOS/nixpkgs/blob/master/CONTRIBUTING.md
[104] https://github.com/nixos/nixpkgs/commit/d17f0f9cbca38fabb71624f069cd4c0d6feace92
[105] https://github.com/NixOS/nixpkgs/commit/fd268b4852d39c18e604c584dd49a611dc795a9b

2.4.2 Automatic function calls

Create a new file `hello.nix`, which could be a typical package recipe as found in Nixpkgs: A function that takes an attribute set, with attributes corresponding to derivations in the top-level package set, and returns a derivation.

Listing 1: hello.nix

```
{ writeShellScriptBin }:
writeShellScriptBin "hello" ''
  echo "Hello, world!"
''
```

Detailed explanation

`hello.nix` declares a function which takes as argument an attribute set with one element `writeShellScriptBin`. `writeShellScriptBin`[106] is a function that happens to exist in Nixpkgs, a build helper[107] that returns a derivation. The derivation output in this case contains an executable shell script in `$out/bin/hello` that prints "Hello world" when run.

Now create a file `default.nix` with the following contents:

Listing 2: default.nix

```
let
  pkgs = import <nixpkgs> { };
in
pkgs.callPackage ./hello.nix { }
```

Realise the derivation in `default.nix` and run the executable that is produced:

```
$ nix-build
$ ./result/bin/hello
Hello, world!
```

The argument `writeShellScriptBin` gets filled in automatically when the function in `hello.nix` is evaluated. For every attribute in the function's argument, `callPackage` passes an attribute from the `pkgs` attribute set if it exists.

It may appear cumbersome to create the extra file `hello.nix` for the package in such a simple setup. We have done so because this is exactly how Nixpkgs is organised: Every package recipe is a file that declares a function. This function takes as arguments the package's dependencies.

2.4.3 Parameterised builds

Change the `default.nix` to produce an attribute set of derivations, with the attribute `hello` containing the original derivation:

Listing 3: default.nix

```
let
  pkgs = import <nixpkgs> { };
in
{
  hello = pkgs.callPackage ./hello.nix { };
}
```

When building the attribute `hello`, by accessing it with the `-A` / `--attr` option[108], the result will be the same as before:

[106] https://nixos.org/manual/nixpkgs/unstable/#trivial-builder-writeShellScriptBin
[107] https://nixos.org/manual/nixpkgs/unstable/#part-builders
[108] https://nix.dev/manual/nix/2.19/command-ref/nix-build#opt-attr

```
$ nix-build -A hello
$ ./result/bin/hello
Hello, world!
```

Also change `hello.nix` to add an additional parameter `audience` with default value `"world"`:

Listing 4: hello.nix

```
{
  writeShellScriptBin,
  audience ? "world",
}:
writeShellScriptBin "hello" ''
  echo "Hello, ${audience}!"
''
```

This also does not change the result.

Things get more interesting when changing `default.nix` to make use of this new argument. Pass the parameter `audience` in the second argument to `callPackage`:

Listing 5: default.nix

```
 let
   pkgs = import <nixpkgs> { };
 in
 {
-  hello = pkgs.callPackage ./hello.nix { };
+  hello = pkgs.callPackage ./hello.nix { audience = "people"; };
 }
```

This attribute is passed on to the argument of the function defined in `hello.nix`: The same syntax can also be used to explicitly set the automatically discovered arguments, such as `writeShellScriptBin`, but that doesn't make sense here.

Try it out:

```
$ nix-build -A hello
$ ./result/bin/hello
Hello, people!
```

This pattern is used widely in Nixpkgs: For example, functions which represent Go programs often have a parameter `buildGoModule`. It is common to find expressions like `callPackage ./go-program.nix { buildGoModule = buildGo116Module; }` to change the default Go compiler version. Nixpkgs is therefore not simply a huge library of pre-configured packages, but a collection of functions – package *recipes* – for customising packages and even entire ecosystems (for example "All Python packages using my custom interpreter") on the fly without duplicating code.

2.5 Overrides

`callPackage` adds more convenience by allowing parameters to be customised *after the fact* using the returned derivation's `override` function.

Add a third attribute `hello-folks` to `default.nix` and set it to `hello.override` called with a new value for `audience`:

Listing 6: default.nix

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

(continues on next page)

(continued from previous page)

```
 in
-{
+rec {
   hello = pkgs.callPackage ./hello.nix { audience = "people"; };
+  hello-folks = hello.override { audience = "folks"; };
 }
```

Note

The resulting attribute set is now recursive (by the keyword `rec`). That is, attribute values can refer to names from within the same attribute set.

`override` passes `audience` to the original function in `hello.nix` - it *overrides* whatever arguments have been passed in the original `callPackage` that produced the derivation `hello`. All the other parameters will remain the same. This is especially useful and can be often found on packages that provide many options to customise a package.

Building `hello-folks` attribute and running the resulting executable will again produce a new version of the script:

```
$ nix-build -A hello-folks
$ ./result/bin/hello
Hello, folks!
```

A real-world example is the `neovim`[109] package recipe, which has overridable arguments such as `extraLuaPackages`, `extraPythonPackages`, or `withRuby`. Currently these parameters are only discoverable by reading the source code, which can be found by following the link to Source on search.nixos.org/packages[110].

2.5.1 Interdependent package sets

You can actually create your own version of `callPackage`! This comes in handy for package sets where the recipes depend on each other.

Note

The following examples do not show the "called" files, as they are not necessary for understanding the principle.

Consider the following recursive attribute set of derivations:

Listing 7: default.nix

```nix
let
  pkgs = import <nixpkgs> { };
in
rec {
  a = pkgs.callPackage ./a.nix { };
  b = pkgs.callPackage ./b.nix { inherit a; };
  c = pkgs.callPackage ./c.nix { inherit b; };
  d = pkgs.callPackage ./d.nix { };
  e = pkgs.callPackage ./e.nix { inherit c d; };
}
```

Note

[109] https://search.nixos.org/packages?show=neovim
[110] https://search.nixos.org/packages

Here, `inherit a;` is equivalent to `a = a;`.

Previously declared derivations are passed as arguments to other derivations through `callPackage`.

In this case you have to remember to manually specify all arguments required by each package in the respective *Nix file* that are not in Nixpkgs. If `./b.nix` requires an argument `a` but there is no `pkgs.a`, the function call will produce an error. This can become quite tedious quickly, especially for larger package sets.

Use `lib.callPackageWith` to create your own `callPackage` based on an attribute set.

Listing 8: default.nix

```
let
  pkgs = import <nixpkgs> { };
  callPackage = pkgs.lib.callPackageWith (pkgs // packages);
  packages = {
    a = callPackage ./a.nix { };
    b = callPackage ./b.nix { };
    c = callPackage ./c.nix { };
    d = callPackage ./d.nix { };
    e = callPackage ./e.nix { };
  };
in
packages
```

This requires some explanation.

First of all note that instead of a recursive attribute set, the names we operate on are now assigned in a `let` binding. It has the same property as recursive sets: Names on the left can be used in expressions on the right of the equal sign (=). This is how we can refer to `packages` when we merge its contents with the pre-existing attribute set `pkgs` using the `//` operator.

Your custom `callPackages` now makes available all the attributes in `pkgs` *and* `packages` to the called package function (the same names from `packages` taking precedence), and `packages` is being built up recursively with each call.

The last bit may make your head spin. This construction is only possible because the Nix language is lazily evaluated. That is, values are only computed when they are actually needed. It allows passing `packages` around without having fully defined it.

Each package's dependencies are now implicit at this level (they are still explicit in each of the package files), and `callPackage` resolves them *automagically*. This relieves you from dealing with them manually, and precludes configuration errors that may only surface late into a lengthy build process.

Of course this small example is still manageable in the original form. And the implicitly recursive variant can obscure the structure for software developers not familiar with lazy evaluation, making it harder to read for them than it was before. But this benefit really pays off for large constructions, where it is the amount of code that would obscure the structure, and where manual modifications would become cumbersome and error-prone.

2.5.2 Summary

Using `callPackage` does not only follow Nixpkgs conventions, which makes your code easier to follow for experienced Nix users, but it also gives you some benefits for free:

1. Parametrized builds
2. Overrideable builds
3. Concise implementation of interdependent package sets

2.5.3 References

- Nixpkgs manual: `callPackageWith`[111]

2.5.4 Next steps

- *Working with local files* (page 57) - learn to package your own projects with Nix
- *Module system deep dive* (page 75) - learn to wield the functional programming magic behind NixOS

2.6 Working with local files

To build a local project in a Nix derivation, source files must be accessible to its `builder` executable[112]. Since by default, the `builder` runs in an isolated environment[113] that only allows reading from the Nix store, the Nix language has built-in features to copy local files to the store and expose the resulting store paths.

Using these features directly can be tricky however:

- Coercion of paths to strings, such as the wide-spread pattern of `src = ./.`, makes the derivation dependent on the name of the current directory. Furthermore, it always adds the entire directory to the store, including unneeded files, which causes unnecessary new builds when they change.
- The `builtins.path`[114] function (and equivalently `lib.sources.cleanSourceWith`[115]) can address these problems. However, it's often hard to express the desired path selection using the `filter` function interface.

In this tutorial you'll learn how to use the Nixpkgs `lib.fileset` library[116] to work with local files in derivations. It abstracts over built-in functionality and offers a safer and more convenient interface.

2.6.1 File sets

A *file set* is a data type representing a collection of local files. File sets can be created, composed, and manipulated with the various functions of the library.

You can explore and learn about the library with `nix repl`[117]:

```
$ nix repl -f channel:nixos-23.11
...
nix-repl> fs = lib.fileset
```

The `trace`[118] function pretty-prints the files included in a given file set:

```
nix-repl> fs.trace ./. null
trace: /home/user (all files in directory)
null
```

All functions that expect a file set for an argument can also accept a path[119]. Such path arguments are then implicitly turned into sets[120] that contain *all* files under the given path. In the previous trace this is indicated by `(all files in directory)`.

[111] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.customisation.callPackageWith
[112] https://nix.dev/manual/nix/stable/language/derivations#attr-builder
[113] https://nix.dev/manual/nix/stable/command-ref/conf-file.html#conf-sandbox
[114] https://nix.dev/manual/nix/stable/language/builtins.html#builtins-path
[115] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.sources.cleanSourceWith
[116] https://nixos.org/manual/nixpkgs/stable/#sec-functions-library-fileset
[117] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-repl
[118] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.trace
[119] https://nix.dev/manual/nix/stable/language/values#type-path
[120] https://nixos.org/manual/nixpkgs/stable/#sec-fileset-path-coercion

Tip

The `trace` function pretty-prints its first argument and returns its second argument. But since you often just need the pretty-printing in `nix repl`, you can omit the second argument:

```
nix-repl> fs.trace ./.
trace: /home/user (all files in directory)
«lambda @ /nix/store/1czr278x24s3bl6qdnifpvm5z03wfi2p-nixpkgs-src/lib/fileset/
↪default.nix:555:8»
```

Even though file sets conceptually contain local files, these files are *never* added to the Nix store unless explicitly requested. Therefore you don't have to worry as much about accidentally copying secrets into the world-readable store.

In this example, although we pretty-printed the home directory, no files were copied. This is in contrast to coercion of paths to strings such as in `"${./.}"`, which copies the whole directory to the Nix store on evaluation!

Warning

When using the `flakes` and `nix-command` experimental features[121], a local directory within a Flake is always copied into the Nix store *completely* unless it is a Git repository!

This implicit coercion also works for files:

```
$ touch some-file
```

```
nix-repl> fs.trace ./some-file
trace: /home/user
trace: - some-file (regular)
```

In addition to the included file, this also prints its file type[122].

2.6.2 Example project

To further experiment with the library, make a sample project. Create a new directory, enter it, and set up `npins` to pin the Nixpkgs dependency:

```
$ mkdir fileset
$ cd fileset
$ nix-shell -p npins --run "npins init --bare; npins add github nixos nixpkgs --
↪branch nixos-23.11"
```

Then create a `default.nix` file with the following contents:

Listing 9: default.nix

```
{
  system ? builtins.currentSystem,
  sources ? import ./npins,
}:
let
  pkgs = import sources.nixpkgs {
    config = { };
    overlays = [ ];
```

(continues on next page)

[121] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-flake
[122] https://nix.dev/manual/nix/stable/language/builtins.html#builtins-readFileType

(continued from previous page)

```
    inherit system;
  };
in
pkgs.callPackage ./build.nix { }
```

Add two source files to work with:

```
$ echo hello > hello.txt
$ echo world > world.txt
```

2.6.3 Adding files to the Nix store

Files in a given file set can be added to the Nix store with `toSource`[123]. The argument to this function requires a `root` attribute to determine which source directory to copy to the store. Only the files in the `fileset` attribute are included in the result.

Define `build.nix` as follows:

Listing 10: build.nix

```
{ stdenv, lib }:
let
  fs = lib.fileset;
  sourceFiles = ./hello.txt;
in

fs.trace sourceFiles

stdenv.mkDerivation {
  name = "fileset";
  src = fs.toSource {
    root = ./.;
    fileset = sourceFiles;
  };
  postInstall = ''
    mkdir $out
    cp -v hello.txt $out
  '';
}
```

The call to `fs.trace` prints the file set that will be used as a derivation input.

Try building it:

Note

It will take a while to fetch Nixpkgs the first time around.

```
$ nix-build
trace: /home/user/fileset
trace: - hello.txt (regular)
this derivation will be built:
  /nix/store/3ci6avmjaijx5g8jhb218i183xi7bi2n-fileset.drv
...
```

(continues on next page)

[123] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.toSource

(continued from previous page)

```
'hello.txt' -> '/nix/store/sa4g6h13v0zbpfw6pzva860kp5aks44n-fileset/hello.txt'
...
/nix/store/sa4g6h13v0zbpfw6pzva860kp5aks44n-fileset
```

But the real benefit of the file set library comes from its facilities for composing file sets in different ways.

2.6.4 Difference

To be able to copy both files `hello.txt` and `world.txt` to the output, add the whole project directory as a source again:

Listing 11: build.nix

```
 { stdenv, lib }:
 let
   fs = lib.fileset;
-  sourceFiles = ./hello.txt;
+  sourceFiles = ./.;
 in

 fs.trace sourceFiles

 stdenv.mkDerivation {
   name = "fileset";
   src = fs.toSource {
     root = ./.;
     fileset = sourceFiles;
   };
   postInstall = ''
     mkdir $out
-    cp -v hello.txt $out
+    cp -v {hello,world}.txt $out
   '';
 }
```

This will work as expected:

```
$ nix-build
trace: /home/user/fileset (all files in directory)
this derivation will be built:
  /nix/store/fsihp8872vv9ngbkc7si5jcbigs81727-fileset.drv
...
'hello.txt' -> '/nix/store/wmsxfgbylagmf033nkazr3qfc96y7mwk-fileset/hello.txt'
'world.txt' -> '/nix/store/wmsxfgbylagmf033nkazr3qfc96y7mwk-fileset/world.txt'
...
/nix/store/wmsxfgbylagmf033nkazr3qfc96y7mwk-fileset
```

However, if you run `nix-build` again, the output path will be different!

```
$ nix-build
trace: /home/user/fileset (all files in directory)
this derivation will be built:
  /nix/store/nlh7ismrf27xsnl3m20vfz6rvwlbbbca-fileset.drv
...
'hello.txt' -> '/nix/store/xknflcvjaa8dj6a6vkg629zmcrgz10rh-fileset/hello.txt'
'world.txt' -> '/nix/store/xknflcvjaa8dj6a6vkg629zmcrgz10rh-fileset/world.txt'
...
```

(continues on next page)

(continued from previous page)

```
/nix/store/xknflcvjaa8dj6a6vkg629zmcrgz10rh-fileset
```

The problem here is that `nix-build` by default creates a `result` symlink in the working directory, which points to the store path just produced:

```
$ ls -l result
result -> /nix/store/xknflcvjaa8dj6a6vkg629zmcrgz10rh-fileset
```

Since `src` refers to the whole directory, and its contents change when `nix-build` succeeds, Nix will have to start over every time.

Note

This will also happen without the file set library, e.g. when setting `src = ./.;` directly.

The `difference`[124] function subtracts one file set from another. The result is a new file set that contains all files from the first argument that aren't in the second argument.

Use it to filter out `./result` by changing the `sourceFiles` definition:

Listing 12: build.nix

```
 { stdenv, lib }:
 let
   fs = lib.fileset;
-  sourceFiles = ./.;
+  sourceFiles = fs.difference ./. ./result;
 in
```

Building this, the file set library will specify which files are taken from the directory:

```
$ nix-build
trace: /home/user/fileset
trace: - build.nix (regular)
trace: - default.nix (regular)
trace: - hello.txt (regular)
trace: - npins (all files in directory)
trace: - world.txt (regular)
this derivation will be built:
  /nix/store/zr19bv51085zz005yk7pw4s9sglmafvn-fileset.drv
...
'hello.txt' -> '/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset/hello.txt'
'world.txt' -> '/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset/world.txt'
...
/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset
```

An attempt to repeat the build will re-use the existing store path:

```
$ nix-build
trace: /home/user/fileset
trace: - build.nix (regular)
trace: - default.nix (regular)
trace: - hello.txt (regular)
trace: - npins (all files in directory)
trace: - world.txt (regular)
/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset
```

[124] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.difference

2.6.5 Missing files

Removing the `./result` symlink creates a new problem, though:

```
$ rm result
$ nix-build
error: lib.fileset.difference: Second argument (negative set)
  (/home/user/fileset/result) is a path that does not exist.
  To create a file set from a path that may not exist, use `lib.fileset.
↪maybeMissing`.
```

Follow the instructions in the error message, and use `maybeMissing`[125] to create a file set from a path that may not exist (in which case the file set will be empty):

Listing 13: build.nix

```
 { stdenv, lib }:
 let
   fs = lib.fileset;
-  sourceFiles = fs.difference ./. ./result;
+  sourceFiles = fs.difference ./. (fs.maybeMissing ./result);
 in
```

This now works, using the whole directory since `./result` is not present:

```
$ nix-build
trace: /home/user/fileset (all files in directory)
this derivation will be built:
  /nix/store/zr19bv51085zz005yk7pw4s9sglmafvn-fileset.drv
...
/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset
```

Another build attempt will produce a different trace, but the same output path:

```
$ nix-build
trace: /home/user/fileset
trace: - build.nix (regular)
trace: - default.nix (regular)
trace: - hello.txt (regular)
trace: - npins (all files in directory)
trace: - world.txt (regular)
/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset
```

2.6.6 Union (explicitly exclude files)

There is still a problem: Changing *any* of the included files causes the derivation to be built again, even though it doesn't depend on those files.

Append an empty line to `build.nix`:

```
$ echo >> build.nix
```

Again, Nix will start from scratch:

```
$ nix-build
trace: /home/user/fileset
trace: - default.nix (regular)
trace: - npins (all files in directory)
```

(continues on next page)

[125] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.maybeMissing

(continued from previous page)

```
trace: - build.nix (regular)
trace: - string.txt (regular)
this derivation will be built:
  /nix/store/zmgpqlpfz2jq0w9rdacsnpx8ni4n77cn-filesets.drv
...
/nix/store/6pffjljjy3c7kla60nljk3fad4q4kkzn-filesets
```

One way to fix this is to use `unions`[126].

Create a file set containing a union of the files to exclude (`fs.unions [ ... ]`), and subtract it (`difference`) from the complete directory (`./.`):

Listing 14: build.nix

```
  sourceFiles =
    fs.difference
      ./.
      (fs.unions [
        (fs.maybeMissing ./result)
        ./default.nix
        ./build.nix
        ./npins
      ]);
```

This will work as expected:

```
$ nix-build
trace: /home/user/fileset
trace: - hello.txt (regular)
trace: - world.txt (regular)
this derivation will be built:
  /nix/store/gr2hw3gdjc28fmv0as1ikpj7lya4r51f-fileset.drv
...
/nix/store/ckn40y7hgqphhbhyrq64h9r6rvdh973r-fileset
```

Changing any of the excluded files now doesn't necessarily cause a new build anymore:

```
$ echo >> build.nix
```

```
$ nix-build
trace: /home/user/fileset
trace: - hello.txt (regular)
trace: - world.txt (regular)
/nix/store/ckn40y7hgqphhbhyrq64h9r6rvdh973r-fileset
```

2.6.7 Filter

The `fileFilter`[127] function allows filtering file sets such that each included file satisfies the given criteria.

Use it to select all files with a name ending in `.nix`:

Listing 15: build.nix

```
  sourceFiles =
    fs.difference
```

(continues on next page)

[126] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.unions
[127] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.fileFilter

(continued from previous page)

```
       ./.
       (fs.unions [
         (fs.maybeMissing ./result)
-        ./default.nix
-        ./build.nix
+        (fs.fileFilter (file: file.hasExt "nix") ./.)
         ./npins
       ]);
```

This does not change the result, even if we add a new `.nix` file.

```
$ nix-build
trace: /home/user/fileset
trace: - hello.txt (regular)
trace: - world.txt (regular)
/nix/store/ckn40y7hgqphhbhyrq64h9r6rvdh973r-fileset
```

Notably, the approach of using `difference ./.` explicitly selects the files to *exclude*, which means that new files added to the source directory are included by default. Depending on your project, this might be a better fit than the alternative in the next section.

2.6.8 Union (explicitly include files)

To contrast the previous approach, `unions` can also be used to select only the files to *include*. This means that new files added to the current directory would be ignored by default.

Create some additional files:

```
$ mkdir src
$ touch build.sh src/select.{c,h}
```

Then create a file set from only the files to be included explicitly:

Listing 16: build.nix

```
{ stdenv, lib }:
let
  fs = lib.fileset;
  sourceFiles = fs.unions [
    ./hello.txt
    ./world.txt
    ./build.sh
    (fs.fileFilter
      (file: file.hasExt "c" || file.hasExt "h")
      ./src
    )
  ];
in

fs.trace sourceFiles

stdenv.mkDerivation {
  name = "fileset";
  src = fs.toSource {
    root = ./.;
    fileset = sourceFiles;
  };
  postInstall = ''
```

(continues on next page)

(continued from previous page)

```
    cp -vr . $out
  '';
}
```

The `postInstall` script is simplified to rely on the sources to be pre-filtered appropriately:

```
$ nix-build
trace: /home/user/fileset
trace: - build.sh (regular)
trace: - hello.txt (regular)
trace: - src (all files in directory)
trace: - world.txt (regular)
this derivation will be built:
  /nix/store/sjzkn07d6a4qfp60p6dc64pzvmmdafff-fileset.drv
...
'.' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset'
'./build.sh' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/build.sh'
'./hello.txt' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/hello.txt'
'./world.txt' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/world.txt'
'./src' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/src'
'./src/select.c' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/src/
↪select.c'
'./src/select.h' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/src/
↪select.h'
...
/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset
```

Only the specified files are used, even when a new one is added:

```
$ touch src/select.o README.md

$ nix-build
trace: - build.sh (regular)
trace: - hello.txt (regular)
trace: - src
trace:   - select.c (regular)
trace:   - select.h (regular)
trace: - world.txt (regular)
/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset
```

2.6.9 Matching files tracked by Git

If a directory is part of a Git repository, passing it to `gitTracked`[128] gives you a file set that only includes files tracked by Git.

Create a local Git repository and add all files except `src/select.o` and `./result` to it:

```
$ git init
Initialized empty Git repository in /home/user/fileset/.git/
$ git add -A
$ git reset src/select.o result
```

Re-use this selection of files with `gitTracked`:

[128] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.gitTracked

Listing 17: build.nix

```
  sourceFiles = fs.gitTracked ./.;
```

Build it again:

```
$ nix-build
warning: Git tree '/home/user/fileset' is dirty
trace: /home/vg/src/nix.dev/fileset
trace: - README.md (regular)
trace: - build.nix (regular)
trace: - build.sh (regular)
trace: - default.nix (regular)
trace: - hello.txt (regular)
trace: - npins (all files in directory)
trace: - src
trace:   - select.c (regular)
trace:   - select.h (regular)
trace: - world.txt (regular)
this derivation will be built:
  /nix/store/p9aw3fl5xcjbgg9yagykywvskzgrmk5y-fileset.drv
...
/nix/store/cw4bza1r27iimzrdbfl4yn5xr36d6k5l-fileset
```

This includes too much though, as not all of these files are needed to build the derivation as originally intended.

Note

When using the `flakes` and `nix-command` experimental features[129], this function isn't needed, because `nix build` by default only allows access to files tracked by Git. However, in order to provide the same developer experience for stable Nix, use of this function is nevertheless recommended.

2.6.10 Intersection

This is where `intersection` comes in. It allows creating a file set that consists only of files that are in *both* of two given file sets.

Select all files that are both tracked by Git *and* relevant for the build:

Listing 18: build.nix

```
  sourceFiles =
    fs.intersection
      (fs.gitTracked ./.)
      (fs.unions [
        ./hello.txt
        ./world.txt
        ./build.sh
        ./src
      ]);
```

This will produce the same output as in the other approach and therefore re-use a previous build result:

```
$ nix-build
warning: Git tree '/home/user/fileset' is dirty
```

(continues on next page)

[129] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-flake

(continued from previous page)

```
trace: - build.sh (regular)
trace: - hello.txt (regular)
trace: - src
trace:   - select.c (regular)
trace:   - select.h (regular)
trace: - world.txt (regular)
/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset
```

2.6.11 Conclusion

We have shown some examples on how to use all of the fundamental file set functions. For more complex use cases, they can be composed as needed.

For the complete list and more details, see the `lib.fileset` reference documentation[130].

2.7 Cross compilation

Nixpkgs offers powerful tools to cross-compile software for various system types.

2.7.1 What do you need?

- Experience using C compilers
- Basic knowledge of the *Nix language* (page 12)

2.7.2 Platforms

When compiling code, we can distinguish between the **build platform**, where the executable is *built*, and the **host platform**, where the compiled executable *runs*.[139]

Native compilation is the special case where those two platforms are the same. **Cross compilation** is the general case where those two platforms are not.

Cross compilation is needed when the host platform has limited resources (such as CPU) or when it's not easily accessible for development.

The `nixpkgs` package collection has world-class support for cross compilation, after many years of hard work by the Nix community.

2.7.3 What's a target platform?

There is a third concept for a platform we call a **target platform**.

The target platform is relevant to cases where you want to build a compiler binary. In such cases, you would build a compiler on the *build platform*, run it to compile code on the *host platform*, and run the final executable on the *target platform*.

Since this is rarely needed, we will assume that the target is identical to the host.

2.7.4 Determining the host platform config

The build platform is determined automatically by Nix during the configure phase.

The host platform is best determined by running this command on the host platform:

[130] https://nixos.org/manual/nixpkgs/stable/#sec-functions-library-fileset
[139] Terminology for cross compilation platforms differs between build systems. We have chosen to follow autoconf terminology[140].
[140] https://www.gnu.org/software/autoconf/manual/autoconf-2.69/html_node/Hosts-and-Cross_002dCompilation.html

```
$ $(nix-build '<nixpkgs>' -I nixpkgs=channel:nixos-23.11 -A gnu-config)/config.
↪guess
aarch64-unknown-linux-gnu
```

In case this is not possible (for example, when the host platform is not easily accessible for development), the platform config has to be constructed manually via the following template:

```
<cpu>-<vendor>-<os>-<abi>
```

This string representation is used in `nixpkgs` for historic reasons.

Note that `<vendor>` is often `unknown` and `<abi>` is optional. There's also no unique identifier for a platform, for example `unknown` and `pc` are interchangeable (which is why the script is called `config.guess`).

If you can't install Nix, find a way to run `config.guess` (usually comes with the autoconf package) from the OS you're able to run on the host platform.

Some other common examples of platform configs:

- aarch64-apple-darwin14
- aarch64-pc-linux-gnu
- x86_64-w64-mingw32
- aarch64-apple-ios

Note

macOS/Darwin is a special case, as not the whole OS is open-source. It's only possible to cross compile between `aarch64-darwin` and `x86_64-darwin`. `aarch64-darwin` support was recently added, so cross compilation is barely tested.

2.7.5 Choosing the host platform with Nix

`nixpkgs` comes with a set of predefined host platforms for cross compilation called `pkgsCross`.

It is possible to explore them in `nix repl`:

Note

Starting with Nix 2.19[131], `nix repl` requires the `-f` / `--file` flag:

```
$ nix repl -f '<nixpkgs>' -I nixpkgs=channel:nixos-23.11
```

```
$ nix repl '<nixpkgs>' -I nixpkgs=channel:nixos-23.11
Welcome to Nix 2.18.1. Type :? for help.

Loading '<nixpkgs>'...
Added 14200 variables.

nix-repl> pkgsCross.<TAB>
pkgsCross.aarch64-android            pkgsCross.musl-power
pkgsCross.aarch64-android-prebuilt   pkgsCross.musl32
pkgsCross.aarch64-darwin             pkgsCross.musl64
pkgsCross.aarch64-embedded           pkgsCross.muslpi
pkgsCross.aarch64-multiplatform      pkgsCross.or1k
```

(continues on next page)

[131] https://nix.dev/manual/nix/latest/release-notes/rl-2.19

(continued from previous page)

```
pkgsCross.aarch64-multiplatform-musl  pkgsCross.pogoplug4
pkgsCross.aarch64be-embedded          pkgsCross.powernv
pkgsCross.amd64-netbsd                pkgsCross.ppc-embedded
pkgsCross.arm-embedded                pkgsCross.ppc64
pkgsCross.armhf-embedded              pkgsCross.ppc64-musl
pkgsCross.armv7a-android-prebuilt     pkgsCross.ppcle-embedded
pkgsCross.armv7l-hf-multiplatform     pkgsCross.raspberryPi
pkgsCross.avr                         pkgsCross.remarkable1
pkgsCross.ben-nanonote                pkgsCross.remarkable2
pkgsCross.fuloongminipc               pkgsCross.riscv32
pkgsCross.ghcjs                       pkgsCross.riscv32-embedded
pkgsCross.gnu32                       pkgsCross.riscv64
pkgsCross.gnu64                       pkgsCross.riscv64-embedded
pkgsCross.i686-embedded               pkgsCross.scaleway-c1
pkgsCross.iphone32                    pkgsCross.sheevaplug
pkgsCross.iphone32-simulator          pkgsCross.vc4
pkgsCross.iphone64                    pkgsCross.wasi32
pkgsCross.iphone64-simulator          pkgsCross.x86_64-embedded
pkgsCross.mingw32                     pkgsCross.x86_64-netbsd
pkgsCross.mingwW64                    pkgsCross.x86_64-netbsd-llvm
pkgsCross.mmix                        pkgsCross.x86_64-unknown-redox
pkgsCross.msp430
```

These attribute names for cross compilation packages have been chosen somewhat freely over the course of time. They usually do not match the corresponding platform config string.

You can retrieve the platform string from `pkgsCross.<platform>.stdenv.hostPlatform.config`:

```
nix-repl> pkgsCross.aarch64-multiplatform.stdenv.hostPlatform.config
"aarch64-unknown-linux-gnu"
```

If the host platform you seek hasn't been defined yet, please contribute it upstream[132].

2.7.6 Specifying the host platform

The mechanism for setting up cross compilation works as follows:

1. Take the build platform configuration and apply it to the current package set, called `pkgs` by convention.

   The build platform is implied in `pkgs = import <nixpkgs> {}` to be the current system. This produces a build environment `pkgs.stdenv` with all the dependencies present to compile on the build platform.

2. Apply the appropriate host platform configuration to all the packages in `pkgsCross`.

   Taking `pkgs.pkgsCross.<host>.hello` will produce the package `hello` compiled on the build platform to run on the `<host>` platform.

There are multiple equivalent ways to access packages targeted to the host platform.

1. Explicitly pick the host platform package from within the build platform environment:

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23.
↪11";
  pkgs = import nixpkgs {};
in
pkgs.pkgsCross.aarch64-multiplatform.hello
```

2. Pass the host platform to `crossSystem` when importing `nixpkgs`. This configures `nixpkgs` such that all its packages are build for the host platform:

[132] https://github.com/NixOS/nixpkgs/blob/master/lib/systems/examples.nix

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23.11";
  pkgs = import nixpkgs { crossSystem = { config = "aarch64-unknown-linux-gnu"; }; };
in
pkgs.hello
```

Equivalently, you can pass the host platform as an argument to `nix-build`:

```
$ nix-build '<nixpkgs>' -I nixpkgs=channel:nixos-23.11 \
  --arg crossSystem '{ config = "aarch64-unknown-linux-gnu"; }' \
  -A hello
```

2.7.7 Cross compiling for the first time

To cross compile a package like hello[133], pick the platform attribute — `aarch64-multiplatform` in our case — and run:

```
$ nix-build '<nixpkgs>' -I nixpkgs=channel:nixos-23.11 \
  -A pkgsCross.aarch64-multiplatform.hello
...
/nix/store/1dx87l5rav8679lqigf9xxkb7wvh2m4k-hello-aarch64-unknown-linux-gnu-2.12.1
```

Note

The hash of the package in the store path changes with the updates to the channel.

Search for a package[134] attribute name to find the one you're interested in building.

2.7.8 Real-world cross compiling of a Hello World example

To show off the power of cross compilation in Nix, let's build our own Hello World program by cross compiling it as static executables to `armv6l-unknown-linux-gnueabihf` and `x86_64-w64-mingw32` (Windows) platforms and run the resulting executable with an emulator[135].

Given we have a `cross-compile.nix`:

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23.11";
  pkgs = import nixpkgs {};

  # Create a C program that prints Hello World
  helloWorld = pkgs.writeText "hello.c" ''
    #include <stdio.h>

    int main (void)
    {
      printf ("Hello, world!\n");
      return 0;
    }
  '';

```

(continues on next page)

[133] https://www.gnu.org/software/hello/
[134] https://search.nixos.org/packages
[135] https://en.wikipedia.org/wiki/Emulator

(continued from previous page)

```
  # A function that takes host platform packages
  crossCompileFor = hostPkgs:
    # Run a simple command with the compiler available
    hostPkgs.runCommandCC "hello-world-cross-test" {} ''
      # Wine requires home directory
      HOME=$PWD

      # Compile our example using the compiler specific to our host platform
      $CC ${helloWorld} -o hello

      # Run the compiled program using user mode emulation (Qemu/Wine)
      # buildPackages is passed so that emulation is built for the build platform
      ${hostPkgs.stdenv.hostPlatform.emulator hostPkgs.buildPackages} hello > $out

      # print to stdout
      cat $out
    '';
in {
  # Statically compile our example using the two platform hosts
  rpi = crossCompileFor pkgs.pkgsCross.raspberryPi;
  windows = crossCompileFor pkgs.pkgsCross.mingwW64;
}
```

If we build this example and print both resulting derivations, we should see "Hello, world!" for each:

```
$ cat $(nix-build cross-compile.nix)
Hello, world!
Hello, world!
```

2.7.9 Developer environment with a cross compiler

In the *tutorial for declarative reproducible environments* (page 8), we looked at how Nix helps us provide tooling and system libraries for our project.

It's also possible to provide an environment with a compiler configured for **cross-compilation to static binaries using musl**.

Given we have a `shell.nix`:

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23.11";
  pkgs = (import nixpkgs {}).pkgsCross.aarch64-multiplatform;
in

# callPackage is needed due to https://github.com/NixOS/nixpkgs/pull/126844
pkgs.pkgsStatic.callPackage ({ mkShell, zlib, pkg-config, file }: mkShell {
  # these tools run on the build platform, but are configured to target the host␣
↪platform
  nativeBuildInputs = [ pkg-config file ];
  # libraries needed for the host platform
  buildInputs = [ zlib ];
}) {}
```

And `hello.c`:

```
#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
int main (void)
{
  printf ("Hello, world!\n");
  return 0;
}
```

We can cross compile it:

```
$ nix-shell --run '$CC hello.c -o hello' shell.nix
```

And confirm it's aarch64:

```
$ nix-shell --run 'file hello' shell.nix
hello: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), statically linked,
↪ with debug_info, not stripped
```

2.7.10 Next steps

- The official binary cache[136] has a limited number of binaries for packages that are cross compiled, so to save time recompiling, configure *your own binary cache and CI with GitHub Actions* (page 142).
- While many compilers in Nixpkgs support cross compilation, not all of them do.

  Additionally, supporting cross compilation is not trivial work and due to many possible combinations of what would need to be tested, some packages might not build.

  A detailed explanation how of cross compilation is implemented in Nix[137] can help with fixing those issues.
- The Nix community has a dedicated Matrix room[138] for help with cross compiling.

2.8 Module system

Much of the power in Nixpkgs and NixOS comes from the module system.

The module system is a Nix language library that enables you to

- Declare one attribute set using many separate Nix expressions.
- Impose type constraints on values in that attribute set.
- Define values for the same attribute in different Nix expressions and merge these values automatically according to their type.

These Nix expressions are called modules and must have a particular structure.

In this tutorial series you'll learn

- What a module is and how to create one.
- What options are and how to declare them.
- How to express dependencies between modules.

[136] https://cache.nixos.org
[137] https://nixos.org/manual/nixpkgs/stable/#chap-cross
[138] https://matrix.to/#/#cross-compiling:nixos.org

2.8.1 What do you need?

- Familiarity with data types and general programming concepts
- A *Nix installation* (page 1) to run the examples
- Intermediate proficiency in reading and writing the *Nix language* (page 12)

2.8.2 How long will it take?

This is a very long tutorial. Prepare for at least 3 hours of work.

A basic module

What is a module?

- A module is a function that takes an attribute set and returns an attribute set.
- It may declare options, telling which attributes are allowed in the final outcome.
- It may define values, for options declared by itself or other modules.
- When evaluated by the module system, it produces an attribute set based on the declarations and definitions.

The simplest possible module is a function that takes any attributes and returns an empty attribute set:

Listing 19: options.nix

```
{ ... }:
{
}
```

To define any values, the module system first has to know which ones are allowed. This is done by declaring *options* that specify which attributes can be set and used elsewhere.

Declaring options

Options are declared under the top-level `options` attribute with `lib.mkOption`[141].

Listing 20: options.nix

```
{ lib, ... }:
{
  options = {
    name = lib.mkOption { type = lib.types.str; };
  };
}
```

Note

The `lib` argument is passed automatically by the module system. This makes Nixpkgs library functions[142] available in each module's function body.

The ellipsis `...` is necessary because the module system can pass arbitrary arguments to modules.

The attribute `type` in the argument to `lib.mkOption` specifies which values are valid for an option. There are several types available under `lib.types`[143].

Here we have declared an option `name` of type `str`: The module system will expect a string when a value is defined.

[141] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.options.mkOption
[142] https://nixos.org/manual/nixpkgs/stable/#chap-functions
[143] https://nixos.org/manual/nixos/stable/#sec-option-types-basic

Now that we have declared an option, we would naturally want to give it a value.

Defining values

Options are set or *defined* under the top-level `config` attribute:

Listing 21: config.nix

```
{ ... }:
{
  config = {
    name = "Boaty McBoatface";
  };
}
```

In our option declaration, we created an option `name` with a string type. Here, in our option definition, we have set that same option to a string.

Option declarations and option definitions don't need to be in the same file. Which modules will contribute to the resulting attribute set is specified when setting up module system evaluation.

Evaluating modules

Modules are evaluated by `lib.evalModules`[144] from the Nixpkgs library. It takes an attribute set as an argument, where the `modules` attribute is a list of modules to merge and evaluate.

The output of `evalModules` contains information about all evaluated modules, and the final values appear in the attribute `config`.

Listing 22: default.nix

```
let
  pkgs = import <nixpkgs> {};
  result = pkgs.lib.evalModules {
    modules = [
      ./options.nix
      ./config.nix
    ];
  };
in
result.config
```

Here's a helper script to parse and evaluate our `default.nix` file with `nix-instantiate --eval`[145] and print the output as JSON:

Listing 23: eval.bash

```
nix-shell -p jq --run "nix-instantiate --eval --json --strict | jq"
```

As long as every definition has a corresponding declaration, evaluation will be successful. If there is an option definition that has not been declared, or the defined value has the wrong type, the module system will throw an error.

Running the script (`./eval.bash`) should show an output that matches what we have configured:

```
{
  "name": "Boaty McBoatface"
}
```

[144] https://nixos.org/manual/nixpkgs/stable/#module-system-lib-evalModules
[145] https://nix.dev/manual/nix/stable/command-ref/nix-instantiate

Module system deep dive

Or: *Wrapping the world in modules*

In this tutorial you will follow an extensive demonstration of how to wrap an existing API with Nix modules.

Overview

This tutorial follows @infinisil[146]'s presentation on modules[147] (source[148]) for participants of Summer of Nix[149] 2021.

It may help playing it alongside this tutorial to better keep track of changes to the code you will work on.

What will you learn?

You'll write modules to interact with the Google Maps API[150], declaring module options which represent map geometry, location pins, and more.

During the tutorial, you will first write some *incorrect* configurations, creating opportunities to discuss the resulting error messages and how to resolve them, particularly when discussing type checking.

What do you need?

You will use two helper scripts for this exercise. Download `map.sh` and `geocode.sh` to your working directory.

Warning

To run the examples in this tutorial, you will need a Google API key[151] in `$XDG_DATA_HOME/google-api/key`.

The empty module

Write the following into a file called `default.nix`:

Listing 24: default.nix

```
{ ... }:
{

}
```

Declaring options

We will need some helper functions, which will come from the Nixpkgs library[152], which is passed by the module system as `lib`:

Listing 25: default.nix

```
- { ... }:
+ { lib, ... }:
{
```

(continues on next page)

[146] https://github.com/infinisil
[147] https://infinisil.com/modules.mp4
[148] https://github.com/tweag/summer-of-nix-modules
[149] https://github.com/ngi-nix/summer-of-nix
[150] https://developers.google.com/maps/documentation/maps-static
[151] https://developers.google.com/maps/documentation/maps-static/start#before-you-begin
[152] https://github.com/NixOS/nixpkgs/tree/master/lib

(continued from previous page)

```
}
```

Using `lib.mkOption`[153], declare the `scripts.output` option to have the type `lines`:

Listing 26: default.nix

```
 { lib, ... }: {

+ options = {
+   scripts.output = lib.mkOption {
+     type = lib.types.lines;
+   };
+ };

 }
```

The `lines` type means that the only valid values are strings, and that multiple definitions should be joined with newlines.

Note

The name and attribute path of the option is arbitrary. Here we use `scripts`, because we will add another script later, and call this one `output`, because it will output the resulting map.

Evaluating modules

Write a new file `eval.nix` to call `lib.evalModules`[154] and evaluate the module in `default.nix`:

Listing 27: eval.nix

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
pkgs.lib.evalModules {
  modules = [
    ./default.nix
  ];
}
```

Run the following command:

Warning

This will result in an error.

```
nix-instantiate --eval eval.nix -A config.scripts.output
```

[153] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.options.mkOption
[154] https://nixos.org/manual/nixpkgs/unstable/#module-system-lib-evalModules

Detailed explanation

`nix-instantiate --eval`[155] parses and evaluates the Nix file at the specified path, and prints the result. `evalModules` produces an attribute set where the final configuration values appear in the `config` attribute. Therefore we evaluate the Nix expression in `eval.nix` at the attribute path[156] `config.scripts.output`.

The error message indicates that the `scripts.output` option is used but not defined: a value must be set for the option before accessing it. You will do this in the next steps.

Type checking

As previously mentioned, the `lines` type only permits string values.

Warning

In this section, you will set an invalid value and encounter a type error.

What happens if you instead try to assign an integer to the option?

Add the following lines to `default.nix`:

Listing 28: default.nix

```
 { lib, ... }: {

  options = {
    scripts.output = lib.mkOption {
      type = lib.types.lines;
    };
  };

+ config = {
+   scripts.output = 42;
+ };
 }
```

Now try to execute the previous command, and witness your first module error:

```
$ nix-instantiate --eval eval.nix -A config.scripts.output
error:
...
       error: A definition for option `scripts.output' is not of type `strings␣
↪concatenated with "\n"'. Definition values:
       - In `/home/nix-user/default.nix': 42
```

The definition `scripts.output = 42;` caused a type error: integers are not strings concatenated with the newline character.

To make this module pass the type checks and successfully evaluate the `scripts.output` option, you will now assign a string to `scripts.output`.

In this case, you will assign a shell command that runs the `map` script in the current directory. That in turn calls the Google Maps Static API to generate a world map. The output is passed on to display it with `feh`[157], a minimalistic image viewer.

Update `default.nix` by changing the value of `scripts.output` to the following string:

[155] https://nix.dev/manual/nix/stable/command-ref/nix-instantiate
[156] https://nix.dev/manual/nix/stable/language/operators#attribute-selection
[157] https://feh.finalrewind.org/

Listing 29: default.nix

```
   config = {
-    scripts.output = 42;
+    scripts.output = ''
+      ./map.sh size=640x640 scale=2 | feh -
+    '';
   };
```

Interlude: reproducible scripts

That simple command will likely not work as intended on your system, as it may lack the required dependencies (curl and feh). We can solve this by packaging the raw `map` script with `pkgs.writeShellApplication`.

First, make available a `pkgs` argument in your module evaluation by adding a module that sets `config._module.args`:

Listing 30: eval.nix

```
 pkgs.lib.evalModules {
   modules = [
+    ({ config, ... }: { config._module.args = { inherit pkgs; }; })
     ./default.nix
   ];
 }
```

Note

This mechanism is currently only documented in the module system code[158], and that documentation is incomplete and out of date.

Then change `default.nix` to have the following contents:

Listing 31: default.nix

```
{ pkgs, lib, ... }: {

  options = {
    scripts.output = lib.mkOption {
      type = lib.types.package;
    };
  };

  config = {
    scripts.output = pkgs.writeShellApplication {
      name = "map";
      runtimeInputs = with pkgs; [ curl feh ];
      text = ''
        ${./map.sh} size=640x640 scale=2 | feh -
      '';
    };
  };
}
```

This will access the previously added `pkgs` argument so we can use dependencies, and copy the `map` file in the current

[158] https://github.com/NixOS/nixpkgs/blob/master/lib/modules.nix#L140-L182

directory into the Nix store so it's available to the wrapped script, which will also live in the Nix store.

Run the script with:

```
nix-build eval.nix -A config.scripts.output
./result/bin/map
```

To iterate more quickly, open a new terminal and set up `entr`[159] to re-run the script whenever any source file in the current directory changes:

```
nix-shell -p entr findutils bash --run \
  "ls *.nix | \
   entr -rs ' \
     nix-build eval.nix -A config.scripts.output --no-out-link \
     | xargs printf -- \"%s/bin/map\" \
     | xargs bash \
   ' \
  "
```

This command does the following:

- List all `.nix` files
- Make `entr` watch them for changes. Terminate the invoked command on each change with `-r`.
- On each change:
  - Run the `nix-build` invocation as above, but without adding a `./result` symlink
  - Take the resulting store path and append `/bin/map` to it
  - Run the executable at the path constructed this way

Declaring more options

Rather than setting all script parameters directly, we will to do that through the module system. This will not just add some safety through type checking, but also allow to build abstractions to manage growing complexity and changing requirements.

Let's begin by introducing another option, `requestParams`, which will represent the parameters of the request made to the Google Maps API.

Its type will be `listOf <elementType>`, which is a list of elements of one type.

Instead of `lines`, in this case you will want the type of the list elements to be `str`, a generic string type.

The difference between `str` and `lines` is in their merging behavior: Module option types not only check for valid values, but also specify how multiple definitions of an option are to be combined into one.

- For `lines`, multiple definitions get merged by concatenation with newlines.
- For `str`, multiple definitions are not allowed. This is not a problem here, since one can't define a list element multiple times.

Make the following additions to your `default.nix` file:

Listing 32: default.nix

```
      scripts.output = lib.mkOption {
        type = lib.types.package;
      };
+
+     requestParams = lib.mkOption {
+       type = lib.types.listOf lib.types.str;
```

(continues on next page)

[159] https://github.com/eradman/entr

(continued from previous page)

```
+    };
   };

  config = {
    scripts.output = pkgs.writeShellApplication {
      name = "map";
      runtimeInputs = with pkgs; [ curl feh ];
      text = ''
        ${./map.sh} size=640x640 scale=2 | feh -
      '';
    };
+
+    requestParams = [
+      "size=640x640"
+      "scale=2"
+    ];
   };
 }
```

Dependencies between options

A given module generally declares one option that produces a result to be used elsewhere, in this case `scripts.output`.

Options can depend on other options, making it possible to build more useful abstractions.

Here, we want the `scripts.output` option to use the values of `requestParams` as arguments to the `./map` script.

Accessing option values

To make option values available to a module, the arguments of the function declaring the module must include the `config` attribute.

Update `default.nix` to add the `config` attribute:

Listing 33: default.nix

```
-{ pkgs, lib, ... }: {
+{ pkgs, lib, config, ... }: {
```

When a module that sets options is evaluated, the resulting values can be accessed by their corresponding attribute names under `config`.

Note

Option values can't be accessed directly from the same module.

The module system evaluates all modules it receives, and any of them can define a particular option's value. What happens when an option is set by multiple modules is determined by that option's type.

Warning

The `config` *argument* is **not** the same as the `config` *attribute*:

- The `config` *argument* holds the result of the module system's lazy evaluation, which takes into account all modules passed to `evalModules` and their `imports`.

- The `config` *attribute* of a module exposes that particular module's option values to the module system for evaluation.

Now make the following changes to `default.nix`:

Listing 34: default.nix

```
   config = {
     scripts.output = pkgs.writeShellApplication {
       name = "map";
       runtimeInputs = with pkgs; [ curl feh ];
       text = ''
-        ${./map.sh} size=640x640 scale=2 | feh -
+        ${./map.sh} ${lib.concatStringsSep " "
+          config.requestParams} | feh -
       '';
```

Here, the value of the `config.requestParams` attribute is populated by the module system based on the definitions in the same file.

Note

Lazy evaluation in the Nix language allows the module system to make a value available in the `config` argument passed to the module which defines that value.

`lib.concatStringsSep " "` is then used to join each list element from the value of `config.requestParams` into a single string, with the list elements of `requestParams` separated by a space character.

The result of this represents the list of command line arguments to pass to the `./map` script.

Conditional definitions

Sometimes, you will want option values to be, well, optional. This can be useful when defining a value for an option is not required, as in the following case.

You will define a new option, `map.zoom`, to control the zoom level of the map. The Google Maps API will infer a zoom level if no corresponding argument is passed, a situation you can represent with the `nullOr <type>`, which represents values of type `<type>` or `null`. This *does not* automatically mean that when the option isn't defined, the value of such an option is `null` – we still need to define a default value.

Add the `map` attribute set with the `zoom` option into the top-level `options` declaration, like so:

Listing 35: default.nix

```
     requestParams = lib.mkOption {
       type = lib.types.listOf lib.types.str;
     };
+
+    map = {
+      zoom = lib.mkOption {
+        type = lib.types.nullOr lib.types.int;
+        default = null;
+      };
+    };
   };
```

To make use of this, use the `mkIf <condition> <definition>` function, which only adds the definition if the condition evaluates to `true`. Make the following additions to the `requestParams` list in the `config` block:

Listing 36: default.nix

```
     requestParams = [
       "size=640x640"
       "scale=2"
+      (lib.mkIf (config.map.zoom != null)
+        "zoom=${toString config.map.zoom}")
     ];
   };
```

This will only add a `zoom` parameter to the script invocation if the value of `config.map.zoom` is not `null`.

Default values

Let's say that in our application we want to have a different default behavior that sets the zoom level to `10`, such that automatic zooming has to be enabled explicitly.

This can be done with the `default` argument to `mkOption`[160]. Its value will be used if the value of the option declaring it is not specified otherwise.

Add the corresponding line:

Listing 37: default.nix

```
     map = {
       zoom = lib.mkOption {
         type = lib.types.nullOr lib.types.int;
+        default = 10;
       };
     };
   };
```

Wrapping shell commands

You have now declared options controlling the map dimensions and zoom level, but have not provided a way to specify where the map should be centered.

Add the `center` option now, possibly with your own location as default value:

Listing 38: default.nix

```
         type = lib.types.nullOr lib.types.int;
         default = 10;
       };
+
+      center = lib.mkOption {
+        type = lib.types.nullOr lib.types.str;
+        default = "switzerland";
+      };
     };
   };
```

To implement this behavior, you will use the `geocode` utility, which turns location names into coordinates. There are multiple ways of making a new package accessible, but as an exercise, you will add it as an option in the module system.

First, add a new option to accommodate the package:

[160] https://github.com/NixOS/nixpkgs/blob/master/lib/options.nix

Listing 39: default.nix

```
   options = {
     scripts.output = lib.mkOption {
       type = lib.types.package;
     };
+
+    scripts.geocode = lib.mkOption {
+      type = lib.types.package;
+    };
```

Then define the value for that option where you make the raw script reproducible by wrapping a call to it in `writeShellApplication`:

Listing 40: default.nix

```
   config = {
+    scripts.geocode = pkgs.writeShellApplication {
+      name = "geocode";
+      runtimeInputs = with pkgs; [ curl jq ];
+      text = ''exec ${./geocode.sh} "$@"'';
+    };
+
     scripts.output = pkgs.writeShellApplication {
       name = "map";
       runtimeInputs = with pkgs; [ curl feh ];
```

Add another `mkIf` call to the list of `requestParams` now where you access the wrapped package through `config.scripts.geocode`, and run the executable `/bin/geocode` inside:

Listing 41: default.nix

```
       "scale=2"
       (lib.mkIf (config.map.zoom != null)
         "zoom=${toString config.map.zoom}")
+      (lib.mkIf (config.map.center != null)
+        "center=\"$(${config.scripts.geocode}/bin/geocode ${
+          lib.escapeShellArg config.map.center
+        })\"")
     ];
   };
```

This time, you've used `escapeShellArg` to pass the `config.map.center` value as a command-line argument to `geocode`, string interpolating the result back into the `requestParams` string which sets the `center` value.

Wrapping shell command execution in Nix modules is a helpful technique for controlling system changes, as it uses the more ergonomic attributes and values interface rather than dealing with the peculiarities of escaping manually.

Splitting modules

The module schema[161] includes the `imports` attribute, which allows incorporating further modules, for example to split a large configuration into multiple files.

In particular, this allows you to separate option declarations from where they are used in your configuration.

Create a new module, `marker.nix`, where you can declare options for defining location pins and other markers on the map:

[161] https://nixos.org/manual/nixos/stable/#sec-writing-modules

Listing 42: marker.nix

```
{ lib, config, ... }: {

}
```

Reference this new file in `default.nix` using the `imports` attribute:

Listing 43: default.nix

```
 { pkgs, lib, config, ... }: {

+  imports = [
+    ./marker.nix
+  ];
+
```

The `submodule` type

We want to set multiple markers on the map. A marker is a complex type with multiple fields.

This is where one of the most useful types included in the module system's type system comes into play: `submodule`. This type allows you to define nested modules with their own options.

Here, you will define a new `map.markers` option whose type is a list of submodules, each with a nested `location` type, allowing you to define a list of markers on the map.

Each assignment of markers will be type-checked during evaluation of the top-level `config`.

Make the following changes to `marker.nix`:

Listing 44: marker.nix

```
-{ lib, config, ... }: {
+{ lib, config, ... }:
+let
+  markerType = lib.types.submodule {
+    options = {
+      location = lib.mkOption {
+        type = lib.types.nullOr lib.types.str;
+        default = null;
+      };
+    };
+  };
+in {
+
+  options = {
+    map.markers = lib.mkOption {
+      type = lib.types.listOf markerType;
+    };
+  };
```

Defining options in other modules

Because of the way the module system composes option definitions, you can freely assign values to options defined in other modules.

In this case, you will use the `map.markers` option to produce and add new elements to the `requestParams` list, making your declared markers appear on the returned map – but from the module declared in `marker.nix`.

To implement this behavior, add the following `config` block to `marker.nix`:

Listing 45: marker.nix

```
+  config = {
+
+    map.markers = [
+      { location = "new york"; }
+    ];
+
+    requestParams = let
+      paramForMarker =
+        builtins.map (marker: "$(${config.scripts.geocode}/bin/geocode ${
+          lib.escapeShellArg marker.location})") config.map.markers;
+    in [ "markers=\"${lib.concatStringsSep "|" paramForMarker}\"" ];
+  };
```

Warning

To avoid confusion with the `map` option setting and the final `config.map` configuration value, here we use the `map` function explicitly as `builtins.map`.

Here, you again used `escapeShellArg` and string interpolation to generate a Nix string, this time producing a pipe-separated list of geocoded location attributes.

The `requestParams` value was also set to the resulting list of strings, which gets appended to the `requestParams` list defined in `default.nix`, thanks to the default merging behavior of the `list` type.

When defining multiple markers, determining an appropriate center or zoom level for the map may be challenging; it's easier to let the API do this for you.

To achieve this, make the following additions to `marker.nix`, above the `requestParams` declaration:

Listing 46: marker.nix

```
+    map.center = lib.mkIf
+      (lib.length config.map.markers >= 1)
+      null;
+
+    map.zoom = lib.mkIf
+      (lib.length config.map.markers >= 2)
+      null;
+
     requestParams = let
       paramForMarker = marker:
         let
```

In this case, the default behavior of the Google Maps API when not passed a center or zoom level is to pick the geometric center of all the given markers, and to set a zoom level appropriate for viewing all markers at once.

Nested submodules

Next, we want to allow multiple named users to define a list of markers each.

For that you'll add a `users` option with type `lib.types.attrsOf <subtype>`, which will allow you to define `users` as an attribute set, whose values have type `<subtype>`.

Here, that subtype will be another submodule which allows declaring a departure marker, suitable for querying the API for the recommended route for a trip.

This will again make use of the `markerType` submodule, giving a nested structure of submodules.

To propagate marker definitions from `users` to the `map.markers` option, make the following changes.

In the `let` block:

Listing 47: marker.nix

```
+  userType = lib.types.submodule {
+    options = {
+      departure = lib.mkOption {
+        type = markerType;
+        default = {};
+      };
+    };
+  };
+
 in {
```

This defines a submodule type for a user, with a `departure` option of type `markerType`.

In the `options` block, above `map.markers`:

Listing 48: marker.nix

```
+    users = lib.mkOption {
+      type = lib.types.attrsOf userType;
+    };
```

That allows adding a `users` attribute set to `config` in any submodule that imports `marker.nix`, where each attribute will be of type `userType` as declared in the previous step.

In the `config` block, above `map.center`:

Listing 49: marker.nix

```
   config = {

-    map.markers = [
-      { location = "new york"; }
-    ];
+    map.markers = lib.filter
+      (marker: marker.location != null)
+      (lib.concatMap (user: [
+        user.departure
+      ]) (lib.attrValues config.users));

     map.center = lib.mkIf
       (lib.length config.map.markers >= 1)
```

This takes all the `departure` markers from all users in the `config` argument, and adds them to `map.markers` if their `location` attribute is not `null`.

The `config.users` attribute set is passed to `attrValues`, which returns a list of values of each of the attributes in the set (here, the set of `config.users` you've defined), sorted alphabetically (which is how attribute names are stored in the Nix language).

Back in `default.nix`, the resulting `map.markers` option value is still accessed by `requestParams`, which in turn is used to generate arguments to the script that ultimately calls the Google Maps API.

Defining the options in this way allows you to set multiple `users.<name>.departure.location` values and generate a map with the appropriate zoom and center, with pins corresponding to the set of `departure.location` values for *all* `users`.

In the 2021 Summer of Nix, this formed the basis of an interactive multi-person map demo.

The `strMatching` type

Now that the map can be rendered with multiple markers, it's time to add some style customizations.

To tell the markers apart, add another option to the `markerType` submodule, to allow labeling each marker pin.

The API documentation states that these labels must be either an uppercase letter or a number[162].

You can implement this with the `strMatching "<regex>"` type, where `<regex>` is a regular expression that will accept any matching values, in this case an uppercase letter or number.

In the `let` block:

Listing 50: marker.nix

```
          type = lib.types.nullOr lib.types.str;
          default = null;
        };
+
+       style.label = lib.mkOption {
+         type = lib.types.nullOr
+           (lib.types.strMatching "[A-Z0-9]");
+         default = null;
+       };
      };
    };
```

Again, `types.nullOr` allows for `null` values, and the default has been set to `null`.

In the `paramForMarker` function:

Listing 51: marker.nix

```
      requestParams = let
-       paramForMarker =
-         builtins.map (marker: "$(${config.scripts.geocode}/bin/geocode ${
-           lib.escapeShellArg marker.location})") config.map.markers;
-     in [ "markers=\"${lib.concatStringsSep "|" paramForMarker}\"" ];
+       paramForMarker = marker:
+         let
+           attributes =
+             lib.optional (marker.style.label != null)
+             "label:${marker.style.label}"
+             ++ [
+               "$(${config.scripts.geocode}/bin/geocode ${
+                 lib.escapeShellArg marker.location
+               })"
+             ];
+         in "markers=\"${lib.concatStringsSep "|" attributes}\"";
+       in
+         builtins.map paramForMarker config.map.markers;
```

Note how we now create a unique `marker` for each user by concatenating the `label` and `location` attributes together, and assigning them to the `requestParams`. The label for each `marker` is only propagated to the CLI parameters if `marker.style.label` is set.

[162] https://developers.google.com/maps/documentation/maps-static/start#MarkerStyles

Functions as submodule arguments

Right now, if a label is not explicitly set, none will show up. But since every `users` attribute has a name, we could use that as an automatic value instead.

This `firstUpperAlnum` function allows you to retrieve the first character of the username, with the correct type for passing to `departure.style.label`:

Listing 52: marker.nix

```
{ lib, config, ... }:
 let
+  # Returns the uppercased first letter
+  # or number of a string
+  firstUpperAlnum = str:
+    lib.mapNullable lib.head
+    (builtins.match "[^A-Z0-9]*([A-Z0-9]).*"
+    (lib.toUpper str));

   markerType = lib.types.submodule {
     options = {
```

By transforming the argument to `lib.types.submodule` into a function, you can access arguments within it.

One special argument automatically available to submodules is `name`, which when used in `attrsOf`, gives you the name of the attribute the submodule is defined under:

Listing 53: marker.nix

```
-  userType = lib.types.submodule {
+  userType = lib.types.submodule ({ name, ... }: {
     options = {
       departure = lib.mkOption {
         type = markerType;
         default = {};
       };
     };
-  };
```

In this case, you don't easily have access to the name from the marker submodules `label` option, where you otherwise could set a `default` value.

Instead you can use the `config` section of the `user` submodule to set a default, like so:

Listing 54: marker.nix

```
+
+    config = {
+      departure.style.label = lib.mkDefault
+        (firstUpperAlnum name);
+    };
+  });

 in {
```

Note

Module options have a *priority*, represented as an integer, which determines the precedence for setting the option to a particular value. When merging values, the priority with lowest numeric value wins.

The `lib.mkDefault` modifier sets the priority of its argument value to 1000, the lowest precedence.

This ensures that other values set for the same option will prevail.

The `either` and `enum` types

For better visual contrast, it would be helpful to have a way to change the *color* of a marker.

Here you will use two new type-functions for this:

- `either <this> <that>`, which takes two types as arguments, and allows either of them
- `enum [ <allowed values> ]`, which takes a list of allowed values, and allows any of them

In the `let` block, add the following `colorType` option, which can hold strings containing either some given color names or an RGB value add the new compound type:

Listing 55: marker.nix

```
     ...
     (builtins.match "[^A-Z0-9]*([A-Z0-9]).*"
     (lib.toUpper str));

+  # Either a color name or `0xRRGGBB`
+  colorType = lib.types.either
+    (lib.types.strMatching "0x[0-9A-F]{6}")
+    (lib.types.enum [
+      "black" "brown" "green" "purple" "yellow"
+      "blue" "gray" "orange" "red" "white" ]);
+
   markerType = lib.types.submodule {
     options = {
       location = lib.mkOption {
```

This allows either strings that match a 24-bit hexadecimal number or are equal to one of the specified color names.

At the bottom of the `let` block, add the `style.color` option and specify a default value:

Listing 56: marker.nix

```
           (lib.types.strMatching "[A-Z0-9]");
         default = null;
       };
+
+      style.color = lib.mkOption {
+        type = colorType;
+        default = "red";
+      };
     };
   };
```

Now add an entry to the `paramForMarker` list which makes use of the new option:

Listing 57: marker.nix

```
               (marker.style.label != null)
               "label:${marker.style.label}"
             ++ [
+              "color:${marker.style.color}"
               "$(${config.scripts.geocode}/bin/geocode ${
```

(continues on next page)

(continued from previous page)

```
                lib.escapeShellArg marker.location
              })"
```

In case you set many different markers, it would be helpful to have the ability to change their size individually.

Add a new `style.size` option to `marker.nix`, allowing you to choose from the set of pre-defined sizes:

Listing 58: marker.nix

```
        type = colorType;
        default = "red";
      };
+
+     style.size = lib.mkOption {
+       type = lib.types.enum
+         [ "tiny" "small" "medium" "large" ];
+       default = "medium";
+     };
    };
  };
```

Now add a mapping for the size parameter in `paramForMarker`, which selects an appropriate string to pass to the API:

Listing 59: marker.nix

```
    requestParams = let
      paramForMarker = marker:
        let
+         size = {
+           tiny = "tiny";
+           small = "small";
+           medium = "mid";
+           large = null;
+         }.${marker.style.size};
+
```

Finally, add another `lib.optional` call to the `attributes` string, making use of the selected size:

Listing 60: marker.nix

```
          attributes =
            lib.optional
              (marker.style.label != null)
              "label:${marker.style.label}"
+           ++ lib.optional
+             (size != null)
+             "size:${size}"
            ++ [
              "color:${marker.style.color}"
              "$(${config.scripts.geocode}/bin/geocode ${
```

The `pathType` submodule

So far, you've created an option for declaring a *departure* marker, as well as several options for configuring the marker's visual representation.

Now we want to compute and display a route from the user's location to some destination.

The new option defined in the next section will allow you to set an *arrival* marker, which together with a departure allows you to draw *paths* on the map using the new module defined below.

To start, create a new `path.nix` file with the following contents:

Listing 61: path.nix

```
{ lib, config, ... }:
let
  pathType = lib.types.submodule {
    options = {
      locations = lib.mkOption {
        type = lib.types.listOf lib.types.str;
      };
    };
  };
in
{
  options = {
    map.paths = lib.mkOption {
      type = lib.types.listOf pathType;
    };
  };
  config = {
    requestParams =
      let
        attrForLocation = loc:
          "$(${config.scripts.geocode}/bin/geocode ${lib.escapeShellArg loc})";
        paramForPath = path:
          let
            attributes =
              builtins.map attrForLocation path.locations;
          in
          ''path="${lib.concatStringsSep "|" attributes}"'';
      in
      builtins.map paramForPath config.map.paths;
  };
}
```

The `path.nix` module declares an option for defining a list of paths on our `map`, where each path is a list of strings for geographic locations.

In the `config` attribute we augment the API call by setting the `requestParams` option value with the coordinates transformed appropriately, which will be concatenated with request parameters set elsewhere.

Now import this new `path.nix` module from your `marker.nix` module:

Listing 62: marker.nix

```
 in {

+  imports = [
+    ./path.nix
+  ];
+
   options = {

     users = lib.mkOption {
```

Copy the `departure` option declaration to a new `arrival` option in `marker.nix`, to complete the initial path

implementation:

Listing 63: marker.nix

```
          type = markerType;
          default = {};
        };
+
+       arrival = lib.mkOption {
+         type = markerType;
+         default = {};
+       };
      };
```

Next, add an `arrival.style.label` attribute to the `config` block, mirroring the `departure.style.label`:

Listing 64: marker.nix

```
      config = {
        departure.style.label = lib.mkDefault
          (firstUpperAlnum name);
+       arrival.style.label = lib.mkDefault
+         (firstUpperAlnum name);
      };
    });
```

Finally, update the return list in the function passed to `concatMap` in `map.markers` to also include the `arrival` marker for each user:

Listing 65: marker.nix

```
      map.markers = lib.filter
        (marker: marker.location != null)
        (lib.concatMap (user: [
-         user.departure
+         user.departure user.arrival
        ]) (lib.attrValues config.users));

      map.center = lib.mkIf
```

Now you have the basis to define paths on the map, connecting pairs of departure and arrival points.

In the path module, define a path connecting every user's departure and arrival locations:

Listing 66: path.nix

```
    config = {
+
+     map.paths = builtins.map (user: {
+       locations = [
+         user.departure.location
+         user.arrival.location
+       ];
+     }) (lib.filter (user:
+       user.departure.location != null
+       && user.arrival.location != null
+     ) (lib.attrValues config.users));
+
      requestParams = let
```

(continues on next page)

(continued from previous page)

```
      attrForLocation = loc:
        "$(geocode ${lib.escapeShellArg loc})";
```

The new `map.paths` attribute contains a list of all valid paths defined for all users.

A path is valid only if the `departure` and `arrival` attributes are set for that user.

The `between` constraint on integer values

Your users have spoken, and they demand the ability to customize the styles of their paths with a `weight` option.

As before, you'll now declare a new submodule for the path style.

While you could also directly declare the `style.weight` option, in this case you should use the submodule to be able reuse the path style type later.

Add the `pathStyleType` submodule option to the `let` block in `path.nix`:

Listing 67: path.nix

```
 { lib, config, ... }:
 let
+
+  pathStyleType = lib.types.submodule {
+    options = {
+      weight = lib.mkOption {
+        type = lib.types.ints.between 1 20;
+        default = 5;
+      };
+    };
+  };
+
   pathType = lib.types.submodule {
```

Note

The `ints.between <lower> <upper>` type allows integers in the given (inclusive) range.

The path weight will default to 5, but can be set to any integer value in the 1 to 20 range, with higher weights producing thicker paths on the map.

Now add a `style` option to the `options` set further down the file:

Listing 68: path.nix

```
     options = {
       locations = lib.mkOption {
         type = lib.types.listOf lib.types.str;
       };
+
+      style = lib.mkOption {
+        type = pathStyleType;
+        default = {};
+      };
     };

   };
```

Finally, update the `attributes` list in `paramForPath`:

Listing 69: path.nix

```
       paramForPath = path:
         let
           attributes =
-            builtins.map attrForLocation path.locations;
+            [
+              "weight:${toString path.style.weight}"
+            ]
+            ++ builtins.map attrForLocation path.locations;
         in "path=\"${lib.concatStringsSep "|" attributes}\"";
```

The `pathStyle` submodule

Users still can't actually customize the path style yet. Introduce a new `pathStyle` option for each user.

The module system allows you to declare values for an option multiple times, and if the types permit doing so, takes care of merging each declaration's values together.

This makes it possible to have a definition for the `users` option in the `marker.nix` module, as well as a `users` definition in `path.nix`:

Listing 70: path.nix

```
 in {
   options = {
+
+    users = lib.mkOption {
+      type = lib.types.attrsOf (lib.types.submodule {
+        options.pathStyle = lib.mkOption {
+          type = pathStyleType;
+          default = {};
+        };
+      });
+    };
+
     map.paths = lib.mkOption {
       type = lib.types.listOf pathType;
     };
```

Then add a line using the `user.pathStyle` option in `map.paths` where each user's paths are processed:

Listing 71: path.nix

```
         user.departure.location
         user.arrival.location
       ];
+      style = user.pathStyle;
     }) (lib.filter (user:
       user.departure.location != null
       && user.arrival.location != null
```

Path styling: color

As with markers, paths should have customizable colors.

You can accomplish this using types you've already encountered by now.

Add a new `colorType` block to `path.nix`, specifying the allowed color names and RGB/RGBA hexadecimal

values:

Listing 72: path.nix

```
 { lib, config, ... }:
 let

+  # Either a color name, `0xRRGGBB` or `0xRRGGBBAA`
+  colorType = lib.types.either
+    (lib.types.strMatching "0x[0-9A-F]{6}([0-9A-F]{2})?")
+    (lib.types.enum [
+      "black" "brown" "green" "purple" "yellow"
+      "blue" "gray" "orange" "red" "white"
+    ]);
+
   pathStyleType = lib.types.submodule {
```

Under the `weight` option, add a new `color` option to use the new `colorType` value:

Listing 73: path.nix

```
         type = lib.types.ints.between 1 20;
         default = 5;
       };
+
+      color = lib.mkOption {
+        type = colorType;
+        default = "blue";
+      };
     };
   };
```

Finally, add a line using the `color` option to the `attributes` list:

Listing 74: path.nix

```
           attributes =
             [
               "weight:${toString path.style.weight}"
+              "color:${path.style.color}"
             ]
             ++ map attrForLocation path.locations;
         in "path=${
```

Further styling

Now that you've got this far, to further improve the aesthetics of the rendered map, add another style option allowing paths to be drawn as *geodesics*, the shortest "as the crow flies" distance between two points on Earth.

Since this feature can be turned on or off, you can do this using the `bool` type, which can be `true` or `false`.

Make the following changes to `path.nix` now:

Listing 75: path.nix

```
         type = colorType;
         default = "blue";
       };
+
```

(continues on next page)

(continued from previous page)

```
+      geodesic = lib.mkOption {
+        type = lib.types.bool;
+        default = false;
+      };
     };
   };
```

Make sure to also add a line to use that value in `attributes` list, so the option value is included in the API call:

Listing 76: path.nix

```
              [
                "weight:${toString path.style.weight}"
                "color:${path.style.color}"
+               "geodesic:${lib.boolToString path.style.geodesic}"
              ]
              ++ map attrForLocation path.locations;
          in "path=${
```

Wrapping up

In this tutorial, you've learned how to write custom Nix modules to bring external services under declarative control, with the help of several new utility functions from the Nixpkgs `lib`.

You defined several modules in multiple files, each with separate submodules making use of the module system's type checking.

These modules exposed features of the external API in a declarative way.

You can now conquer the world with Nix.

2.9 NixOS

Learn how to configure, test, and install or deploy NixOS.

2.9.1 Creating NixOS images

- *NixOS virtual machines* (page 97)
- *Building a bootable ISO image* (page 102)
- *Building and running Docker images* (page 103)

2.9.2 Testing and deploying NixOS configurations

- *Integration testing with NixOS virtual machines* (page 105)
- *Provisioning remote machines via SSH* (page 110)
- *Installing NixOS on a Raspberry Pi* (page 115)
- *Deploying NixOS using Terraform* (page 119)

2.9.3 Scaling up

- *Setting up an HTTP binary cache* (page 122)
- *Setting up distributed builds* (page 126)

NixOS virtual machines

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 Linux system with virtualisation support
- (optional) A graphical environment for running a graphical virtual machine
- A working Nix installation[163]
- Basic knowledge of the *Nix language* (page 12)

Important

A NixOS configuration is a Nix language function following the NixOS module[164] convention. For a thorough treatment of the module system, check the *Module system deep dive* (page 75) tutorial.

Starting from a default NixOS configuration

Note

This tutorial starts with building up your `configuration.nix` from first principles, explaining each step. If you prefer, you can skip ahead to the *sample configuration* (page 98) section.

We start with a minimal `configuration.nix`:

```
{ config, pkgs, ... }:

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

  system.stateVersion = "24.05";
}
```

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

```
  users.users.alice = {
    isNormalUser = true;
    extraGroups = [ "wheel" ];
  };
```

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:

[163] https://nix.dev/install-nix
[164] https://nixos.org/manual/nixos/stable/index.html#sec-writing-modules

```
    initialPassword = "test";
```

We add two lightweight programs as an example:

```
  environment.systemPackages = with pkgs; [
    cowsay
    lolcat
  ];
```

Warning

Do not use plain text passwords outside of this example unless you know what you are doing. See `initialHashedPassword`[165] or `ssh.authorizedKeys`[166] for more secure alternatives.

Sample configuration

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

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

  users.users.alice = {
    isNormalUser = true;
    extraGroups = [ "wheel" ]; # Enable ‘sudo’ for the user.
    initialPassword = "test";
  };

  environment.systemPackages = with pkgs; [
    cowsay
    lolcat
  ];

  system.stateVersion = "24.05";
}
```

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-24.05 -I nixos-
↪config=./configuration.nix
```

This command builds the attribute `vm` from the `nixos-24.05` release of NixOS, using the NixOS configuration as specified in the relative path.

Detailed explanation

- The positional argument to `nix-build`[167] is a path to the derivation to be built. That path can be obtained from *a Nix expression that evaluates to a derivation* (page 36).

[165] https://nixos.org/manual/nixos/stable/options.html#opt-users.extraUsers._name_.initialHashedPassword
[166] https://nixos.org/manual/nixos/stable/options.html#opt-users.extraUsers._name_.openssh.authorizedKeys.keys
[167] https://nix.dev/manual/nix/stable/command-ref/nix-build.html

The virtual machine build helper is defined in NixOS, which is part of the `nixpkgs` repository[168]. Therefore we use the *lookup path* (page 25) `<nixpkgs/nixos>`.

- The `-A` option[169] specifies the attribute to pick from the provided Nix expression `<nixpkgs/nixos>`.

  To build the virtual machine, we choose the `vm` attribute as defined in `nixos/default.nix`[170].

- The `-I` option[171] prepends entries to the search path.

  Here we set `nixpkgs` to refer to a *specific version of Nixpkgs* (page 156) and set `nix-config` to the `configuration.nix` file in the current directory.

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:

```
$ QEMU_KERNEL_PARAMS=console=ttyS0 ./result/bin/run-nixos-vm -nographic; reset
```

This command will run QEMU in the current terminal due to `-nographic`. `console=ttyS0` will also show the boot process, which ends at the console login screen.

Log in as `alice` with the password `test`. Check that the programs are indeed available as specified:

```
$ cowsay hello | lolcat
```

Exit the virtual machine by shutting it down:

```
$ sudo poweroff
```

Note

If you forgot to add the user to `wheel` or didn't set a password, stop the virtual machine from a different terminal:

```
$ sudo pkill qemu
```

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
```

[168] https://github.com/NixOS/nixpkgs
[169] https://nix.dev/manual/nix/stable/command-ref/opt-common.html#opt-attr
[170] https://github.com/NixOS/nixpkgs/blob/7c164f4bea71d74d98780ab7be4f9105630a2eba/nixos/default.nix#L19
[171] https://nix.dev/manual/nix/stable/command-ref/opt-common.html#opt-I

Running GNOME on a graphical VM

To create a virtual machine with a graphical user interface, add the following lines to the configuration:

```
  # Enable the X11 windowing system.
  services.xserver.enable = true;

  # Enable the GNOME Desktop Environment.
  services.xserver.displayManager.gdm.enable = true;
  services.xserver.desktopManager.gnome.enable = true;
```

These three lines activate X11, the GDM display manager (to be able to login) and Gnome as desktop manager.

Tip

You can also use the `installation-cd-graphical-gnome.nix` module to generate the configuration file from scratch:

```
nix-shell -I nixpkgs=channel:nixos-24.05 -p "$(cat <<EOF
  let
    pkgs = import <nixpkgs> { config = {}; overlays = []; };
    iso-config = pkgs.path + /nixos/modules/installer/cd-dvd/installation-cd-
↪graphical-gnome.nix;
    nixos = pkgs.nixos iso-config;
  in nixos.config.system.build.nixos-generate-config
EOF
)"
```

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

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

```
{ 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" ];
    initialPassword = "test";
  };

  system.stateVersion = "24.05";
}
```

To get graphical output, run the virtual machine without special options:

```
$ nix-build '<nixpkgs/nixos>' -A vm -I nixpkgs=channel:nixos-24.05 -I nixos-
↪config=./configuration.nix
$ ./result/bin/run-nixos-vm
```

Running Sway as Wayland compositor on a VM

To change to a Wayland compositor, disable `services.xserver.desktopManager.gnome` and enable `programs.sway`:

Listing 77: configuration.nix

```
-  services.xserver.desktopManager.gnome.enable = true;
+  programs.sway.enable = true;
```

Note

Running Wayland compositors in a virtual machine might lead to complications with the display drivers used by QEMU. You need to choose from the available drivers one that is compatible with Sway. See QEMU User Documentation[172] for options. One possibility is the `virtio-vga` driver:

```
$ ./result/bin/run-nixos-vm -device virtio-vga
```

Arguments to QEMU can also be added to the configuration file:

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

  services.xserver.enable = true;

  services.xserver.displayManager.gdm.enable = true;
  programs.sway.enable = true;

  imports = [ <nixpkgs/nixos/modules/virtualisation/qemu-vm.nix> ];
  virtualisation.qemu.options = [
    "-device virtio-vga"
  ];

  users.users.alice = {
    isNormalUser = true;
    extraGroups = [ "wheel" ];
    initialPassword = "test";
  };

  system.stateVersion = "24.05";
}
```

The NixOS manual has chapters on X11[173] and Wayland[174] listing alternative window managers.

References

- NixOS Manual: NixOS Configuration[175].
- NixOS Manual: Modules[176].
- NixOS Manual Options reference[177].

[172] https://www.qemu.org/docs/master/system/qemu-manpage.html
[173] https://nixos.org/manual/nixos/stable/#sec-x11
[174] https://nixos.org/manual/nixos/stable/#sec-wayland
[175] https://nixos.org/manual/nixos/stable/index.html#ch-configuration
[176] https://nixos.org/manual/nixos/stable/index.html#sec-writing-modules
[177] https://nixos.org/manual/nixos/stable/options.html

- NixOS Manual: Changing the configuration[178].
- NixOS source code: `configuration template` in `tools.nix`[179].
- NixOS source code: `vm` attribute in `default.nix`[180].
- Nix manual: `nix-build`[181].
- Nix manual: common command-line options[182].
- QEMU User Documentation[183] for more runtime options
- NixOS option search: `virtualisation.qemu`[184] for declarative virtual machine configuration

Next steps

- *Module system deep dive* (page 75)
- *Integration testing with NixOS virtual machines* (page 105)
- *Building a bootable ISO image* (page 102)

Building a bootable ISO image

Note

If you need to build images for a different platform, see Cross compiling[185].

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" "f2fs" "xfs"
↪"ntfs" "cifs" ];
}
```

Generate an ISO with the above configuration:

```
$ NIX_PATH=nixpkgs=https://github.com/NixOS/nixpkgs/archive/
↪74e2faf5965a12e8fa5cff799b1b19c6cd26b0e3.tar.gz nix-shell -p nixos-generators --
↪run "nixos-generate --format iso --configuration ./myimage.nix -o result"
```

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

[178] https://nixos.org/manual/nixos/stable/#sec-changing-config
[179] https://github.com/NixOS/nixpkgs/blob/4e0525a8cdb370d31c1e1ba2641ad2a91fded57d/nixos/modules/installer/tools/tools.nix#L122-L226
[180] https://github.com/NixOS/nixpkgs/blob/master/nixos/default.nix
[181] https://nix.dev/manual/nix/stable/command-ref/nix-build.html
[182] https://nix.dev/manual/nix/stable/command-ref/opt-common.html
[183] https://www.qemu.org/docs/master/system/qemu-manpage.html
[184] https://search.nixos.org/options?query=virtualisation.qemu
[185] https://github.com/nix-community/nixos-generators#user-content-cross-compiling

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

Next steps

- Take a look at this list of formats that generators support[186] to find your cloud provider or virtualization technology.
- Take a look at the alternative guide to create a NixOS live CD[187]

Building and running Docker images

Docker[188] 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[189] 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[190] provides `dockerTools` to create Docker images:

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

pkgs.dockerTools.buildImage {
  name = "hello-docker";
  config = {
    Cmd = [ "${pkgsLinux.hello}/bin/hello" ];
  };
}
```

Note

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

- *Set up a remote build machine* (page 126) to build on Linux
- *Cross compile to Linux* (page 67) 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.

[186] https://github.com/nix-community/nixos-generators#user-content-supported-formats
[187] https://wiki.nixos.org/wiki/Creating_a_NixOS_live_CD
[188] https://www.docker.com/
[189] https://docs.docker.com/get-docker/
[190] https://github.com/NixOS/nixpkgs

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.gz.drv
warning: unknown setting 'experimental-features'
building '/nix/store/qpgdp0qpd8ddi1ld72w02zkmm7n87b92-docker-layer-hello-docker.drv
↪'...
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-docker.
↪tar.gz.drv'...
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-hello-
↪docker.tar.gz
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[191] is a much better place for that.

[191] https://docs.docker.com/

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[192].
- You might like to browse through more examples of Docker images built with Nix[193].
- Take a look at Arion[194], a `docker-compose` wrapper with first-class support for Nix.
- Build docker images on a *CI with GitHub Actions* (page 142).

Integration testing with NixOS virtual machines

What will you learn?

This tutorial introduces Nixpkgs functionality for testing NixOS configurations. It also shows how to set up distributed test scenarios that involve multiple machines.

What do you need?

- A working *Nix installation* (page 1) on Linux, or NixOS[195]
- Basic knowledge of the *Nix language* (page 12)
- Basic knowledge of *NixOS configuration* (page 97)

Introduction

Nixpkgs provides a test environment[196] to automate integration testing for distributed systems. It allows defining tests based on a set of declarative NixOS configurations and using a Python shell to interact with them through QEMU[197] as the backend. Those tests are widely used to ensure that NixOS works as intended, so in general they are called NixOS Tests[198]. They can be written and launched outside of NixOS, on any Linux machine[208].

Integration tests are reproducible due to the design properties of Nix, making them a valuable part of a continuous integration (CI) pipeline.

The `testers.runNixOSTest` function

NixOS VM tests are defined using the `testers.runNixOSTest` function. The pattern for NixOS VM tests looks like this:

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in

pkgs.testers.runNixOSTest {
  name = "test-name";
```

(continues on next page)

[192] https://nixos.org/nixpkgs/manual/#sec-pkgs-dockerTools
[193] https://github.com/NixOS/nixpkgs/blob/master/pkgs/build-support/docker/examples.nix
[194] https://docs.hercules-ci.com/arion/
[195] https://nixos.org/manual/nixos/stable/index.html#sec-installation
[196] https://nixos.org/manual/nixos/stable/index.html#sec-nixos-tests
[197] https://www.qemu.org/
[198] https://nixos.org/manual/nixos/stable/index.html#sec-nixos-tests
[208] Support for running NixOS VM tests on macOS[209] is also implemented but currently undocumented[210].
[209] https://github.com/NixOS/nixpkgs/issues/108984
[210] https://github.com/NixOS/nixpkgs/issues/254552

(continued from previous page)

```
  nodes = {
    machine1 = { config, pkgs, ... }: {
      # ...
    };
    machine2 = { config, pkgs, ... }: {
      # ...
    };
  };
  testScript = { nodes, ... }: ''
    # ...
  '';
}
```

The function `testers.runNixOSTest` takes a module[199] to specify the test options[200]. Because this module only sets configuration values, one can use the abbreviated module notation.

The following configuration values must be set:

- `name`[201] defines the name of the test.
- `nodes`[202] contains a set of named configurations, because a test script can involve more than one virtual machine. Each virtual machine is created from a NixOS configuration.
- `testScript`[203] defines the Python test script, either as literal string or as a function that takes a `nodes` attribute. This Python test script can access the virtual machines via the names used for the `nodes`. It has super user rights in the virtual machines. In the Python script each virtual machine is accessible via the `machine` object. NixOS provides various methods[204] to run tests on these configurations.

The test framework automatically starts the virtual machines and runs the Python script.

Minimal example

As a minimal test on the default configuration, we will check if the user `root` and `alice` can run Firefox. We will build the example up from scratch.

1. Use a *pinned version of Nixpkgs* (page 156), and *explicitly set configuration options and overlays* (page 146) to avoid them being inadvertently overridden by global configuration:

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11
↪";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in

pkgs.testers.runNixOSTest {
  # ...
}
```

2. Label the test with a descriptive name:

```
name = "minimal-test";
```

3. Because this example only uses one virtual machine, the node we specify is simply called `machine`. This name is arbitrary and can be chosen freely. As configuration you use the relevant parts of the default configuration, *that we used in a previous tutorial* (page 97):

[199] https://nixos.org/manual/nixos/stable/#sec-writing-modules
[200] https://nixos.org/manual/nixos/stable/index.html#sec-test-options-reference
[201] https://nixos.org/manual/nixos/stable/index.html#test-opt-name
[202] https://nixos.org/manual/nixos/stable/index.html#test-opt-nodes
[203] https://nixos.org/manual/nixos/stable/index.html#test-opt-testScript
[204] https://nixos.org/manual/nixos/stable/index.html#ssec-machine-objects

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

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

4. This is the test script:

```
machine.wait_for_unit("default.target")
machine.succeed("su -- alice -c 'which firefox'")
machine.fail("su -- root -c 'which firefox'")
```

   This Python script refers to `machine` which is the name chosen for the virtual machine configuration used in the `nodes` attribute set.

   The script waits until systemd reaches `default.target`. It uses the `su` command to switch between users and the `which` command to check if the user has access to `firefox`. It expects that the command `which firefox` to succeed for user `alice` and to fail for `root`.

   This script will be the value of the `testScript` attribute.

The complete `minimal-test.nix` file content looks like the following:

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in

pkgs.testers.runNixOSTest {
  name = "minimal-test";

  nodes.machine = { config, pkgs, ... }: {

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

    system.stateVersion = "23.11";
  };

  testScript = ''
    machine.wait_for_unit("default.target")
    machine.succeed("su -- alice -c 'which firefox'")
    machine.fail("su -- root -c 'which firefox'")
  '';
}
```

Running tests

To set up all machines and run the test script:

```
$ nix-build minimal-test.nix
```

```
...
test script finished in 10.96s
cleaning up
killing machine (pid 10)
(0.00 seconds)
/nix/store/bx7z3imvxxpwkkza10vb23czhw7873w2-vm-test-run-minimal-test
```

Interactive Python shell in the virtual machine

When developing tests or when something breaks, it's useful to interactively tinker with the test or access a terminal for a machine.

To start an interactive Python session with the testing framework:

```
$ $(nix-build -A driverInteractive minimal-test.nix)/bin/nixos-test-driver
```

Here you can run any of the testing operations. Execute the `testScript` attribute from `minimal-test.nix` with the `test_script()` function.

If a virtual machine is not yet started, the test environment takes care of it on the first call of a method on a `machine` object.

But you can also manually trigger the start of the virtual machine with:

```
>>> machine.start()
```

for a specific node,

or

```
>>> start_all()
```

for all nodes.

You can enter a interactive shell on the virtual machine using:

```
>>> machine.shell_interact()
```

and run shell commands like:

```
uname -a
```

```
Linux server 5.10.37 #1-NixOS SMP Fri May 14 07:50:46 UTC 2021 x86_64 GNU/Linux
```

Re-running successful tests

Because test results are kept in the Nix store, a successful test is cached. This means that Nix will not run the test a second time as long as the test setup (node configuration and test script) stays semantically the same. Therefore, to run a test again, one needs to remove the result.

If you would try to delete the result using the symbolic link, you will get the following error:

```
nix-store --delete ./result
```

```
finding garbage collector roots...
0 store paths deleted, 0.00 MiB freed
error: Cannot delete path '/nix/store/4klj06bsilkqkn6h2sia8dcsi72wbcfl-vm-test-run-
↪unnamed' since it is still alive. To find out why, use: nix-store --query --roots
```

Instead, remove the symbolic link and only then remove the cached result:

```
rm ./result
nix-store --delete /nix/store/4klj06bsilkqkn6h2sia8dcsi72wbcfl-vm-test-run-unnamed
```

This can be also done with one command:

```
result=$(readlink -f ./result) rm ./result && nix-store --delete $result
```

Tests with multiple virtual machines

Tests can involve multiple virtual machines, for example to test client-server-communication.

The following example setup includes:

- A virtual machine named `server` running nginx[205] with default configuration.
- A virtual machine named `client` that has `curl` available to make an HTTP request.
- A `testScript` orchestrating testing logic between `client` and `server`.

The complete `client-server-test.nix` file content looks like the following:

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in

pkgs.testers.runNixOSTest {
  name = "client-server-test";

  nodes.server = { pkgs, ... }: {
    networking = {
      firewall = {
        allowedTCPPorts = [ 80 ];
      };
    };
    services.nginx = {
      enable = true;
      virtualHosts."server" = {};
    };
  };

  nodes.client = { pkgs, ... }: {
    environment.systemPackages = with pkgs; [
      curl
    ];
  };

  testScript = ''
    server.wait_for_unit("default.target")
    client.wait_for_unit("default.target")
    client.succeed("curl http://server/ | grep -o \"Welcome to nginx!\"")
```

(continues on next page)

[205] https://nginx.org/en/

(continued from previous page)

```
  '';
}
```

The test script performs the following steps:

1) Start the server and wait for it to be ready.
2) Start the client and wait for it to be ready.
3) Run `curl` on the client and use `grep` to check the expected return string. The test passes or fails based on the return value.

Run the test:

```
$ nix-build client-server-test.nix
```

Additional information regarding NixOS tests

- Running integration tests on CI requires hardware acceleration, which many CIs do not support.

  To run integration tests in *GitHub Actions* (page 142) see how to disable hardware acceleration[206].

- NixOS comes with a large set of tests that can serve as educational examples.

  A good inspiration is Matrix bridging with an IRC[207].

Next steps

- *Module system deep dive* (page 75)
- *Building a bootable ISO image* (page 102)
- *Building and running Docker images* (page 103)

Provisioning remote machines via SSH

It is possible to replace any Linux installation with a NixOS configuration on running systems using `nixos-anywhere`[211] and `disko`[212].

Introduction

In this tutorial, you will deploy a NixOS configuration to a running computer.

What will you learn?

You'll learn how to

- Specify a minimal NixOS configuration with a declarative disk layout and SSH access
- Check that a configuration is valid
- Deploy and update a NixOS configuration on a remote machine

[206] https://github.com/cachix/install-nix-action#how-do-i-run-nixos-tests
[207] https://github.com/NixOS/nixpkgs/blob/master/nixos/tests/matrix/appservice-irc.nix
[211] https://nix-community.github.io/nixos-anywhere/
[212] https://github.com/nix-community/disko

What do you need?

- Familiarity with the *Nix language* (page 12)
- Familiarity with the *Module system* (page 72)

For a successful unattended installation, ensure for the *target machine* that:

- It is a QEMU virtual machine running Linux
  - With `kexec`[213] support
  - On the `x86-64` or `aarch64` instruction set architecture (ISA)
  - With at least 1 GB of RAM

  This may also be a live system booted from USB, such as the NixOS installer[214].
- The IP address is configured automatically with DHCP
- You can login via SSH
  - With public key authentication (prefered), or password
  - As user `root` or another user with `sudo` permissions

The *local machine* only needs a working *Nix installation* (page 1).

We call the *target machine* `target-machine` in this tutorial. Replace it with the actual hostname or IP address.

Prepare the environment

Create a new project directory and enter it with your shell:

```
mkdir remote
cd remote
```

Specify dependencies (page 136) on `nixpkgs`, `disko`, and `nixos-anywhere`:

```
$ nix-shell -p npins
[nix-shell:remote]$ npins init
[nix-shell:remote]$ npins add github nix-community disko
[nix-shell:remote]$ npins add github nix-community nixos-anywhere
```

Create a new file `shell.nix` which provides all needed tooling using the pinned dependencies:

```
let
  sources = import ./npins;
  pkgs = import sources.nixpkgs {};
in

pkgs.mkShell {
  nativeBuildInputs = with pkgs; [
    npins
    nixos-anywhere
    nixos-rebuild
  ];
  shellHook = ''
    export NIX_PATH="nixpkgs=${sources.nixpkgs}:nixos-config=$PWD/configuration.nix
↪"
  '';
}
```

[213] https://en.wikipedia.org/wiki/Kexec
[214] https://nixos.org/download/#download-nixos-accordion

Now exit the temporary environment and enter the newly specified one:

```
[nix-shell:remote]$ exit
$ nix-shell
```

This shell environment is ready to use well-defined versions of Nixpkgs with `nixos-anywhere` and `nixos-rebuild`.

Important

Run all following commands in this environment.

Create a NixOS configuration

The new NixOS configuration will consist of the general system configuration and a disk layout specification.

The disk layout in this example describes a single disk with a master boot record[215] (MBR) and EFI system partition[216] (ESP) partition, and a root file system that takes all remaining available space. It will work on both EFI and BIOS systems.

Create a new file `single-disk-layout.nix` with the disk layout specification:

```
{ ... }:

{
  disko.devices.disk.main = {
    type = "disk";
    content = {
      type = "gpt";
      partitions = {
        MBR = {
          priority = 0;
          size = "1M";
          type = "EF02";
        };
        ESP = {
          priority = 1;
          size = "500M";
          type = "EF00";
          content = {
            type = "filesystem";
            format = "vfat";
            mountpoint = "/boot";
          };
        };
        root = {
          priority = 2;
          size = "100%";
          content = {
            type = "filesystem";
            format = "ext4";
            mountpoint = "/";
          };
        };
      };
```

(continues on next page)

[215] https://en.wikipedia.org/wiki/Master_boot_record
[216] https://en.wikipedia.org/wiki/EFI_system_partition

(continued from previous page)

```
    };
  };
}
```

Create the file `configuration.nix`, which imports the disk layout definition and specifies which disk to format:

Tip

If you don't know the target disk's device identifier, list all devices on the *target machine* with `lsblk`:

```
$ ssh target-machine lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda      8:0    0   256G  0 disk
├─sda1   8:1    0 248.5G  0 part /nix/store
│                               /
└─sda2   8:2    0   7.5G  0 part [SWAP]
sr0     11:0    1  1024M  0 rom
```

In this example, the disk name is `sda`. The block device path is then `/dev/sda`. Note that value for later.

```nix
{ modulesPath, ... }:

let
  diskDevice = "/dev/sda";
  sources = import ./npins;
in
{
  imports = [
    (modulesPath + "/profiles/qemu-guest.nix")
    (sources.disko + "/module.nix")
    ./single-disk-layout.nix
  ];

  disko.devices.disk.main.device = diskDevice;

  boot.loader.grub = {
    devices = [ diskDevice ];
    efiSupport = true;
    efiInstallAsRemovable = true;
  };

  services.openssh.enable = true;

  users.users.root.openssh.authorizedKeys.keys = [
    "<your SSH key here>"
  ];

  system.stateVersion = "24.11";
}
```

Important

Replace `/dev/sda` with your disk block device path.

Replace the `<your SSH key here>` string with the SSH public key that you want to use for future logins as

user `root`.

Detailed explanation

The `diskDevice` variable in the `let` block defines the path of the disk block device:

```
let
  diskDevice = "/dev/sda";
  sources = import ./npins;
in
```

It is used to set the target for the partitioning and formatting as described in the disk layout specification. It is also used in the boot loader configuration to make it bootable on both legacy BIOS as well as UEFI systems:

```
  disko.devices.disk.main.device = diskDevice;

  boot.loader.grub = {
    devices = [ diskDevice ];
    efiSupport = true;
    efiInstallAsRemovable = true;
  };
```

The `qemu-guest.nix` module makes this system compatible for running inside a QEMU virtual machine:

```
  imports = [
    (modulesPath + "/profiles/qemu-guest.nix")
    (sources.disko + "/module.nix")
    ./single-disk-layout.nix
  ];
```

From a disk layout specification, the `disko` library generates a partitioning script and the portion of the NixOS configuration that mounts the partitions accordingly at boot time. The first line imports the library, the second line applies the disk layout:

```
  imports = [
    (modulesPath + "/profiles/qemu-guest.nix")
    (sources.disko + "/module.nix")
    ./single-disk-layout.nix
  ];
```

Test the disk layout

Check that the disk layout is valid:

```
nix-build -E "((import <nixpkgs> {}).nixos [ ./configuration.nix ]).installTest"
```

This command runs the complete installation in a virtual machine by building a derivation in the `installTest` attribute provided by the `disko` module.

Deploy the system

To deploy the system, build the configuration and the corresponding disk formatting script, and run `nixos-anywhere` using the results:

Important

Replace `target-host` with the hostname or IP address of your *target machine*.

```
toplevel=$(nixos-rebuild build --no-flake)
diskoScript=$(nix-build -E "((import <nixpkgs> {}).nixos [ ./configuration.nix ]).
↪diskoScript")
nixos-anywhere --store-paths "$diskoScript" "$toplevel" root@target-host
```

Note

If you don't have public key authentication: Set the environment variable `SSH_PASS` to your password then append the `--env-password` flag to the `nixos-anywhere` command.

`nixos-anywhere` will now log into the target system, partition, format, and mount the disk, and install the NixOS configuration. Then, it reboots the system.

Update the system

To update the system, run `npins` and re-deploy the configuration:

```
npins update nixpkgs
nixos-rebuild switch --no-flake --target-host root@target-host
```

`nixos-anywhere` is not needed any more, unless you want to change the disk layout.

Next steps

- *Setting up an HTTP binary cache* (page 122)
- *Setting up post-build hooks* (page 140)

References

- `nixos-anywhere` project page[217]
- `disko` project repository[218]
- Collection of disk layout examples[219]

Installing NixOS on a Raspberry Pi

This tutorial assumes you have a Raspberry Pi 4 Model B with 4GB RAM[220].

Before starting this tutorial, make sure you have all the necessary hardware[221]:

- HDMI cable/adapter.
- 8GB+ SD card.
- SD card reader (in case your machine doesn't have an SD slot).
- Power cable for your Raspberry Pi.
- USB keyboard.

Note

This tutorial was written for the Raspberry Pi 4B. Using a previously supported model like the 3B or 3B+ is possible with some modifications to this tutorial.

[217] https://nix-community.github.io/nixos-anywhere/
[218] https://github.com/nix-community/disko
[219] https://github.com/nix-community/disko/tree/master/example
[220] https://www.raspberrypi.org/products/raspberry-pi-4-model-b/
[221] https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up/1

Booting NixOS live image

> **Note**
>
> Booting from USB may require an EEPROM firmware upgrade. This tutorial boots from an SD card to avoid such hiccups.

To prepare the AArch64 image on another device with Nix, run the following commands:

```
$ nix-shell -p wget zstd

[nix-shell:~]$ wget https://hydra.nixos.org/build/226381178/download/1/nixos-sd-
↪image-23.11pre500597.0fbe93c5a7c-aarch64-linux.img.zst
[nix-shell:~]$ unzstd -d nixos-sd-image-23.11pre500597.0fbe93c5a7c-aarch64-linux.
↪img.zst
[nix-shell:~]$ dmesg --follow
```

> **Note**
>
> You can download a recent image from Hydra[222], clicking on the latest successful build (marked with a green checkmark), and copying the link to the build product image.

> **Note**
>
> It may be more convenient to use a software like Etcher[223] to flash the image to your SD card if you are on a system where it's available.

Your terminal should be printing kernel messages as they come in.

Plug in your SD card and your terminal should print what device it got assigned, for example `/dev/sdX`.

Press Ctrl+C to stop `dmesg --follow`.

Copy NixOS to your SD card by replacing `sdX` with the name of your device in the following command:

```
[nix-shell:~]$ sudo dd if=nixos-sd-image-23.11pre500597.0fbe93c5a7c-aarch64-linux.
↪img of=/dev/sdX bs=4096 conv=fsync status=progress
```

Once that command exits, **move the SD card into your Raspberry Pi and power it on**.

You should be greeted with a fresh shell!

In case the image doesn't boot, it's worth updating the firmware[224] and booting the image again.

Getting internet connection

Run `sudo -i` to get a root shell for the rest of the tutorial.

At this point you'll need an internet connection. If you can use an ethernet cable, plug it in and skip to the next section.

If you're connecting to wifi, run `iwconfig` to find the name of your wireless network interface. If it's `wlan0`, replace `SSID` and `passphrase` with your data and run:

[222] https://hydra.nixos.org/job/nixos/trunk-combined/nixos.sd_image.aarch64-linux
[223] https://www.balena.io/etcher/
[224] https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#bootloader_update_stable

```
# wpa_supplicant -B -i wlan0 -c <(wpa_passphrase 'SSID' 'passphrase') &
```

Once you see in your terminal that connection is established, run `host nixos.org` to check that the DNS resolves correctly.

In case you've made a typo, run `pkill wpa_supplicant` and start over.

Updating firmware

To benefit from updates and bug fixes from the vendor, we'll start by updating Raspberry Pi firmware:

```
# nix-shell -p raspberrypi-eeprom
# mount /dev/disk/by-label/FIRMWARE /mnt
# BOOTFS=/mnt FIRMWARE_RELEASE_STATUS=stable rpi-eeprom-update -d -a
```

Installing and configuring NixOS

Now we'll install NixOS with our own configuration, here creating a `guest` user and enabling the SSH daemon.

In the `let` binding below, change the value of the `SSID` and `SSIDpassword` variables to the `SSID` and `passphrase` values you used previously:

```
{ config, pkgs, lib, ... }:

let
  user = "guest";
  password = "guest";
  SSID = "mywifi";
  SSIDpassword = "mypassword";
  interface = "wlan0";
  hostname = "myhostname";
in {

  boot = {
    kernelPackages = pkgs.linuxKernel.packages.linux_rpi4;
    initrd.availableKernelModules = [ "xhci_pci" "usbhid" "usb_storage" ];
    loader = {
      grub.enable = false;
      generic-extlinux-compatible.enable = true;
    };
  };

  fileSystems = {
    "/" = {
      device = "/dev/disk/by-label/NIXOS_SD";
      fsType = "ext4";
      options = [ "noatime" ];
    };
  };

  networking = {
    hostName = hostname;
    wireless = {
      enable = true;
      networks."${SSID}".psk = SSIDpassword;
      interfaces = [ interface ];
    };
  };
```

(continues on next page)

(continued from previous page)

```
  environment.systemPackages = with pkgs; [ vim ];

  services.openssh.enable = true;

  users = {
    mutableUsers = false;
    users."${user}" = {
      isNormalUser = true;
      password = password;
      extraGroups = [ "wheel" ];
    };
  };

  hardware.enableRedistributableFirmware = true;
  system.stateVersion = "23.11";
}
```

To save time on typing the whole configuration, download it:

```
# curl -L https://tinyurl.com/tutorial-nixos-install-rpi4 > /etc/nixos/
↪configuration.nix
```

Note

Credentials you write into a NixOS configuration will be stored in plain text in your `/nix/store` when that configuration is built.

If you **don't** want this to happen, you can enter your credentials at a console or use one of the community's solutions for encrypted secrets.

Due to the way the `nixos-sd-image` is designed, NixOS is actually *already installed* at this point, so we only need to `nixos-rebuild` with our new configuration:

```
# nixos-rebuild boot
# reboot
```

If your system doesn't boot, select the oldest configuration in the bootloader menu to get back to the live image and start over.

Making changes

It booted, congratulations!

To make further changes to the configuration, search through NixOS options[225], edit `/etc/nixos/configuration.nix`, and update your system:

```
$ sudo -i
# nixos-rebuild switch
```

Next steps

- Once you have a working OS, try upgrading it with `nixos-rebuild switch --upgrade` to install more recent package versions, and reboot to the old configuration if something broke.

[225] https://search.nixos.org/options

- To enable hardware acceleration for a nice graphical desktop experience, add the `nixos-hardware`[226] module to your configuration:

```
imports = [
  "${fetchTarball "https://github.com/NixOS/nixos-hardware/tarball/master"}/
↪raspberry-pi/4"
];
```

  We recommend pinning the reference to `nixos-hardware`: *Pinning Nixpkgs* (page 156)

- To tweak bootloader options affecting hardware, see `config.txt` options[227]. You can change these options by running `mount /dev/disk/by-label/FIRMWARE /mnt` and opening `/mnt/config.txt`.

Deploying NixOS using Terraform

Assuming you're familiar with the basics of Terraform[228], by the end of this tutorial you will have provisioned an Amazon Web Services (AWS) instance with Terraform, and will be able to use Nix to deploy incremental changes to NixOS running on the instance.

We'll look at how to boot a NixOS machine and how to deploy the incremental changes.

Booting NixOS image

1. Start by providing the Terraform executable:

```
$ nix-shell -p terraform
```

2. We are using Terraform Cloud[229] as a state/locking backend[230]:

```
$ terraform login
```

3. Make sure to create an organization[231], like `myorganization`, in your Terraform Cloud account.
4. Inside `myorganization`, create a workspace[232] by choosing **CLI-driven workflow** and pick a name, like `myapp`.
5. Inside your workspace, under `Settings / General`, change Execution Mode to `Local`.
6. Inside a new directory, create a `main.tf` file with the following contents. This will start an AWS instance with the NixOS image using one SSH keypair and an SSH security group:

```
terraform {
    backend "remote" {
        organization = "myorganization"

        workspaces {
            name = "myapp"
        }
    }
}

provider "aws" {
    region = "eu-central-1"
}
```

(continues on next page)

[226] https://github.com/nixos/nixos-hardware
[227] https://www.raspberrypi.org/documentation/configuration/config-txt/
[228] https://www.terraform.io/intro/index.html
[229] https://app.terraform.io
[230] https://www.terraform.io/docs/state/purpose.html
[231] https://app.terraform.io/app/organizations/new
[232] https://app.terraform.io/app/cachix/workspaces/new

(continued from previous page)

```
module "nixos_image" {
    source  = "git::https://github.com/tweag/terraform-nixos.git//aws_image_nixos?
↪ref=5f5a0408b299874d6a29d1271e9bffeee4c9ca71"
    release = "20.09"
}

resource "aws_security_group" "ssh_and_egress" {
    ingress {
        from_port   = 22
        to_port     = 22
        protocol    = "tcp"
        cidr_blocks = [ "0.0.0.0/0" ]
    }

    egress {
        from_port       = 0
        to_port         = 0
        protocol        = "-1"
        cidr_blocks     = ["0.0.0.0/0"]
    }
}

resource "tls_private_key" "state_ssh_key" {
    algorithm = "RSA"
}

resource "local_file" "machine_ssh_key" {
    sensitive_content = tls_private_key.state_ssh_key.private_key_pem
    filename          = "${path.module}/id_rsa.pem"
    file_permission   = "0600"
}

resource "aws_key_pair" "generated_key" {
    key_name   = "generated-key-${sha256(tls_private_key.state_ssh_key.public_key_
↪openssh)}"
    public_key = tls_private_key.state_ssh_key.public_key_openssh
}

resource "aws_instance" "machine" {
    ami             = module.nixos_image.ami
    instance_type   = "t3.micro"
    security_groups = [ aws_security_group.ssh_and_egress.name ]
    key_name        = aws_key_pair.generated_key.key_name

    root_block_device {
        volume_size = 50 # GiB
    }
}

output "public_dns" {
    value = aws_instance.machine.public_dns
}
```

The only NixOS specific snippet is:

```
module "nixos_image" {
  source = "git::https://github.com/tweag/terraform-nixos.git/aws_image_nixos?
↪ref=5f5a0408b299874d6a29d1271e9bffeee4c9ca71"
  release = "20.09"
}
```

Note

The `aws_image_nixos` module will return a NixOS AMI given a NixOS release number[233] so that the `aws_instance` resource can reference the AMI in instance_type[234] argument.

5. Make sure to configure AWS credentials[235].
6. Applying the Terraform configuration should get you a running NixOS:

```
$ terraform init
$ terraform apply
```

Deploying NixOS changes

Once the AWS instance is running a NixOS image via Terraform, we can teach Terraform to always build the latest NixOS configuration and apply those changes to your instance.

1. Create `configuration.nix` with the following contents:

```
{ config, lib, pkgs, ... }: {
  imports = [ <nixpkgs/nixos/modules/virtualisation/amazon-image.nix> ];

  # Open https://search.nixos.org/options for all options
}
```

2. Append the following snippet to your `main.tf`:

```
module "deploy_nixos" {
    source = "git::https://github.com/tweag/terraform-nixos.git//deploy_nixos?
↪ref=5f5a0408b299874d6a29d1271e9bffeee4c9ca71"
    nixos_config = "${path.module}/configuration.nix"
    target_host = aws_instance.machine.public_ip
    ssh_private_key_file = local_file.machine_ssh_key.filename
    ssh_agent = false
}
```

3. Deploy:

```
$ terraform init
$ terraform apply
```

Caveats

- The `deploy_nixos` module requires NixOS to be installed on the target machine and Nix on the host machine.
- The `deploy_nixos` module doesn't work when the client and target architectures are different (unless you use distributed builds[236]).

[233] https://status.nixos.org
[234] https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance#instance_type
[235] https://registry.terraform.io/providers/hashicorp/aws/latest/docs#authentication
[236] https://nix.dev/manual/nix/stable/advanced-topics/distributed-builds.html

- If you need to inject a value into Nix, there is no elegant solution.
- Each machine is evaluated separately, so note that your memory requirements will grow linearly with the number of machines.

Next steps

- It's possible to switch to Google Compute Engine[237].
- The `deploy_nixos` module[238] supports a number of arguments, for example to upload keys.

Setting up an HTTP binary cache

A binary cache stores pre-built Nix store objects[239] and provides them to other machines over the network. Any machine with a Nix store can be a binary cache for other machines.

Introduction

In this tutorial you will set up a Nix binary cache that will serve store objects from a NixOS machine over HTTP or HTTPS.

What will you learn?

You'll learn how to:

- Set up signing keys for your cache
- Enable the right services on the NixOS machine serving the cache
- Check that the setup works as intended

What do you need?

- A working *Nix installation* (page 1) on your local machine
- SSH access to a NixOS machine to use as a cache

  If you're new to NixOS, learn about the *module system* (page 72) and configure your first system with *NixOS virtual machines* (page 97).
- (optional) A public IP and DNS domain

  If you don't host yourself, check NixOS friendly hosters[240] on the NixOS Wiki. Follow the tutorial on *Provisioning remote machines via SSH* (page 110) to deploy your NixOS configuration.

For a cache on a local network, we assume:

- The hostname is `cache` (replace it with yours, or an IP address)
- The host serves store objects via HTTP on port 80 (this is the default)

For a publicly accessible cache, we assume:

- The domain name is `cache.example.com` (replace it with yours)
- The host serves store objects via HTTPS on port 443 (this is the default)

[237] https://github.com/tweag/terraform-nixos/tree/master/google_image_nixos#readme
[238] https://github.com/tweag/terraform-nixos/tree/master/deploy_nixos#readme
[239] https://nix.dev/manual/nix/latest/store/store-object
[240] https://wiki.nixos.org/wiki/NixOS_friendly_hosters

How long will it take?

- 25 minutes

Set up services

For the NixOS machine hosting the cache, create a new configuration module in `binary-cache.nix`:

```
{ config, ... }:

{
  services.nix-serve = {
    enable = true;
    secretKeyFile = "/var/secrets/cache-private-key.pem";
  };

  services.nginx = {
    enable = true;
    recommendedProxySettings = true;
    virtualHosts.cache = {
      locations."/".proxyPass = "http://${config.services.nix-serve.bindAddress}:$
↪{toString config.services.nix-serve.port}";
    };
  };

  networking.firewall.allowedTCPPorts = [
    config.services.nginx.defaultHTTPListenPort
  ];
}
```

The options under `services.nix-serve`[241] configure the binary cache service.

`nix-serve` doesn't support IPv6 or SSL/HTTPS. The `services.nginx`[242] options are used to set up a proxy, which does support IPv6, to handle requests to the hostname `cache`.

Important

There is an *optional HTTPS section* (page 125) at the end of this tutorial.

Add the new NixOS module to the existing machine configuration:

```
{ config, ... }:

{
  imports = [
    ./binary-cache.nix
  ];

  # ...
}
```

From your local machine, deploy the new configuration:

```
nixos-rebuild switch --no-flake --target-host root@cache
```

[241] https://search.nixos.org/options?query=services.nix-serve
[242] https://search.nixos.org/options?query=services.nginx

Note

The binary cache daemon will report errors because there is no secret key file, yet.

Generate a signing key pair

A pair of private and public keys is required to ensure that the store objects in the cache are authentic.

To generate a key pair for the binary cache, replace the example hostname `cache.example.com` with your hostname:

```
nix-store --generate-binary-cache-key cache.example.com cache-private-key.pem␣
↪cache-public-key.pem
```

`cache-private-key.pem` will be used by the binary cache daemon to sign the binaries as they are served. Copy it to the location configured in `services.nix-serve.secretKeyFile` on the machine hosting the cache:

```
scp cache-private-key.pem root@cache:/var/secrets/cache-private-key.pem
```

Up until now, the binary cache daemon was in a restart loop due to the missing secret key file. Check that it now works correctly:

```
ssh root@cache systemctl status nix-serve.service
```

Important

Configure Nix to use a custom binary cache (page 133) using `cache-public-key.pem` on your local machine.

Test availability

The following steps check if everything is set up correctly and may help with identifying problems.

Check general availability

Test if the binary cache, reverse proxy, and firewall rules work as intended by querying the cache:

```
$ curl http://cache/nix-cache-info
StoreDir: /nix/store
WantMassQuery: 1
Priority: 30
```

Check store object signing

To test if store objects are signed correctly, inspect the metadata of a sample derivation. On the binary cache host, build the `hello` package and get the `.narinfo` file from the cache:

```
$ hash=$(nix-build '<nixpkgs>' -A pkgs.hello | awk -F '/' '{print $4}' | awk -F '-
↪' '{print $1}')
$ curl "http://cache/$hash.narinfo" | grep "Sig: "
...
Sig: cache.example.
↪org:GyBFzocLAeLEFd0hr2noK84VzPUw0ArCNYEnrm1YXakdsC5FkO2Bkj2JH8Xjou+wxeXMjFKa0YP2AML7nBWsAg==
```

Make sure that the output contains this line prefixed with `Sig:` and shows the public key you generated.

Serving the binary cache via HTTPS

If the binary cache is publicly accessible, it is possible to enforce HTTPS with Let's Encrypt[243] SSL certificates. Edit your `binary-cache.nix` like this and make sure to replace the example URL and mail address with yours:

```
   services.nginx = {
     enable = true;
     recommendedProxySettings = true;
-    virtualHosts.cache = {
+    virtualHosts."cache.example.com" = {
+      enableACME = true;
+      forceSSL = true;
       locations."/".proxyPass = "http://${config.services.nix-serve.bindAddress}:$
↪{toString config.services.nix-serve.port}";
     };
   };

+   security.acme = {
+     acceptTerms = true;
+     certs = {
+       "cache.example.com".email = "you@example.com";
+     };
+   };

   networking.firewall.allowedTCPPorts = [
     config.services.nginx.defaultHTTPListenPort
+    config.services.nginx.defaultSSLListenPort
   ];
```

Rebuild the system to deploy these changes:

```
nixos-rebuild switch --no-flake --target-host root@cache.example.com
```

Next steps

If your binary cache is already a remote build machine[244], it will serve all store objects in its Nix store.

- *Configure Nix to use a custom binary cache* (page 133) using the binary cache's hostname and the generated public key
- *Setting up post-build hooks* (page 140) to upload store objects to the binary cache
- *Setting up distributed builds* (page 126)

To save storage space, please refer to the following NixOS configuration attributes:

- `nix.gc`[245]: Options for automatic garbage collection
- `nix.optimise`[246]: Options for periodic Nix store optimisation

Alternatives

- `nix-serve-ng`[247]: A drop-in replacement for `nix-serve`, written in Haskell

[243] https://letsencrypt.org/
[244] https://nix.dev/manual/nix/latest/advanced-topics/distributed-builds
[245] https://search.nixos.org/options?query=nix.gc
[246] https://search.nixos.org/options?query=nix.optimise
[247] https://github.com/aristanetworks/nix-serve-ng

- The SSH Store[248], Experimental SSH Store[249], and the S3 Binary Cache Store[250] can also be used to serve a cache. There are many commercial providers for S3-compatible storage, for example:
  - Amazon S3
  - Tigris
  - Cloudflare R2
- attic[251]: Nix binary cache server backed by an S3-compatible storage provider
- Cachix[252]: Nix binary cache as a service

References

- Nix Manual on HTTP Binary Cache Store[253]
- `services.nix-serve` module options[254]
- `services.nginx` module options[255]

Setting up distributed builds

Nix can speed up builds by spreading the work across multiple computers at once.

Introduction

In this tutorial, you'll set up a separate build machine and configure your local machine to offload builds to it.

What will you learn?

You'll learn how to

- Create a new user for remote build access from a local machine to the remote builder
- Configure remote builders with a sustainable setup
- Test remote builder connectivity and authentication
- Configure the local machine to automatically distribute builds

What do you need?

- Familiarity with the *Nix language* (page 12)
- Familiarity with the *Module system* (page 72)
- A *local machine* (example hostname: `localmachine`)

  The computer *with Nix installed* (page 1) that distributes builds to other machines.
- A *remote machine* (example hostname: `remotemachine`)

  A computer running NixOS that accepts build jobs from the *local machine*. Follow *Provisioning remote machines via SSH* (page 110) to set up a remote NixOS system.

[248] https://nix.dev/manual/nix/latest/store/types/ssh-store
[249] https://nix.dev/manual/nix/latest/store/types/experimental-ssh-store
[250] https://nix.dev/manual/nix/latest/store/types/s3-binary-cache-store
[251] https://github.com/zhaofengli/attic
[252] https://www.cachix.org
[253] https://nix.dev/manual/nix/latest/store/types/http-binary-cache-store
[254] https://search.nixos.org/options?query=services.nix-serve
[255] https://search.nixos.org/options?query=services.nginx

How long will it take?

- 25 minutes

Create an SSH key pair

The *local machine*'s Nix daemon runs as the `root` user and will need the *private* key file to authenticate itself to remote machines. The *remote machine* will need the *public* key to recognize the *local machine*.

On the *local machine*, run the following command as `root` to create an SSH key pair:

```
# ssh-keygen -f /root/.ssh/remotebuild
```

Note

The name and location of the key pair files can be freely chosen.

Set up the remote builder

In the NixOS configuration directory of the *remote machine*, create the file `remote-builder.nix`:

```
{
  users.users.remotebuild = {
    isNormalUser = true;
    createHome = false;
    group = "remotebuild";

    openssh.authorizedKeys.keyFiles = [ ./remotebuild.pub ];
  };

  users.groups.remotebuild = {};

  nix.settings.trusted-users = [ "remotebuild" ];
}
```

Copy the file `remotebuild.pub` into this directory.

This configuration module creates a new user `remotebuild` with no home directory. The `root` user on the *local machine* will be able to log into the remote builder via SSH using the previously generated SSH key.

Add the new NixOS module to the existing configuration of the *remote machine*:

```
{
  imports = [
    ./remote-builder.nix
  ];

  # ...
}
```

Activate the new configuration as root:

```
nixos-rebuild switch --no-flake --target-host root@remotemachine
```

Test authentication

Make sure that the SSH connection and authentication work. On the *local machine*, run as `root`:

```
# ssh remotebuild@remotemachine -i /root/.ssh/remotebuild "echo hello"
Could not chdir to home directory /home/remotebuild: No such file or directory
hello
```

If the `hello` message is visible, the authentication works. The `Could not chdir to ...` message confirms that the remote user has no home directory.

This test login also adds the host key of the remote builder to the `/root/.ssh/known_hosts` file of the local machine. Future logins will not be interrupted by host key checks.

Set up distributed builds

Note

If your *local machine* runs NixOS, skip this section and *configure Nix through module options* (page 129).

Configure Nix to use the remote builder by adding to the Nix configuration file[256] as `root`:

```
# cat << EOF >> /etc/nix/nix.conf
builders = ssh-ng://remotebuild@remotebuilder $(nix-instantiate --eval -E builtins.
↪currentSystem) /root/.ssh/remotemachine - - nixos-test,big-parallel,kvm
builders-use-substitutes = true
```

Detailed explanation

The first line registers the remote machine as a remote builder by specifying

- The protocol, user, and hostname
- The *local machine*'s system type[257]

  This will delegate jobs for that system type to the *remote machine*.
- The location of the SSH key
- A list of supported system features[258]

  This particular list must be specified in order to delegate building compilers and running *NixOS VM tests* (page 105) to remote machines.

See the reference documentation on the `builders` setting[259] for details.

The second line instructs all remote builders to obtain dependencies from its own binary caches instead of from the *local machine*. This assumes that the remote builders' internet connection is at least as fast as the local machine's internet connection.

To activate this configuration, restart the Nix daemon:

Linux

On Linux with `systemd`, run as `root`:

```
# systemctl restart nix-daemon.service
```

[256] https://nix.dev/manual/nix/2.23/command-ref/conf-file
[257] https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-system
[258] https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-system-features
[259] https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-builders

macOS

On macOS, run as `root`:

```
# sudo launchctl stop org.nixos.nix-daemon
# sudo launchctl start org.nixos.nix-daemon
```

NixOS

If your *local machine* runs NixOS, in its configuration directory create the file `distributed-builds.nix`:

```
{ pkgs, ... }:
{
  nix.distributedBuilds = true;
  nix.settings.builders-use-substitutes = true;

  nix.buildMachines = [
    {
      hostName = "remotebuilder";
      sshUser = "remotebuild";
      sshKey = "/root/.ssh/remotebuild";
      system = pkgs.stdenv.hostPlatform.system;
      supportedFeatures = [ "nixos-test" "big-parallel" "kvm" ];
    }
  ];
}
```

Detailed explanation

This configuration module enables distributed builds and adds the remote builder, specifying:

- The SSH hostname and username
- The location of the SSH key
- Which *local machine*'s system type[260]

  This will delegate jobs for that system type to the *remote machine*.
- A list of supported system features[261]

  This particular list must be specified in order to delegate building compilers and running *NixOS VM tests* (page 105) to remote machines.

See the NixOS option documentation on `nix.buildMachines`[262] for details.

The `builders-use-substitutes` instructs all remote builders to obtain dependencies from its own binary caches instead of from the *local machine*. This assumes that the remote builders' internet connection is at least as fast as the local machine's internet connection.

Add the new NixOS module to the existing machine configuration:

```
{
  imports = [
    ./distributed-builds.nix
  ];

  # ...
}
```

Activate the new configuration as `root`:

```
# nixos-rebuild switch
```

Test distributed builds

Try building an new derivation on the *local machine*:

```
$ nix-build --max-jobs 0 -E << EOF
(import <nixpkgs> {}).writeText "test" "$(date)"
EOF
this derivation will be built:
  /nix/store/9csjdxv6ir8ccnjl6ijs36izswjgchn0-test.drv
building '/nix/store/9csjdxv6ir8ccnjl6ijs36izswjgchn0-test.drv' on 'ssh://
↪remotebuilder'...
Could not chdir to home directory /home/remotebuild: No such file or directory
copying 0 paths...
copying 1 paths...
copying path '/nix/store/hvj5vyg4723nly1qh5a8daifbi1yisb3-test' from 'ssh://
↪remotebuilder'...
/nix/store/hvj5vyg4723nly1qh5a8daifbi1yisb3-test
```

The resulting derivation changes on every invocation because it depends on the current system time, and thus can never be in the local cache. The `--max-jobs 0` command line argument[263] forces Nix to build it on the remote builder.

The last output line contains the output path and indicates that build distribution works as expected.

Optimise the remote builder configuration

To maximise parallelism, enable automatic garbage collection, and prevent Nix builds from consuming all memory, add the following lines to your `remote-builder.nix` configuration module:

```
 {
   users.users.remotebuild = {
     isNormalUser = true;
     createHome = false;
     group = "remotebuild";

     openssh.authorizedKeys.keyFiles = [ ./remotebuild.pub ];
   };

   users.groups.remotebuild = {};

-  nix.settings.trusted-users = [ "remotebuild" ];
+  nix = {
+    nrBuildUsers = 64;
+    settings = {
+      trusted-users = [ "remotebuild" ];
+
+      min-free = 10 * 1024 * 1024;
+      max-free = 200 * 1024 * 1024;

+      max-jobs = "auto";
+      cores = 0;
+    };
+  };
```

(continues on next page)

[260] https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-system
[261] https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-system-features
[262] https://search.nixos.org/options?query=nix.buildMachines
[263] https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-max-jobs

(continued from previous page)

```
+  systemd.services.nix-daemon.serviceConfig = {
+    MemoryAccounting = true;
+    MemoryMax = "90%";
+    OOMScoreAdjust = 500;
+  };
}
```

Tip

Refer to the Nix reference manual[264] for details on the options available in `nix.settings`[265].

Remote builders can have different performance characteristics. For each `nix.buildMachines` item, set the `maxJobs`, `speedFactor`, and `supportedFeatures` attributes correctly for each different remote builder. This helps Nix on the *local machine* distributing builds the optimal way.

Tip

Refer to the NixOS option documentation on `nix.buildMachines`[266] for details.

Set the `nix.buildMachines.*.publicHostKey` field to each remote builder's public host key to secure build distribution against man-in-the-middle scenarios.

Next steps

- *Configure Nix to use a custom binary cache* (page 133) on each remote builder
- *Setting up post-build hooks* (page 140) to upload store objects to a binary cache

To set up multiple builders, repeat the instructions in the *Set up the remote builder* (page 127) section for each remote builder. Add all new remote builders to the `nix.buildMachines` attribute shown in the *Set up distributed builds* (page 128) section.

Alternatives

- nixbuild.net[267] - Nix remote builders as a service
- Hercules CI[268] - Continuous integration with automatic build distribution
- garnix[269] - Hosted continuous integration with build distribution

References

- Nix reference manual: Settings for distributed builds[270]

[264] https://nix.dev/manual/nix/2.23/command-ref/conf-file
[265] https://search.nixos.org/options?show=nix.settings
[266] https://search.nixos.org/options?query=nix.buildMachines
[267] https://nixbuild.net
[268] https://hercules-ci.com/
[269] https://garnix.io/
[270] https://nix.dev/manual/nix/latest/command-ref/conf-file#conf-builders

CHAPTER

THREE

GUIDES

These sections contains guides to getting things done.

3.1 Recipes

3.1.1 Configure Nix to use a custom binary cache

Nix can be configured to use a binary cache with the `substituters`[271] and `trusted-public-keys`[272] settings, either exclusively or in addition to cache.nixos.org[273].

Tip

Follow the tutorial to *set up an HTTP binary cache* (page 122) and create a key pair for signing store objects.

For example, given a binary cache at `https://example.org` with public key `My56...Q==%`, and some derivation in `default.nix`, make Nix exclusively use that cache once by passing settings as command line flags[274]:

```
$ nix-build --substituters https://example.org --trusted-public-keys example.
↪org:My56...Q==%
```

To permanently use the custom cache in addition to the public cache, add to the Nix configuration file[275]:

```
$ echo "extra-substituters = https://example.org" >> /etc/nix/nix.conf
$ echo "extra-trusted-public-keys = example.org:My56...Q==%" >> /etc/nix/nix.conf
```

To always use only the custom cache:

```
$ echo "substituters = https://example.org" >> /etc/nix/nix.conf
$ echo "trusted-public-keys = example.org:My56...Q==%" >> /etc/nix/nix.conf
```

NixOS

On NixOS, Nix is configured through the `nix.settings`[276] option:

```
{ ... }: {
  nix.settings = {
    substituters = [ "https://example.org" ];
    trusted-public-keys = [ "example.org:My56...Q==%" ];
  };
}
```

[271] https://nix.dev/manual/nix/2.21/command-ref/conf-file.html#conf-substituters
[272] https://nix.dev/manual/nix/2.21/command-ref/conf-file.html#conf-trusted-public-keys
[273] http://cache.nixos.org
[274] https://nix.dev/manual/nix/2.21/command-ref/conf-file#command-line-flags
[275] https://nix.dev/manual/nix/2.21/command-ref/conf-file#configuration-file

3.1.2 Automatic environment activation with `direnv`

Instead of manually activating the environment for each project, you can reload a *declarative shell* (page 8) every time you enter the project's directory or change the `shell.nix` inside it.

1. Make nix-direnv available[277]
2. Hook it into your shell[278]

For example, write a `shell.nix` with the following contents:

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in

pkgs.mkShellNoCC {
  packages = with pkgs; [
    hello
  ];
}
```

From the top-level directory of your project run:

```
$ echo "use nix" > .envrc && direnv allow
```

The next time you launch your terminal and enter the top-level directory of your project, `direnv` will automatically launch the shell defined in `shell.nix`

```
$ cd myproject
$ which hello
/nix/store/1gxz5nfzfnhyxjdyzi04r86sh61y4i00-hello-2.12.1/bin/hello
```

`direnv` will also check for changes to the `shell.nix` file.

Make the following addition:

```
 let
   nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
   pkgs = import nixpkgs { config = {}; overlays = []; };
 in

 pkgs.mkShellNoCC {
   packages = with pkgs; [
     hello
   ];
+
+  shellHook = ''
+    hello
+  '';
 }
```

The running environment should reload itself after the first interaction (run any command or press `Enter`).

[276] https://search.nixos.org/options?show=nix.settings
[277] https://github.com/nix-community/nix-direnv
[278] https://direnv.net/docs/hook.html

```
Hello, world!
```

3.1.3 Dependencies in the development shell

When *packaging software in `default.nix`* (page 40), you'll want a *development environment in `shell.nix`* (page 8) to enter it conveniently with `nix-shell` or *automatically with `direnv`* (page 134).

How to share the package's dependencies in `default.nix` with the development environment in `shell.nix`?

Summary

Use the `inputsFrom` attribute to `pkgs.mkShellNoCC`[279]:

```
# default.nix
let
  pkgs = import <nixpkgs> {};
  build = pkgs.callPackage ./build.nix {};
in
{
  inherit build;
  shell = pkgs.mkShellNoCC {
    inputsFrom = [ build ];
  };
}
```

Import the `shell` attribute in `shell.nix`:

```
# shell.nix
(import ./.).shell
```

Complete example

Assume your build is defined in `build.nix`:

```
# build.nix
{ cowsay, runCommand }:
runCommand "cowsay-output" { buildInputs = [ cowsay ]; } ''
  cowsay Hello, Nix! > $out
''
```

In this example, `cowsay` is declared as a build-time dependency using `buildInputs`.

Further assume your project is defined in `default.nix`:

```
# default.nix
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
{
  build = pkgs.callPackage ./build.nix {};
}
```

Add an attribute to `default.nix` specifying an environment:

[279] https://nixos.org/manual/nixpkgs/stable/#sec-pkgs-mkShell-attributes

```
 let
   nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
   pkgs = import nixpkgs { config = {}; overlays = []; };
 in
 {
   build = pkgs.callPackage ./build.nix {};
+  shell = pkgs.mkShellNoCC {
+  };
 }
```

Move the `build` attribute into the `let` binding to be able to re-use it. Then take the package's dependencies into the environment with `inputsFrom`[280]:

```
 let
   nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
   pkgs = import nixpkgs { config = {}; overlays = []; };
+  build = pkgs.callPackage ./build.nix {};
 in
 {
-  build = pkgs.callPackage ./build.nix {};
+  inherit build;
   shell = pkgs.mkShellNoCC {
+    inputsFrom = [ build ];
   };
 }
```

Finally, import the `shell` attribute in `shell.nix`:

```
# shell.nix
(import ./.).shell
```

Check the development environment, it contains the build-time dependency `cowsay`:

```
$ nix-shell --pure
[nix-shell]$ cowsay shell.nix
```

Next steps

- *Towards reproducibility: pinning Nixpkgs* (page 11)
- *Automatic environment activation with direnv* (page 134)
- *Setting up a Python development environment* (page 138)
- *Packaging existing software with Nix* (page 40)

3.1.4 Automatically managing remote sources with `npins`

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* (page 11).

For more automation around handling remote sources, set up `npins`[281] in your project:

```
$ nix-shell -p npins --run "npins init --bare; npins add github nixos nixpkgs --
↪branch nixos-23.11"
```

[280] https://nixos.org/manual/nixpkgs/stable/#sec-pkgs-mkShell-attributes
[281] https://github.com/andir/npins/

This command will fetch the latest revision of the Nixpkgs 23.11 release branch. In the current directory it will generate `npins/sources.json`, which will contain a pinned reference to the obtained revision. It will also create `npins/default.nix`, which exposes those dependencies as an attribute set.

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

```
{
  sources ? import ./npins,
  system ? builtins.currentSystem,
  pkgs ? import sources.nixpkgs { inherit system; config = {}; overlays = []; },
}:
{
  package = pkgs.hello;
}
```

`nix-build` will call the top-level function with the empty attribute set `{}`, or with the attributes passed via `--arg`[282] or `--argstr`[283]. This pattern allows *overriding remote sources* (page 137) programmatically.

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

```
 {
   sources ? import ./npins,
   system ? builtins.currentSystem,
   pkgs ? import sources.nixpkgs { inherit system; config = {}; overlays = []; },
 }:
-{
+rec {
   package = pkgs.hello;
+  shell = pkgs.mkShellNoCC {
+    inputsFrom = [ package ];
+    packages = with pkgs; [
+      npins
+    ];
+  };
 }
```

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

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

See *Dependencies in the development shell* (page 135) 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 npins with a different version of Nixpkgs:

```
$ nix-shell
[nix-shell]$ mkdir old
[nix-shell]$ cd old
[nix-shell]$ npins init --bare
[nix-shell]$ npins add github nixos nixpkgs --branch nixos-21.11
```

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

[282] https://nix.dev/manual/nix/stable/command-ref/nix-build#opt-arg
[283] https://nix.dev/manual/nix/stable/command-ref/nix-build#opt-argstr

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

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 ./npins'
```

Migrating from `niv`

A previous version of this guide recommended using `niv`[284], a similar pin manager written in Haskell.

If you have a project using `niv`, you can import remote source definitions into `npins`:

```
npins import-niv
```

Warning

All the imported entries will be updated, so they won't necessarily point to the same commits as before!

Next steps

- Check the built-in help for more information:

  ```
  npins --help
  ```

- For more details and examples of the different ways to specify remote sources, see *Towards reproducibility: pinning Nixpkgs* (page 11).

3.1.5 Setting up a Python development environment

In this example you will build a Python web application using the Flask[285] web framework as an exercise. To make best use of it you should be familiar with *defining declarative shell environments* (page 8).

Create a new file called `myapp.py` and add the following code:

```
#!/usr/bin/env python

from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return {
        "message": "Hello, Nix!"
    }

def run():
    app.run(host="0.0.0.0", port=5000)
```

(continues on next page)

[284] https://github.com/nmattia/niv/
[285] https://flask.palletsprojects.com

(continued from previous page)

```
if __name__ == "__main__":
    run()
```

This is a simple Flask application which serves a JSON document with the message `"Hello, Nix!"`.

Create a new file `shell.nix` to declare the development environment:

```
{ pkgs ? import (fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11
↪") {} }:

pkgs.mkShellNoCC {
  packages = with pkgs; [
    (python3.withPackages (ps: [ ps.flask ]))
    curl
    jq
  ];
}
```

This describes a shell environment with an instance of `python3` that includes the `flask` package using `python3.withPackages`[286]. It also contains `curl`[287], a utility to perform web requests, and `jq`[288], a tool to parse and format JSON documents.

Both of them are not Python packages. If you went with Python's virtualenv[289], it would not be possible to add these utilities to the development environment without additional manual steps.

Run `nix-shell` to enter the environment you just declared:

```
$ nix-shell
these 2 derivations will be built:
  /nix/store/5yvz7zf8yzck6r9z4f1br9sh71vqkimk-builder.pl.drv
  /nix/store/aihgjkf856dbpjjqalgrdmxyyd8a5j2m-python3-3.9.13-env.drv
these 93 paths will be fetched (109.50 MiB download, 468.52 MiB unpacked):
  /nix/store/0xxjx37fcy2nl3yz6igmv4mag2a7giq6-glibc-2.33-123
  /nix/store/138azk9hs5a2yp3zzx6iy1vdwi9q26wv-hook
...

[nix-shell:~]$
```

Start the web application within this shell environment:

```
[nix-shell:~]$ python ./myapp.py
 * Serving Flask app 'myapp'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.␣
↪Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.1.100:5000
Press CTRL+C to quit
```

You now have a running Python web application. Try it out!

Open a new terminal to start another session of the shell environment and follow the commands below:

[286] https://nixos.org/manual/nixpkgs/stable/#python.withpackages-function
[287] https://search.nixos.org/packages?show=curl
[288] https://search.nixos.org/packages?show=jq
[289] https://virtualenv.pypa.io/en/latest/

```
$ nix-shell

[nix-shell:~]$ curl 127.0.0.1:5000
{"message":"Hello, Nix!"}

[nix-shell:~]$ curl 127.0.0.1:5000 | jq '.message'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    26  100    26    0     0  13785      0 --:--:-- --:--:-- --:--:-- 26000
"Hello, Nix!"
```

As demonstrated, you can use both `curl` and `jq` to test the running web application without any manual installation.

You can commit the files we created to version control and share them with other people. Others can now use the same shell environment as long as they have *Nix installed* (page 1).

Next steps

- *Packaging existing software with Nix* (page 40)
- *Working with local files* (page 57)
- *Automatic environment activation with direnv* (page 134)
- *Automatically managing remote sources with npins* (page 136)

3.1.6 Setting up post-build hooks

This guide shows how to use the Nix `post-build-hook`[290] configuration option to automatically upload build results to an S3-compatible binary cache[291].

Implementation caveats

This is a simple and working example, but it is not suitable for all use cases.

The post-build hook program runs after each executed build, and blocks the build loop. The build loop exits if the hook program fails.

Concretely, this implementation will make Nix slow or unusable when the network connection is slow or unreliable. A more advanced implementation might pass the store paths to a user-supplied daemon or queue for processing the store paths outside of the build loop.

Prerequisites

This tutorial assumes you have configured an S3-compatible binary cache[292], and that the `root` user's default AWS profile can upload to the bucket.

Set up a signing key

Use `nix-store --generate-binary-cache-key`[293] to create a pair of cryptographic keys. You will sign paths with the private key, and distribute the public key for verifying the authenticity of the paths.

```
$ nix-store --generate-binary-cache-key example-nix-cache-1 /etc/nix/key.private /
↪etc/nix/key.public
$ cat /etc/nix/key.public
example-nix-cache-1:1/cKDz3QCCOmwcztD2eV6Coggp6rqc9DGjWv7C0G+rM=
```

[290] https://nix.dev/manual/nix/2.22/command-ref/conf-file#conf-post-build-hook
[291] https://nix.dev/manual/nix/2.22/store/types/s3-binary-cache-store
[292] https://nix.dev/manual/nix/2.22/store/types/s3-binary-cache-store#authenticated-writes-to-your-s3-compatible-binary-cache
[293] https://nix.dev/manual/nix/2.22/command-ref/nix-store/generate-binary-cache-key

Configure Nix to use a custom binary cache (page 133) on any machine that will access the bucket. For example, add the cache URL to `substituters`[294] and the public key to `trusted-public-keys`[295] in `nix.conf`:

```
substituters = https://cache.nixos.org/ s3://example-nix-cache
trusted-public-keys = cache.nixos.org-
↪1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY= example-nix-cache-1:1/
↪cKDz3QCCOmwcztD2eV6Coggp6rqc9DGjWv7C0G+rM=
```

Machines that build for the cache must sign derivations using the private key. The path to the file containing the private key you just generated must be added to the `secret-key-files`[296] setting for those machines:

```
secret-key-files = /etc/nix/key.private
```

Implementing the build hook

Write the following script to `/etc/nix/upload-to-cache.sh`:

```
#!/bin/sh
set -eu
set -f # disable globbing
export IFS=' '
echo "Uploading paths" $OUT_PATHS
exec nix copy --to "s3://example-nix-cache" $OUT_PATHS
```

The `$OUT_PATHS` variable is a space-separated list of Nix store paths. In this case, we expect and want the shell to perform word splitting to make each output path its own argument to `nix store sign`. Nix guarantees the paths will not contain any spaces, however a store path might contain glob characters. The `set -f` disables globbing in the shell.

Make sure the hook program is executable by the `root` user:

```
# chmod +x /etc/nix/upload-to-cache.sh
```

Updating Nix configuration

Set the `post-build-hook`[297] configuration option on the local machine to run the hook:

```
post-build-hook = /etc/nix/upload-to-cache.sh
```

Then restart the `nix-daemon` an all involved machines, e.g. with

```
pkill nix-daemon
```

Testing

Build any derivation, for example:

```
$ nix-build -E '(import <nixpkgs> {}).writeText "example" (builtins.toString␣
↪builtins.currentTime)'
this derivation will be built:
  /nix/store/s4pnfbkalzy5qz57qs6yybna8wylkig6-example.drv
building '/nix/store/s4pnfbkalzy5qz57qs6yybna8wylkig6-example.drv'...
running post-build-hook '/home/grahamc/projects/github.com/NixOS/nix/post-hook.sh'.
↪..
```

(continues on next page)

[294] https://nix.dev/manual/nix/2.22/command-ref/conf-file#conf-substituters
[295] https://nix.dev/manual/nix/2.22/command-ref/conf-file#conf-trusted-public-keys
[296] https://nix.dev/manual/nix/2.22/command-ref/conf-file#conf-secret-key-files
[297] https://nix.dev/manual/nix/2.22/command-ref/conf-file#conf-post-build-hook

(continued from previous page)

```
post-build-hook: Signing paths /nix/store/ibcyipq5gf91838ldx40mjsp0b8w9n18-example
post-build-hook: Uploading paths /nix/store/ibcyipq5gf91838ldx40mjsp0b8w9n18-
↪example
/nix/store/ibcyipq5gf91838ldx40mjsp0b8w9n18-example
```

To check that the hook took effect, delete the path from the store, and try substituting it from the binary cache:

```
$ rm ./result
$ nix-store --delete /nix/store/ibcyipq5gf91838ldx40mjsp0b8w9n18-example
$ nix-store --realise /nix/store/ibcyipq5gf91838ldx40mjsp0b8w9n18-example
copying path '/nix/store/m8bmqwrch6l3h8s0k3d673xpmipcdpsa-example from 's3://
↪example-nix-cache'...
warning: you did not specify '--add-root'; the result might be removed by the␣
↪garbage collector
/nix/store/m8bmqwrch6l3h8s0k3d673xpmipcdpsa-example
```

Conclusion

You have configured Nix to automatically sign and upload every local build to a remote S3-compatible binary cache.

Before deploying this to production, be sure to consider the *implementation caveats* (page 140).

3.1.7 Continuous integration with GitHub Actions

In this guide, we'll show you **a few short steps** to get started using GitHub Actions[298] 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) is the most straightforward.

Caching builds using Cachix

Using Cachix[299] 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[300] 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`).

[298] https://github.com/features/actions
[299] https://cachix.org/
[300] https://app.cachix.org/cache

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@v4
    - uses: cachix/install-nix-action@v25
      with:
        nix_path: nixpkgs=channel:nixos-unstable
    - uses: cachix/cachix-action@v14
      with:
        name: mycache
        # If you chose signing key for write access
        signingKey: '${{ secrets.CACHIX_SIGNING_KEY }}'
        # If you chose API tokens for write access OR if you have a private cache
        authToken: '${{ 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.

Next steps

- See GitHub Actions workflow syntax[301]
- To quickly setup a Nix project read through Getting started Nix template[302].

3.2 Best practices

3.2.1 URLs

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

RFC 45[303] was accepted to deprecate unquoted URLs and provides a number of arguments how this feature does more harm than good.

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

3.2.2 Recursive attribute set `rec { ... }`

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

Example:

[301] https://docs.github.com/en/actions/reference/workflow-syntax-for-github-actions
[302] https://github.com/nix-dot-dev/getting-started-nix-template
[303] https://github.com/NixOS/rfcs/pull/45

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

```
{ 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:

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

Tip

Self-reference can be achieved by explicitly naming the attribute set:

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

3.2.3 `with` scopes

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

```
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` is 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[304].

[304] https://github.com/NixOS/nix/issues/490

Tip

Do not use `with` at the top of a Nix file. Explicitly assign names in a `let` expression.

Example:

```
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

```
buildInputs = with pkgs; [ curl jq ];
```

with the following:

```
buildInputs = builtins.attrValues {
  inherit (pkgs) curl jq;
};
```

3.2.4 `<...>` lookup paths

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

`<...>` is special syntax that was introduced in 2011[305] to conveniently access values from the environment variable `$NIX_PATH`[306].

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[307] 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* (page 11).

Do not use lookup paths, except in minimal examples.

[305] https://github.com/NixOS/nix/commit/1ecc97b6bdb27e56d832ca48cdafd3dbb5185a04
[306] https://nix.dev/manual/nix/stable/command-ref/env-common.html#env-NIX_PATH
[307] https://nix.dev/manual/nix/stable/command-ref/nix-channel.html

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`[308] option.

3.2.5 Reproducible Nixpkgs configuration

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

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

However, even when `<nixpkgs>` is replaced as shown in *Towards reproducibility: pinning Nixpkgs* (page 11), the result may still not be fully reproducible. This is because, for historical reasons, the Nixpkgs top-level expression[309] 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`[310] and `overlays`[311] when importing Nixpkgs:

```
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 to reduce distractions.

3.2.6 Updating nested attribute sets

The attribute set update operator[312] merges two attribute sets.

Example:

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

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

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

Example:

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

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

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

[308] https://search.nixos.org/options?show=nix.nixPath
[309] https://github.com/NixOS/nixpkgs/blob/master/default.nix
[310] https://nixos.org/manual/nixpkgs/stable/#chap-packageconfig
[311] https://nixos.org/manual/nixpkgs/stable/#chap-overlays
[312] https://nix.dev/manual/nix/stable/language/operators.html#update

Tip

Use the `pkgs.lib.recursiveUpdate`[313] Nixpkgs function:

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

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

3.2.7 Reproducible source paths

```
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 be 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`[314] 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:

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

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

3.3 Troubleshooting

This page is a collection of tips to solve problems you may encounter using Nix.

3.3.1 What to do if a binary cache is down or unreachable?

Pass `--option substitute false`[315] to Nix commands.

3.3.2 How to force Nix to re-check if something exists in the binary cache?

Nix keeps track of what's available in binary caches so it doesn't have to query them on every command. This includes negative answers, that is, if a given store path cannot be substituted.

[313] https://nixos.org/manual/nixpkgs/stable/#function-library-lib.attrsets.recursiveUpdate
[314] https://nix.dev/manual/nix/stable/language/builtins.html#builtins-path
[315] https://nix.dev/manual/nix/stable/command-ref/conf-file#conf-substitute

Pass the `--narinfo-cache-negative-ttl`[316] option to set the cache timeout in seconds.

3.3.3 How to fix: `error: querying path in database: database disk image is malformed`

This is a known issue[317]. Try:

```
$ sqlite3 /nix/var/nix/db/db.sqlite "pragma integrity_check"
```

Which will print the errors in the database[318]. If the errors are due to missing references, the following may work:

```
$ mv /nix/var/nix/db/db.sqlite /nix/var/nix/db/db.sqlite-bkp
$ sqlite3 /nix/var/nix/db/db.sqlite-bkp ".dump" | sqlite3 /nix/var/nix/db/db.sqlite
```

3.3.4 How to fix: `error: current Nix store schema is version 10, but I only support 7`

This is a known issue[319].

It means that using a new version of Nix upgraded the SQLite schema of the database[320], and then you tried to use an older version Nix.

The solution is to dump the database, and use the old Nix version to re-import the data:

```
$ /path/to/nix/unstable/bin/nix-store --dump-db > /tmp/db.dump
$ mv /nix/var/nix/db /nix/var/nix/db.toonew
$ mkdir /nix/var/nix/db
$ nix-store --load-db < /tmp/db.dump
```

3.3.5 How to fix: `writing to file: Connection reset by peer`

This may mean you are trying to import a too large file or directory into the Nix store[321], or your machine is running out of resources, such as disk space or memory.

Try to reduce the size of the directory to import, or run garbage collection[322].

3.3.6 macOS update breaks Nix installation

This is a known issue[323]. The Nix installer[324] modifies `/etc/zshrc`. When macOS is updated, it will typically overwrite `/etc/zshrc` again.

As a workaround, add the following code snippet to the end of `/etc/zshrc` and restart the shell:

```
if [ -e '/nix/var/nix/profiles/default/etc/profile.d/nix-daemon.sh' ]; then
  . '/nix/var/nix/profiles/default/etc/profile.d/nix-daemon.sh'
fi
```

[316] https://nix.dev/manual/nix/stable/command-ref/conf-file.html#conf-narinfo-cache-negative-ttl
[317] https://github.com/NixOS/nix/issues/1353
[318] https://nix.dev/manual/nix/stable/glossary#gloss-nix-database
[319] https://github.com/NixOS/nix/issues/1251
[320] https://nix.dev/manual/nix/stable/glossary#gloss-nix-database
[321] https://nix.dev/manual/nix/stable/glossary#gloss-store
[322] https://nix.dev/manual/nix/stable/command-ref/nix-collect-garbage
[323] https://github.com/NixOS/nix/issues/3616
[324] https://nix.dev/manual/nix/latest/installation/installing-binary

3.4 Frequently Asked Questions

3.4.1 Nix

How to format Nix language code automatically?

`nixfmt`[325] is the official formatter for *Nix language* code. Please refer to its source repository for installation instructions.

`nixfmt` is used to format all code[326] in *Nixpkgs*.

How to convert between paths and strings in the Nix language?

See the Nix reference manual on string interpolation[327] and operators on paths and strings[328]

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 https://github.com/nix-community/home-manager

What's the recommended process for building custom packages?

Please read *Packaging existing software with Nix* (page 40).

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

Please read *Packaging existing software with Nix* (page 40) and the Nixpkgs contributing guide[329].

3.4.2 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[330] (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* (page 40).

  - Modify the program's ELF header[331] to include paths to libraries using `autoPatchelfHook`[332].

    Do this if building from source isn't feasible.

[325] https://github.com/NixOS/nixfmt
[326] https://github.com/NixOS/nixpkgs/blob/master/ci/default.nix
[327] https://nix.dev/manual/nix/2.19/language/string-interpolation
[328] https://nix.dev/manual/nix/2.19/language/operators#string-concatenation
[329] https://github.com/NixOS/nixpkgs/blob/master/CONTRIBUTING.md
[330] https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html
[331] https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
[332] https://nixos.org/manual/nixpkgs/stable/#setup-hook-autopatchelfhook

  - Wrap the program to run in an FHS-like environment using `buildFHSEnv`[333].

    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`[334]. 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`[335]:

```
$ 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/test-
↪driver.pl
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.

[333] https://nixos.org/manual/nixpkgs/stable/#sec-fhs-environments
[334] https://github.com/Mic92/nix-ld
[335] https://nixos.org/manual/nixpkgs/stable/#sec-steam-run

How to bootstrap NixOS inside an existing Linux installation?

There are a couple of tools:

- https://github.com/nix-community/nixos-anywhere
- https://github.com/jeaye/nixos-in-place
- https://github.com/elitak/nixos-infect
- https://github.com/cleverca22/nix-tests/tree/master/kexec

CHAPTER

FOUR

REFERENCE

These sections contains collections of detailed technical descriptions.

4.1 Glossary

Nix
: Build system and package manager.

  Read /nɪks/ (“Niks”).

  See also

  - *Nix reference manual* (page 154)
  - Nix source code[336]

Nix language
: Programming language to declare packages and configurations for *Nix*.

  See also

  - *Nix language basics* (page 12)
  - Nix language reference[337]

Nix expression
: Expression written in the *Nix language*.

Nix file
: File (`.nix`) containing a *Nix expression*.

Nixpkgs
: Software distribution built with *Nix*.

  Read /nɪks ˈpækɪʤɪz/ (“Nix packages”).

  See also

  - Nixpkgs reference manual[338]
  - Nixpkgs source code[339]

[336] https://github.com/NixOS/nix
[337] https://nix.dev/manual/nix/stable/language
[338] https://nixos.org/manual/nixpkgs
[339] https://github.com/NixOS/nixpkgs

NixOS
: Linux distribution based on *Nix* and *Nixpkgs*.

  Read /nɪks oʊ ɛs/ (“Niks Oh Es”).

See also

- NixOS reference manual[340]
- NixOS source code[341]

4.2 Nix reference manual

The Nix reference manual is available for multiple versions:

- Nix pre-release[342]

  Development build from the `master` branch of the Nix repository[343]

- Nix 2.29[344] (single page[345])

  Latest Nix release

- Nix 2.28[346] (single page[347])

  Shipped with the rolling release of *Nixpkgs* and *NixOS*

- Nix 2.28[348] (single page[349])

  Shipped with the current stable release of *Nixpkgs* and *NixOS*: 25.05

- Nix 2.24[350] (single page[351])

  Shipped with the previous stable release of *Nixpkgs* and *NixOS*: 24.11

Tip

More information on Nixpkgs and NixOS releases: *Which channel branch should I use?* (page 162)

4.3 Further reading

4.3.1 Nix language tutorials

- Nix language one-pager reference[352] (Vincent Ambo, 2019-2021)

  Overview of common language features and widely used idioms.

- A tour of Nix[353] (Joachim Schiele, 2015-2022)

  Interactive exercises with the Nix language.

[340] https://nixos.org/manual/nixos
[341] https://github.com/NixOS/nixpkgs/tree/master/nixos
[342] https://nix.dev/manual/nix/development/
[343] https://github.com/NixOS/nix
[344] https://nix.dev/manual/nix/latest/
[345] https://nix.dev/manual/nix/latest/nix-2.29.html
[346] https://nix.dev/manual/nix/rolling/
[347] https://nix.dev/manual/nix/rolling/nix-2.28.html
[348] https://nix.dev/manual/nix/stable/
[349] https://nix.dev/manual/nix/stable/nix-2.28.html
[350] https://nix.dev/manual/nix/prev-stable/
[351] https://nix.dev/manual/nix/prev-stable/nix-2.24.html
[352] https://github.com/tazjin/nix-1p
[353] https://nixcloud.io/tour

- Video: Nix language overview[354] (Wil Taylor, 2021)

  Overview of language features.

- Video: Reading the Nix language[355] (Jonas Chevalier, 2019)

  Introduction to reading Nix language code.

- Video: How and Why it Works[356] (Graham Christensen, 2019)

  Introduction to writing derivations.

4.3.2 Other articles

- Nix Pills[357]

  A low-level tutorial on building software packages with Nix, showing in detail how Nixpkgs is constructed.

- Customizing packages in Nix[358] (2022)

  An overview of different methods to customize Nix packages.

- Manage your dot files with Home Manager[359] (Mattia Gheda, 2021)

  A tutorial for getting started with Home Manager.

- Nix Shorts[360]

  A series of posts on basics of how packaging with Nix works.

- NixOS and Flakes - An unofficial book for beginners[361] (2023)

  This tutorial is an introduction to *NixOS* using the experimental Nix *Flakes* (page 159) functionality.

4.3.3 Other videos

- Nixology[362] (Burke Libbey, 2020)

  Video series introducing fundamental Nix concepts.

- The Nix Hour[363] (Silvan Mosberger, since 2022)

  Weekly series exploring topics and answering questions from all around the Nix ecosystem.

- Nixpkgs[364] (Jon Ringer, 2020-22)

  Video series with tutorials on various activities around Nixpkgs.

- NixOS[365] (Wil Taylor, 2021)

  Series of tutorials on getting started with NixOS.

- NixOS Foundation on YouTube[366]

  The official NixOS Foundation channel.

[354] https://www.youtube.com/watch?v=eCapIx9heBw&list=PL-saUBvIJzOkjAw_vOac75v-x6EzNzZq-&index=5
[355] https://youtu.be/hbJkMl631FE?t=1533
[356] https://youtu.be/hbJkMl631FE?t=4806
[357] https://nixos.org/nixos/nix-pills/index.html
[358] https://bobvanderlinden.me/customizing-packages-in-nix/
[359] https://ghedam.at/24353/tutorial-getting-started-with-home-manager-for-nix
[360] https://github.com/justinwoo/nix-shorts
[361] https://nixos-and-flakes.thiscute.world
[362] https://www.youtube.com/playlist?list=PLRGI9KQ3_HP_OFRG6R-p4iFgMSK1t5BHs
[363] https://www.youtube.com/playlist?list=PLyzwHTVJlRc8yjlx4VR4LU5A5O44og9in
[364] https://www.youtube.com/@jonringer117/videos
[365] https://www.youtube.com/playlist?list=PL-saUBvIJzOkjAw_vOac75v-x6EzNzZq-
[366] https://www.youtube.com/@NixOS-Foundation/playlists

- NixCon on YouTube[367]

  Recordings of NixCon talks and discussions.

4.4 Pinning Nixpkgs

Specifying remote Nix expressions, such as the one provided by Nixpkgs, can be done in several ways:

- `$NIX_PATH` environment variable[368]
- `-I` option[369] to most of commands like `nix-build`, `nix-shell`, etc.
- `fetchurl`[370], `fetchTarball`[371], `fetchGit`[372] or Nixpkgs fetchers[373] in Nix expressions

4.4.1 Possible URL values

- Local file path:

```
./path/to/expression.nix
```

  Using `./.` means that the expression is located in a file `default.nix` in the current directory.

- Pinned to a specific commit:

```
https://github.com/NixOS/nixpkgs/archive/
↪eabc38219184cc3e04a974fe31857d8e0eac098d.tar.gz
```

- Using the latest channel version, meaning all tests have passed:

```
http://nixos.org/channels/nixos-22.11/nixexprs.tar.xz
```

- Shorthand syntax for channels:

```
channel:nixos-22.11
```

- Using the latest channel version, hosted by GitHub:

```
https://github.com/NixOS/nixpkgs/archive/nixos-22.11.tar.gz
```

- Using the latest commit on the release branch, but not tested yet:

```
https://github.com/NixOS/nixpkgs/archive/release-21.11.tar.gz
```

4.4.2 Examples

-
```
$ nix-build -I ~/dev
```

-
```
$ nix-build -I nixpkgs=http://nixos.org/channels/nixos-22.11/nixexprs.tar.xz
```

-
```
$ nix-build -I nixpkgs=channel:nixos-22.11
```

-
```
$ NIX_PATH=nixpkgs=http://nixos.org/channels/nixos-22.11/nixexprs.tar.xz nix-
↪build
```

[367] https://www.youtube.com/@NixCon
[368] https://nix.dev/manual/nix/stable/command-ref/env-common.html#env-NIX_PATH
[369] https://nix.dev/manual/nix/stable/command-ref/opt-common.html#opt-I
[370] https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchurl
[371] https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchTarball
[372] https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchGit
[373] https://nixos.org/manual/nixpkgs/stable/#chap-pkgs-fetchers

- ```
  $ NIX_PATH=nixpkgs=channel:nixos-22.11 nix-build
  ```

- In the Nix language:

```
let
  pkgs = import (fetchTarball "https://github.com/NixOS/nixpkgs/archive/nixos-22.11.tar.gz") {};
in pkgs.stdenv.mkDerivation { ... }
```

4.4.3 Finding specific commits and releases

status.nixos.org[374] provides:

- Latest tested commits for each release - use when pinning to specific commits
- List of active release channels - use when tracking latest channel versions

The complete list of channels is available at nixos.org/channels[375].

Tip

More information on Nixpkgs and NixOS releases: *Which channel branch should I use?* (page 162)

[374] https://status.nixos.org/
[375] https://nixos.org/channels
```

CHAPTER

FIVE

CONCEPTS

These sections contains explanations of history and ideas in the Nix ecosystem.

5.1 Flakes

What is usually referred to as "flakes" is:

- A policy for managing dependencies between *Nix expressions*.
- An experimental feature[376] in Nix, implementing that policy and supporting functionality.

5.1.1 What are flakes?

Technically, a flake[377] is a file system tree that contains a file named `flake.nix` in its root directory.

Flakes add the following behavior to Nix:

1. A `flake.nix` file enforces a schema[378], where:
   - Other flakes are referenced as dependencies providing *Nix language* code or other files.
   - The values produced by the *Nix expression*s in `flake.nix` are structured according to pre-defined use cases.
2. References to other flakes can be specified using a dedicated URL-like syntax[379]. A flake registry[380] allows using symbolic identifiers for further brevity. References can be automatically locked to their current specific version and later updated programmatically.
3. A new command line interface[381], implemented as a separate experimental feature, leverages flakes by accepting flake references in order to build, run, or deploy software defined as a flake.

Nix handles flakes differently than regular *Nix files* in the following ways:

- The `flake.nix` file is checked for schema validity.

  In particular, the metadata fields cannot be arbitrary Nix expressions. This is to prevent complex, possibly non-terminating computations while querying the metadata.

- The entire flake directory is copied to Nix store before evaluation.

  This allows for effective evaluation caching, which is relevant for large expressions such as Nixpkgs, but also requires copying the entire flake directory again on each change.

- No external variables, parameters, or impure language values are allowed.

  It means full reproducibility of a Nix expression, and, by extension, the resulting build instructions by default, but also prohibits parameterisation of results by consumers.

[376] https://nix.dev/manual/nix/stable/contributing/experimental-features
[377] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-flake.html#description
[378] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-flake.html#flake-format
[379] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-flake.html#flake-references
[380] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-registry.html
[381] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix.html

5.1.2 Why are flakes controversial?

Flakes (page 159) were inspired by Shea Levy's NixCon 2018 talk[382], formally proposed in RFC 49[383], and have been in development since 2019. Nix introduced the implementation as its first experimental feature[384] in 2021.

The subject is considered controversial among Nix users and developers in terms of design, implementation quality, and decision making process. In particular:

- The RFC was closed without conclusion, and some fundamental issues are not yet resolved. For example:
  - The notion of a global flake registry[385] saw substantial criticism[386] that was never addressed. While the source references of registry entries can be pinned[387], local registry names in Nix expressions introduce mutable system state[388] and are thus, in that regard, no improvement over channels as managed by `nix-channel`[389].
  - It is impossible to parameterise flakes[390]. This means that flakes downgrade ease of use of the `system` parameter[391] of derivations, for producers and consumers.
  - the flakes proposal was criticised for trying to solve too many problems at once[392] and at the wrong abstraction layer[393]. Part of this is that the new command line interface and flakes are closely tied to each other[394].
- As predicted by RFC reviewers[395], the original implementation introduced regressions[396] in the Nix 2.4 release[397], breaking some stable functionality without a major version[398] increment.
- Copying sources to the Nix store prior to evaluation adds a significant performance penalty[399], especially for large repositories such as *Nixpkgs*. Work to address this has been in progress since May 2022[400], but risks introducing its own set of issues[401].
- New Nix users were and still are encouraged by various individuals to adopt flakes despite there being no stability guarantees and no timeline to conclude the experiment.

This led to a situation where the stable interface was only sparsely maintained for multiple years, and repeatedly suffered breakages due to ongoing development. Meanwhile, the new interface was adopted widely enough for evolving its design without negatively affecting users to become very challenging.

As of the 2023[402] survey, 84% of the respondents rely on experimental features. *Nixpkgs* as a contrasting example, while featuring a `flake.nix` for compatibility, does not depend on Nix experimental features in its code base.

5.1.3 Should I use flakes in my project?

You have to judge for yourself based on your needs.

Flakes (page 159) emphasize reproducible artifacts and convenience for their consumers, while classic Nix tools center around composable building blocks and customisation options for developers. Both paradigms have their own set of unique concepts and support tooling that have to be learned, with varying ease of use, implementation quality,

[382] https://www.youtube.com/watch?v=DHOLjsyXPtM
[383] https://github.com/NixOS/rfcs/pull/49
[384] https://nix.dev/manual/nix/stable/contributing/experimental-features
[385] https://github.com/NixOS/flake-registry
[386] https://github.com/NixOS/rfcs/pull/49#issuecomment-635635333
[387] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-registry-pin
[388] https://github.com/NixOS/nix/issues/7422
[389] https://nix.dev/manual/nix/stable/command-ref/nix-channel
[390] https://github.com/NixOS/nix/issues/2861
[391] https://github.com/NixOS/nix/issues/3843
[392] https://github.com/nixos/rfcs/pull/49#issuecomment-521998933
[393] https://discourse.nixos.org/t/nixpkgs-cli-working-group-member-search/30517
[394] https://discourse.nixos.org/t/2023-03-06-nix-team-meeting-minutes-38/26056#cli-stabilisation-announcement-draft-4
[395] https://github.com/NixOS/rfcs/pull/49#issuecomment-588990425
[396] https://discourse.nixos.org/t/nix-2-4-and-what-s-next/16257
[397] https://nix.dev/manual/nix/stable/release-notes/rl-2.4.html
[398] https://semver.org/
[399] https://github.com/NixOS/nix/issues/3121
[400] https://github.com/NixOS/nix/pull/6530
[401] https://github.com/NixOS/nix/pull/6530#issuecomment-1850565931
[402] https://discourse.nixos.org/t/nix-community-survey-2023-results/33124

and support status. At the moment, neither the stable nor the experimental interface is clearly superior to the other in all aspects.

Flakes and the `nix` command suite bring multiple improvements that are relevant for both software users and package authors:

- The new command-line interface, together with flakes, makes dealing with existing packages significantly more convenient in many cases.
- The constraints imposed on flakes strengthen reproducibility by default, and enable some performance improvements when interacting with a large Nix package repository like *Nixpkgs*.
- Flake references allow for easier handling of version upgrades for existing packages or project dependencies.
- The flake schema[403] helps with composing Nix projects from multiple sources in an orderly fashion.

At the same time, flakes have *fundamental architectural issues* (page 160) and a number of problems with the implementation[404], and there is no coordinated effort to resolve them systematically. There are also still many open design questions around the `nix` command line interface[405], some of which are currently being worked on.

While flakes reduce complexity in some regards, they also introduce some with additional mechanisms. You will have to learn more about the system to fully understand how it works.

Other than that, and below the surface of the flake schema, Nix and the Nix language work exactly the same in both cases. In principle, the same level of reproducibility can be achieved with or without flakes. In particular, the process of adding software to *Nixpkgs* or maintaining *NixOS* modules and configurations is not affected by flakes at all. There is also no evidence that flakes could help solving the scalability challenges of either.

Finally, there are downsides to relying on experimental features[406] in general:

- Interfaces and behavior of experimental features could still be changed by Nix developers. This may require you to adapt your code at some point in the future, which will be more effort when it has grown in complexity. Currently there is no concrete timeline for stabilising flakes.[407] In contrast, stable features in Nix can be considered stable indefinitely.
- The Nix maintainer team[408] focuses on fixing bugs and regressions in stable interfaces, supporting well-understood use cases, as well as improving the internal design and overall contributor experience in order to ease future development. Improvements to experimental features have low priority.
- The Nix documentation team[409] focuses on improving documentation and learning materials for stable features and common principles. When using flakes, you will have to rely more heavily on user-to-user support, third-party documentation, and the source code.

5.1.4 Further reading

- Flakes aren't real and cannot hurt you: a guide to using Nix flakes the non-flake way[410] (Jade Lovelace, January 2024)
- Nix Flakes is an experiment that did too much at once…[411] (comments[412]) (Samuel Dionne-Riel, September 2023)
- Experimental does not mean unstable[413] (comments[414]) (Graham Christensen, September 2023)
- The Nix Hour: comparing flakes to traditional Nix[415] (Silvan Mosberger, November 2022)

[403] https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-flake.html#flake-format
[404] https://github.com/NixOS/nix/issues?q=is%3Aissue+is%3Aopen+label%3Aflakes+sort%3Areactions-%2B1-desc
[405] https://github.com/NixOS/nix/issues?q=is%3Aissue+is%3Aopen+label%3Anew-cli+sort%3Areactions-%2B1-desc
[406] https://nix.dev/manual/nix/stable/contributing/experimental-features
[407] https://discourse.nixos.org/t/stabilising-the-new-nix-command-line-interface/35531#p-123372-how-does-this-relate-to-flakes-3
[408] https://nixos.org/community/teams/nix.html
[409] https://nixos.org/community/teams/documentation.html
[410] https://jade.fyi/blog/flakes-arent-real/
[411] https://samuel.dionne-riel.com/blog/2023/09/06/flakes-is-an-experiment-that-did-too-much-at-once.html
[412] https://discourse.nixos.org/t/nix-flakes-is-an-experiment-that-did-too-much-at-once/32707
[413] https://determinate.systems/posts/experimental-does-not-mean-unstable
[414] https://discourse.nixos.org/t/experimental-does-not-mean-unstable-detsyss-perspective-on-nix-flakes/32703
[415] https://www.youtube.com/watch?v=atmoYyBAhF4

5.2 Frequently Asked Questions

5.2.1 What is the origin of the name Nix?

> The name *Nix* is derived from the Dutch word *niks*, meaning *nothing*; build actions do not see anything that has not been explicitly declared as an input.
>
> — Nix: A Safe and Policy-Free System for Software Deployment[416], LISA XVIII, 2004

The Nix logo is inspired by an idea for the Haskell logo[417] and the fact that *nix* is Latin for *snow*[418].

5.2.2 What are flakes?

See *Flakes* (page 159).

5.2.3 Which channel branch should I use?

Nixpkgs and NixOS have both stable and rolling releases.

These releases are distributed in variants called "channel branches": Git branches used for releases, which are also converted to Nix channels.

Tip

Consult the `nix-channel`[419] entry in the Nix Reference Manual for more information on channels, and the Nixpkgs contributing guide[420] on the Nixpkgs branching strategy.

Stable

Stable releases receive conservative updates to fix bugs or security vulnerabilities; otherwise package versions are not changed. A new stable release is made every six months.

- On Linux (including NixOS and WSL), use `nixos-*`[421].

  These branches point to commits where most Linux packages got pre-built and can be fetched from the binary cache. Furthermore, these commits passed the full NixOS test suite.

- On macOS/Darwin, use `nixpkgs-*-darwin`[422]

  These branches point to commits where most Darwin packages got pre-built and can be fetched from the binary cache.

- On any other platform it doesn't matter which one of the above is used.

  Hydra doesn't pre-build any binaries for other platforms.

All of these "channel branches" follow the corresponding `release-*`[423] branch.

Example

`nixos-23.05` and `nixpkgs-23.05-darwin` are both based on `release-23.05`.

[416] https://edolstra.github.io/pubs/nspfssd-lisa2004-final.pdf
[417] https://wiki.haskell.org/File:Sgf-logo-blue.png
[418] https://nix-dev.science.uu.narkive.com/VDaaP1BY/nix-logo
[419] https://nix.dev/manual/nix/2.22/command-ref/nix-channel
[420] https://github.com/NixOS/nixpkgs/blob/master/CONTRIBUTING.md#branch-conventions
[421] https://github.com/NixOS/nixpkgs/branches/all?query=nixos-
[422] https://github.com/NixOS/nixpkgs/branches/all?query=nixpkgs-
[423] https://github.com/NixOS/nixpkgs/branches/all?query=release-

Rolling

Rolling releases follow `master`[424], the main development branch.

- On Linux (including NixOS and WSL), use `nixos-unstable`[425].
- On any other platform, use `nixpkgs-unstable`[426].

`*-small`[427] channel branches have passed a smaller test suite, which means they are more up-to-date with respect to their base branch, but offer fewer stability guarantees.

5.2.4 Are there any impurities left in sandboxed builds?

Yes. There is:

- CPU architecture—great effort being made to avoid compilation of native instructions in favour of hardcoded supported ones.
- System's current time/date.
- The filesystem used for building (see also `TMPDIR`[428]).
- Linux kernel parameters, such as:
  - IPv6 capabilities[429].
  - binfmt interpreters, e.g., those configured with `boot.binfmt.emulatedSystems`[430].
- Timing behaviour of the build system—parallel Make build does not get the correct inputs in some cases.
- Insertion of random values, e.g., from `/dev/random` or `/dev/urandom`.
- Differences between Nix versions. For instance, a new Nix version might introduce a new environment variable. A statement like `env > $out` is not promised by Nix to result in the same output, going into the future.

[424] https://github.com/NixOS/nixpkgs/branches/all?query=master
[425] https://github.com/NixOS/nixpkgs/branches/all?query=nixos-unstable
[426] https://github.com/NixOS/nixpkgs/branches/all?query=nixpkgs-unstable
[427] https://github.com/NixOS/nixpkgs/branches/all?query=-small
[428] https://nix.dev/manual/nix/stable/command-ref/env-common.html#env-TMPDIR
[429] https://github.com/NixOS/nix/issues/5615
[430] https://search.nixos.org/options?show=boot.binfmt.emulatedSystems

CHAPTER

SIX

CONTRIBUTING

6.1 How to contribute

The Nix ecosystem is developed by many volunteers and a few paid developers, maintaining one of the largest open source software distributions in the world. Keeping it working and up to date – and improving it continuously – would not be possible without your support!

This guide shows how you can contribute to *Nix*, *Nixpkgs* or *NixOS*. It assumes that you're already somewhat proficient with basic concepts and workflows, which are outlined in the *beginner tutorial series* (page 3). The most important aspects are *the Nix language* (page 12), the various Nixpkgs mechanisms for *constructing derivations to build software* (page 40), *the module system* (page 72), and *NixOS integration tests* (page 105).

Important

If you cannot contribute time, consider donating to the NixOS Foundation on Open Collective[431].

Currently the focus is on funding in-person events[432] to share knowledge and grow the community of developers proficient with Nix. With enough budget, it would be possible to pay for ongoing maintenance and development of critical infrastructure and code – demanding work that we cannot expect to be done by volunteers indefinitely.

6.1.1 Getting started

Start with asking informed questions, after reading *reference documentation* (page 153) and the code relevant to what you care about.

Join our community communication platforms[433] to get in contact with other users and developers. Check out and consider participating in our community teams[434] if you're interested in a particular topic.

All the source code and documentation is on GitHub[435], and you need a GitHub account to propose changes. Technical discussions happen in issue and pull request comments.

Tip

If you are new to Nix, consider *contributing documentation* (page 169) first.

This is where we need the most help and where it is the easiest to begin.

Documentation and contribution guides are often incomplete or outdated, much as we would like them to be otherwise. We're working on it. You can help and improve the situation for everyone by immediately solving problems with the contribution workflow as you encounter them. This may slow you down with addressing your original concern. But it will make it a lot easier for anyone to make meaningful contributions in the future. And it will lead to better code and documentation in the long run.

[431] https://opencollective.com/nixos
[432] https://github.com/NixOS/foundation/issues?q=is%3Aissue%20label%3Afunding-request%20
[433] https://nixos.org/community
[434] https://nixos.org/community/#governance-teams
[435] https://github.com/NixOS

6.1.2 Report an issue

Note

For asking general questions about the code or how to do things, please use our community communication platforms[436]

To state techincal problems and propose solutions, open GitHub issues and close them when the problem is resolved or invalidated.

We can only fix issues that we know of, so please report any issue you encounter.

- Report issues with *Nix* (including the Nix reference manual[437]) at https://github.com/NixOS/nix/issues.
- Report issues with *Nixpkgs* or *NixOS* (including packages, configuration modules, the Nixpkgs manual[438], and the NixOS manual[439]) at https://github.com/NixOS/nixpkgs/issues.

Make sure that there is not already an open issue for your problem. Please follow the issue template and fill in all requested information.

Take particular care to provide a minimal, easy-to-understand example to reproduce the problem you are facing. You should also show what you have found in attempts to solve the problem yourself. This makes it much more likely for the issue to be resolved eventually, and is important for multiple reasons:

- A reproducible sample is concise and unambiguous.

  This helps with triaging issues, understanding the problem, finding the root cause, and developing a solution. Your preliminary research further helps maintainers with analysis.

- It allows anyone to determine if the issue is still relevant.

  Issues can remain unaddressed for a long time. Deciding what to do with them, even after months or years have passed, requires checking if the underlying problem persists or was resolved. This has to be easy to do: then anyone can help out with triaging, and notify maintainers to close or re-prioritise issues.

- The sample can be used for a regression test when solving the problem.

Tip

Ideally you would also propose or sketch a solution. The perfect issue is, in fact, a pull request that solves the problem directly and ensures with tests that it cannot occur again.

Important

Please open issues to request new features (such as packages, modules, commands, …) only if your are willing and able to implement them yourself. Then the issue can be used to gauge user interest, to determine if the feature fits into the project, and to discuss implementation strategies.

6.1.3 Contribute to Nix

Nix is the cornerstone of the ecosystem, and is mostly written in C++.

If you want to help with development, check the contributing guide in the Nix repository on GitHub[440].

[436] https://nixos.org/community
[437] https://nix.dev/manual/nix/stable
[438] https://nixos.org/manual/nixpkgs/stable
[439] https://nixos.org/manual/nixos/stable
[440] https://github.com/NixOS/nix/blob/master/CONTRIBUTING.md

6.1.4 Contribute to Nixpkgs

Tip

For a verbal introduction, watch the NixCon 2024 talk Becoming a Nixpkgs Contributor[441].

Nixpkgs is a large software project with various areas of development. You can find inspiration for things to improve in the Nixpkgs issue tracker[442].

If you want to help, start with the contributing guide in the Nixpkgs repository on GitHub[443] to get an overview of the code and the contribution process. There are also programming-language-specific instructions[444] for adding packages.

6.1.5 Contribute to NixOS

NixOS is a collectively developed Linux distribution that can be configured conveniently in a highly flexible way through declarative programming interfaces. The code for modules and default configurations is in the `nixos` directory of the `nixpkgs` GitHub repository[445].

See the NixOS manual's development section[446] to get started with making improvements. Contributor documentation specific to NixOS is still lacking, but most conventions for *Nixpkgs contributions* (page 167) apply. Help with improving that is greatly appreciated.

Check issues labeled `good-first-bug`[447] if you're a new contributor. If you know your way around, working on popular issues[448] will be highly appreciated by other NixOS users.

6.2 How to get help

If you prepared a pull request and need help moving forward, check *How to get help* (page 167) for

6.3 How to get help

If you need assistance with one of your contributions, there are a few places you can go for help.

6.3.1 How to find maintainers

For better efficiency and higher chance of success, you should try contacting individuals or groups with more specific knowledge first:

- If your contribution is for a package in Nixpkgs, look for its maintainers in the `maintainers`[449] attribute.
- Check if any teams are responsible for the relevant subsystem:
  - On the NixOS website[450].
  - In the list of Nixpkgs maintainer teams[451].

[441] https://www.youtube.com/watch?v=eijTOBBbCv4
[442] https://github.com/NixOS/nixpkgs/issues?q=is%3Aopen+is%3Aissue+-label%3A%226.topic%3A+nixos%22+-label%3A%226.topic%3A+module+system%22+-label%3A%226.+topic%3A+nixos-container%22+sort%3Areactions-%2B1-desc
[443] https://github.com/NixOS/nixpkgs/blob/master/CONTRIBUTING.md
[444] https://nixos.org/manual/nixpkgs/unstable/#chap-language-support
[445] https://github.com/NixOS/nixpkgs/tree/master/nixos
[446] https://nixos.org/manual/nixos/stable/index.html#ch-development
[447] https://github.com/NixOS/nixpkgs/issues?q=is%3Aopen+label%3A%223.skill%3A+good-first-bug%22+label%3A%226.topic%3A+nixos%22
[448] https://github.com/NixOS/nixpkgs/issues?q=is%3Aopen+is%3Aissue+label%3A%226.topic%3A+nixos%22+sort%3Areactions-%2B1-desc
[449] https://nixos.org/manual/nixpkgs/stable/#var-meta-maintainers
[450] https://nixos.org/community/#governance-teams
[451] https://github.com/NixOS/nixpkgs/blob/master/maintainers/team-list.nix

  - In the `CODEOWNERS` files for Nixpkgs[452] or Nix[453].

- Check the output of `git blame`[454] or `git log`[455] for the files you need help with. Take note of the email addresses of people who committed relevant code.

6.3.2 Which communication channels to use

Once you've found the people you're looking for, you can contact them on one of the community communication platforms[456]:

- GitHub[457]

  All the source code is maintained on GitHub. This is the right place to discuss implementation details.

  In issue comments or pull request descriptions, mention the GitHub username[458] found in the `maintainers-list.nix` file[459].

- Discourse[460]

  Discourse is used for announcements, coordination, and open-ended questions.

  Try the GitHub username found in the `maintainers-list.nix` file[461] to mention or directly contact a specific user. Note that some people use a different username on Discourse.

- Matrix[462]

  Matrix is used for short-lived, timely exchanges, and direct messages.

  To contact a maintainer, use their Matrix handle found in the `maintainers-list.nix` file[463]. If no Matrix handle is present for a specific maintainer, try searching for their GitHub username, as most people tend to use the same one across channels.

  Maintainer teams sometimes have their own public Matrix room.

- Email

  Use email addresses found with `git log`.

- Meetings and events

  Check the official NixOS Calendar[464] and the Discourse community calendar[465] for real-time or in-person events. Some community teams hold regular meetings and publish their meeting notes.

6.3.3 Other venues

If you haven't found any specific users or groups that could help you with your contribution, you can resort to asking the community at large, using one of the following official communication channels:

- A room related to your question in the NixOS Matrix space[466].
- The *Help* category[467] on Discourse.

[452] https://github.com/NixOS/nixpkgs/blob/master/ci/OWNERS
[453] https://github.com/NixOS/nix/blob/master/.github/CODEOWNERS
[454] https://git-scm.com/docs/git-blame
[455] https://www.git-scm.com/docs/git-log
[456] https://nixos.org/community
[457] https://github.com/nixos
[458] https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax#mentioning-people-and-teams
[459] https://github.com/NixOS/nixpkgs/blob/master/maintainers/maintainer-list.nix
[460] https://discourse.nixos.org
[461] https://github.com/NixOS/nixpkgs/blob/master/maintainers/maintainer-list.nix
[462] https://matrix.to/#/#community:nixos.org
[463] https://github.com/NixOS/nixpkgs/blob/master/maintainers/maintainer-list.nix
[464] https://calendar.google.com/calendar/u/0/embed?src=b9o52fobqjak8oq8lfkhg3t0qg@group.calendar.google.com
[465] https://discourse.nixos.org/t/community-calendar/18589
[466] https://matrix.to/#/#community:nixos.org
[467] https://discourse.nixos.org/c/learn/9

- The general `#nix`[468] room on Matrix.

6.4 Contributing documentation

Thank you for your interest to help improve documentation in the Nix ecosystem! This project would not be possible without your support.

6.4.1 Getting started

Check the overview of *documentation resources* (page 170). Documentation contributions should follow the *style guide* (page 173).

Get in touch with the Nix documentation team[469] if you need more guidance.

Important

If you cannot contribute time, consider donating to the NixOS Foundation's documentation project on Open Collective[470] to fund ongoing maintenance and development of reference documentation and learning materials.

6.4.2 Feedback

Feedback is also a valuable contribution. Please share your thoughts in the Documentation category on Discourse[471].

Nix beginners

Try to use official documentation as your primary resource, however incomplete it may appear.

Please open issues and report all problems or questions that arise. Also state your learning goals and the paths you have taken so far.

Sharing your first-hand experience will help guide our efforts and solve recurrent problems with documentation for you and everyone else.

Nix educators

You will probably have observed where learners get stuck most often, and which typical needs and questions they have. You may have your own written notes for classes, trainings, or presentations.

Please share your experience to help us improve upstream documentation and beginner materials, so you can focus on providing the best value to your students.

Domain experts using Nix

If you are proficient in applying Nix to a domain-specific problem, and want to share your expertise on best practices, please check the existing content.

- Does existing material on your subject meet your standards?
- How could we improve it?
- Is there a popular application of Nix' capabilities not yet covered?
- We would be glad to incorporate your insights.

[468] https://matrix.to/#/#nix:nixos.org
[469] https://nixos.org/community/teams/documentation
[470] https://opencollective.com/nixos/projects/nix-documentation
[471] https://discourse.nixos.org/c/dev/documentation/25

6.4.3 Licensing and attribution

When opening pull requests with your own contributions, you agree to licensing your work under CC-BY-SA 4.0[472].

When adding material made by others, make sure it has a license that permits this. In that case, unambiguously state source, authors, and license in the newly added content. Ideally, notify the authors *before* using their work.

Add the original author as co-author[473] to the first commit of your pull request, which should contain the original document verbatim, so we can track authorship and changes through version history.

Using free licenses other than CC-BY-SA 4.0 is possible for individual documents. By contributing changes to those documents you agree to license your work accordingly.

Note

If you have written a tutorial or guide related to Nix, please consider licensing it under CC-BY-SA 4.0! This will allow us to feature your work as official documentation if it complements or improves upon existing materials.

Documentation resources

This is an overview of documentation resources for Nix, Nixpkgs, and NixOS, with suggestions how you can help to improve them.

Reference manuals

The reference manuals document interfaces and behavior, show examples, and define component-specific terms.

- *Nix reference manual* (page 154)
  - source[474]
  - issues[475]
  - pull requests[476]
- Nixpkgs reference manual[477]
  - source[478]
  - issues[479]
  - pull requests[480]
- NixOS reference manual[481]
  - source[482]
  - issues[483]
  - pull requests[484]

[472] https://creativecommons.org/licenses/by-sa/4.0/
[473] https://docs.github.com/en/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/creating-a-commit-with-multiple-authors
[474] https://github.com/NixOS/nix/tree/master/doc/manual
[475] https://github.com/NixOS/nix/issues?q=is%3Aopen+is%3Aissue+label%3Adocumentation
[476] https://github.com/NixOS/nix/pulls?q=is%3Aopen+is%3Apr+label%3Adocumentation
[477] https://nixos.org/manual/nixpkgs
[478] https://github.com/NixOS/nixpkgs/tree/master/doc
[479] https://github.com/NixOS/nixpkgs/issues?q=is%3Aopen+is%3Aissue+label%3A%226.topic%3A+documentation%22+-label%3A%226.topic%3A+nixos%22
[480] https://github.com/NixOS/nixpkgs/pulls?q=is%3Aopen+is%3Apr+label%3A%226.topic%3A+documentation%22+-label%3A%226.topic%3A+nixos%22
[481] https://nixos.org/manual/nixos
[482] https://github.com/NixOS/nixpkgs/tree/master/nixos/doc/manual
[483] https://github.com/NixOS/nixpkgs/issues?q=is%3Aopen+is%3Aissue+label%3A%226.topic%3A+documentation%22+label%3A%226.topic%3A+nixos%22+
[484] https://github.com/NixOS/nixpkgs/pulls?q=is%3Aopen+is%3Apr+label%3A%226.topic%3A+documentation%22+label%3A%226.topic%3A+nixos%22+

The respective manual sections are maintained by developers of the code being documented.

How to help:

- Add links to definitions, commands, options, etc. where only the name is mentioned
- Ensure consistent use of technical terms
- Check that examples are self-contained and follow best practices
- Expand on sections that appear incomplete

NixOS Wiki

NixOS Wiki[485] is a collection of NixOS user guides, configuration examples, and troubleshooting tips. It is meant to be complementary to the NixOS reference manual.

It is collectively edited by the NixOS user community.

How to help:

- Improve discoverability by adding categorisation and links to reference documentation
- Remove redundant or outdated information
- Add guides and sample configurations for your use cases

nix.dev

The purpose of nix.dev[486] (source[487]) is to orient beginners in the Nix ecosystem.

The documentation team maintains nix.dev as editors.

How to help:

- Work on open issues[488]
- Review pull requests[489]
- Add guides or tutorials following the proposed outline[490]. New articles can be based on videos such as:
  - The Nix Hour[491] recordings
  - some of the ~100 NixCon[492] recordings
  - Nix video guides[493] by @jonringer.
  - Summer of Nix 2022 talks[494]

  Since writing a guide or tutorial is a lot of work, please make sure to coordinate with maintainers, for example by commenting on or opening an issue.

Discourse

Nix users exchange information and support each other on these Discourse categories:

- Help[495]

[485] https://wiki.nixos.org/
[486] https://nix.dev
[487] https://github.com/nixos/nix.dev
[488] https://github.com/nixos/nix.dev/issues
[489] https://github.com/nixos/nix.dev/pulls
[490] https://github.com/NixOS/nix.dev/issues/572
[491] https://www.youtube.com/watch?v=wwV1204mCtE&list=PLyzwHTVJlRc8yjlx4VR4LU5A5O44og9in
[492] https://www.youtube.com/c/NixCon
[493] https://www.youtube.com/user/elitespartan117j27
[494] https://www.youtube.com/playlist?list=PLt4-_lkyRrOMWyp5G-m_d1wtTcbBaOxZk
[495] https://discourse.nixos.org/c/learn/9

- Guides[496]
- Links[497]

How to help:

- Ask informed questions, show your work
- Answer other people's questions
- Address recurrent questions by updating or adding a NixOS Wiki article, nix.dev guide or tutorial, or one of the reference manuals.
- Encourage and help people to incorporate their insights into official documentation

Nix Pills

Nix Pills[498] is a series of low-level tutorials on building software packages with Nix, showing in detail how Nixpkgs is made from first principles.

The Nix Pills are not actively maintained.

Documentation framework

We aim to build our documentation according to the Diátaxis framework for technical documentation[499], which divides documentation into four categories:

- *Tutorials* (page 172)
- *Guides* (page 173)
- *Reference* (page 172)
- *Concept* (page 173)

We've found that contributors struggle to understand the differences between these categories, and while we *strongly* recommend reading up on the Diataxis framework, we can summarize them as follows:

Reference

Reference material should

- Focus on "what's there", simply listing which functions, classes, etc. exist
- Use terse language, with the text and layout optimized for scanning and random access
- Show relevant and complete usage examples
- Link to related items for better discoverability

Tutorials

Tutorials walk the user through a particular activity to teach them about common tools and patterns in the ecosystem. While the activity itself is important, the goal is also to connect the dots between other things the reader has learned.

The structure of tutorials should minimise the cognitive load on learners, and actively avoid choices and opportunities for user errors.

[496] https://discourse.nixos.org/c/howto/15
[497] https://discourse.nixos.org/c/links/12
[498] https://nixos.org/guides/nix-pills/
[499] https://diataxis.fr

Guides

Guides are a list of steps showing how to achieve a specific goal or solve a specific problem. The goal is to help the reader reach a specific end, not understand the underlying theory or broader context.

A guide assumes that the reader already has the background to understand the topic at hand and therefore doesn't need to explain the introduction of each new concept.

Concepts

Concepts describe the internals of a piece of code or how to think about a particular idea or entity in the ecosystem. A concept can also describe the historical context behind why something works the way that it does today.

If you find yourself wanting to write about the nitty gritty details of how something works, you most likely want to write an explanation.

Guides vs. Tutorials

We find that contributors primarily struggle with the difference between a Guide and a Tutorial.

Here are several explanations to help you understand the difference.

- A guide is used in a "working" context where the reader just wants a sequence of instructions to achieve an outcome.
  - In this context the reader may already know or may not care how or why these instructions work, they just want to know what to do in order to achieve the desired result.
- A tutorial is used in a "learning" context where the reader is following a sequence of instructions to gain practice performing a certain task.
  - Some small bits of motivation or explanation are helpful in this context to help a reader connect the dots with other things they may have already learned, but the focus is on the activity, not on *how* or *why*.

A helpful analogy is landing an airplane in two different contexts.

Let's say the pilot is unconscious and you now have to land the plane to avoid a crash landing. In this context you just want to know how not to die. You don't care about how or why, you just want to be on the ground in one piece. This is the context for a guide.

A pilot training in a flight simulator wants to practice landing the plane. The pilot-in-training needs practice knowing when to deploy the landing gear, when to adjust flaps, etc. Actually landing the plane during the flight simulation is less important than learning the individual skills that make up a successful landing. This is the context for a tutorial.

Finally, one last way of thinking about the difference between How-to Guide and Tutorial is like this:

- Guide: "step 1: do this, step 2: do that, etc"
- Tutorial: "take my hand as I show you how to do this"

Style guide

This document outlines the guidelines we use when writing documentation.

Writing style

Aim for clarity and brevity

> I would have written a shorter letter, but I did not have the time.
>
> — Blaise Pascal[500]

[500] https://en.m.wikiquote.org/w/index.php?title=Blaise_Pascal&oldid=2978584#Quotes

Readers' time and attention is limited. Take the time to be extraordinarily respectful with their cognitive resources.

The same holds for communication directed to contributors and maintainers: This is a public project, and many people will read what you write. Use this leverage with care.

- Follow the evidence-based plain language guidelines[501].
  - Don't use jargon. Readers may not be familiar with particular technical terms.
  - Don't use long, complicated words if there are shorter, simpler words that convey the same meaning.
- Use the imperative voice when giving instructions. For example, write:

  > Add the `python310` package to `buildInputs`.

  Don't use a conversational tone, as it distracts from the contents. For example, don't write:

  > Going forward, let's now add the `python310` package to `buildInputs` as we have seen in the previous tutorial.

Use inclusive language

Adapted from Contributor Covenant[502] and The Carpentries Code of Conduct[503]:

- Use welcoming and inclusive language
- Show empathy and respect towards other people
- Be respectful of different viewpoints and experiences
- Give and gracefully accept constructive criticism
- Focus on what is best for the community

Avoid idioms as they can be hard to understand for non-native English speakers.

Don't try to be funny. Humor is highly culturally sensitive. At best, jokes may obfuscate the relevant instructions. At worst, jokes may offend readers and invalidate our effort to help them learn.

Don't use references to popular culture. What you may consider well-known may be entirely obscure and distracting to people from different backgrounds.

Voice

Describe the subject factually and use the imperative voice in direct instructions.

Do not assume a personal relationship with readers, prefer clarity and brevity to emotional appeal.

Use "you" to refer to the reader and only use "we" to refer to the authors. Both should be rarely needed.

For example:

> You will have to deploy secrets to the remote machine. We chose to show the explicit, manual process using `scp` here, but there are various tools to automate that.

Be correct, cite sources

The only thing worse than no documentation is *incorrect documentation*. One way to ensure correctness is by citing your sources. If you make a claim about how something works (e.g. that a command line argument exists), link to official documentation for that subject. We would like to maintain a network of documentation, so linking to other documentation helps to reinforce the documentation ecosystem.

It is explicitly encouraged to update or restructure the manuals where appropriate, to improve the overall experience.

[501] https://www.plainlanguage.gov/guidelines/

[502] https://github.com/EthicalSource/contributor_covenant/blob/cd7fcf684249786b7f7d47ba49c23a6bcb3233eb/content/version/2/1/code_of_conduct.md

[503] https://github.com/carpentries/docs.carpentries.org/blob/fb188fa8d7f57ad85eb525091e335ed0d8fea16d/source/policies/coc/index.md#L13-L19

Markup and source

Code samples

Always motivate code before showing it, describing in words what it is for or what it will do.

Counter-example

````
Run this command:

```bash
:(){ :|:& };:
```
````

Non-trivial examples may need additional explanation, especially if they use concepts from outside the given context. Use a collapsed content box for explanation that would distract from the reading flow.

Example

````
Set off a [fork bomb](https://en.wikipedia.org/wiki/Fork_bomb):

```bash
:(){ :|:& };:
```

:::{dropdown} Detailed explanation
This Bash command defines and executes a function `:` that recursively spawns␣
↪copies of itself, quickly consuming system resources
:::
````

Always explain code in the text itself. Use comments in code samples very sparingly, for instance to highlight a particular aspect.

Readers tend to glance over large amounts of code when scanning for information, even if most of it is comments. Especially beginners will likely find reading more complex-looking code strenuous and may therefore avoid it altogether.

If a code sample appears to require a lot of inline explanation, consider replacing it with a simpler one. If that's not possible, break the example down into multiple parts, explain them separately, and then show the combined result at the end.

Code samples that are *intended* to work should work.

If you are going to present an example that does not work (e.g. you're illustrating a common mistake) explain so beforehand. Many readers will get stuck trying to make example code work without reading ahead to find out that the code isn't intended to work.

Code samples should all include a programming language when applicable for syntax highlighting when rendered e.g.

````
```python
print("Hello, World!")
```
````

Headers

Reserve the largest header (`#`) for the title.

Use Markdown headers `##` through `###` to divide up content in the body of the document. Finer grained headings are not necessarily better.

One line per sentence

Write one sentence per line.

This makes long sentences immediately visible. It also makes it easier to review changes and provide direct suggestions, since the GitHub review interface is line-oriented.

Links

Use reference links[504] – sparingly – to ease source readability. Put definitions close to their first use.

Example

```
We follow the [Diátaxis](https://diataxis.fr/) approach to structure␣
↪documentation.
This framework distinguishes between [tutorials], [guides], [reference], and␣
↪[explanation].

[tutorials]: https://diataxis.fr/tutorials/
[guides]: https://diataxis.fr/how-to-guides/
[reference]: https://diataxis.fr/reference/
[explanation]: https://diataxis.fr/explanation/
```

Unless explicitly required to point to the latest version of an external resource, all references should be permanent links[505].

Many web services offer permalinks, such as:

- GitHub URLs to specific commits[506]
- Wikipedia URLs to specific page versions[507]
- Internet Archive "Save Page Now" for persisting web pages[508]

How to write a tutorial

This is a guide to writing tutorials about Nix.

By tutorials we mean *lessons* as described in the Diátaxis framework for technical documentation[509], and recommend becoming familiar with Diátaxis before proceeding. Especially note the difference between tutorials and guides[510].

Target audience

The main target audience of Nix tutorials are software developers with at least basic experience on the Linux command line.

[504] https://github.github.com/gfm/#reference-link
[505] https://en.wikipedia.org/wiki/Permalink
[506] https://docs.github.com/en/repositories/working-with-files/using-files/getting-permanent-links-to-files
[507] https://en.wikipedia.org/wiki/Wikipedia:Linking_to_Wikipedia#Permanent_links_to_old_versions_of_pages
[508] https://web.archive.org/save
[509] https://diataxis.fr/
[510] https://diataxis.fr/tutorials-how-to/

Experts answering questions immediately, personalised instructions and training, and other forms of apprenticeship are known to be the most effective support for learning Nix. **These tutorials are targeted at those who don't have access to any of that**, and should therefore be written to be suitable for self-directed learning. This is achieved by following the structure outlined here, which is primarily characterised by aiming to avoid and close all information gaps for the learner.

As a byproduct, a well-written tutorial will be useful as lecture notes for use in interactive training sessions. Therefore, the secondary target audience are instructors teaching Nix.

Process

Writing a high-quality tutorial takes some time – for you and others. Most of that time is typically spent collaboratively:

- Figuring out the right approach
- Ensuring that the instructions are neither too sparse nor too dense for learners
- Finding ever-more succinct and clear ways to convey ideas

Follow these steps to avoid re-doing work

Pick a topic

There is a tracking issue[511] for tutorials that the documentation team has decided should exist as part of the tutorial series. Pick an issue that covers a topic that you're either knowledgeable about or have a particular interest in.

Check referenced issues and pull request to make sure you won't duplicate work that someone else has already started!

There are more tutorial requests[512] than captured in the outline. Open a new issue[513] if what you wanted to work on isn't tracked anywhere. This is an opportunity for you to clarify your goals, and an opportunity for everyone else to find out that there's interest in that subject.

Submit an pull request with an outline

Submit a pull request with an outline of the tutorial following *our recommended structure* (page 177). The outline should contain bullet points on each section's content. Reference the tracking issue from the pull request description to announce that you're working on a tutorial.

A review will ensure you're going in the right direction in terms of learning objectives and technical implementation.

Expand on the outline

Elaborate the contents of the tutorial following your outline and the *Style guide* (page 173).

A review will ensure that you get all the required information in the right order without overwhelming learners.

Follow up on review comments

Revise your tutorial based on the detailed feedback. We recommend testing your tutorial with friends or coworkers. This will both help with revealing implicit prerequisites as well as provide a realistic estimate of the reading time.

In a final review will check that everything is technically correct.

Structure

Each tutorial should answer the following questions.

In addition, we strongly recommend the book How Learning Works (summary)[514] as a guide for designing learning materials.

[511] https://github.com/NixOS/nix.dev/issues/572
[512] https://github.com/NixOS/nix.dev/issues?q=is%3Aopen+is%3Aissue+label%3Atutorial
[513] https://github.com/NixOS/nix.dev/issues/new?&template=tutorial.md
[514] https://www.lesswrong.com/posts/mAdMkFqWzbJRB544m/book-review-how-learning-works

What will you learn?

Describe the problem statement and learning goals.

The learning goal of a tutorial is always acquiring a skill, which is distinguished by being applicable to a set of situations with recurrent patterns.

What do you need?

State the prerequisite knowledge and skills. The tutorial should always be written such that the stated prerequisites are sufficient to achieve learning goals.

Examples:

- links to previous chapters
- domain-specific skills or knowledge

How long does it take?

Estimate the reading time. This is important for learners to make sure they have the capacity to achieve the planned tasks and thus avoid frustration that may prevent them from continuing on their journey into the Nix ecosystem.

The estimate will depend on the learner's pre-existing knowledge and proficiency. You can note how optional skills or knowledge may influence reading time.

What to do?

Provide steps to achieve the learning goal. These should take the form of direct instructions which repeatably lead to the desired outcome.

It is also worthwhile to add contextual explanations within `:::{dropdown}` blocks. This can help with understanding while keeping distractions minimal.

What did you learn?

Provide exercises or worked examples, and other means of self-assessment.

This is also a good place to offer the readers ways to give feedback or ask the authors questions in order to continue improving the tutorial.

Next steps

Depending on how well a use case is explored, point the reader to

- reference manuals
- guides or other tutorials
- links to known-good external resources, with summaries
- overview of available support tools, and their state of maturity and maintenance
- overview of ideas, and state of community discussion.

We recommend making an explicit separation of practical from theoretical learning resources, as then readers will be able to quickly decide to either get things done or learn more.

External resources should have a summary to set expectations, ideally including reading time. Blog posts should have their original title in the link, and `(<author>, <year>)`: Give authors credit, give readers an idea of how up to date the information is.

CHAPTER

SEVEN

ACKNOWLEDGEMENTS

7.1 Sponsoring

The following people and organisations have contributed to make this effort possible:

- @fricklerhandwerk[515] serves as the team lead since 2023-02, sponsored by Antithesis[516] from 2023-02 to 2024-04
- @zmitchell[517] led the Learning Journey Working Group from 2023-03 to 2023-08, sponsored by flox[518]
- @infinisil[519] worked on the team between 2022-11 and 2024-05, sponsored by Tweag[520]
- @lucperkins[521] served as the team lead from 2022-11 to 2023-01, sponsored by Determinate Systems[522]
- @fricklerhandwerk[523] served as the team lead from 2022-05 to 2022-10, sponsored by Tweag[524]

7.2 History

Many thanks to past contributors, who helped make Nix documentation what it is today:

- @infinisil[525] helped lead the team between 2022-11 and 2024-05. During that time he provided diligent technical reviews of countless contributions, reworked the contribution guides for Nixpkgs[526], and rewrote his module system tutorial[527] for publication.
- @olafklingt[528] volunteered on the team from 2022-10 to 2024-05, and was a formal member between 2022-10 and 2024-05. He added an introduction to NixOS virtual machines[529] and greatly simplified the tutorial on NixOS VM tests[530], and kept them up to date. Both articles enjoy great popularity and are central elements of our tutorial series.
- @brianmcgee[531] was part of the team from 2023-03 to 2023-10 and contributed to the Learning Journey Working Group effort.

[515] https://github.com/fricklerhandwerk
[516] https://antithesis.com
[517] https://github.com/zmitchell
[518] https://floxdev.com
[519] https://github.com/infinisil
[520] https://tweag.io
[521] https://github.com/lucperkins
[522] https://determinate.systems
[523] https://github.com/fricklerhandwerk
[524] https://tweag.io
[525] https://github.com/infinisil
[526] https://github.com/NixOS/nixpkgs/blob/master/CONTRIBUTING.md
[527] https://nix.dev/tutorials/module-system/deep-dive
[528] https://github.com/olafklingt
[529] https://nix.dev/tutorials/nixos/nixos-configuration-on-vm
[530] https://nix.dev/tutorials/nixos/integration-testing-using-virtual-machines
[531] https://github.com/brianmcgee

- @zmitchell[532] led the Learning Journey Working Group[533] from 2023-03 to 2023-08 that added a number of tutorials. He published regular updates on developments in documentation[534] in that period.
- @Mic92[535] was a founding member and part of the team from 2022-05 to 2023-01. Jörg had written a great deal of documentation on the NixOS Wiki, and shared his experience to set a direction for the documentation team.
- @domenkozar[536] was a founding member and part of the team from 2022-05 to 2023-01. Domen originally started nix.dev, wrote many basic tutorials, and funded editorial work through Cachix[537]. He helped bootstrap the documentation team, handed out permissions, and advised us on many aspects. Domen donated nix.dev to the NixOS Foundation 2023-07.

[532] https://github.com/zmitchell
[533] https://discourse.nixos.org/search?q=learning%20journey%20working%20group%20-%20meeting%20notes%20in%3Atitle%20order%3Alatest_topic
[534] https://discourse.nixos.org/search?q=This%20Month%20in%20Nix%20Docs%20in%3Atitle%20before%3A2023-10-30%20order%3Alatest_topic
[535] https://github.com/Mic92
[536] https://github.com/domenkozar
[537] https://www.cachix.org/

INDEX

N