Learning NixOS

A non-linear, wiki-style guide on learning to work with Nix and NixOS. Work in progress, and constantly improving.

Concepts

This chapter introduces you to a number of different core concepts in Nix and NixOS, as well as various of the important tools in the ecosystem.

Concepts

What is Nix?

Nix is a next-generation package and system manager.

Many other package managers suffer from dependency conflict issues, and many systems built on them 'decay' over time, becoming messier, slower, and more prone to crashes over time. Nix does not suffer from these issues, because of a few unique properties:

- **Deterministic:** Building the same thing in Nix with the same configuration always results in the same output. No matter where you build it, what else is installed, or how you've configured your system. If it works in one place, it works everywhere; if it *breaks* in one place, it breaks everywhere in exactly the same way. No more "works on my system".
- **Isolated:** Every package has its own fully-declared set of dependencies, that doesn't conflict with any other package on the system. Dependencies are deduplicated where possible, but it's completely possible to have an unlimited amount of different versions of the same dependency, used by different applications - *without dependency conflicts*.
- **Customizable:** All packages can be freely modified, right from within your system configuration, without needing complex custom repository infrastructure or tooling. That includes making patches to dependencies, even for a single application, without affecting anything else on the system! Packaging your own applications is easier, too.
- **End-to-end:** 'Packages' aren't just software. Nix can build your system configuration, too, with all of the same guarantees, and it can seamlessly extend to container or even multi-system management - whether it's synchronizing the software on your desktop and your laptop, or managing a large fleet of production servers.
- **Centralized configuration:** All your configuration options - for your OS, for your services, and so on - are contained within the `configuration.nix` file (or, optionally, any files you import from it). This means you can use a single syntax for configuring everything, and Nix will take care of converting it to the right configuration format for the software you are using.
- **No runtime overhead:** Best of all, Nix can do all of the above *without needing VMs or containers*. That means that there's no runtime overhead, and it works absolutely fine in graphical environments like on a desktop or laptop, too! You *can* use VMs or containers, of course, where you need them - they're just a part of your system configuration.

These properties give you a lot of nice features:

- **Reliability:** A Nix-based system is *much* more reliable than what you could get from another Linux distribution (that isn't based on the Nix model, anyway). Mystery failures are eliminated, problems are easier to diagnose, and they can be reliably reproduced by people helping you out, even if they are running a totally different system configuration.
- **Rollbacks:** Change your kernel settings? Install some experimental drivers or software? No problem. If something breaks, you can just roll back to a previous version of your

system - even if your system doesn't boot anymore.

- **Ease of installation:** No need to resolve dependency conflicts, or mess around with `virtualenv`, or pick between two pieces of software that require different conflicting dependencies. Software *just works*. Even if it's ancient software that requires ancient dependency versions to run that are supported by nothing else, you can make it work without breaking any other software.
- **Easy synchronization:** Synchronizing software or system configurations between computers is as simple as copying over your `configuration.nix`, either manually or through something like Git or even Dropbox. Since the entire system is built from that file, you can apply it to any amount of computers without issues, even if they were running a totally different configuration before.
- **No 'cruft' or 'decay':** Because your system state is derived entirely from the configuration you give it, and not from piecemeal "change this, change that" instructions, your system doesn't accumulate 'cruft' over time like you may be used to from other distributions. A 5-year-old NixOS installation will run just as smoothly and reliably as a 2-day-old one, because effectively every configuration change is a (fast) reinstallation. The only exception is stateful data generated by applications, and this can be eliminated too if you really want.
- **Automated package testing:** Because all builds are isolated and deterministic, it's possible to do fully automated package tests, to verify that a package really works as advertised. This is done for (a subset of) the official package set, too.

The properties above are not *entirely* without tradeoffs - make sure to read the section below about the tradeoffs in the FAQ before diving into Nix and NixOS.

What is Nix (the language)?

Confusingly, the name "Nix" is not just used for the package manager, but also for the *language* that you use to write packages or system configurations. Sometimes, people call it 'nixlang' to differentiate it from the package manager, and we'll do the same in this documentation.

nixlang is a little different from what you might be used to. It's a bit like a declarative language such as JSON, but also a bit like a 'real' programming language, with support for functions and variables (sort of). An excellent step-by-step introduction to the language can be found here - it's a fairly simple language, but because it has some unusual characteristics, you should definitely give that a read.

This language is used throughout Nix, and in all of the tooling surrounding it. It's the language you use for writing package definitions, modifying your system configuration, managing multiple servers, and even for writing package tests. Because it allows creating functions and other abstractions, it can support configuration at any scale, without becoming complex to use for the simple cases.

If you're curious why Nix has its own custom language, and why it doesn't just use something that already exists, have a look in the FAQ.

What is nixpkgs?

You'll often run across the name 'nixpkgs' in this documentation. Nix itself is really just the package manager and build tool - it doesn't come with any software packages, and expects the user to point it at some sort of 'package set'.

That's where nixpkgs comes in - it's the officially maintained package set for Nix, and it's what almost every Nix user uses. It contains a wide selection of software - comparable to what you might find in most Linux distributions, and often even exceeding them - as well as all the bits and pieces for NixOS.

You're not limited to using nixpkgs, of course. It's just selected as a default when you install Nix, and you're free to add other package sets, or write an 'overlay' that extends nixpkgs with additional packages. For example, Mozilla maintains a nixpkgs overlay for their Rust and Firefox projects; and many users maintain Flakes, which can provide their own package sets.

If you're in a more experimental mood, you could even totally remove nixpkgs, and write your own package set from scratch. This is something that most users won't want (or need) to do, though.

Concepts

What is NixOS?

While Nix can run as a stand-alone package manager on any Linux system, and even on macOS, there's only so much that it can do without control over the rest of the system. NixOS is a Linux distribution that takes the concept of Nix a step further, by making it possible to use Nix for managing *your entire system* - from software, to services, to kernel settings, to container management, all using the same language.

This wiki is for learning both Nix *and* NixOS - NixOS-specific sections will be marked as such.

Concepts

What is NixOps?

Nix (and NixOS) themselves only manage a single machine. If you want to manage *multiple* machines, especially if they are many servers, you can use a tool like NixOps - it's an 'orchestration tool' like Ansible, Chef, or Puppet, but with the guarantees of Nix. Like all of the other tools, you use nix-lang for specifying your systems.

If you're curious about what NixOS with NixOps does better than other orchestration tools, give this excellent article a read.

> NixOps is only one of many deployment tools for Nix and NixOS, and it is mentioned here because it was the first one. If you only need to manage one or a few servers, there are many other options that are often simpler, such as morph. These will likely get their own wiki articles at a later time.

Concepts

What is Hydra?

Hydra is, more or less, a build server. Unsurprisingly, it uses Nix and nixlang for specifying what to build. It's used to build the binary packages for nixpkgs, for example, as well as for running automated tests to ensure that packages actually work. If you're just using Nix or NixOS as an end user, you probably don't need to care about this.

Because Hydra supports deployment operations after a successful build-and-testing cycle, you could also technically consider it a Continuous Deployment system.

What is a derivation?

You can think of a derivation as a set of build instructions, somewhat similar to how IKEA furniture comes with an assembly manual. The furniture (or package, or configuration file, or...) still needs to be built, but the build instructions (the derivation) have the information on how to do so. Nix takes these instructions, and uses them to create a *build result*.

Derivations are described using the Nix language (nixlang), and they may build *anything* - it doesn't need to be a software package! You might have a derivation for every software package that is being a built, and a derivation for your system configuration, and a derivation for each (automatically generated) configuration file for the software on your system, and so on.

Derivations also keep track track of their dependencies; that is, which *other* derivations are referenced inside of its instructions. Nix needs this information to make sure that everything is built in the correct order, and correctly linked together.

> There is a `derivation` function in the standard library of Nix, but in practice you will probably never use it. Instead, you will most likely be using `mkDerivation`, which is a wrapper function in nixpkgs that automatically handles some things for you. This is explained further in the chapter about nixpkgs.

What is the Nix store?

The Nix store is a folder, located at `/nix/store` by default, that contains every build result from a derivation that Nix has ever generated. These build results stay in the Nix store until they are explicitly garbage-collected. Each entry in the Nix store is prefixed by a hash of the derivation that was used to build - this is how Nix avoids building the same thing more than once, and how it ensures that there is a unique reference to every possible version and variant of a piece of software.

You should not ever need to touch the Nix store manually; it is entirely under the control of Nix. However, Nix does provide several utilities for managing the store, such as `nix-collect-garbage`.

The high-level workflow of Nix

You can use Nix to build many different things for many different purposes. However, the basic workflow is always the same:

1. You specify some kind of Nix expression. This can be a simple expression, a whole system configuration, a Morph deployment... anything that is written in Nixlang.
2. You *evaluate* that expression, and recursively evaluate everything it references, using Nix. Wherever Nix encounters derivations, it will build them and turn them into the store path of their build result. It then returns the result of the 'top-level' expression you asked it to evaluate.
3. You *apply* the result of that expression in some way. For a NixOS configuration that means setting it as the default boot target in the bootloader, for a Morph deployment that means that Morph will send it to the remote server over SSH, for a VM build that means it gives you a link to the generated VM image, and so on.

Importantly, when using NixOS, this same workflow also applies to changing any of your system configuration, and even installing packages! You never "install a package" as a discrete action - rather, you add an item to the list of packages that should exist on your system, and rebuild the configuration using `nixos-rebuild`.

In that process, Nix will notice that a package is referenced that it doesn't have yet, so it builds or downloads it. Then `nixos-rebuild` changes the system environment to the new version of your system, where this package is part of the environment. The end result is that the package is now available for you to use.

This is often one of the things that people have the most trouble with, when learning NixOS - unlearning the idea of "installing packages" as a command that you run, and instead thinking of your system configuration as having 'versions' that do or do not have certain packages available. It's a little bit like a version control such as Git.

Isn't that a really limiting workflow?

Yes and no! It's true that this workflow makes some things a bit harder, and that it can take some time to get used to. However, NixOS can do everything that more traditional Linux distributions can do in terms of configuration, and - thanks to this workflow - can even do some things that they *can't* do, like going back to past versions of your system, or having virtual environments on the same system that look completely different.

Because every version of the system is itself a build result from a derivation, it can be referenced and managed in the same way that a piece of software might be. Opening a shell that points at a particular environment, generating a virtual machine image out of an environment, it's all possible.

Frequently Asked Questions

General

Are there any downsides?

Yes. Here are some of the most common issues that people run in today:

- **Poor user experience:** While the *concepts* behind Nix are great and could make system management a lot easier, the current generation of Nix tools can still be rather awkward to use. A lot of user experience kinks haven't been quite worked out yet, although this is a problem that's being worked on, and solvable in the long term.
- **Poor documentation:** The documentation for Nix and associated tools isn't great yet. This wiki is a (hopefully successful!) attempt at improving that, but you'll most likely still run into quite a few 'documentation deserts' while using Nix, especially when just getting started. Please do contact me about documentation you're missing - it'll help me improve the wiki!
- **Incompatibility with other Linux systems:** This is a less solvable problem, mostly specific to NixOS. To be able to provide the guarantees that it does, Nix makes some unusual decisions in how it structures your system. This means that, for example, `/usr` doesn't exist - and neither do many of the other 'well-known locations' that tools and users alike expect.
  This isn't a problem for packaged software, but it means that random 'static Linux binaries' downloaded from the internet won't work on NixOS out of the box. This includes AppImages, and many proprietary games. Some workarounds exist (in the form of `steam-run` and `appimage-run`), but it's still not the click-and-run experience that you get on other distributions.
  Similarly, configuration files aren't in a big bucket that you can edit at will - it's not possible to edit configuration files from outside of your Nix configuration, and various other such limitations exist. This isn't *necessarily* a problem (since you can still modify configuration files through your Nix configuration, for example), but it means you will have to adjust your habits and essentially re-learn a lot of Linux things.
  While work is ongoing on reducing the impact of these differences to end users, NixOS will never work *exactly* like the distributions you're used to - the model that those distributions use is fundamentally incompatible with the guarantees that Nix provides.

Should I use Nix?

That depends. Given how different Nix is from 'traditional' package management systems and Linux distributions, it will take quite some time to learn how all of it works.

If you decide to use Nix on another distribution, then you can do this while you continue using your other package manager, so you won't get 'blocked' on not understanding something about Nix yet. However, the benefits you get from Nix will also be limited to *just* package management.

If you decide to use NixOS, then you can use the full range of system management niceties available - but there won't be an 'escape hatch' when you're not sure how to do something, so this is a somewhat steeper learning curve. You should expect to be figuring things out for at least a week, when taking this route.

It really all comes down to this: you'll have to spend effort upfront to learn about how Nix works, and that can be quite a bit of work. But in the long term you'll get a much more reliable system out of it, and it'll make your life easier. Whether you're willing to make that tradeoff, is up to you.

The Nix language (nixlang)

Why a custom language? Why not use something that already exists?

Most languages are one of roughly two types:

- **Declarative:** These are languages like JSON, YAML, XML, TOML, and so on. They're typically very simple, only supporting statically specified data or maybe really basic references, without any logic. They're especially suited for serialization, and for cases where you need to extensively inspect the information written in it, like in configuration files.
- **Imperative:** These are what you might think of when you hear 'programming language'; Python, C++, JavaScript, Rust, Lua, and so on. They typically specify a sequence of instructions that modify state and might produce output. They're especially suited towards implementing arbitrary logic, like in software, but it's hard to 'inspect' the state of the application.

The problem is that Nix needs a little bit of both. It's mostly data-oriented - a system configuration is data, package metadata is data - but it's also about data that's complex enough and sometimes repetitive enough that you want to be able to use logic to *produce* it.

nixlang is precisely that - a simple and mostly-declarative language with enough abstraction that you can programmatically generate data, but not so much that it becomes difficult to inspect.

One particularly unique aspect of it is that it's lazily-evaluated; that is, instead of executing some logic and storing the result in a variable, you store the *logic itself* in a variable, and the first time that variable is accessed, the logic is executed and the result remembered for future access. This is what makes it possible to have tens of thousands of packages in a Nix package set, without needing to execute *all* logic every time you need a single package.

Installing software

This chapter describes the different ways in which you can install software on NixOS - as well as ways to run software *without* permanently installing it.

Installing software globally

Probably the most common case, is wanting to install software system-wide. This is not *technically* system-wide in NixOS, due to its internal isolation properties, but the difference doesn't really matter for your day-to-day use.

Installing software globally is done by adding it to your `environment.systemPackages`. For example, like so:

```
{
  # ... other configuration goes here ...

  environment.systemPackages = [
    pkgs.htop
    pkgs.iotop
  ];

  # ... other configuration goes here ...
}
```

In this example, two packages are installed: `htop`, and `iotop`. As you can see, they are prefixed with pkgs. - this means that they are *attributes* (ie. properties) of the pkgs *binding* (ie. variable).

The type of `environment.systemPackages` is a *list*, containing *derivations*.

Once you have added the package(s) that you want to install to your `systemPackages`, you need to rebuild the system to make the changes take effect.

Running unpackaged software

This chapter guides you through a number of ways in which you can run software that isn't packaged for NixOS (yet), including proprietary and custom software and games that likely never will be packaged.

Introduction

NixOS is very strict in its approach to system purity; there is essentially no global environment, and this also means that a lot of assumptions about what a 'standard Linux system' looks like, do not hold up when you are using NixOS. This is a common cause of problems, when people try to run software on NixOS that was not specifically packaged for it.

In this chapter, we'll go through a few ways in which you can deal with this situation, depending on what format your software is available in.

Running an AppImage

If you try to run an AppImage with the usual `./application-name` invocation, you'll find that they won't run. This is because the AppImage runtime expects some files to exist in a global location that NixOS doesn't put there.

In practice, this is usually not a problem - in *most* cases, you can use `appimage-run` to run an AppImage on NixOS. `appimage-run` is a tool that creates a virtual environment with all the stuff that an AppImage expects to be there, and then runs it within that environment for you.

To run an AppImage called `application-name`, you should ensure that `appimage-run` is installed into your environment (eg. by adding it to your `systemPackages`), and then run:

```
appimage-run ./application-name
```

> In some cases, this doesn't work either. That usually happens when the AppImage expects some additional library to exist on the system, but the appimage-run environment doesn't have it. On most Linux distributions, this would be fixed by installing the library it wants, but on NixOS that's a little more complicated. This guide doesn't currently cover that case (yet), and for now I would recommend to try another approach in this chapter instead.

Security considerations

Storing secrets and the Nix store

Nix stores every derivation that it builds in the Nix store. However, to make sure that everything in the Nix store is perfectly deterministic and usable by anyone, it needs to set the attributes of all files to a fixed value - this means that every file creation and modification date is set to UNIX timestamp `0`, but it *also* means that every file is made world-readable.

That's a problem when you're handling sensitive data!

For this reason, you should avoid storing any kind of secret or sensitive data in the Nix store, like passwords, credentials, API keys, SSH keys, and so on. Most NixOS modules will provide a mechanism for specifying secrets in some out-of-band way, usually by expecting you to specify the path to a 'key file' - a file somewhere outside of the Nix store, that is not managed by Nix, which contains the secret value(s). However, not all modules have been updated to do this yet, so for now you should always pay attention to where your secrets are being stored.

Why can't Nix just manage the secrets outside of the Nix store?

For the Nix model to work as designed, it needs to have a single universal index of build artifacts (store paths), and that is the Nix store. If secrets were managed outside of the store, Nix would not be able to provide all of its guarantees. That is why this task is usually left to tools that are dedicated to the purpose - for example, Morph can handle this for you for server deployments.

Deploying with morph

Introduction to Morph

Morph is a simple, stateless deployment tool for NixOS. Essentially, it's a way to manage one or more servers remotely from a central NixOS configuration, deploying system updates and configuration changes over SSH.

You write configuration for Morph with the same language, syntax and tools as you would use for a local NixOS configuration; in fact, there is very little difference in format between a Morph and a NixOS configuration. In many cases, it's nothing more than an extra object wrapped around the system configurations.

For example, consider the following (simplified) NixOS system configuration, derived from an example in the Morph repository:

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

  services.nginx.enable = true;

  fileSystems = {
    "/" = {
      label = "nixos";
      fsType = "ext4";
    };
  };
}
```

The equivalent Morph configuration would look like this:

> This example is simplified, and you should not actually try deploy this to a server! It will break your connection, as the example configuration does not include an SSH daemon.

```
{
  network = {
    pkgs = import (builtins.fetchTarball "https://github.com/NixOS/nixpkgs/archive/nixos-
unstable.tar.gz") {};
```

```
    description = "Basic Morph configuration";
  };

  "server01.example.com" = { pkgs }: {
    boot.loader.systemd-boot.enable = true;
    boot.loader.efi.canTouchEfiVariables = true;

    services.nginx.enable = true;

    fileSystems = {
      "/" = {
        label = "nixos";
        fsType = "ext4";
      };
    };
  };
}
```

As you can see, the configuration actually didn't change at all - all we've done is add a wrapper object which specifies the default `nixpkgs` source, and the hostname to deploy the configuration on. This is all we need to have a working Morph deployment!

Aside from pushing updates to your server(s), Morph does a few other essential things:

- **Health checks**, ie. verifying that a deployment succeeded by checking that expected services are available afterwards.
- **Uploading secrets,** such as application keys and credentials that should not be in the Nix store.

However, that's about all that it does. This is usually all you need for simpler deployments, but if you need more complex procedures, then a different deployment tool may be a better choice.

> It is easy to change your deployment tool later on, because Morph syntax and configuration structure is so close to that of NixOS itself! This means it's completely okay to start with Morph, and then move to something else if it turns out not to be sufficient.

Stateless?

Some deployment tools, even those for NixOS, are stateful; they keep a database on your own computer of the current state of each server, then each time they are run, they calculate what changed and push those changes to the server. One of the downsides of this model is that you

must always deploy to a server from the same computer, since the database can only exist in one place at a time. However, this kind of state can be necessary if the tool also needs to manage *resources* such as IaaS services.

When you are just deploying to a standard Linux system like a VPS or bare-metal dedicated server, though, the added complexity and limitations of a stateful deployment tool are not really worth it, and it's often much simpler to use a stateless deployment tool instead. These don't need to keep local state, and always ensure that the remote system is in the exact configuration that your configuration file specifies. Morph is one of those tools.