NixOS in Production

The NixOS handbook for professional use ONLY

Gabriella Gonzalez

NixOS in Production

The NixOS handbook for professional use ONLY

Gabriella Gonzalez

This book is available at http://leanpub.com/nixos-in-production

This version was published on 2024-09-23

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. Introduction

This book is a guide to using NixOS in production, where NixOS is an operating system built on top of the Nix package manager.

This guide assumes that you have some familiarity with NixOS, so if you have never used NixOS before and you're looking for a tutorial or introduction then this book might not be for you. Some chapters may review basic concepts, but in general they will not be written for a beginner audience.

However, this book will appeal to you if you wish to answer the following questions:

- What real-world use cases does NixOS address better than the alternatives?
- What does a mature NixOS enterprise look like?
- How do I smoothly migrate an organization to adopt NixOS?
- What potential pitfalls of NixOS should I be mindful to avoid?
- How can I effectively support and debug NixOS when things go wrong?

I'm writing this book because I cultivated years of professional experience doing all of the above, back when no such resource existed. I learned NixOS the hard way and I'm writing this book so that you don't have to make the same mistakes I did.

Currently, most educational resources for NixOS (including the NixOS manual) are written with desktop users in mind, whereas I view NixOS as far better suited as a production operating system. This book attempts to fill that documentation gap by catering to professional NixOS users instead of hobbyists.

Continue reading on if you want to use NixOS "for real" and build a career around one of the hottest emerging DevOps technologies. This book will improve your NixOS proficiency and outline a path towards using NixOS to improve your organization's operational maturity and reliability.

2. What is NixOS for?

Some NixOS users might try to "convert" others to NixOS using a pitch that goes something like this:

> NixOS is a Linux distribution built on top of the Nix package manager. It uses declarative configuration and allows reliable system upgrades.
>
> Source: Wikipedia - NixOS[1]

This sort of feature-oriented description explains what NixOS *does*, but does not quite explain what NixOS *is for*. What sort of useful things can you do with NixOS? When is NixOS the best solution? What types of projects, teams, or organizations should prefer using NixOS over other the alternatives?

Come to think of it, what *are* the alternatives? Is NixOS supposed to replace Debian? Or Docker? Or Ansible? Or Vagrant? Where does NixOS fit in within the modern software landscape?

In this chapter I'll help you better understand when you should recommend NixOS to others and (just as important!) when you should gently nudge people away from NixOS. Hopefully this chapter will improve your overall understanding of NixOS's "niche".

Desktop vs. Server

The title of this book might have tipped you off that I will endorse NixOS for use as a server operating system rather than a desktop operating system.

I would not confidently recommend NixOS as a desktop operating system because:

- *NixOS expects users to be developers who are more hands-on with their system*

  NixOS does not come preinstalled on most computers and the installation guide assumes quite a bit of technical proficiency. For example, NixOS is typically configured via text files and upgrades are issued from the command line.

- *Nixpkgs doesn't enjoy mainstream support for desktop-oriented applications*

  … especially games and productivity tools. Nixpkgs is a fairly large software distribution, especially compared to other Linux software distributions, but most desktop applications will not support the Nix ecosystem out-of-the-box.

- *The NixOS user experience differs from what most desktop users expect*

  Most desktop users (especially non-technical users) expect to install packages by either downloading the package from the publisher's web page or by visiting an "app store" of some sort. They don't expect to modify a text configuration file in order to install package.

[1] https://en.wikipedia.org/wiki/NixOS

However, the above limitations don't apply when using NixOS as a server operating system:

- *Servers are managed by technical users comfortable with the command-line*

  Server operating systems are often headless[2] machines that only support a command-line interface. In fact, a typical Ops team would likely frown upon managing a server in any other way.

- *Nixpkgs provides amazing support for server-oriented software and services*

  `nginx`, `postgres`, `redis`, ... you name it, Nixpkgs most likely has the service you need and it's a dream to set up.

- *End users can more easily self-serve if they stray from the beaten path*

  Server-oriented software is more likely to be open source than desktop-oriented software and therefore easier to package.

Furthermore, NixOS possesses several unique advantages compared to other server-oriented operating systems:

- *NixOS can be managed entirely declaratively*

  You can manage every single aspect of a NixOS server using a single, uniform, declarative option system. This goes hand-in-hand with GitOps[3] for managing a fleet of machines (which I'll cover in a future chapter).

- *NixOS upgrades are fast, safe and reliable*

  Upgrades are atomic, meaning that you can't leave a system in an unrecoverable state by canceling the upgrade midway (e.g. `Ctrl-C` or loss of power). You can also build the desired system ahead of time if you want to ensure that the upgrade is quick and smooth.

- *NixOS systems are lean, lean, lean*

  If you like Alpine Linux then you'll *love* NixOS. NixOS systems tend to be very light on disk, memory, and CPU resources because you only pay for what you use. You can achieve astonishingly small system footprints whether you run services natively on the host or inside of containers.

- *NixOS systems have better security-related defaults*

  You get several security improvements for free or almost free by virtue of using NixOS. For example, your system's footprint is immutable and internal references to filepaths or executables are almost always fully qualified.

[2] https://en.wikipedia.org/wiki/Headless_computer
[3] https://www.redhat.com/en/topics/devops/what-is-gitops

On-premises vs. Software as a service

"Server operating systems" is still a fairly broad category and we can narrow things down further depending on where we deploy the server:

- *On-premises ("on-prem" for short)*

  "On-premises" software runs within the end user's environment. For example, if the software product is a server then an on-premises deployment runs within the customer's data center, either as a virtual machine or a physical rack server.

- *Software as a service ("SaaS" for short)*

  The opposite of on-premises is "off-premises" (more commonly known as "software as a service"). This means that you centrally host your software, either in your data center or in the cloud, and customers interact with the software via a web interface or API.

NixOS is better suited for SaaS than on-prem deployments, because NixOS fares worse in restricted network environments where network access is limited or unavailable.

You can still deploy NixOS for on-prem deployments and I will cover that in a later chapter, but you will have a much better time using NixOS for SaaS deployments.

Virtualization

You might be interested in how NixOS fares with respect to virtualization or containers, so I'll break things down into these four potential use cases:

- *NixOS without virtualization*

  You can run NixOS on a bare metal machine (e.g. a desktop computer or physical rack server) without any virtual machines or containers. This implies that services run directly on the bare metal machine.

- *NixOS as a host operating system*

  You can also run NixOS on a bare metal machine (i.e the "host") but then on that machine you run containers or virtual machines (i.e. the "guests"). Typically, you do this if you want services to run inside the guest machines.

- *NixOS as a guest operating system*

  Virtual machines or OS containers[4] can run a fully-fledged operating system inside of them, which can itself be a NixOS operating system. I consider this similar in spirit to the "NixOS without virtualization" case above because in both cases the services are managed by NixOS.

[4] https://blog.risingstack.com/operating-system-containers-vs-application-containers/

- *Application containers*

  Containers technically do not need to run an entire operating system and can instead run a single process (e.g. one service). You can do this using Nixpkgs, which provides support for building application containers.

So which use cases are NixOS/Nixpkgs well-suited for? If I had to rank these deployment models then my preference (in descending order) would be:

- *NixOS as a guest operating system*

  Specifically, this means that you would run NixOS as a virtual machine on a cloud provider (e.g. AWS) and all of your services run within that NixOS guest machine with no intervening containers.

  I prefer this because this is the leanest deployment model and the lowest maintenance to administer.

- *NixOS without virtualization*

  This typically entails running NixOS on a physical rack server and you still use NixOS to manage all of your services without containers.

  This can potentially be the most cost-effective deployment model if you're willing to manage your own hardware (including RAID and backup/restore) or you operate your own data center.

- *NixOS as a host operating system - Static containers*

  NixOS also works well when you want to statically specify a set of containers to run. Not only can NixOS run Docker containers or OCI containers, but NixOS also provides special support for "NixOS containers" (which are `systemd-nspawn` containers under the hood) or application containers built by Nixpkgs.

  I rank this lower than "NixOS without virtualization" because NixOS obviates some (but not all) of the reasons for using containers. In other words, once you switch to using NixOS you might find that you can do just fine without containers or at least use them much more sparingly.

- *NixOS as a host operating system - Dynamic containers*

  You can also use NixOS to run containers dynamically, but NixOS is not special in this regard. At best, NixOS might simplify administering a container orchestration service (e.g. `kubernetes`).

- *Application containers sans NixOS*

  This is technically a use case for Nixpkgs and not NixOS, but I mention it for completeness. Application containers built by Nixpkgs work best if you are trying to introduce the Nix ecosystem (but not NixOS) within a legacy environment.

  However, you lose out on the benefits of using NixOS because, well, you're no longer using NixOS.

The killer app for NixOS

Based on the above guidelines, we can outline the ideal use case for NixOS:

- NixOS shines as a server operating system for SaaS deployments
- Services should preferably be statically defined via the NixOS configuration
- NixOS can containerize these services, but it's simpler to skip the containers

If your deployment model matches that outline then NixOS is not only a safe choice, but likely the best choice! You will be in great company if you use NixOS in this way.

You can still use NixOS in other capacities, but the further you depart from the above "killer app" the more you will need to roll up your sleeves.

Profile of a NixOS adopter

NixOS is a DevOps[5] tool, meaning that NixOS blurs the boundary between software development and operations.

The reason why NixOS fits the DevOps space so well is because NixOS unifies all aspects of managing a system through the uniform NixOS options interface. In other words, you can use NixOS options to configure operational details (e.g. RAID, encryption, boot loaders) and also software development details (e.g. dependency versions, patches, and even small amounts of inline code).

This means that a DevOps engineer or DevOps team is best situated to introduce NixOS within an engineering organization.

DevOps is more of a set of cultural practices than a team, but some organizations explicitly create a DevOps team or hire engineers for their DevOps expertise in order to support tools (like NixOS) that enable those cultural practices.

What does NixOS replace?

If NixOS is a server operating system, does that mean that NixOS competes with other server operating systems like Ubuntu Server, Debian or Fedora? Not exactly.

NixOS competes more with the Docker ecosystem, meaning that a lot of the value that NixOS adds overlaps with Docker:

- *NixOS supports declarative system specification*

  ... analogous to `docker compose`.

[5] https://www.atlassian.com/devops

- *NixOS provides better isolation*

  … analogous to containers.

- *NixOS uses the Nix package manager to declaratively assemble software*

  … analogous to `Dockerfiles`.

You *can* use NixOS in conjunction with Docker containers since NixOS supports declaratively launching containers, but you probably want to avoid buying further into the broader Docker ecosystem if you use NixOS. You don't want to be in a situation where your engineering organization fragments and does everything in two different ways: the NixOS way and the Docker way.

For those familiar with the Gentoo Linux distribution, **NixOS is like Gentoo, but for Docker**[6]. Similar to Gentoo, NixOS is an operating system that provides unparalleled control over the machine while targeting use cases and workflows similar to the Docker ecosystem.

[6]The reason why is that writing a (meaningful) test for our TODO list example would require executing JavaScript using something like Selenium, which will significantly increase the size of the example integration test. `postgrest`, on the other hand, is easier to test from the command line.

3. The big picture

Before diving in further you might want to get some idea of what a "real" NixOS software enterprise looks like. Specifically:

- What are the guiding principles for a NixOS-centric software architecture?
- How does a NixOS-centric architecture differ from other architectures?
- What would a "NixOS team" need to be prepared to support and maintain?

Here I'll do my best to answer those questions so that you can get a better idea of what you would be signing up for.

The Zen of NixOS

I like to use the term "master cue" to denote an overarching sign that indicates that you're doing things right. This master cue might not tell you *how* to do things right, but it can still provide a high-level indicator of whether you are on the right track.

The master cue for NixOS is very similar to the master cue for the Nix ecosystem, which is this:

> Every common build/test/deploy-related activity should be possible with at most one command using Nix's command line interface.

I say "*at most* one command" because some activities (like continuous deployment) should ideally require no human intervention at all. However, activities that do require human intervention should in principle be compressible into a single Nix command.

I can explain this by providing an example of a development workflow that *disregards* this master cue:

Suppose that you want to test your local project's changes within the context of some larger system at work (i.e. an integration test[1]). Your organization's process for testing your code might hypothetically look like this:

- Create and publish a branch in version control recording your changes
- Manually trigger some workflow to build a software artifact containing your changes
- Update some configuration file to reference the newly-built software artifact
- Run the appropriate integration test

Now what if I told you that the entire integration testing process from start to finish could be:

[1] https://en.wikipedia.org/wiki/Integration_testing

- Run `nix flake check`

In other words:

- *There would be no need to create or publish your branch*

  You could test uncommitted changes straight from your local project checkout.

- *There would be no multi-step publication process*

  All of the intermediate build products and internal references would be handled transparently by the Nix build tool.

- *The test itself would be managed by the Nix build tool*

  In other words, Nix would treat your integration test no differently than any other build product. Tests and their outputs *are* build products.

- *There would be no need to select the appropriate tests to rerun*

  The Nix build tool would automatically infer which tests depended on your project and rerun those. Other test runs and their results would be cached if their dependency tree did not include your changes.

Some of these potential improvements are not specific to the Nix ecosystem. After all, you could attempt to create a script that automates the more painstaking multi-step process. However, you would likely need to reinvent large portions of the Nix ecosystem for this automation to be sufficiently robust and efficient. For example:

- *Do you maintain a file server for storing intermediate build products?*

  You're likely implementing your own version of the Nix store and caching system

- *Do you generate unique labels for build products to isolate parallel workflows?*

  In the best case scenario, you label build products by a hash of their dependencies and you've reinvented the Nix store's hashing scheme. In the worst case scenario you're doing something less accurate (e.g. using timestamps in the labels instead of hashes).

- *Do you have a custom script that updates references to these build products?*

  This would be reinventing Nix's language support for automatically updating dependency references.

- *Do you need to isolate your integration tests or run them in parallel?*

  You would likely reimplement the NixOS test framework.

You can save yourself a lot of headaches by taking time to learn and use the Nix ecosystem as idiomatically as possible instead of learning these lessons the hard way.

GitOps

NixOS exemplifies the Infrastructure as Code (IaC)[2] paradigm, meaning that every aspect of your organization (including hardware/systems/software) is stored in code or configuration files that are the source of truth for how everything is built. In particular, you don't make undocumented changes to your infrastructure that cause it to diverge from what is recorded within those files.

This book will espouse a specific flavor of Infrastructure of Code known as GitOps[3] where:

- *The code and configuration files are (primarily) declarative*

  In other words, they tend to specify the desired state of the system rather than a sequence of steps to get there.

- *These files are stored in version control*

  Proponents of this approach most commonly use `git` as their version control software, which is why it's called "GitOps".

- *Pull requests are the change management system*

  In other words, the pull request review process determines whether you have sufficient privileges, enough vetting, or the correct approvals from relevant maintainers.

DevOps

NixOS also exemplifies the DevOps[4] principle of breaking down boundaries between software developers ("Dev") and operations ("Ops"). Specifically, NixOS goes further in this regard than most other tools by unifying both software configuration and system configuration underneath the NixOS option system. These NixOS options fall into roughly three categories:

- *Systems configuration*

  These are options that are mostly interesting to operations engineers, such as:
  - log rotation policies
  - kernel boot parameters
  - disk encryption settings

- *Hybrid systems/software options*

  These are options that live in the grey area between Dev and Ops, such as:
  - Service restart policies
  - Networking
  - Credentials/secrets management

[2] https://en.wikipedia.org/wiki/Infrastructure_as_code
[3] https://about.gitlab.com/topics/gitops/
[4] https://en.wikipedia.org/wiki/DevOps

- *Software configuration*

  These are options that are mostly interesting to software engineers, such as:

  - Patches
  - Command-line arguments
  - Environment variables

In extreme cases, you can even embed non-Nix code inside of Nix and do "pure software development". In other words, you can author inline code written within another language inside of a NixOS configuration file. I'll include one example of this later on in the "Our first web server" chapter.

Architecture

A NixOS-centric architecture tends to have the following key pieces of infrastructure:

- *Version control*

  If you're going to use GitOps then you had better use `git`! More specifically, you'll likely use a `git` hosting provider like GitHub[5] or GitLab[6] which supports pull requests and continuous integration.

  Most companies these days use version control, so this is not a surprising requirement.

- *Product servers*

  These are the NixOS servers that actually host your product-related services.

- *A central build server (the "hub")*

  This server initiates builds for continuous integration, which are delegated to builders.

- *Builders for each platform*

  These builders perform the actual Nix builds. However, remember that integration tests will be Nix builds, too, so these builders also run integration tests.

  These builders will come in two flavors:

  - Builders for the hub (the "spokes")
  - Builders for developers to use

- *A cache*

  In simpler setups the "hub" can double as a cache, but as you grow you will likely want to upload build products to a dedicated cache.

[5] https://github.com/
[6] https://about.gitlab.com/

- *One or more "utility" servers*

  A "utility" server is a NixOS server that you can use to host IT infrastructure and miscellaneous utility services to support developers (e.g. web pages, chat bots).

  This server will play a role analogous to a container engine or virtual machine hypervisor in other software architectures, except that we won't necessarily be using virtual machines or containers: many things will run natively on the host as NixOS services. Of course, you can also use this machine to run a container engine or hypervisor in addition to running things natively on the host.

A "utility" server should **not** be part of your continuous integration or continuous deployment pipeline. You should think of such a server as a "junk drawer" for stuff that does not belong in CI/CD.

Moreover, you will either need a cloud platform (e.g. AWS[7]) or data center for hosting these machines. In this book we'll primarily focus on hosting infrastructure on AWS.

These are not the only components you will need to build out your product, but these should be the only components necessary to support DevOps workflows, including continuous integration and continuous deployment.

Notably absent from the above list are:

- *Container-specific infrastructure*

  A NixOS-centric architecture already mitigates some of the need for containerizing services, but the architecture doesn't change much even if you do use containers, because containers can be built by Nixpkgs, distributed via the cache, and declaratively deployed to any NixOS machine.

- *Programming-language-specific infrastructure*

  If Nixpkgs supports a given language then we require no additional infrastructure to support building and deploying that language. However, we might still host language-specific amenities on our utility server, such as generated documentation.

- *Continuous-deployment services*

  NixOS provides out-of-the-box services that we can use for continuous deployment, which we will cover in a later chapter.

- *Cloud/Virtual development environments*

  Nix's support for development shells (e.g. `nix develop`) will be our weapon of choice here.

[7] https://aws.amazon.com/

Scope

So far I've explained NixOS in high-level terms, but you might prefer a more down-to-earth picture of the day-to-day requirements and responsibilities for a professional NixOS user.

To that end, here is a checklist that will summarize what you would need to understand in order to effectively introduce and support NixOS within an organization:

- Infrastructure setup
  - Continuous integration
  - Builders
  - Caching
- Development
  - NixOS module system
  - Project organization
  - NixOS best practices
  - Quality controls
- Testing
  - Running virtual machines
  - Automated testing
- Deployment
  - Provisioning a new system
  - Upgrading a system
  - Dealing with restricted networks
- System administration
  - Infrastructure as code
  - Disk management
  - Filesystem
  - Networking
  - Users and authentication
  - Limits and quotas
- Security
  - System hardening
  - Patching dependencies
- Diagnostics and Debugging
  - Nix failures
  - Test failures
  - Production failures
  - Useful references
- Fielding inquiries
  - System settings

    - Licenses
    - Vulnerabilities
- Non-NixOS Integrations
    - Images
    - Containers

This book will cover all of the above topics and more, although they will not necessarily be grouped or organized in that exact order.

4. Setting up your development environment

I'd like you to be able to follow along with the examples in this book, so this chapter provides a quick setup guide to bootstrap from nothing to deploying a blank NixOS system that you can use for experimentation.

Installing Nix

In order to follow along with this book you will need the following requirements:

- Nix version 2.18.1 or newer
- Flake support enabled

  Specifically, you'll need to enable the following experimental features in your Nix configuration file[1]:

  ```
  extra-experimental-features = nix-command flakes repl-flake
  ```

You've likely already installed Nix if you're reading this book, but I'll still cover how to do this because I have a few tips to share that can help you author a more reliable installation script for your colleagues.

Needless to say, if you or any of your colleagues are using NixOS as your development operating system then you don't need to install Nix and you can skip to the Running a NixOS Virtual Machine section below.

Default installation

If you go to the download page for Nix[2] it will tell you to run something similar to this:

```
$ sh <(curl --location https://nixos.org/nix/install)
```

Throughout this book I'll use consistently long option names instead of short names (e.g. `--location` instead of `-L`), for two reasons:

- Long option names are more self-documenting
- Long option names are easier to remember

For example, `tar --extract --file` is clearer and a better mnemonic than `tar xf`.

You may freely use shorter option names if you prefer, though, but I still highly recommend using long option names at least for non-interactive scripts.

[1] https://nixos.org/manual/nix/stable/command-ref/conf-file.html
[2] https://nixos.org/download.html

Depending on your platform the download instructions might also tell you to pass the `--daemon` or `--no-daemon` option to the installation script to specify a single-user or multi-user installation. For simplicity, the instructions in this chapter will omit the `--daemon` / `--no-daemon` flag, but keep in mind the following platform-specific advice:

- *On macOS the installer defaults to a multi-user Nix installation*

  macOS doesn't even support a single-user Nix installation, so this is a good default.

- *On Windows the installer defaults to a single-user Nix installation*

  This default is also the recommended option.

- *On Linux the installer defaults to a single-user Nix installation*

  This is the one case where the default behavior is questionable. Multi-user Nix installations are typically better if your Linux distribution supports Systemd, so you should explicitly specify `--daemon` if you use `systemd`.

Pinning the version

First, we will want to pin the version of Nix that you install if you're creating setup instructions for others to follow. For example, this book will be based on Nix version 2.18.1, and you can pin the Nix version like this:

```
$ VERSION='2.18.1'
$ URL="https://releases.nixos.org/nix/nix-${VERSION}/install"
$ sh <(curl --location "${URL}")
```

... and you can find the full set of available releases by visiting the release file server[3].

Feel free to use a Nix version newer than 2.18.1 if you want. The above example installation script only pins the version 2.18.1 because that's what happened to be the latest stable version at the time of this writing. That's also the Nix version that the examples from this book have been tested against.

The only really important thing is that everyone within your organization uses the same version of Nix, if you want to minimize your support burden.

However, there are a few more options that the script accepts that we're going to make good use of, and we can list those options by supplying `--help` to the script:

```
$ VERSION='2.18.1'
$ URL="https://releases.nixos.org/nix/nix-${VERSION}/install"
$ sh <(curl --location "${URL}") --help
```

[3] https://releases.nixos.org/?prefix=nix/

```
Nix Installer [--daemon|--no-daemon] [--daemon-user-count INT] [--no-channel-add] [--no-modify-profile\
] [--nix-extra-conf-file FILE]
Choose installation method.

 --daemon:    Installs and configures a background daemon that manages the store,
              providing multi-user support and better isolation for local builds.
              Both for security and reproducibility, this method is recommended if
              supported on your platform.
              See https://nixos.org/manual/nix/stable/installation/installing-binary.html#multi-user-i\
nstallation

 --no-daemon: Simple, single-user installation that does not require root and is
              trivial to uninstall.
              (default)

 --no-channel-add:    Don't add any channels. nixpkgs-unstable is installed by default.

 --no-modify-profile: Don't modify the user profile to automatically load nix.

 --daemon-user-count: Number of build users to create. Defaults to 32.

 --nix-extra-conf-file: Path to nix.conf to prepend when installing /etc/nix/nix.conf

 --tarball-url-prefix URL: Base URL to download the Nix tarball from.
```

You might wonder if you can use the `--tarball-url-prefix` option for distributing a custom build of Nix, but that's not what this option is for. You can only use this option to download Nix from a different location (e.g. an internal mirror), because the new download still has to match the same integrity check as the old download.

Don't worry, though; there still is a way to distribute a custom build of Nix, and we'll cover that in a later chapter.

Configuring the installation

The extra options of interest to us are:

- `--nix-extra-conf-file`

  This lets you extend the installed `nix.conf` if you want to make sure that all users within your organization share the same settings.

- `--no-channel-add`

  You can (and should) enable this option within a professional organization to disable the preinstallation of any channels.

These two options are crucial because we are going to use them to systematically replace Nix channels with flakes.

Nix channels are a trap and I treat them as a legacy Nix feature poorly suited for professional development, despite how ingrained they are in the Nix ecosystem.

The issue with channels is that they essentially introduce impurity into your builds by depending on the `NIX_PATH` and there aren't great solutions for enforcing that every Nix user or every machine within your organization has the exact same `NIX_PATH`.

Moreover, Nix now supports flakes, which you can think of as a more modern alternative to channels. Familiarity with flakes is not a precondition to reading this book, though: I'll teach you what you need to know.

So what we're going to do is:

- *Disable channels by default*

  Developers can still opt in to channels by installing them, but disabling channels by default will discourage people from contributing Nix code that depends on the `NIX_PATH`.

- *Append the following setting to `nix.conf` to enable the use of flakes:*

  ```
  extra-experimental-features = nix-command flakes repl-flake
  ```

So the final installation script we'll end up with is:

```
$ VERSION='2.18.1'
$ URL="https://releases.nixos.org/nix/nix-${VERSION}/install"
$ CONFIGURATION="
extra-experimental-features = nix-command flakes repl-flake
extra-trusted-users = ${USER}
"
$ sh <(curl --location "${URL}") \
    --no-channel-add \
    --nix-extra-conf-file <(<<< "${CONFIGURATION}")
```

The prior script only works if your shell is Bash or Zsh and all shell commands throughout this book assume the use of one of those two shells.

For example, the above command uses support for process substitution (which is not available in a POSIX shell environment) because otherwise we'd have to create a temporary file to store the `CONFIGURATION` and clean up the temporary file afterwards (which is tricky to do 100% reliably). Process substitution is also more reliable than a temporary file because it happens entirely in memory and the intermediate result can't be accidentally deleted.

Running a NixOS virtual machine

Now that you've installed Nix I'll show you how to launch a NixOS virtual machine (VM) so that you can easily test the examples throughout this book.

macOS-specific instructions

If you are using macOS, then follow the instructions in the Nixpkgs manual[4] to set up a local Linux builder. We'll need this builder to create other NixOS machines, since they require Linux build products.

In particular, you will need to leave that builder running in the background while following the remaining examples in this chapter. In other words, in one terminal window you will need to run:

```
$ nix run 'nixpkgs#darwin.linux-builder'
```

... and you will need that to be running whenever you need to build a NixOS system. However, you can shut down the builder when you're not using it by giving the builder the `shutdown now` command.

The `nix run nixpkgs#darwin.linux-builder` command is not enough to set up Linux builds on macOS. Read and follow the full set of instructions from the Nixpkgs manual linked above.

If you are using Linux (including NixOS or the Windows Subsystem for Linux) you can skip to the next step.

Platform-independent instructions

Run the following command to generate your first project:

```
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10#setup'
```

... that will generate the following `flake.nix` file:

```
{ inputs = {
    flake-utils.url = "github:numtide/flake-utils/v1.0.0";

    nixpkgs.url = "github:NixOS/nixpkgs/23.11";
  };

  outputs = { flake-utils, nixpkgs, ... }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = nixpkgs.legacyPackages."${system}";

        base = { lib, modulesPath, ... }: {
          imports = [ "${modulesPath}/virtualisation/qemu-vm.nix" ];

          # https://github.com/utmapp/UTM/issues/2353
          networking.nameservers = lib.mkIf pkgs.stdenv.isDarwin [ "8.8.8.8" ];
```

[4] https://nixos.org/manual/nixpkgs/stable/#sec-darwin-builder

```
          virtualisation = {
            graphics = false;

            host = { inherit pkgs; };
          };
        };

        machine = nixpkgs.lib.nixosSystem {
          system = builtins.replaceStrings [ "darwin" ] [ "linux" ] system;

          modules = [ base ./module.nix ];
        };

        program = pkgs.writeShellScript "run-vm.sh" ''
          export NIX_DISK_IMAGE=$(mktemp -u -t nixos.qcow2)

          trap "rm -f $NIX_DISK_IMAGE" EXIT

          ${machine.config.system.build.vm}/bin/run-nixos-vm
        '';

      in
        { packages = { inherit machine; };

          apps.default = {
            type = "app";

            program = "${program}";
          };
        }
    );
}
```

... and also the following `module.nix` file:

```
# module.nix

{ services.getty.autologinUser = "root";
}
```

Then run this command within the same directory to run our test virtual machine:

```
$ nix run
warning: creating lock file '…/flake.lock'
trace: warning: system.stateVersion is not set, defaulting to 23.11. …
…

Run 'nixos-help' for the NixOS manual.

nixos login: root (automatic login)


[root@nixos:~]#
```

You can then shut down the virtual machine by entering `shutdown now`.

If you're unable to shut down the machine gracefully for any reason you can shut down the machine non-gracefully by typing `Ctrl-a + c` to open the `qemu` prompt and then entering `quit` to exit.

Also, don't worry about the `system.stateVersion` warning for now. We'll fix that later.

If you were able to successfully launch and shut down the virtual machine then you're ready to follow along with the remaining examples throughout this book. If you see an example in this book that begins with this line:

```
# module.nix

…
```

… then that means that I want you to save that example code to the `module.nix` file and then restart the virtual machine by running `nix run`.

For example, let's test that right now; save the following file to `module.nix`:

```
# module.nix

{ services.getty.autologinUser = "root";

  services.postgresql.enable = true;
}
```

… then start the virtual machine and log into the machine. As the `root` user, run:

```
[root@nixos:~]# sudo --user postgres psql
psql (14.5)
Type "help" for help.

postgres=#
```

… and now you should have command-line access to a `postgres` database.

The run script in the `flake.nix` file ensures that the virtual machine does not persist state in between runs so that you can safely experiment inside of the virtual machine without breaking upcoming examples.

5. Our first web server

Now that we can build and run a local NixOS virtual machine we can create our first toy web server. We will use this server throughout this book as the running example which will start off simple and slowly grow in maturity as we increase the realism of the example and build out the supporting infrastructure.

Hello, world!

We'll begin from the template project from "Setting up your development environment". You can either begin from the previous chapter by running the following command (if you haven't done so already):

```
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10#setup'
```

... or if you want to skip straight to the final result at the end of this chapter you can run:

```
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10#server'
```

Let's modify `module.nix` to specify a machine that serves a simple static "Hello, world!" page on `http://localhost`:

```
# module.nix

{ pkgs, ... }:

{ services = {
    getty.autologinUser = "root";

    nginx = {
      enable = true;

      virtualHosts.localhost.locations."/" = {
        index = "index.html";

        root = pkgs.writeTextDir "index.html" ''
          <html>
          <body>
          Hello, world!
          </body>
          </html>
        '';
      };
    };
  };
```

```
  networking.firewall.allowedTCPPorts = [ 80 ];

  virtualisation.forwardPorts = [
    { from = "host"; guest.port = 80; host.port = 8080; }
  ];

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

You can read the above code as saying:

- *Enable `nginx` which currently only listens on `localhost`*
  In other words, `nginx` will only respond to requests addressed to `localhost` (e.g. `127.0.0.1`).
- *Serve a static web page*
  … which is a bare-bones "Hello, world!" HTML page.
- *Open port 80 on the virtual machine's firewall*
  … since that is the port that `nginx` will listen on by default until we create a certificate and enable TLS.
- *Forward port 8080 on the "host" to port 80 on the "guest"*
  The "guest" is the virtual machine and the "host" is your development machine.
- *Allow the `root` user to log in with an empty password*
- *Set the system's "state version" to 23.11*

You always want to specify a system state version that matches the starting version of Nixpkgs for that machine and *never change it* afterwards. In other words, even if you upgrade Nixpkgs later on you would keep the state version the same.

Nixpkgs uses the state version to migrate your NixOS system because in order to migrate your system each migration needs to know where your system started from.

Two common mistakes NixOS users sometimes make are:

- *updating the state version when they upgrade Nixpkgs*
  This will cause the machine to never be migrated because Nixpkgs will think that the machine was never deployed to an older version.
- *specifying a uniform state version across a fleet of NixOS machines*
  For example, you might have one NixOS machine in your data center that was first deployed using Nixpkgs 23.05 and another machine in your data center that was first deployed using Nixpkgs 23.11. If you try to change their state versions to match then one or the other might not upgrade correctly.

If you deploy that using `nix run` you can open the web page in your browser by visiting `http://localhost:8080`[1] which should display the following contents:

> Hello, world!

In general I don't recommend testing things by hand like this. Remember the "master cue":

> Every common build/test/deploy-related activity should be possible with at most a single command using Nix's command line interface.

In a later chapter we'll cover how to automate this sort of testing using NixOS's support for integration tests. These tests will also take care of starting up and tearing down the virtual machine for you so that you don't have to do that by hand either.

DevOps

The previous example illustrates how NixOS promotes DevOps on a small scale. If the inline web page represents the software development half of the project (the "Dev") and the `nginx` configuration represents the operational half of the project (the "Ops") then we can in principle store both the "Dev" and the "Ops" halves of our project within the same file. As an extreme example, we can even template the web page with system configuration options!

```
# module.nix

{ config, lib, pkgs, ... }:

{ services = {
    getty.autologinUser = "root";

    nginx = {
      enable = true;

      virtualHosts.localhost.locations."/" = {
        index = "index.html";

        root = pkgs.writeTextDir "index.html" ''
         <html>
         <body>
         This server's firewall has the following open ports:

         <ul>
         ${
         let
           renderPort = port: "<li>${toString port}</li>\n";

         in
```

[1] http://localhost:8080

```
          lib.concatMapStrings renderPort config.networking.firewall.allowedTCPPorts
        }
        </ul>
        </body>
        </html>
      '';
    };
  };
};

networking.firewall.allowedTCPPorts = [ 80 ];

virtualisation.forwardPorts = [
  { from = "host"; guest.port = 80; host.port = 8080; }
];

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

If you restart the machine and refresh http://localhost:8080[2] the page should now display:

> This server's firewall has the following open ports:
>
> - 80

There are less roundabout ways to query our system's configuration that don't involve serving a web page. For example, using the same `flake.nix` file we can more directly query the open ports using:

```
$ nix eval '.#machine.config.networking.firewall.allowedTCPPorts'
[ 80 ]
```

TODO list

Now we're going to create the first prototype of a toy web application: a TODO list implemented entirely in client-side JavaScript (later on we'll add a backend service).

Create a subdirectory named `www` within your current directory:

```
$ mkdir www
```

... and then save a file named `index.html` with the following contents underneath that subdirectory:

[2] http://localhost:8080

```
<html>
<body>
<button id='add'>+</button>
</body>
<script>
let add = document.getElementById('add');
function newTask() {
    let subtract = document.createElement('button');
    subtract.textContent = "-";
    let input = document.createElement('input');
    input.setAttribute('type', 'text');
    let div = document.createElement('div');
    div.replaceChildren(subtract, input);
    function remove() {
      div.replaceChildren();
      div.remove();
    }
    subtract.addEventListener('click', remove);
    add.before(div);
}
add.addEventListener('click', newTask);
</script>
</html>
```

In other words, the above file should be located at `www/index.html` relative to the directory containing your `module.nix` file.

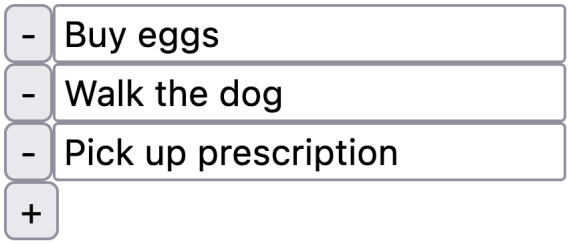
Now save the following NixOS configuration to `module.nix`:

```
# module.nix

{ services = {
    getty.autologinUser = "root";

    nginx = {
      enable = true;

      virtualHosts.localhost.locations."/" = {
        index = "index.html";

        root = ./www;
      };
    };
  };

  networking.firewall.allowedTCPPorts = [ 80 ];

  virtualisation.forwardPorts = [
    { from = "host"; guest.port = 80; host.port = 8080; }
  ];

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

If you restart the virtual machine and refresh the web page you'll see a single + button:

Each time you click the + button it will add a TODO list item consisting of:

- A text entry field to record the TODO item
- A - button to delete the TODO item



Passing through the filesystem

The previous NixOS configuration requires rebuilding and restarting the virtual machine every time we change the web page. If you try to change the `./www/index.html` file while the virtual machine is running you won't see any changes take effect.

However, we can pass through our local filesystem to the virtual machine so that we can easily test changes. To do so, add the following option to the configuration:

```
virtualisation.sharedDirectories.www = {
  source = "$WWW";
  target = "/var/www";
};
```

... and change `module.nix` to reference `/var/www`, like this:

```
  virtualHosts.localhost.locations."/" = {
    index = "index.html";
    root = "/var/www";
  };
```

Finally, restart the machine, except with a slightly modified version of our original `nix run` command:

```
$ WWW="$PWD/www" nix run
```

Now, we only need to refresh the page to view any changes we make to `index.html` and we no longer need to restart the virtual machine.

Exercise: Add a "TODO list" heading (i.e. `<h1>TODO list</h1>`)to the web page and refresh the page to confirm that your changes took effect.

6. NixOS option definitions

By this point in the book you may have copied and pasted some NixOS code, but perhaps you don't fully understand what is going on, especially if you're not an experienced NixOS user. This chapter will slow down and help you solidify your understanding of the NixOS module system so that you can improve your ability to read, author, and debug modules.

Throughout this book I'll consistently use the following terminology to avoid ambiguity:

- "Option declarations" will refer to the `options` attribute of a NixOS module
- "Option definitions" will refer to the `config` attribute of a NixOS module

Along the same lines:

- "Declare an option" will mean to set an attribute nested underneath `options`
- "Define an option" will mean to set an attribute nested underneath `config`

In this chapter and the next chapter we'll focus mostly on option *definitions* and later on we'll cover option *declarations* in more detail.

Anatomy of a NixOS module

In the most general case, a NixOS module has the following "shape":

```
# Module arguments which our system can use to refer to its own configuration
{ config, lib, pkgs, ... }:

{ # Other modules to import
  imports = [
    …
  ];

  # Options that this module declares
  options = {
    …
  };

  # Options that this module defines
  config = {
    …
  };
}
```

In other words, in the fully general case a NixOS module is a function whose output is an attribute set with three attributes named `imports`, `options`, and `config`.

Nix supports data structures known "attribute sets" which are analogous to "maps" or "records" in other programming languages.

To be precise, Nix uses the following terminology:

- *an "attribute set" is a data structure associating keys with values*
  For example, this is a nested attribute set:

  ```
  { bio = { name = "Alice"; age = 24; };
    job = "Software engineer";
  }
  ```

- an "attribute" is Nix's name for a key or a field of an "attribute set"
  For example, `bio`, `job`, `name`, and `age` are all attributes in the above example.

- *an "attribute path" is a chain of one or more attributes separated by dots*
  For example, `bio.name` is an attribute path.

I'm explaining all of this because I'll use the terms "attribute set", "attribute", and "attribute path" consistently throughout the text to match Nix's official terminology (even though no other language uses those terms).

Syntactic sugar

All elements of a NixOS module are optional and NixOS supports "syntactic sugar" to simplify several common cases. For example, you can omit the module arguments if you don't use them:

```
{ imports = [
    …
  ];

  options = {
    …
  };

  config = {
    …
  };
}
```

You can also omit any of the `imports`, `options`, or `config` attributes, too, like in this module, which only imports other modules:

```
{ imports = [
    ./physical.nix
    ./logical.nix
  ];
}
```

... or this `config`-only module:

```
{ config = {
    services = {
      apache-kafka.enable = true;

      zookeeper.enable = true;
    };
  };
}
```

Additionally, the NixOS module system provides special support for modules which only define options by letting you elide the `config` attribute and promote the options defined within to the "top level". As an example, we can simplify the previous NixOS module to this:

```
{ services = {
    apache-kafka.enable = true;

    zookeeper.enable = true;
  };
}
```

You might wonder if there should be some sort of coding style which specifies whether people should include or omit these elements of a NixOS module. For example, perhaps you might require that all elements are present, for consistency, even if they are empty or unused.

My coding style for NixOS modules is:

- *you should permit omitting the module arguments*
- *you should permit omitting the `imports`, `options`, or `config` attributes*
- *you should **avoid** eliding the `config` attribute*
  In other words, if you do define any options, always nest them underneath the `config` attribute.

NixOS modules are not language features

The Nix programming language does not provide any built-in support for NixOS modules. This sometimes confuses people new to either the Nix programming language or the NixOS module system.

The NixOS module system is a domain-specific language implemented within the Nix programming language. Specifically, the NixOS module system is (mostly) implemented within the `lib/modules.nix` file included in Nixpkgs[1]. If you ever receive a stack trace related to the NixOS module system you will often see functions from `modules.nix` show up in the stack trace, because they are ordinary functions and not language features.

In fact, a NixOS module in isolation is essentially "inert" from the Nix language's point of view. For example, if you save the following NixOS module to a file named `example.nix`:

```
{ config = {
    services.openssh.enable = true;
  };
}
```

... and you evaluate that, the result will be the same, just without the syntactic sugar:

```
$ nix eval --file ./example.nix
{ config = { services = { openssh = { enable = true; }; }; }; }
```

The Nix programming language provides "syntactic sugar" for compressing nested attributes by chaining them using a dot (`.`). In other words, this Nix expression:

```
{ config = {
    services.openssh.enable = true;
  };
}
```

... is the same thing as this Nix expression:

```
{ config = {
    services = {
      openssh = {
        enable = true;
      };
    };
  };
}
```

... and they are both also the same thing as this Nix expression:

```
{ config.services.openssh.enable = true; }
```

Note that this syntactic sugar is a feature of the *Nix programming language*, not the NixOS module system. In other words, this feature works even for Nix expressions that are not destined for use as NixOS modules.

Along the same lines, the following NixOS module:

[1] https://github.com/NixOS/nixpkgs/blob/23.11/lib/modules.nix

```
{ config, ... }:

{ config = {
    services.apache-kafka.enable = config.services.zookeeper.enable;
  };
}
```

... is just a function. If we save that to `example.nix` and then evaluate that the interpreter will simply say that the file evaluates to a "lambda" (an anonymous function):

```
$ nix eval --file ./example.nix
<LAMBDA>
```

... although we can get a more useful result within the `nix repl` by calling our function on a sample argument:

```
$ nix repl
…
nix-repl> example = import ./example.nix

nix-repl> input = { config = { services.zookeeper.enable = true; }; }

nix-repl> output = example input

nix-repl> :p output
{ config = { services = { apache-kafka = { enable = true; }; }; }; }

nix-repl> output.config.services.apache-kafka.enable
true
```

This illustrates that our NixOS module really is just a function whose input is an attribute set and whose output is also an attribute set. There is nothing special about this function other than it happens to be the same shape as what the NixOS module system accepts.

NixOS

So if NixOS modules are just pure functions or pure attribute sets, what turns those functions or attribute sets into a useful operating system? In other words, what puts the "NixOS" in the "NixOS module system"?

The answer is that this actually happens in two steps:

- *All NixOS modules your system depends on are combined into a single, composite attribute set*

  In other words all of the `imports`, `options` declarations, and `config` settings are fully resolved, resulting in one giant attribute set. The code for combining these modules lives in `lib/modules.nix`[2] in Nixpkgs.

[2] https://github.com/NixOS/nixpkgs/blob/23.11/lib/modules.nix

- *The final composite attribute set contains a special attribute that builds the system*

  Specifically, there will be a `config.system.build.toplevel` attribute path which contains a derivation you can use to build a runnable NixOS system. The top-level code for assembling an operating system lives in `nixos/modules/system/activation/top-level.nix`[3] in Nixpkgs.

This will probably make more sense if we use the NixOS module system ourselves to create a fake placeholder value that will stand in for a real operating system.

First, we'll create our own `top-level.nix` module that will include a fake `config.system.build.toplevel` attribute path that is a string instead of a derivation for building an operating system:

```
# top-level.nix

{ config, lib, ... }:

{ imports = [ ./other.nix ];

  options = {
    system.build.toplevel = lib.mkOption {
      description = "A fake NixOS, modeled as a string";

      type = lib.types.str;
    };
  };

  config = {
    system.build.toplevel =
      "Fake NixOS - version ${config.system.nixos.release}";
  };
}
```

That imports a separate `other.nix` module which we also need to create:

```
# other.nix

{ lib, ... }:

{ options = {
    system.nixos.release = lib.mkOption {
      description = "The NixOS version";

      type = lib.types.str;
    };
  };

  config = {
    system.nixos.release = "23.11";
  };
}
```

We can then materialize the final composite attribute set like this:

[3] https://github.com/NixOS/nixpkgs/blob/23.11/nixos/modules/system/activation/top-level.nix

```
$ nix repl github:NixOS/nixpkgs/23.11
…
nix-repl> result = lib.evalModules { modules = [ ./top-level.nix ]; }

nix-repl> :p result.config
{ system = { build = { toplevel = "Fake NixOS - version 23.11"; }; nixos = { release = "23.11"; }; }; }

nix-repl> result.config.system.build.toplevel
"Fake NixOS - version 23.11"
```

In other words, `lib.evalModules` is the magic function that combines all of our NixOS modules into a composite attribute set.

NixOS essentially does the same thing as in the above example, except on a much larger scale. Also, in a real NixOS system the final `config.system.build.toplevel` attribute path stores a buildable derivation instead of a string.

Recursion

The NixOS module system lets modules refer to the final composite configuration using the `config` function argument that is passed into every NixOS module. For example, this is how our `top-level.nix` module was able to refer to the `system.nixos.release` option that was set in the `other.nix` module:

```
# This represents the final composite configuration
# |
{ config, lib, ... }:

{ …

  config = {
    system.build.toplevel =
      "Fake NixOS - version ${config.system.nixos.release}";
                            # |
                            # … which we can use within our configuration
  };
}
```

You're not limited to referencing configuration values set in other NixOS modules; you can even reference configuration values set within the same module. In other words, NixOS modules support recursion[4] where modules can refer to themselves.

As a concrete example of recursion, we can safely merge the `other.nix` module into the `top-level.nix` module:

[4] https://en.wikipedia.org/wiki/Recursion

```
{ config, lib, ... }:

{ options = {
    system.build.toplevel = lib.mkOption {
      description = "A fake NixOS, modeled as a string";

      type = lib.types.str;
    };

    system.nixos.release = lib.mkOption {
      description = "The NixOS version";

      type = lib.types.str;
    };
  };

  config = {
    system.build.toplevel =
      "Fake NixOS - version ${config.system.nixos.release}";

    system.nixos.release = "23.11";
  };
}
```

... and this would still work, even though this module now refers to its own configuration values. The Nix interpreter won't go into an infinite loop because the recursion is still well-founded.

We can better understand why this recursion is well-founded by simulating how `lib.evalModules` works by hand. Conceptually what `lib.evalModules` does is:

- combine all of the input modules
- compute the fixed point[5] of this composite module

We'll walk through this by performing the same steps as `lib.evalModules`. First, to simplify things we'll consolidate the prior example into a single flake that we can evaluate as we go:

```
# Save this to `./evalModules/flake.nix`

{ inputs.nixpkgs.url = "github:NixOS/nixpkgs/23.11";

  outputs = { nixpkgs, ... }:
    let
      other =
        { lib, ... }:
        { # To save space, this example compresses the code a bit
          options.system.nixos.release = lib.mkOption {
            description = "The NixOS version";
            type = lib.types.str;
          };
          config.system.nixos.release = "23.11";
        };
```

[5] https://en.wikipedia.org/wiki/Fixed_point_(mathematics)

```
      topLevel =
        { config, lib, ... }:
        { imports = [ other ];
          options.system.build.toplevel = lib.mkOption {
            description = "A fake NixOS, modeled as a string";
            type = lib.types.str;
          };
          config.system.build.toplevel =
            "Fake NixOS - version ${config.system.nixos.release}";
        };

    in
      nixpkgs.lib.evalModules { modules = [ topLevel ]; };
}
```

You can evaluate the above flake like this:

```
$ nix eval './evalModules#config'
{ system = { build = { toplevel = "Fake NixOS - version 23.11"; }; nixos = { release = "23.11"; }; }; }

$ nix eval './evalModules#config.system.build.toplevel'
"Fake NixOS - version 23.11"
```

Various `nix` commands (like `nix eval`) take a flake reference as an argument which has the form:

```
${URI}#${ATTRIBUTE_PATH}
```

In the previous example, the `URI` was `./evalModules` (a file path in this case) and the `ATTRIBUTE_PATH` was `config.system.build.toplevel`.

However, if you use `zsh` as your shell with `EXTENDED_GLOB` glob support (i.e. `setopt extended_glob`) then `zsh` interprets `#` as a special character. This is why all of the examples from this book quote the flake reference as a precaution, but if you're not using `zsh` or its extended globbing support then you can remove the quotes, like this:

```
$ nix eval ./evalModules#config.system.build.toplevel
```

If you run into an error like:

```
error: getting status of '/nix/store/…-source/evalModules': No such file or directory
```

... this can happen because you created the `./evalModules` directory inside of a `git` repository. When you use flakes inside of a repository you need to explicitly add them and all files they depend on to the repository using:

```
$ git add ./evalModules/flake.nix
```

Technically, the bare minimum you need to do is actually:

```
$ git add --intent-to-add ./evalModules/flake.nix
```

... which comes in handy if you don't plan to ever actually commit the file.

The first thing that `lib.evalModules` does is to merge the `other` module into the `topLevel` module, which we will simulate by hand by performing the same merge ourselves:

```
{ inputs.nixpkgs.url = "github:NixOS/nixpkgs/23.11";

  outputs = { nixpkgs, ... }:
    let
      topLevel =
        { config, lib, ... }:
        { options.system.nixos.release = lib.mkOption {
            description = "The NixOS version";
            type = lib.types.str;
          };
          options.system.build.toplevel = lib.mkOption {
            description = "A fake NixOS, modeled as a string";
            type = lib.types.str;
          };
          config.system.nixos.release = "23.11";
          config.system.build.toplevel =
            "Fake NixOS - version ${config.system.nixos.release}";
        };

    in
      nixpkgs.lib.evalModules { modules = [ topLevel ]; };
}
```

After that we compute the fixed point of our module by passing the module's output as its own input, the same way that `evalModules` would:

```
{ inputs.nixpkgs.url = "github:NixOS/nixpkgs/23.11";

  outputs = { nixpkgs, ... }:
    let
      topLevel =
        { config, lib, ... }:
        { options.system.nixos.release = lib.mkOption {
            description = "The NixOS version";
            type = lib.types.str;
          };
          options.system.build.toplevel = lib.mkOption {
            description = "A fake NixOS, modeled as a string";
            type = lib.types.str;
          };
          config.system.nixos.release = "23.11";
          config.system.build.toplevel =
            "Fake NixOS - version ${config.system.nixos.release}";
        };

      result = topLevel {
        inherit (result) config options;
        inherit (nixpkgs) lib;
      };

    in
      result;
}
```

This walkthrough grossly oversimplifies what `evalModules` does. For starters, we've completely ignored how `evalModules` uses the `options` declarations to:

- check that configuration values match their declared types
- replace missing configuration values with their default values

However, this oversimplification is fine for now.

The last step is that when `nix eval` accesses the `config.system.build.toplevel` field of the `result`, the Nix interpreter conceptually performs the following substitutions:

```
result.config.system.build.toplevel
```

```
# Substitute `result` with its right-hand side
= (topLevel {
    inherit (result) config options;
    inherit (nixpkgs) lib;
  }).config.system.build.toplevel
```

```
# `inherit` is syntactic sugar for this equivalent Nix expression
= ( topLevel {
      config = result.config;
      options = result.options;
      lib = nixpkgs.lib;
    }
  ).config.system.build.toplevel


# Evaluate the `topLevel` function
= ( { options.system.nixos.release = lib.mkOption {
        description = "The NixOS version";
        type = lib.types.str;
      };
      options.system.build.toplevel = lib.mkOption {
        description = "A fake NixOS, modeled as a string";
        type = lib.types.str;
      };
      config.system.nixos.release = "23.11";
      config.system.build.toplevel =
        "Fake NixOS - version ${result.config.system.nixos.release}";
    }
  ).config.system.build.toplevel


# Access the `config.system.build.toplevel` attribute path
= "Fake NixOS - version ${result.config.system.nixos.release}"


# Substitute `result` with its right-hand side (again)
= "Fake NixOS - version ${
  (topLevel {
      inherit (result) config options;
      inherit (nixpkgs) lib;
    }
  ).config.system.nixos.release
}"


# Evaluate the `topLevel` function (again)
= "Fake NixOS - version ${
  ( { options.system.nixos.release = lib.mkOption {
        description = "The NixOS version";
        type = lib.types.str;
      };
      options.system.build.toplevel = lib.mkOption {
        description = "A fake NixOS, modeled as a string";
        type = lib.types.str;
      };
      config.system.nixos.release = "23.11";
      config.system.build.toplevel =
        "Fake NixOS - version ${result.config.system.nixos.release}";
    }
  ).config.system.nixos.release
}"
```

```
# Access the `config.system.nixos.release` attribute path
= "Fake NixOS - version ${"23.11"}"


# Evaluate the string interpolation
= "Fake NixOS - version 23.11"
```

So even though our NixOS module is defined recursively in terms of itself, that recursion is still well-founded and produces an actual result.

7. Advanced option definitions

NixOS option definitions are actually much more sophisticated than the previous chapter let on and in this chapter we'll cover some common tricks and pitfalls.

Make sure that you followed the instructions from the "Setting up your development environment" chapter if you would like to test the examples in this chapter.

Imports

The NixOS module system lets you import other modules by their path, which merges their option declarations and option definitions with the current module. But, did you know that the elements of an `imports` list don't have to be paths?

You can put inline NixOS configurations in the `imports` list, like these:

```
{ imports = [
    { services.openssh.enable = true; }

    { services.getty.autologinUser = "root"; }
  ];
}
```

... and they will behave as if you had imported files with the same contents as those inline configurations.

In fact, anything that is a valid NixOS module can go in the import list, including NixOS modules that are functions:

```
{ imports = [
    { services.openssh.enable = true; }

    ({ lib, ... }: { services.getty.autologinUser = lib.mkDefault "root"; })
  ];
}
```

I will make use of this trick in a few examples below, so that we can simulate modules importing other modules within a single file.

`lib` utilities

Nixpkgs provides several utility functions for NixOS modules that are stored underneath the "`lib`" hierarchy, and you can find the source code for those functions in `lib/modules.nix`[1].

[1] https://github.com/NixOS/nixpkgs/blob/23.11/lib/modules.nix

If you want to become a NixOS module system expert, take the time to read and understand all of the code in `lib/modules.nix`.

Remember that the NixOS module system is implemented as a domain-specific language in Nix and `lib/modules.nix` contains the implementation of that domain-specific language, so if you understand everything in that file then you understand essentially all that there is to know about how the NixOS module system works under the hood.

That said, this chapter will still try to explain things enough so that you don't have to read through that code.

You do not need to use or understand all of the functions in `lib/modules.nix`, but you do need to familiarize yourself with the following four primitive functions:

- `lib.mkMerge`
- `lib.mkOverride`
- `lib.mkIf`
- `lib.mkOrder`

By "primitive", I mean that these functions cannot be implemented in terms of other functions. They all hook into special behavior built into `lib.evalModules`.

`mkMerge`

The `lib.mkMerge` function merges a list of "configuration sets" into a single "configuration set" (where "configuration set" means a potentially nested attribute set of configuration option settings).

For example, the following NixOS module:

```
{ lib, ... }:

{ config = lib.mkMerge [
    { services.openssh.enable = true; }
    { services.getty.autologinUser = "root"; }
  ];
}
```

... is equivalent to this NixOS module:

```
{ config = {
    services.openssh.enable = true;
    services.getty.autologinUser = "root";
  };
}
```

You might wonder whether you should merge modules using `lib.mkMerge` or merge them using the `imports` list. After all, we could have also written the previous `mkMerge` example as:

```
{ imports = [
    { services.openssh.enable = true; }
    { services.getty.autologinUser = "root"; }
  ];
}
```

... and that would have produced the same result. So which is better?

The short answer is: `lib.mkMerge` is usually what you want.

The long answer is that the main trade-off between `imports` and `lib.mkMerge` is:

- *The `imports` section can merge NixOS modules that are functions*
  `lib.mkMerge` can only merge configuration sets and not functions.

- *The list of `imports` can't depend on any configuration values*
  In practice, this means that you can easily trigger an infinite recursion if you try to do anything fancy using `imports` and you can typically fix the infinite recursion by switching to `lib.mkMerge`.

The latter point is why you should typically prefer using `lib.mkMerge`.

Merging options

You can merge configuration sets that define same option multiple times, like this:

```
{ lib, ... }:

{ config = lib.mkMerge [
    { networking.firewall.allowedTCPPorts = [ 80 ]; }
    { networking.firewall.allowedTCPPorts = [ 443 ]; }
  ];
}
```

... and the outcome of merging two identical attribute paths depends on the option's "type".

For example, the `networking.firewall.allowedTCPPorts` option's type is:

```
$ nix eval '.#machine.options.networking.firewall.allowedTCPPorts.type.description'
"list of 16 bit unsigned integer; between 0 and 65535 (both inclusive)"
```

If you specify a list-valued option twice, the lists are combined, so the above example reduces to this:

```
{ lib, ... }:

{ config = lib.mkMerge [
    { networking.firewall.allowedTCPPorts = [ 80 443 ]; }
  ];
}
```

... and we can even prove that by querying the final value of the option from the command line:

```
$ nix eval '.#machine.config.networking.firewall.allowedTCPPorts'
[ 80 443 ]
```

However, you might find the `nix repl` more convenient if you prefer to interactively browse the available options. Run this command:

```
$ nix repl '.#machine'
…
Added 7 variables.
```

... which will load your NixOS system into the REPL and now you can use tab-completion to explore what is available:

```
nix-repl> config.<TAB>
config.appstream          config.nix
config.assertions         config.nixops
…
nix-repl> config.networking.<TAB>
config.networking.bonds
config.networking.bridges
…
nix-repl> config.networking.firewall.<TAB>
config.networking.firewall.allowPing
config.networking.firewall.allowedTCPPortRanges
…
nix-repl> config.networking.firewall.allowedTCPPorts
[ 80 443 ]
```

> **Exercise**: Try to save the following NixOS module to `module.nix`, which specifies the same option twice without using `lib.mkMerge`:
>
> ```
> { lib, ... }:
>
> { config = {
>     networking.firewall.allowedTCPPorts = [ 80 ];
>     networking.firewall.allowedTCPPorts = [ 443 ];
>   };
> }
> ```
>
> This will fail to deploy. Do you understand why? Specifically, is the failure a limitation of the NixOS module system or the Nix programming language?

You can also nest `lib.mkMerge` underneath an attribute. For example, this:

```
{ config = lib.mkMerge [
    { networking.firewall.allowedTCPPorts = [ 80 ]; }
    { networking.firewall.allowedTCPPorts = [ 443 ]; }
  ];
}
```

... is the same as this:

```
{ config.networking = lib.mkMerge [
    { firewall.allowedTCPPorts = [ 80 ]; }
    { firewall.allowedTCPPorts = [ 443 ]; }
  ];
}
```

... is the same as this:

```
{ config.networking.firewall = lib.mkMerge [
    { allowedTCPPorts = [ 80 ]; }
    { allowedTCPPorts = [ 443 ]; }
  ];
}
```

... is the same as this:

```
{ config.networking.firewall.allowedTCPPorts = lib.mkMerge [ [ 80 ] [ 443 ] ];
}
```

... is the same as this:

```
{ config.networking.firewall.allowedTCPPorts = [ 80 443 ]; }
```

Conflicts

Duplicate options cannot necessarily always be merged. For example, if you merge two configuration sets that disagree on whether to enable a service:

```
{ lib, ... }:

{ config = {
    services.openssh.enable = lib.mkMerge [ true false ];
  };
}
```

... then that will fail at evaluation time with this error:

```
error: The option `services.openssh.enable' has conflicting definition values:
       - In `/nix/store/…-source/module.nix': true
       - In `/nix/store/…-source/module.nix': false
(use '--show-trace' to show detailed location information)
```

This is because `services.openssh.enable` is declared to have a boolean type, and you can only merge multiple boolean values if all occurrences agree. You can verify this yourself by changing both occurrences to `true`, which will fix the error.

As a general rule of thumb:

- *Most scalar option types will fail to merge distinct values*

  e.g. boolean values, strings, integers.

- *Most complex option types will successfully merge in the obvious way*

  e.g. lists will be concatenated and attribute sets will be combined.

The most common exception to this rule of thumb is the "lines" type (`lib.types.lines`), which is a string option type that you can define multiple times. `services.zookeeper.extraConf` is an example of one such option that has this type:

```
{ lib, ... }:

{ config = {
    services.zookeeper = {
      enable = true;

      extraConf = lib.mkMerge [ "initLimit=5" "syncLimit=2" ];
    };
  };
}
```

... and merging multiple occurrences of that option concatenates them as lines by inserting an intervening newline character:

```
$ nix eval '.#machine.config.services.zookeeper.extraConf'
"initLimit=5\nsyncLimit=2"
```

`mkOverride`

The `lib.mkOverride` function specifies the "priority" of an option definition, which comes in handy if you want to override a configuration value that another NixOS module already defined.

Higher priority overrides

This most commonly comes up when we need to override an option that was already defined by one of our dependencies (typically a NixOS module provided by Nixpkgs). One example would be overriding the restart frequency of `nginx`:

```
{ config = {
    services.nginx.enable = true;

    systemd.services.nginx.serviceConfig.RestartSec = "5s";
  };
}
```

The above naïve attempt will fail at evaluation time with:

```
error: The option `systemd.services.nginx.serviceConfig.RestartSec' has conflicting definition values:
       - In `/nix/store/…-source/nixos/modules/services/web-servers/nginx/default.nix': "10s"
       - In `/nix/store/…-source/module.nix': "5s"
(use '--show-trace' to show detailed location information)
```

The problem is that when we enable `nginx` that automatically defines a whole bunch of other NixOS options, including `systemd.services.nginx.serviceConfig.RestartSec`[2]. This option is a scalar string option that disallows multiple distinct values because the NixOS module system by default has no way to known which one to pick to resolve the conflict.

However, we can use `mkOverride` to annotate our value with a higher priority so that it overrides the other conflicting definition:

```
{ lib, ... }:

{ config = {
    services.nginx.enable = true;

    systemd.services.nginx.serviceConfig.RestartSec = lib.mkOverride 50 "5s";
  };
}
```

… and now that works, since we specified a new priority of `50` that takes priority over the default priority of `100`. There is also a pre-existing utility named `lib.mkForce` which sets the priority to 50, so we could have also used that instead:

```
{ lib, ... }:

{ config = {
    services.nginx.enable = true;

    systemd.services.nginx.serviceConfig.RestartSec = lib.mkForce "5s";
  };
}
```

[2] https://github.com/NixOS/nixpkgs/blob/23.11/nixos/modules/services/web-servers/nginx/default.nix#L1234

You do **not** want to do this:

```
{ lib, ... }:

{ config = {
    services.nginx.enable = true;

    systemd.services.nginx.serviceConfig = lib.mkForce { RestartSec = "5s" };
  };
}
```

That is not equivalent, because it overrides not only the `RestartSec` attribute, but also all other attributes underneath the `serviceConfig` attribute (like `Restart`, `User`, and `Group`, all of which are now gone).

You always want to narrow your use of `lib.mkForce` as much as possible to protect against this common mistake.

The default priority is `100` and **lower** numeric values actually represent **higher** priority. In other words, an option definition with a priority of `50` takes precedence over an option definition with a priority of `100`.

Yes, the NixOS module system confusingly uses lower numbers to indicate higher priorities, but in practice you will rarely see explicit numeric priorities. Instead, people tend to use derived utilities like `lib.mkForce` or `lib.mkDefault` which select the appropriate numeric priority for you.

In extreme cases you might still need to specify an explicit numeric priority. The most common example is when one of your dependencies already define an option using `lib.mkForce` and you need to override *that*. In that scenario you could use `lib.mkOverride 49`, which would take precedence over `lib.mkForce`

```
{ lib, ... }:

{ config = {
    services.nginx.enable = true;

    systemd.services.nginx.serviceConfig.RestartSec = lib.mkMerge [
      (lib.mkForce "5s")
      (lib.mkOverride 49 "3s")
    ];
  };
}
```

... which will produce a final value of:

```
$ nix eval '.#machine.config.systemd.services.nginx.serviceConfig.RestartSec'
"3s"
```

Lower priority overrides

The default values for options also have a priority, which is priority `1500` and there's a `lib.mkOptionDefault` that sets a configuration value to that same priority.

That means that a NixOS module like this:

```
{ lib, ... }:

{ options.foo = lib.mkOption {
    default = 1;
  };
}
```

... is the exact same thing as a NixOS module like this:

```
{ lib, ... }:

{ options.foo = lib.mkOption { };

  config.foo = lib.mkOptionDefault 1;
}
```

However, you will more commonly use `lib.mkDefault` which defines a configuration option with priority `1000`. Typically you'll use `lib.mkDefault` if you want to override the default value of an option, while still allowing a downstream user to override the option yet again at the normal priority (`100`).

`mkIf`

`mkIf` is far-and-away the most widely used NixOS module primitive, because you can use `mkIf` to selectively enable certain options based on the value of another option.

An extremely common idiom from Nixpkgs is to use `mkIf` in conjunction with an `enable` option, like this:

```
# module.nix

let
  # Pretend that this came from another file
  cowsay =
    { config, lib, pkgs, ... }:

    { options.services.cowsay = {
        enable = lib.mkEnableOption "cowsay";

        greeting = lib.mkOption {
          description = "The phrase the cow will greet you with";

          type = lib.types.str;

          default = "Hello, world!";
        };
      };

      config = lib.mkIf config.services.cowsay.enable {
        systemd.services.cowsay = {
          wantedBy = [ "multi-user.target" ];

          script = "${pkgs.cowsay}/bin/cowsay ${config.services.cowsay.greeting}";
```

```
        };
      };
    }

in
  { imports = [ cowsay ];

    config = {
      services.cowsay.enable = true;

      services.getty.autologinUser = "root";
    };
  }
```

If you launch the above NixOS configuration you should be able to verify that the `cowsay` service is running like this:

```
[root@nixos:~]# systemctl status cowsay
● cowsay.service
     Loaded: loaded (/etc/systemd/system/cowsay.service; enabled; preset: enabl>
     Active: inactive (dead) since Sat 2023-11-05 20:11:05 UTC; 43s ago
   Duration: 106ms
    Process: 683 ExecStart=/nix/store/v02wsh00gi1vcblpcl8p103qhlpkaifb-unit-scr>
   Main PID: 683 (code=exited, status=0/SUCCESS)
         IP: 0B in, 0B out
        CPU: 19ms

Nov 05 20:11:04 nixos systemd[1]: Started cowsay.service.
Nov 05 20:11:05 nixos cowsay-start[689]:  _______________
Nov 05 20:11:05 nixos cowsay-start[689]: < Hello, world! >
Nov 05 20:11:05 nixos cowsay-start[689]:  ---------------
Nov 05 20:11:05 nixos cowsay-start[689]:         \   ^__^
Nov 05 20:11:05 nixos cowsay-start[689]:          \  (oo)\_______
Nov 05 20:11:05 nixos cowsay-start[689]:             (__)\       )\/\
Nov 05 20:11:05 nixos cowsay-start[689]:                 ||----w |
Nov 05 20:11:05 nixos cowsay-start[689]:                 ||     ||
Nov 05 20:11:05 nixos systemd[1]: cowsay.service: Deactivated successfully.
```

You might wonder why we need a `mkIf` primitive at all. Couldn't we use an `if` expression like this instead?

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

{ …

  config = if config.services.cowsay.enable then {
    systemd.services.cowsay = {
      wantedBy = [ "multi-user.target" ];

      script = "${pkgs.cowsay}/bin/cowsay ${config.services.cowsay.greeting}";
    };
  } else { };
}
```

The most important reason why this doesn't work is because it triggers an infinite loop:

```
error: infinite recursion encountered

       at /nix/store/vgicc88fhmlh7mwik7gqzzm2jyfva9l9-source/lib/modules.nix:259:21:

          258|           (regularModules ++ [ internalModule ])
          259|           ({ inherit lib options config specialArgs; } // specialArgs);
             |                     ^
          260|         in mergeModules prefix (reverseList collected);
(use '--show-trace' to show detailed location information)
```

The reason why is because the recursion is not well-founded:

```
# This attribute directly depends on itself
# |          |
  config = if config.services.cowsay.enable then {
```

... and the reason why `lib.mkIf` doesn't share the same problem is because `evalModules` pushes `mkIf` conditions to the "leaves" of the configuration tree, as if we had instead written this:

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

{ …

  config = {
    systemd.services.cowsay = {
      wantedBy = lib.mkIf config.services.cowsay.enable [ "multi-user.target" ];

      script =
        lib.mkIf config.services.cowsay.enable
          "${pkgs.cowsay}/bin/cowsay ${config.services.cowsay.greeting}";
    };
  };
}
```

... which makes the recursion well-founded.

The second reason we use `lib.mkIf` is because it correctly handles the fallback case. To see why that matters, consider this example that tries to create a `service.kafka.enable` short-hand synonym for `services.apache-kafka.enable`:

```
let
  kafkaSynonym =
    { config, lib, ... }:

    { options.services.kafka.enable = lib.mkEnableOption "apache";

      config.services.apache-kafka.enable = config.services.kafka.enable;
    };

in
  { imports = [ kafkaSynonym ];

    config.services.apache-kafka.enable = true;
  }
```

The above example leads to a conflict because the `kafkaSynonym` module defines `services.kafka.enable` to `false` (at priority 100), and the downstream module defines `services.apache-kafka.enable` to `true` (also at priority 100).

Had we instead used `mkIf` like this:

```
let
  kafkaSynonym =
    { config, lib, ... }:

    { options.services.kafka.enable = lib.mkEnableOption "apache";

      config.services.apache-kafka.enable =
        lib.mkIf config.services.kafka.enable true;
    };

in
  { imports = [ kafkaSynonym ];

    config.services.apache-kafka.enable = true;
  }
```

... then that would do the right thing because in the default case `services.apache-kafka.enable` would remain undefined, which would be the same thing as being defined as `false` at priority `1500`. That avoids defining the same option twice at the same priority.

`mkOrder`

The NixOS module system strives to make the behavior of our system depend as little as possible on the order in which we import or `mkMerge` NixOS modules. In other words, if we import two modules that we depend on:

```
{ imports = [ ./A.nix ./B.nix ]; }
```

... then ideally the behavior shouldn't change if we import those same two modules in a different order:

```
{ imports = [ ./B.nix ./A.nix ]; }
```

... and in *most cases* that is true. 99% of the time you can safely sort your import list and either your NixOS system will be *exactly* the same as before (producing the exact same Nix store build product) or *essentially* the same as before, meaning that the difference is irrelevant. However, for those 1% of cases where order matters we need the `lib.mkOrder` function.

Here's one example of where ordering matters:

```
let
  moduleA = { pkgs, ... }: {
    environment.defaultPackages = [ pkgs.gcc ];
  };

  moduleB = { pkgs, ... }: {
    environment.defaultPackages = [ pkgs.clang ];
  };

in
  { imports = [ moduleA moduleB ]; }
```

Both the `gcc` package and `clang` package add a `cc` executable to the `PATH`, so the order matters here because the first `cc` on the `PATH` wins.

In the above example, `clang`'s `cc` is the first one on the `PATH`, because we imported `moduleB` second:

```
[root@nixos:~]# readlink $(type -p cc)
/nix/store/6szy6myf8vqrmp8mcg8ps7s782kygy5g-clang-wrapper-11.1.0/bin/cc
```

... but if we flip the order imports:

```
    imports = [ moduleB moduleA ];
```

... then `gcc`'s `cc` comes first on the `PATH`:

```
[root@nixos:~]# readlink $(type -p cc)
/nix/store/9wqn04biky07333wkl35bfjv9zv009pl-gcc-wrapper-9.5.0/bin/cc
```

This sort of order-sensitivity frequently arises for "list-like" option types, including actual lists or string types like `lines` that concatenate multiple definitions.

Fortunately, we can fix situations like these with the `lib.mkOrder` function, which specifies a numeric ordering that NixOS will respect when merging multiple definitions of the same option.

Every option's numeric order is `1000` by default, so if we set the numeric order of `clang` to `1500`:

```
let
  moduleA = { pkgs, ... }: {
    environment.defaultPackages = [ pkgs.gcc ];
  };

  moduleB = { lib, pkgs, ... }: {
    environment.defaultPackages = lib.mkOrder 1500 [ pkgs.clang ];
  };

in
  { imports = [ moduleA moduleB ]; }
```

... then `gcc` will always come first on the `PATH`, no matter which order we import the modules.

You can also use `lib.mkBefore` and `lib.mkAfter`, which provide convenient synonyms for numeric order `500` and `1500`, respectively:

```
mkBefore = mkOrder 500;

mkAfter = mkOrder 1500;
```

... so we could have also written:

```
let
  moduleA = { pkgs, ... }: {
    environment.defaultPackages = [ pkgs.gcc ];
  };

  moduleB = { lib, pkgs, ... }: {
    environment.defaultPackages = lib.mkAfter [ pkgs.clang ];
  };

in
  { imports = [ moduleA moduleB ]; }
```

8. Deploying to AWS using Terraform

Up until now we've been playing things safe and test-driving everything locally on our own machine. We could even prolong this for quite a while because NixOS has advanced support for building and testing clusters of NixOS machines locally using virtual machines. However, at some point we need to dive in and deploy a server if we're going to use NixOS for real.

In this chapter we'll deploy our TODO app to our first "production" server in AWS meaning that you *will* need to create an AWS account[1] to follow along.

AWS prices and offers will vary so this book can't provide any strong guarantees about what this would cost you. However, at the time of this writing the examples in this chapter would fit well within the current AWS free tier, which is 750 hours of a `t3.micro` instance.

Even if there were no free tier, the cost of a `t3.micro` instance is currently ≈1¢ / hour or ≈ $7.50 / month if you never shut it off (and you can shut it off when you're not using it). So at most this chapter should only cost you a few cents from start to finish.

Throughout this book I'll take care to minimize your expenditures by showing how you to develop and test locally as much as possible.

In the spirit of Infrastructure as Code, we'll be using Terraform to declaratively provision AWS resources, but before doing so we need to generate AWS access keys for programmatic access.

Configuring your access keys

To generate your access keys, follow the instructions in Accessing AWS using AWS credentials[2].

In particular, take care to **not** generate access keys for your account's root user. Instead, use the Identity and Access Management (IAM) service to create a separate user with "Admin" privileges and generate access keys for that user. The difference between a root user and an admin user is that an admin user's privileges can later be limited or revoked, but the root user's privileges can never be limited nor revoked.

The above AWS documentation also recommends generating temporary access credentials instead of long-term credentials. However, setting this up properly and ergonomically requires setting up the IAM Identity Center which is only permitted for AWS accounts that have set up an AWS Organization. That is way outside of the scope of this book so instead you should just generate long-term credentials for a non-root admin account.

If you generated the access credentials correctly you should have:

[1] https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/
[2] https://docs.aws.amazon.com/general/latest/gr/aws-sec-cred-types.html

- an access key ID (i.e. `AWS_ACCESS_KEY_ID`)
- a secret access key (i.e. `AWS_SECRET_ACCESS_KEY`)

If you haven't already, configure your development environment to use these tokens by running:

```
$ nix run 'github:NixOS/nixpkgs/23.11#awscli' -- configure --profile nixos-in-production
AWS Access Key ID [None]: …
AWS Secret Access Key [None]: …
Default region name [None]: …
Default output format [None]:
```

If you're not sure what region to use, pick the one closest to you based on the list of AWS service endpoints[3].

A minimal Terraform specification

Now run the following command to bootstrap our first Terraform project:

```
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10#terraform'
```

… which will generate the following files:

- `module.nix` + `www/index.html`

  The NixOS configuration for our TODO list web application, except adapted to run on AWS instead of inside of a `qemu` VM.

- `flake.nix`

  A Nix flake that wraps our NixOS configuration so that we can refer to the configuration using a flake URI.

- `main.tf`

  The Terraform specification for deploying our NixOS configuration to AWS.

- `backend/main.tf`

  This Terraform configuration provisions an S3 bucket for use with Terraform's S3 backend[4]. We won't use this until the very end of this chapter, though, so we'll ignore it for now.

Deploying our configuration

To deploy the Terraform configuration, run the following commands:

[3] https://docs.aws.amazon.com/general/latest/gr/rande.html
[4] https://developer.hashicorp.com/terraform/language/settings/backends/s3

```
$ nix shell 'github:NixOS/nixpkgs/23.05#terraform'
$ terraform init
$ terraform apply
```

… and when prompted to enter the `region`, use the same AWS region you specified earlier when running `aws configure`:

```
var.region
  Enter a value: …
```

After that, `terraform` will display the execution plan and ask you to confirm the plan:

```
module.ami.data.external.ami: Reading...
module.ami.data.external.ami: Read complete after 1s [id=-]

Terraform used the selected providers to generate the following execution plan.
Resource actions are indicated with the following symbols:
  + create
 <= read (data resources)

Terraform will perform the following actions:

…

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

… and if you confirm then `terraform` will deploy that execution plan:

```
…

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:

public_dns = "ec2-….compute.amazonaws.com"
```

The final output will include the URL for your server. If you open that URL in your browser you will see the exact same TODO server as before, except now running on AWS instead of inside of a `qemu` virtual machine. If this is your first time deploying something to AWS then congratulations!

Cleaning up

Once you verify that everything works you can destroy all deployed resources by running:

```
$ terraform apply -destroy
```

`terraform` will prompt you for the same information (i.e. the same region) and also prompt for confirmation just like before:

```
var.region
  Enter a value: …

…

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

… and once you confirm then `terraform` will destroy everything:

```
…

Apply complete! Resources: 0 added, 0 changed, 7 destroyed.
```

Now you can read the rest of this chapter in peace knowing that you are no longer being billed for this example.

Terraform walkthrough

The key file in our Terraform project is `main.tf` containing the Terraform logic for how to deploy our TODO list application.

You can think of a Terraform module as being sort of like a function with side effects, meaning:

- The function has inputs
  Terraform calls these input variables[5].
- The function has outputs
  Terraform calls these output values[6].
- The function does things other than producing output values
  For example, the function might provision a resource[7].
- You can invoke another terraform module like a function call
  In other words, one Terraform module can call another Terraform module by supplying the child module[8] with appropriate function arguments.

Our starting `main.tf` file provides examples of all of the above concepts.

[5] https://developer.hashicorp.com/terraform/language/values/variables
[6] https://developer.hashicorp.com/terraform/language/values/outputs
[7] https://developer.hashicorp.com/terraform/language/resources/syntax
[8] https://developer.hashicorp.com/terraform/language/modules/syntax#calling-a-child-module

Input variables

For example, the beginning of the module declares one input variable:

```
variable "region" {
  type = string
  nullable = false
}
```

… which is analogous to a Nix function like this one that takes the following attribute set as an input:

```
{ region }:
  …
```

When you run `terraform apply` you will be automatically prompted to supply all input variables:

```
$ terraform apply
var.region
  Enter a value: …
```

… but you can also provide the same values on the command line, too, if you don't want to supply them interactively:

```
$ terraform apply -var region=…
```

… and if you really want to make the whole command non-interactive you can also add the `-auto-approve` flag:

```
$ terraform apply -var region=… -auto-approve
```

… so that you don't have to manually confirm the deployment by entering "yes".

Output variables

The end of the Terraform module declares one output value:

```
output "public_dns" {
  value = aws_instance.todo.public_dns
}
```

… which would be like our function returning an attribute set with one attribute:

```
{ region }:

let
  …

in
  { output = aws_instance.todo.public_dns; }
```

… and when the deploy completes Terraform will render all output values:

```
Outputs:

public_dns = "ec2-….compute.amazonaws.com"
```

Resources

In between the input variables and the output values the Terraform module declares several resources. For now, we'll highlight the resource that provisions the EC2 instance:

```
resource "aws_security_group" "todo" {
  …
}

resource "tls_private_key" "nixos-in-production" {
  …
}

resource "local_sensitive_file" "ssh_private_key" {
}

resource "local_file" "ssh_public_key" {
}

resource "aws_key_pair" "nixos-in-production" {
  …
}

resource "aws_instance" "todo" {
  ami = module.ami.ami
  instance_type = "t3.micro"
  security_groups = [ aws_security_group.todo.name ]
  key_name = aws_key_pair.nixos-in-production.key_name

  root_block_device {
    volume_size = 7
  }
}

resource "null_resource" "wait" {
  …
}
```

… and you can think of resources sort of like `let` bindings that provision infrastructure as a side effect:

```
{ region }:

let
  …;

  aws_security_group.todo = aws_security_group { … };

  tls_private_key.nixos-in-production = tls_private_key { … };

  local_sensitive_file.ssh_private_key = local_sensitive_file { … };

  local_file.ssh_public_key = local_file { … };

  aws_key_pair.nixos-in-production = aws_key_pair { … };

  aws_instance.todo = aws_instance {
    ami = module.ami.ami;
    instance_type = "t3.micro";
    security_groups = [ aws_security_group.todo.name ];
    key_name = aws_key_pair.nixos-in-production.key_name;
    root_block_device.volume_size = 7;
  }

  null_resource.wait = null_resource { … };

in
  { output = aws_instance.todo.public_dns; }
```

Our Terraform deployment declares six resources, the first of which declares a security group (basically like a firewall):

```
resource "aws_security_group" "todo" {
  # The "nixos" Terraform module requires SSH access to the machine to deploy
  # our desired NixOS configuration.
  ingress {
      from_port = 22
      to_port = 22
      protocol = "tcp"
      cidr_blocks = [ "0.0.0.0/0" ]
  }

  # We will be building our NixOS configuration on the target machine, so we
  # permit all outbound connections so that the build can download any missing
  # dependencies.
  egress {
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = [ "0.0.0.0/0" ]
  }

  # We need to open port 80 so that we can view our TODO list web page.
  ingress {
    from_port = 80
    to_port = 80
    protocol = "tcp"
```

```
    cidr_blocks = [ "0.0.0.0/0" ]
  }
}
```

The next four resources generate an SSH key pair that we'll use to manage the machine:

```
# Generate an SSH key pair as strings stored in Terraform state
resource "tls_private_key" "nixos-in-production" {
  algorithm = "ED25519"
}

# Synchronize the SSH private key to a local file that the "nixos" module can
# use
resource "local_sensitive_file" "ssh_private_key" {
    filename = "${path.module}/id_ed25519"
    content = tls_private_key.nixos-in-production.private_key_openssh
}

resource "local_file" "ssh_public_key" {
    filename = "${path.module}/id_ed25519.pub"
    content = tls_private_key.nixos-in-production.public_key_openssh
}

# Mirror the SSH public key to EC2 so that we can later install the public key
# as an authorized key for our server
resource "aws_key_pair" "nixos-in-production" {
  public_key = tls_private_key.nixos-in-production.public_key_openssh
}
```

The `tls_private_key` resource[9] is currently not secure because the deployment state is stored locally unencrypted. We will fix this later on in this chapter by storing the deployment state using Terraform's S3 backend[10].

After that we get to the actual server:

```
resource "aws_instance" "todo" {
  # This will be an AMI for a stock NixOS server which we'll get to below.
  ami = module.ami.ami

  # We could use a smaller instance size, but at the time of this writing the
  # t3.micro instance type is available for 750 hours under the AWS free tier.
  instance_type = "t3.micro"

  # Install the security groups we defined earlier
  security_groups = [ aws_security_group.todo.name ]

  # Install our SSH public key as an authorized key
  key_name = aws_key_pair.nixos-in-production.key_name

  # Request a bit more space because we will be building on the machine
  root_block_device {
```

[9] https://registry.terraform.io/providers/hashicorp/tls/latest/docs/resources/private_key
[10] https://developer.hashicorp.com/terraform/language/settings/backends/s3

```
    volume_size = 7
  }

  # We will use this in a future chapter to bootstrap other secrets
  user_data = <<-EOF
    #!/bin/sh
    (umask 377; echo '${tls_private_key.nixos-in-production.private_key_openssh}' > /var/lib/id_ed2551\
9)
    EOF
}
```

Finally, we declare a resource whose sole purpose is to wait until the EC2 instance is reachable via SSH so that the "nixos" module knows how long to wait before deploying the NixOS configuration:

```
# This ensures that the instance is reachable via `ssh` before we deploy NixOS
resource "null_resource" "wait" {
  provisioner "remote-exec" {
    connection {
      host = aws_instance.todo.public_dns
      private_key = tls_private_key.nixos-in-production.private_key_openssh
    }

    inline = [ ":" ]  # Do nothing; we're just testing SSH connectivity
  }
}
```

Modules

Our Terraform module also invokes two other Terraform modules (which I'll refer to as "child modules") and we'll highlight here the module that deploys the NixOS configuration:

```
module "ami" {
  …;
}

module "nixos" {
  source = "github.com/Gabriella439/terraform-nixos-ng//nixos?ref=af1a0af57287851f957be2b524fcdc008a21\
d9ae"
  host = "root@${aws_instance.todo.public_ip}"
  flake = ".#default"
  arguments = [ "--build-host", "root@${aws_instance.todo.public_ip}" ]
  ssh_options = "-o StrictHostKeyChecking=accept-new"
  depends_on = [ null_resource.wait ]
}
```

You can liken child modules to Nix function calls for imported functions:

```
{ region }:

let
  module.ami = …;

  module.nixos =
    let
      source = fetchFromGitHub {
        owner = "Gabriella439";
        repo = "terraform-nixos-ng";
        rev = "af1a0af57287851f957be2b524fcdc008a21d9ae";
        hash = …;
      };

    in
      import source {
        host = "root@${aws_instance.todo_public_ip}";
        flake = ".#default";
        arguments = [ "--build-host" "root@${aws_instance.todo.public_ip}" ];
        ssh_options = "-o StrictHostKeyChecking=accept-new";
        depends_on = [ null_resource.wait ];
      };


  aws_security_group.todo = aws_security_group { … };

  tls_private_key.nixos-in-production = tls_private_key { … };

  local_sensitive_file.ssh_private_key = local_sensitive_file { … };

  local_file.ssh_public_key = local_file { … };

  aws_key_pair.nixos-in-production = aws_key_pair { … };

  aws_instance.todo = aws_instance { … };

  null_resource.wait = null_resource { … };

in
  { output = aws_instance.todo.public_dns; }
```

The first child module selects the correct NixOS AMI to use:

```
module "ami" {
  source = "github.com/Gabriella439/terraform-nixos-ng//ami?ref=af1a0af57287851f957be2b524fcdc008a21d9\
ae"
  release = "23.05"
  region = var.region
  system = "x86_64-linux"
}
```

… and the second child module deploys our NixOS configuration to our EC2 instance:

```
module "nixos" {
  source = "github.com/Gabriella439/terraform-nixos-ng//nixos?ref=af1a0af57287851f957be2b524fcdc008a21\
d9ae"

  host = "root@${aws_instance.todo.public_ip}"

  # Get the NixOS Configuration from the nixosConfigurations.default attribute
  # of our flake.nix file
  flake = ".#default"

  # Build our NixOS configuration on the same machine that we're deploying to
  arguments = [ "--build-host", "root@${aws_instance.todo.public_ip}" ]

  ssh_options = "-o StrictHostKeyChecking=accept-new -i ${local_sensitive_file.ssh_private_key.filenam\
e}"

  depends_on = [ null_resource.wait ]
}
```

In this example we build our NixOS configuration on our web server so that this example can be deployed without any supporting infrastructure. However, you typically will want to build the NixOS configuration on a dedicated builder rather than building on the target server for two reasons:

- You can deploy to a smaller/leaner server
  Building a NixOS system typically requires more disk space, RAM, and network bandwidth than running/deploying the same system. Centralizing that build work onto a larger special-purpose builder helps keep other machines lightweight.

- You can lock down permissions on the target server
  For example, if we didn't have to build on our web server then we wouldn't need to permit outbound connections on that server since it would no longer need to fetch build-time dependencies.

A future chapter will cover how to provision a dedicated builder for this purpose.

S3 Backend

The above Terraform deployment doesn't properly protect the key pair used to `ssh` into and manage the NixOS machine. By default the private key of the key pair is stored in a world-readable `terraform.tfstate` file. However, even if we were to restrict that file's permissions we wouldn't be able to easily share our Terraform deployment with colleagues. In particular, we wouldn't want to add the `terraform.tfstate` file to version control in a shared repository since it contains sensitive secrets.

The good news is that we can fix both of those problems by setting up an S3 backend[11] for Terraform which allows the secret to be securely stored in an S3 bucket that can be shared by multiple people managing the same Terraform deployment.

[11] https://developer.hashicorp.com/terraform/language/settings/backends/s3

The template for this chapter's Terraform configuration already comes with a `backend/` subdirectory containing a Terraform specification that provisions a suitable S3 bucket and DynamoDB table for an S3 backend. All you have to do is run:

```
$ cd ./backend
$ terraform apply
var.region
  Enter a value: …

…

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

Just make sure to use the same region as our original Terraform deployment when prompted.

When the deployment succeeds it will output the name of the randomly-generated S3 bucket, which will look something like this (with a timestamp in place of the `X`s):

```
…
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.

Outputs:

bucket = "nixos-in-productionXXXXXXXXXXXXXXXXXXXXXXXXXX"
```

Then switch back to the original Terraform deployment in the parent directory:

```
$ cd ..
```

… and modify that deployment's `main.tf` to reference the newly-created `bucket` like this:

```
terraform {
  required_version = ">= 1.3.0"

  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.56"
    }
  }

  # This is the new bit you want to add
  backend "s3" {
    bucket = "nixos-in-productionXXXXXXXXXXXXXXXXXXXXXXXXX"
    key = "terraform"
    region = "…"  # Use the same region here that you used before
    dynamodb_table = "terraform-state"
    profile = "nixos-in-production"
  }
}
```

These last few manual steps to update the S3 backend are a bit gross but this is primarily to work around limitations in Terraform. In particular, Terraform doesn't provide a way for our main deployment to automatically reference the S3 backend we created. Terraform specifically prohibits `backend` stanzas from referencing variables so all of the backend options have to be hard-coded values.

Then you can upgrade your existing deployment to reference the S3 backend you just provisioned by re-running `terraform init` with the `-migrate-state` option:

```
$ terraform init -migrate-state
Initializing modules...

Initializing the backend...
Do you want to copy existing state to the new backend?
  Pre-existing state was found while migrating the previous "local" backend to the
  newly configured "s3" backend. No existing state was found in the newly
  configured "s3" backend. Do you want to copy this state to the new "s3"
  backend? Enter "yes" to copy and "no" to start with an empty state.

  Enter a value: yes
```

... and once that's done you can verify that nothing broke by running `terraform apply` again, which should report that no new changes need to be deployed:

```
$ terraform apply
var.region
  Enter a value: …

…

No changes. Your infrastructure matches the configuration.
```

The difference is that now the terraform state is securely stored in an S3 bucket instead of on your filesystem so you'd now be able to store your Terraform configuration in version control and let other developers manage the same deployment. There's just one last thing you need to do, which is to remove the `terraform.tfstate.backup` file, which contains the old (pre-S3-backend) Terraform state, including the secrets:

```
$ rm terraform.tfstate.backup
```

You can also remove the `terraform.tfstate` file, too, since it's empty and no longer used:

```
$ rm terraform.tfstate
```

Future Terraform examples in this book won't include the S3 backend code to keep them shorter, but feel free to reuse the same S3 bucket created in this chapter to upgrade any of those examples with an S3 backend. However, if you do that then keep in mind that you need to use a different key for storing Terraform's state if you want to keep those examples separate.

In other words, when adding the S3 backend to the `terraform` clause, specify a different key for each separate deployment:

```
terraform {
  …

  backend "s3" {
    …
    key = "…"  # This is what needs to be unique per deployment
    …
  }
}
```

This key is used by Terraform to record where to store the deployment's state within the S3 bucket, so if you use the same key for two different deployments they will will interfere with one another.

Version control

Once you create the S3 backend you can safely store your Terraform configuration in version control. Specifically, these are the files that you want to store in version control:

- `flake.nix`, `module.nix`, `www/`

  These provide the configuration for the machine we're deploying so we obviously need to keep these.

- `flake.lock`

  It's also worth keeping this in version control even though it's not strictly necessary. The lock file slightly improves the determinism of the deployment, although the flake included in the template is already fairly deterministic even without the lockfile because it references a specific tag from Nixpkgs.

- `main.tf`, `backend/main.tf`

  We definitely want to keep the Terraform deployments for our main deployment and the S3 backend.

- `terraform.tfstate`

  You don't need to keep this in version control (it's an empty file

Just as important, you do **NOT** want to keep the `id_ed25519` file in version control (since this contains the private key). In fact, the provided template includes a `.gitignore` file to prevent you from accidentally adding the private key to version control.

Terraform will recreate this private key file locally for each developer that manages the deployment. For example, if another developer were to apply the deployment for the first time, they would see this diff:

```
Terraform will perform the following actions:

  # local_sensitive_file.ssh_private_key will be created
  + resource "local_sensitive_file" "ssh_private_key" {
      + content              = (sensitive value)
      + directory_permission = "0700"
      + file_permission      = "0700"
      + filename             = "./id_ed25519"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

... indicating that Terraform will download the private key from the S3 backend and create a secure local copy in order to `ssh` into the machine.

However, it's completely fine to add the public key to version control if you want.

9. Continuous Integration and Deployment

This chapter will cover how to use both continuous integration (a.k.a. "CI") and continuous deployment (a.k.a. "CD"), beginning with a brief explanation of what those terms mean.

Both continuous integration and continuous deployment emphasize *continuously* incorporating code changes into your product. *Continuous integration*[1] emphasizes continuously incorporating code changes into the trunk development branch of your version control repository whereas *Continuous deployment*[2] emphasizes continuously incorporating code changes into production.

Continuous Integration

Colloquially developers often understand "continuous integration" to mean automatically testing pull requests before they are merged into version control. However, continuous integration is about more than just automated tests and is really about ensuring that changes are regularly being integrated into the trunk development branch (and automated tests help with that). For example, if you have long-lived development branches that's not really adhering to the spirit of continuous integration, even if you do put them through automated tests.

I'm mentioning this because this book will offer opinionated guidance that works better if you're not supporting long-lived development branches. You can still modify this book's guidance to your tastes, but in my experience sticking to only one long-lived development branch (the trunk branch) will simplify your architecture, reduce communication overhead between developers, and improve your release frequency. In fact, this specific flavor of continuous integration has a name: trunk-based development[3].

That said, this chapter will focus on how to set up automated tests since that's the part of continuous integration that's NixOS-specific.

The CI solution I endorse for most Nix/NixOS projects is garnix[4] because with garnix you don't have to manage secrets and you don't have to set up your own build servers or cache. In other words, garnix is architecturally simple to install and manage.

However, garnix only works with GitHub (it's a GitHub app[5]) so if you are using a different version control platform then you'll have to use a different CI solution. The two major alternatives that people tend to use are:

- Hydra

[1] https://en.wikipedia.org/wiki/Continuous_integration
[2] https://en.wikipedia.org/wiki/Continuous_deployment
[3] https://trunkbaseddevelopment.com/
[4] https://garnix.io/
[5] https://github.com/apps/garnix-ci

Like garnix, Hydra is a Nix-aware continuous integration service but unlike garnix, Hydra is self-hosted[6]. Hydra's biggest strength is deep visibility into builds in progress and ease of scaling out build capacity but Hydra's biggest weakness is that it is high maintenance to support, and difficult to debug when things go wrong.

- A non-Nix-aware CI service that just runs `nix build`

  For example, you could have a GitHub action or Jenkins job that runs some `nix build` command on all pull requests. The advantage of this approach is that it is very simple but the downside is that the efficiency is worse when you need to scale out your build capacity.

The reason why non-Nix-aware CI solutions tend to do worse at scale is because they typically have their own notion of available builders/agents/slots which does not map cleanly onto Nix's notion of available builders. This means that you have to waste time tuning the two sets of builders to avoid wasting available build capacity and even after tuning you'll probably end up with wasted build capacity.

The reason Hydra doesn't have this problem is because Hydra uses Nix's native notion of build capacity (remote builds[7]) configured via the `nix.distributedBuilds` and `nix.buildMachines` NixOS options. That means that you can easily scale out build capacity by adding more builders[8].

This chapter will focus on setting up garnix since it's dramatically simpler than the alternatives.

Also, we're going to try to minimize the amount of logic that needs to live outside of Nix. For example:

- checks that you'd normally run in a non-Nix-aware job can be incorporated into a Nix build's check phase

- non-Nix-aware jobs that deploy ("push") a machine configuration can be replaced by machines periodically fetching and installing ("pulling") their configuration

  This is covered later in this chapter's Continuous Deployment section. For more discussion on the tradeoffs of "push" vs "pull" continuous deployment, see: Push vs. Pull in GitOps: Is There Really a Difference?[9].

garnix

garnix already has official documentation[10] for how to set it up, but I'll mention here the relevant bits for setting up CI for our own production deployment. We're going to configure this CI to

[6] The reason why is that writing a (meaningful) test for our TODO list example would require executing JavaScript using something like Selenium, which will significantly increase the size of the example integration test. `postgrest`, on the other hand, is easier to test from the command line.

[7] https://nixos.org/manual/nix/stable/advanced-topics/distributed-builds.html

[8] Okay, there is actually a limit to how much you can scale out build capacity. After a certain point you will begin to hit bottlenecks in instantiating derivations at scale, but even in this scenario Hydra still has a higher performance ceiling than the the non-Nix-aware alternatives.

[9] https://thenewstack.io/push-vs-pull-in-gitops-is-there-really-a-difference/

[10] https://garnix.io/docs

build and cache the machine that we deploy to production, which will also ensure that we don't merge any changes that break the build.

This exercise will build upon the same example as the previous chapter on Terraform, and you can reuse the example from that chapter or you can generate the example if you haven't already by running these commands:

```
$ mkdir todo-app
$ cd todo-app
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10#terraform'
```

... or you can skip straight to the final result (minus the secrets file) by running:

```
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10#continuous-deployment'
```

garnix requires the use of Nix flakes in order to support efficient evaluation caching[11] and the good news is that we can already build our NixOS system without any changes to our flake, but it might not be obvious how at first glance.

If we wanted to build our system, we would run:

```
$ nix build '.#nixosConfigurations.default.config.system.build.toplevel'
```

If your development system is Apple Silicon (i.e. `aarch64-darwin`) you will not yet be able to build that locally. Even if you use the Linux builder from the Setting up your development environment chapter that won't work because the builder's architecture (`aarch64-linux`) won't match the architecture of the system we're deploying (`x86_64-linux`).

In a future chapter we'll cover how to set up an `x86_64-linux` remote builders that you can use for testing builds like these, but until then you will have to settle for just *evaluating* the system configuration instead of *building* it, like this:

```
$ nix eval '.#nixosConfigurations.default.config.system.build.toplevel'
```

This will catch errors at the level of Nix evaluation (e.g. Nix syntax errors or bad function calls) but this won't catch errors related to actually building the system.

In fact, if all you care about is evaluation, you can simplify that latter command even further by just running:

```
$ nix flake check
```

... which does exactly the same thing (among other things). However, typically we want to build *and cache* our NixOS system, which is why we don't just run `nix flake check` in CI.

Attributes

Let's step through this attribute path:

[11] https://www.tweag.io/blog/2020-06-25-eval-cache/

```
nixosConfigurations.default.config.system.build.toplevel
```

... to see where each attribute comes from because that will come in handy if you choose to integrate Nix into a non-Nix-aware CI solution:

- `nixosConfigurations`

  This is one of the "standard" output attributes[12] for flakes where we store NixOS configurations that we want to build. This attribute name is not just a convention; NixOS configurations stored under this attribute enjoy special support from Nix tools. Specifically:

  - `nixos-rebuild` only supports systems underneath the `nixosConfigurations` output

    We use `nixos-rebuild` indirectly as part of our Terraform deployment because the `terraform-nixos-ng` module uses `nixos-rebuild` under the hood[13]. In our project's `main.tf` file the `module.nixos.flake` option is set to `.#default` which `nixos-rebuild` replaces with `.#nixosConfigurations.default`[14].
  - `nix flake check` automatically checks the `nixosConfigurations` flake output

    ... as noted in the previous aside.
  - garnix's default configuration[15] builds all of the `nixosConfigurations` flake outputs

    ... so if we stick to using that output then we don't need to specify a non-default configuration.

- `default`

  We can store more than one NixOS system configuration underneath the `nixosConfigurations` output. We can give each system any attribute name, but typically if you only have one system to build then the convention is to name that the `default` system. The command-line tooling does not give this `default` attribute any special treatment, though.

- `config`

  The output of the `nixpkgs.lib.nixosSystem` system is similar in structure to a NixOS module, which means that it has attributes like `config` and `options`. The `config` attribute lets you access the finalized values for all NixOS options.

- `system.build.toplevel`

  This is a NixOS option that stores the final derivation for building our NixOS system. For more details, see the NixOS option definitions chapter.

[12] https://nixos.wiki/wiki/Flakes#Output_schema
[13] https://www.haskellforall.com/2023/01/terraform-nixos-ng-modern-terraform.html
[14] I have no idea why `nixos-rebuild` works this way and doesn't accept the full attribute path including the `nixosConfigurations` attribute.
[15] https://garnix.io/docs/yaml_config

You can use `nix repl` to explore flake outputs by running:

```
$ nix repl .#
Welcome to Nix 2.18.1. Type :? for help.

Loading installable 'path:/Users/gabriella/proj/todo-app#'...
Added 1 variables.
nix-repl>
```

... and then you can use autocompletion within the REPL to see what's available. For example:

```
nix-repl> nixosConfigurations.<TAB>
nix-repl> nixosConfigurations.default.<TAB>
nixosConfigurations.default._module       nixosConfigurations.default.extendModules  nixosConfigurati\
ons.default.options       nixosConfigurations.default.type
nixosConfigurations.default.config        nixosConfigurations.default.extraArgs      nixosConfigurati\
ons.default.pkgs
```

Enabling garnix CI

The only thing you'll need in order to enable garnix CI for your project is to:

- turn your local directory into a `git` repository

  ```
  $ git init
  $ git add --all
  $ git commit --message 'Initial commit'
  ```

- host your `git` repository on GitHub

  ... by following these instructions[16].

  Also, make this repository private, because later in this chapter we're going to practice fetching a NixOS configuration from a private repository.

- enable garnix on that repository

  ... by visiting the garnix GitHub app[17], installing it, and enabling it on your newly-created repository.

... and you're mostly done! You won't see any activity, though, until you create your first pull request so you can verify that garnix is working by creating a pull request to make the following change to the `flake.nix` file:

[16] https://docs.github.com/en/migrations/importing-source-code/using-the-command-line-to-import-source-code/adding-locally-hosted-code-to-github

[17] https://github.com/apps/garnix-ci

```
--- a/flake.nix
+++ b/flake.nix
@@ -7,4 +7,12 @@
       modules = [ ./module.nix ];
     };
   };
+
+  nixConfig = {
+    extra-substituters = [ "https://cache.garnix.io" ];
+
+    extra-trusted-public-keys = [
+      "cache.garnix.io:CTFPyKSLcx5RMJKfLo5EEPUObbA78b0YQ2DTCJXqr9g="
+    ];
+  };
 }
```

Once you create that pull request, garnix will report two status checks on that pull request:

- "Evaluate flake.nix"

  This verifies that your `flake.nix` file is well-formed and also serves as a fallback status check you can use if your flake has no outputs (for whatever reason).
- "nixosConfig default"

  This status check verifies that our `nixosConfigurations.default` output builds correctly and caches it.

The next thing you need to do is to enable branch protection settings so that those new status checks gate merge into your `main` branch. To do that, visit the "Settings → Branches → Add branch protection rule" page of your repository (which you can also find at `https://github.com/${OWNER}/${REPOSITORY}/settings/branch_protection_rules/new` where `${OWNER}` is your username and `${REPOSITORY}` is the repository name you chose). Then select the following options:

- Branch name pattern: `main`
- Require status checks to pass before merging
  - Status checks that are required:
    * "Evaluate flake.nix"
    * "nixosConfig default"

Since this is a tutorial project we won't enable any other branch protection settings, but for a real project you would probably want to enable some other settings (like requiring at least one approval from another contributor).

Once you've made those changes, merge the pull request you just created. You've just set up automated tests for your repository!

Using garnix's cache

One of the reasons I endorse garnix for most Nix/NixOS projects is that they also take care of hosting a cache on your behalf. Anything built by your CI is made available from their cache.

The pull request you just merged configures your flake to automatically make use of garnix's cache. If you were using an `x86_64-linux` machine, you could test this by running:

```
$ nix build .#nixosConfigurations.default.config.system.build.toplevel
```

The above command does not work on other systems (e.g. `aarch64-darwin`), even though the complete build product is cached! You would think that Nix would just download ("substitute") the complete build product even if there were a system mismatch, but this does not work because Nix refuses to substitute certain derivations[18]. The above `nix build` command will only work if your local system is `x86_64-linux` or you have a remote builder configured to build `x86_64-linux` build products because Nix will insist on building some of the build products instead of substituting them.

It is possible to work around this by adding the following two `nix.conf` options (and restarting your Nix daemon):

```
extra-substituters = https://cache.garnix.io
extra-trusted-public-keys = cache.garnix.io:CTFPyKSLcx5RMJKfLo5EEPUObbA78b0YQ2DTCJXqr9g=
```

... and then you can run:

```
$ FLAKE='.#nixosConfigurations.default.config.system.build.toplevel'
$ nix-store --realise "$(nix eval --raw "${FLAKE}.outPath")"
```

... but that's not as great of a user experience.

Continuous Deployment

We're going to be using "pull-based" continuous deployment to manage our server, meaning that our server will periodically fetch the desired NixOS configuration and install that configuration.

NixOS already has a set of `system.autoUpgrade`[19] options for managing a server in this way. What we want is to be able to set at least the following two NixOS options:

```
system.autoUpgrade = {
  enable = true;

  flake = "github:${username}/${repository}#default";
};
```

However, there's a catch: this means that our machine will need to be able to access our private `git` repository. Normally the way you'd do this is to specify an access token in `nix.conf` like this:

[18] https://github.com/NixOS/nix/issues/8677
[19] https://search.nixos.org/options?query=system.autoUpgrade

```
access-tokens = github.com=${SECRET_ACCESS_TOKEN}
```

… but don't want to save this access token in version control in order to deploy our machine.

Another way we could fetch from a private git repository is to specify a flake like this:

```
flake = "git+ssh://git@github.com/${username}/${repository}#default";
```

… which would allow us to access the private git repository using an SSH key pair instead of using a GitHub access token (assuming that we configure GitHub to grant that key pair access to the repository). Either way, we'd need some sort of secret to be present on the machine in order to access the private repository.

So we need some way to securely transmit or install secrets (such as personal access tokens) to our machine, but how do we bootstrap all of that?

For the examples in this book, we're going to reuse the SSH key pair generated for our Terraform deployment as a "primary key pair". In other words, we're going to install the private key of our SSH key pair on the target machine and then use the corresponding public key (which we can freely share) to encrypt other secrets (which only our target machine can decrypt, using the private key). In fact, our original Terraform template already does this:

```
resource "aws_instance" "todo" {
  …

  # We will use this in a future chapter to bootstrap other secrets
  user_data = <<-EOF
    #!/bin/sh
    (umask 377; echo '${tls_private_key.nixos-in-production.private_key_openssh}' > /var/lib/id_ed2551\
9)
    EOF
}
```

We are now living in the future and we're going to use the SSH private key mirrored to `/var/lib/id_ed25519` as the primary key that bootstraps all our other secrets.

This implies that our "admin" (the person deploying our machine using Terraform) will be able to transitively access all other secrets that the machine depends on because the admin has access to the same private key. However, there's no real good way to prevent this sort of privilege escalation, because the admin has `root` access to the machine and good luck granting the machine access to a secret without granting the `root` user access to the same secret.[20]

`sops-nix`

We're going to use `sops-nix`[21] (a NixOS wrapper around `sops`[22]) to securely distribute all other secrets we need to our server. The way that `sops-nix` works is:

[20] There are some ways you can still prevent privilege escalation by the `root` user, like multi-factor authentication, but do you really want some other person to have to multi-factor authenticate every time one of your machines polls GitHub for the latest configuration? It's much simpler to just trust your admin.

[21] https://github.com/Mic92/sops-nix

[22] https://github.com/getsops/sops

- You generate an asymmetric key pair

  In other words, you generate a public key (used to encrypt secrets) and a matching private key (used to decrypt secrets encrypted by the public key). This can be a GPG[23], SSH[24], or age[25] key pair.

  This is our "primary key pair".

- You install the private key on the target machine without using `sops`

  There is no free lunch here. You can't bootstrap secrets on the target machine out of thin air. The private key of our primary key pair needs to already be present on the machine so that the machine can decrypt secrets encrypted by the public key.

- You use `sops` to add new encrypted secrets

  `sops` is a command-line tool that makes it easy to securely edit a secrets file. You can create a new secrets file using just the public key, but if you want to edit an existing secrets file (e.g. to add or remove secrets) you will require both the public key and private key. In practice this means that only an admin can add new secrets.

- You use `sops-nix` decrypt those secrets

  `sops-nix` is a NixOS module that uses the private key (which we installed out-of-band) to decrypt the secrets file and make those secrets available as plain text files on the machine. By default, those text files are only readable by the `root` user but you can customize the file ownership and permissions to your liking.

You might wonder what is the point of using `sops` to distribute secrets to the machine if it requires already having a secret present on the machine (the primary key).

The purpose of `sops` is to provide a uniform interface for adding, versioning, and installing all other secrets. Otherwise, you'd have to roll your own system for doing this once you realize that it's kind of a pain to implement a secret distribution mechanism for each new secret you need.

So `sops` doesn't completely solve the problem of secrets management (you still have to figure out how to install the primary key), but it does make it easier to manage all the other secrets.

age keys

To use the `sops` command-line tool we'll need to convert our SSH primary key pair into an age key pair. This step is performed by the admin who has access to both the SSH public key and the SSH private key and requires the `ssh-to-age` command-line tool, which you can obtain like this:

[23] https://www.gnupg.org/gph/en/manual/c14.html
[24] https://man.openbsd.org/ssh-keygen
[25] https://github.com/FiloSottile/age#readme

```
$ nix shell 'github:NixOS/nixpkgs/23.11#ssh-to-age'
```

The public key of our age key pair will be stored in a `.sops.yaml` configuration file which lives in version control. To create the age public key, run:

```
$ cat > .sops.yaml <<EOF
creation_rules:
- age: '$(ssh-to-age -i id_ed25519.pub)'
EOF
```

The private key of our age key pair is stored locally by the admin so that they can edit secrets. To store the age private key, run:

```
$ # On Linux
$ KEY_FILE=~/.config/sops/age/keys.txt

$ # On MacOS
$ KEY_FILE=~/Library/'Application Support'/sops/age/keys.txt

$ mkdir -p "$(dirname "$KEY_FILE")"
$ (umask 077; ssh-to-age -private-key -i id_ed25519 -o "$KEY_FILE")
```

Now you're ready to start reading and writing secrets!

Creating the secret

Create a fine-grained personal access token[26] by visiting the New fine-grained personal access token[27] page and entering the following settings:

- **Token name**: "todo app - continuous deployment"
- **Expiration**: 30 days
- **Description**: leave empty
- **Resource owner**: your personal GitHub account
- **Repository access**: choose "Only select repositories" and select your `todo-app` repository
- **Repository permissions**: set the "Contents" permission to "Read-only"
- **Account permissions**: do not enable any account permissions

... and then click the "Generate token" button. Keep this page with the generated token open for just a second.

Fetch the `sops` command-line tool by running:

[26] https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens#creating-a-fine-grained-personal-access-token

[27] https://github.com/settings/personal-access-tokens/new

```
$ nix shell 'github:NixOS/nixpkgs/23.11#sops'
```

... and then create a new secrets file by running:

```
$ sops secrets.yaml
```

That will open a new file in your editor with the following contents:

```
hello: Welcome to SOPS! Edit this file as you please!
example_key: example_value
# Example comment
example_array:
    - example_value1
    - example_value2
example_number: 1234.56789
example_booleans:
    - true
    - false
```

We're going to do what file says and edit the file how we please. Delete the entire file and replace it with:

```
github-access-token: 'extra-access-tokens = github.com=github_pat_…'
```

... replacing `github_pat_…` with the personal access token you just generated.

Now if you save, exit, and view the file (without `sops`) you will see something like this:

```
github-access-token: …
sops:
    kms: []
    gcp_kms: []
    azure_kv: []
    hc_vault: []
    age:
        - recipient: …
          enc: |
            -----BEGIN AGE ENCRYPTED FILE-----
            …
            -----END AGE ENCRYPTED FILE-----
    lastmodified: "…"
    mac: ENC[AES256_GCM,data:…,iv:…,tag:…,type:str]
    pgp: []
    unencrypted_suffix: _unencrypted
    version: 3.7.3
```

... and since you're the admin you can still decrypt the file using `sops` to view the secret:

```
$ sops secrets.yaml
```

Anyone who doesn't have access to the private key would instead get an error message like this:

```
Failed to get the data key required to decrypt the SOPS file.

Group 0: FAILED
  …: FAILED
    - | failed to open file: open
      | …/sops/age/keys.txt: no such file or directory

Recovery failed because no master key was able to decrypt the file. In
order for SOPS to recover the file, at least one key has to be successful,
but none were.
```

Installing the secret

Now we can distribute the GitHub personal access token stored inside of `secrets.yaml`. First we'll need to import the `sops-nix` NixOS module by modifying our `flake.nix` file like this:

```
{ inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/23.11";

    sops-nix.url = "github:Mic92/sops-nix/bd695cc4d0a5e1bead703cc1bec5fa3094820a81";
  };

  outputs = { nixpkgs, sops-nix, ... }: {
    nixosConfigurations.default = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";

      modules = [ ./module.nix sops-nix.nixosModules.sops ];
    };
  };

  nixConfig = {
    extra-substituters = [ "https://cache.garnix.io" ];

    extra-trusted-public-keys = [
      "cache.garnix.io:CTFPyKSLcx5RMJKfLo5EEPUObbA78b0YQ2DTCJXqr9g="
    ];
  };
}
```

Then we'll enable continuous deployment by adding the following lines to `module.nix`:

```
{ modulesPath, ... }:

{ …

  sops = {
    defaultSopsFile = ./secrets.yaml;

    age.sshKeyPaths = [ "/var/lib/id_ed25519" ];

    secrets.github-access-token = { };
  };

  nix.extraOptions = "!include /run/secrets/github-access-token";
}
```

That will:

- Install the secret on our server

  That uses the `/var/lib/id_ed25519` private key that Terraform installed to decrypt the `secrets.yaml` file.

- Incorporate the secret into the Nix configuration

  This grants our Nix daemon access to our private `git` repository containing our NixOS configuration.

The `autoUpgrade` service

Finally, to enable continuous deployment, we will enable `system.autoUpgrade`:

```
{ modulesPath, ... }:

{ …

  system.autoUpgrade = {
    enable = true;

    # Replace ${username}/${repository} with your repository's address
    flake = "github:${username}/${repository}#default";

    # Poll the `main` branch for changes once a minute
    dates = "minutely";

    # You need this if you poll more than once an hour
    flags = [ "--option" "tarball-ttl" "0" ];
  };
```

but don't deploy this configuration just yet! First, add all the changes we've made so far to version control:

```
$ nix flake update
$ git fetch origin main
$ git checkout -b continuous-deployment FETCH_HEAD
$ git add .sops.yaml secrets.yaml flake.nix flake.lock module.nix
$ git commit --message 'Enable continuous deployment'
$ git push --set-upstream origin continuous-deployment
```

Then create a pull request from those changes and merge the pull request once it passes CI.

Once you've merged your changes, checkout the `main` branch of your repository:

```
$ git checkout main
$ git pull --ff-only
```

… and deploy those changes using `terraform`, the same way we did in the previous chapter:

```
$ terraform apply
```

Once you've applied those changes your machine will begin automatically pulling its configuration from the `main` branch of your repository.

Testing continuous deployment

There are some diagnostic checks you can do to verify that everything is working correctly, but first you need to log into the machine using:

```
$ ssh -i id_ed25519 "root@${ADDRESS}"
```

… replacing `${ADDRESS}` with the `public_dns` output of the Terraform deployment.

Then you can check that the secret was correctly picked up by the Nix daemon by running:

```
[root@…:~]# nix --extra-experimental-features 'nix-command flakes' show-config | grep access-tokens
access-tokens = github.com=github_pat_…
```

… and you can also monitor the upgrade service by running:

```
[root@…:~]# journalctl --output cat --unit nixos-upgrade --follow
```

… and if things are working then every minute you should see the service output something like this:

```
Starting NixOS Upgrade...
warning: ignoring untrusted flake configuration setting 'extra-substituters'
warning: ignoring untrusted flake configuration setting 'extra-trusted-public-keys'
building the system configuration...
warning: ignoring untrusted flake configuration setting 'extra-substituters'
warning: ignoring untrusted flake configuration setting 'extra-trusted-public-keys'
updating GRUB 2 menu...
switching to system configuration /nix/store/…-nixos-system-unnamed-…
activating the configuration...
setting up /etc...
sops-install-secrets: Imported /etc/ssh/ssh_host_rsa_key as GPG key with fingerprint …
sops-install-secrets: Imported /var/lib/id_ed25519 as age key with fingerprint …
reloading user units for root...
Successful su for root by root
pam_unix(su:session): session opened for user root(uid=0) by (uid=0)
pam_unix(su:session): session closed for user root
setting up tmpfiles
finished switching to system configuration /nix/store/…-nixos-system-unnamed-…
nixos-upgrade.service: Deactivated successfully.
Finished NixOS Upgrade.
nixos-upgrade.service: Consumed … CPU time, no IP traffic.
```

Now let's test that our continuous deployment is working by making a small change so that we don't need to add the `--extra-experimental-features` option to every `nix` command.

Add the following option to `module.nix`:

```
  nix.settings.extra-experimental-features = [ "nix-command" "flakes" ];
```

... and create and merge a pull request for that change. Once your change is merged the machine will automatically pick up the change on the next minute boundary and you can verify the change worked by running:

```
[root@…:~]# nix show-config
```

... which will now work without the `--extra-experimental-features` option.

Congratulations! You've now set up a continuous deployment system!

10. Flakes

This book has leaned pretty heavily on Nix's support for "flakes", but I've glossed over the details of how flakes work despite how much we've already been using them. In this chapter I'll give a more patient breakdown of flakes.

Most of what this chapter will cover is information that you can already find from other resources, like the NixOS Wiki page on Flakes[1] or by running `nix help flake`. However, I'll still try to explain flakes in my own words.

Motivation

You can think of flakes as a package manager for Nix. In other words, if we use Nix to build and distribute packages written in other programming languages (e.g. Go, Haskell, Python), then flakes are how we "build" and distribute Nix packages.

Here are some example Nix packages that are distributed as flakes:

- `nixpkgs`

  This is the most widely used Nix package of all. Nixpkgs is a giant `git` repository hosted on GitHub[2] containing the vast majority of software packaged for Nix. Nixpkgs also includes several important helper functions that you'll need for building even the simplest of packages, so you pretty much can't get anything done in Nix without using Nixpkgs to some degree.

- `flake-utils`

  This is a Nix package containing useful utilities for creating flakes and is itself distributed as a flake.

- `sops-nix`

  This is a flake we just used in the previous chapter to securely distribute secrets.

All three of the above packages provide reusable Nix code that we might want to incorporate into downstream Nix projects. Flakes provide a way for us to depend on and integrate Nix packages like these into our own projects.

Flakes, step-by-step

We can build a better intuition for how flakes work by starting from the simplest possible flake you can write:

[1] https://nixos.wiki/wiki/Flakes#Flake_schema
[2] https://github.com/NixOS/nixpkgs

```
# ./flake.nix

{ outputs = { nixpkgs, ... }: {
    # Replace *BOTH* occurrences of `x86_64-linux` with your machine's system
    #
    # You can query your current system by running:
    #
    # $ nix eval --impure --expr 'builtins.currentSystem'
    packages.x86_64-linux.default = nixpkgs.legacyPackages.x86_64-linux.hello;
  };
}
```

You can then build and run that flake with this command:

```
$ nix run
Hello, world!
```

Flake references

We could have also run the above command as:

```
$ nix run .
```

... or like this:

```
$ nix run '.#default'
```

... or in this fully qualified form:

```
$ # Replace x86_64-linux with your machine's system
$ nix run '.#packages.x86_64-linux.default'
```

In the above command `.#packages.x86_64-linux.default` uniquely identifies a "flake reference" and an attribute path, which are separated by a `#` character:

- The first half (the flake reference) specifies where a flake is located

  In the above example the flake reference is "`.`" (a shorthand for our current directory).
- The second half (the attribute path) specifies which output attribute to use

  In the above example, it is `packages.x86_64-linux.default` and `nix run` uses that output attribute path to select which executable to run.

Usually we don't want to write out something long like `.#packages.x86_64-linux.default` when we use flakes, so flake-enabled Nix commands provide a few convenient shorthands that we can use to shorten the command.

First off, many Nix commands will automatically expand part of the flake attribute path on your behalf. For example, if you run:

```
$ nix run '.#default'
```

... then `nix run` will attempt to expand `.#default` to a fully qualified attribute path of `.#apps."${system}".default` and if the flake does not have that output attribute path then `nix run` will fall back to a fully qualified attribute path of `.#packages."${system}".default`.

Different Nix commands will expand the attribute path differently. For example:

- `nix build` and `nix eval` expand `foo` to `packages."${system}".foo`
- `nix run` expands `foo` to `apps."${system}".foo`

  ... and falls back to `packages."${system}".foo` if that's missing
- `nix develop` expands `foo` to `devShells."${system}".foo`

  ... and falls back to `packages."${system}".foo` if that's missing
- `nixos-rebuild`'s `--flake` option expands `foo` to `nixosConfigurations.foo`
- `nix repl` will not expand attribute paths at all

In each case the `"${system}"` in the expanded attribute path corresponds to your current system, which you can query using this command:

```
$ nix eval --impure --expr 'builtins.currentSystem'
```

You can even omit the attribute path, in which case it will default to an attribute path of `default`. For example, if you run:

```
$ nix run .
```

... then `nix run` will expand `.` to `.#default` (which will in turn expand to `.#packages.${system}.default` for our flake).

Furthermore, you can omit the flake reference, which will default to `.`, so if you run:

```
$ nix run
```

... then that expands to a flake reference of `.` (which will then continue to expand according to the above rules).

Flake URIs

So far these examples have only used a flake reference of `.` (the current directory), but in this book we'll be using several types of flake references, including:

- paths

  These can be relative paths (like `.` or `./utils` or `../bar`), home-anchored paths (like `~/workspace`), or absolute paths (like `/etc/nixos`). In all three cases the path must be a directory containing a `flake.nix` file.

- GitHub URIs

  These take the form `github:${OWNER}/${REPOSITORY}` or `github:${OWNER}/${REPOSITORY}/${REFERENCE}` (where `${REFERENCE}` can be a branch, tag, or revision). Nix will take care of cloning the repository for you in a cached and temporary directory and (by default) look for a `flake.nix` file within the root directory of the repository.

- indirect URIs

  An indirect URI is one that refers to an entry in Nix's "flake registry". If you run `nix registry list` you'll see a list of all your currently configured indirect URIs.

Flake inputs

Normally the way flakes work is that you specify both inputs and outputs, like this:

```
{ inputs = {
    foo.url = "${FLAKE_REFERENCE}";
    bar.url = "${FLAKE_REFERENCE}";
  };

  outputs = { self, foo, bar }: {
    baz = …;
    qux = …;
  };
}
```

In the above example, `foo` and `bar` would be the flake inputs while `baz` and `qux` would be the flake outputs. In other words, the sub-attributes nested underneath the `inputs` attribute are the flake inputs and the attributes generated by the `outputs` function are the flake outputs.

Notice how the `outputs` function takes input arguments which share the same name as the flake inputs because the flakes machinery resolves each input and then passes each resolved input as a function argument of the same name to the `outputs` function.

To illustrate this, if you were to build the `baz` output of the above flake using:

```
$ nix build .#baz
```

… then that would sort of be like building this Nix pseudocode:

```
let
  flake = import ./flake.nix;

  foo = resolveFlakeURI flake.inputs.foo;

  bar = resolveFlakeURI flake.inputs.baz;

  self = flake.outputs { inherit self foo bar; };

in
  self.baz
```

... where `resolveFlakeURI` would be sort of like a function from an input's flake reference to the Nix code packaged by that flake reference.

If you're curious how flake inputs and outputs are actually resolved, it's actually implemented as a function in Nix, which you can find here in the `NixOS/nix` repository[3].

However, if you were paying close attention you might have noticed that our original example flake does not have any `inputs`:

```
{ outputs = { nixpkgs, ... }: {
    packages.x86_64-linux.default = nixpkgs.legacyPackages.x86_64-linux.hello;
  };
}
```

... and the `outputs` function references a `nixpkgs` input which we never specified. The reason this works is because flakes automatically convert missing inputs to "indirect" URIs that are resolved using Nix's flake registry. In other words, it's as if we had written:

```
{ inputs = {
    nixpkgs.url = "nixpkgs";  # Example of an indirect flake reference
  };

  outputs = { nixpkgs, ... }: {
    packages.x86_64-linux.default = nixpkgs.legacyPackages.x86_64-linux.hello;
  };
}
```

An indirect flake reference is resolved by doing a lookup in the flake registry, which you can query yourself like this:

```
$ nix registry list | grep nixpkgs
global flake:nixpkgs github:NixOS/nixpkgs/nixpkgs-unstable
```

... so we could have also written:

[3] https://github.com/NixOS/nix/blob/2.18.1/src/libexpr/flake/call-flake.nix

```
{ inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
  };

  outputs = { nixpkgs, ... }: {
    packages.x86_64-linux.default = nixpkgs.legacyPackages.x86_64-linux.hello;
  };
}
```

... which would have produced the same result: both flake references will attempt to fetch the `nixpkgs-unstable` branch of the `nixpkgs` repository to resolve the `nixpkgs` flake input.

Throughout the rest of this chapter (and book) I'm going to try to make flake references as pure as possible, meaning:

- no indirect flake references
  In other words, instead of `nixpkgs` I'll use something like `github:NixOS/nixpkgs/23.11`.

- all GitHub flake references will include a tag
  In other words, I won't use a flake reference like `github:NixOS/nixpkgs`.

Neither of these precautions are strictly necessary when using flakes because flakes lock their dependencies using a `flake.lock` file which you can (and should) store in version control. However, it's still a good idea to take these precautions anyway even if you include the `flake.lock` file alongside your `flake.nix` file. The more reproducible your flake references, the better you document how to regenerate or update your lock file.

Suppose we were to use our own local `git` checkout of `nixpkgs` instead of a remote `nixpkgs` branch: we'd have to change the `nixpkgs` input to our flake to reference the path to our local repository (since paths are valid flake references), like this:

```
{ inputs = {
    nixpkgs.url = ~/repository/nixpkgs;
  };

  outputs = { nixpkgs, ... }: {
    packages.x86_64-linux.default = nixpkgs.legacyPackages.x86_64-linux.hello;
  };
}
```

... and then we also need to build the flake using the `--impure` flag:

```
$ nix build --impure
```

Without the flag we'd get this error message:

```
error: the path '~/repository/nixpkgs' can not be resolved in pure mode
```

Notice that we're using flake references in two separate ways:

- on the command line
  e.g. `nix build "${FLAKE_REFERENCE}"`
- when specifying flake inputs
  e.g. `inputs.foo.url = "${FLAKE_REFERENCE}";`

However, they differ in two important ways:

- command line flake references never require the `--impure` flag
  In other words, `nix build ~/proj/nixpkgs#foo` is fine, but if you specify `~/proj/nixpkgs` as a flake input then you have to add the `--impure` flag on the command line.

- flake inputs can be specified as attribute sets instead of strings

To elaborate on the latter point, instead of specifying a `nixpkgs` input like this:

```
{ inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
  };

  …
}
```

We could instead specify the same input like this:

```
{ inputs = {
    type = "github";

    owner = "NixOS";

    repo = "nixpkgs";
  };

  …
}
```

Throughout this book I'll consistently use the non-structured (string) representation for flake references to keep things simple.

Flake outputs

We haven't yet covered what we actually *get* when we resolve a flake input. For example, what Nix expression does a flake reference like `github:NixOS/nixpkgs/23.11` resolve to?

The answer is that a flake reference will resolve to the output attributes of the corresponding flake. For a flake like `github:NixOS/nixpkgs/23.11` that means that Nix will:

- clone the `nixpkgs` repository[4]
- check out the `23.11` tag[5] of that repository
- look for a `flake.nix`[6] file in the top-level directory of that repository
- resolve inputs for that flake

  For this particular flake, there are no inputs to resolve.
- look for an `outputs` attribute[7], which will be a function
- computed the fixed-point of that function
- return that fixed-point as the result

  In this case the result would be an attribute set[8] containing five attributes: `lib`, `checks`, `htmlDocs`, `legacyPackages`, and `nixosModules`.

In other words, it would behave like this (non-flake-enabled) Nix code:

```
# nixpkgs.nix

let
  pkgs = import <nixpkgs> { };

  nixpkgs = pkgs.fetchFromGitHub {
    owner = "NixOS";
    repo = "nixpkgs";
    rev = "23.11";
    hash = "sha256-btHN1czJ6rzteeCuE/PNrdssqYD2nIA4w48miQAFloM=";
  };

  flake = import "${nixpkgs}/flake.nix";

  self = flake.outputs { inherit self; };

in
  self
```

... except that with flakes we wouldn't have to figure out what hash to use since that would be transparently managed for us by the flake machinery.

If you were to load the above file into the REPL:

```
$ nix repl --file nixpkgs.nix
```

... you would get the exact same result as if you had loaded the equivalent flake into the REPL:

```
$ nix repl github:NixOS/nixpkgs/23.11
```

In both cases the REPL would now have the `lib`, `checks`, `htmlDocs`, `legacyPackages`, and `nixosModules` attributes in scope since those are the attributes returned by the `outputs` function:

[4] https://github.com/NixOS/nixpkgs
[5] https://github.com/NixOS/nixpkgs/tree/23.11
[6] https://github.com/NixOS/nixpkgs/blob/23.11/flake.nix
[7] https://github.com/NixOS/nixpkgs/blob/23.11/flake.nix#L6
[8] https://github.com/NixOS/nixpkgs/blob/23.11/flake.nix#L16-L74

```
nix-repl> legacyPackages.x86_64-linux.hello
«derivation /nix/store/zjh5kllay6a2ws4w46267i97lrnyya9l-hello-2.12.1.drv»
```

This `legacyPackages.x86_64-linux.hello` attribute path is the same attribute path that our original flake output uses:

```
{ …

  outputs = { nixpkgs, ... }: {
    packages.x86_64-linux.default = nixpkgs.legacyPackages.x86_64-linux.hello;
                                    # ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  };
}
```

There's actually one more thing you can do with a flake, which is to access the original path to the flake. The following flake shows an example of this feature in action:

```
{ inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/23.11";

    flake-utils.url = "github:numtide/flake-utils/v1.0.0";
  };

  outputs = { flake-utils, nixpkgs, ... }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        config = { };

        overlay = self: super: {
          hello = super.hello.overrideAttrs (_: { doCheck = false; });
        };

        overlays = [ overlay ];

        pkgs = import nixpkgs { inherit system config overlays; };
      in
        { packages.default = pkgs.hello; }
    );
}
```

This flake customizes Nixpkgs using an overlay instead of using the "stock" package set, but in order to create a new package set from that overlay we have to `import` the original source directory for Nixpkgs. In the above example, that happens here when we `import` `nixpkgs`:

```
        pkgs = import nixpkgs { inherit system config overlays; };
```

Normally the `import` keyword expects either a file or (in this case) a directory containing a `default.nix` file, but here `nixpkgs` is neither: it's an attribute set containing all of the `nixpkgs` flake's outputs. However, the `import` keyword can still treat `nixpkgs` like a path because it also comes with an `outPath` attribute, so we could have also written:

```
      pkgs = import nixpkgs.outPath { inherit system config overlays; };
```

All flake inputs come with this `outPath` attribute, meaning that you can use a flake input anywhere Nix expects a path and the flake input will be replaced with the path to the directory containing the `flake.nix` file.

Platforms

All of the above examples hard-coded a single system (`x86_64-linux`), but usually you want to support building a package for multiple systems. People typically use the `flake-utils` flake for this purpose, which you can use like this;

```
{ inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/23.11";

    flake-utils.url = "github:numtide/flake-utils/v1.0.0";
  };

  outputs = { flake-utils, nixpkgs, ... }:
    flake-utils.lib.eachDefaultSystem (system: {
      packages.default = nixpkgs.legacyPackages."${system}".hello;
    });
}
```

... and that is essentially the same thing as if we had written:

```
{ inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/23.11";

    flake-utils.url = "github:numtide/flake-utils/v1.0.0";
  };

  outputs = { nixpkgs, ... }: {
    packages.x86_64-linux.default = nixpkgs.legacyPackages.x86_64-linux.hello;
    packages.aarch64-linux.default = nixpkgs.legacyPackages.aarch64-linux.hello;
    packages.x86_64-darwin.default = nixpkgs.legacyPackages.x86_64-darwin.hello;
    packages.aarch64-darwin.default = nixpkgs.legacyPackages.aarch64-darwin.hello;
  };
}
```

We'll be using `flake-utils` throughout the rest of this chapter and you'll see almost all flakes use this, too.

Flake-related commands

The Nix command-line interface provides several commands that are flake-aware, and for the purpose of this chapter we'll focus on the following commands:

- `nix build`

- `nix run`
- `nix shell`
- `nix develop`
- `nix flake check`
- `nix flake init`
- `nix eval`
- `nix repl`
- `nixos-rebuild`

We'll be using the following flake as the running example for our commands:

```
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10?dir=templates/cowsay'
```

... which will have this structure:

```
{ inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/23.11";

    flake-utils.url = "github:numtide/flake-utils/v1.0.0";
  };

  outputs = { flake-utils, nixpkgs, self }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = nixpkgs.legacyPackages."${system}";

      in
        { packages.default = pkgs.cowsay;

          apps = …;

          checks = …;

          devShells = …;
        }) // {
          templates.default = …;
        };
}
```

One of the things you might notice is that the some of the output attributes are nested inside of the call to `eachDefaultSystem`. Specifically, the `packages`, `apps`, `checks`, and `devShells` outputs:

```
  …

  flake-utils.lib.eachDefaultSystem (system:
    let
      pkgs = nixpkgs.legacyPackages."${system}";

    in
      { packages.default = pkgs.cowsay;

        apps = …;

        checks = …;

        devShells = …;
      }) // …
```

For each of these outputs we want to generate system-specific build products, which is why they go inside the call to `eachDefaultSystem`. However, some flake outputs (like templates) are not system-specific, so they would go outside of the call to `eachDefaultSystem`, like this:

```
  flake-utils.lib.eachDefaultSystem (system:
    let
      pkgs = nixpkgs.legacyPackages."${system}";

    in
      { …
      }) // {
        templates.default = …;
      };
```

You can always consult the flake output schema[9] if you're not sure which outputs are system-specific and which ones are not. For example, the sample schema will show something like this:

```
self, ... }@inputs:
{ checks."<system>"."<name>" = derivation;
  …
  packages."<system>"."<name>" = derivation;
  …
  apps."<system>"."<name>" = {
    type = "app";
    program = "<store-path>";
  };
  …
  devShells."<system>"."<name>" = derivation;
  …
  templates."<name>" = {
    path = "<store-path>";
    description = "template description goes here?";
  };
}
```

… and `."<system>".` component of the first four attribute paths indicates that these outputs are system-specific, whereas the `templates."<name>"` attribute path has no system-specific path component.

[9] https://nixos.wiki/wiki/Flakes#Output_schema

The same sample schema also explains which outputs are used by which Nix commands, but we're about to cover that anyway:

`nix build`

The `nix build` command builds output attributes underneath the `packages` attribute path.

For example, if we run:

```
$ nix build
```

... that will build the `.#packages."${system}".default` output, which in our flake is just a synonym for the `cowsay` package from Nixpkgs:

```
…
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = nixpkgs.legacyPackages."${system}";

      in
        { packages.default = pkgs.cowsay;

          …
        })
```

That build will produce this result:

```
$ tree ./result
./result
├── bin
|   ├── cowsay
|   └── cowthink -> cowsay
└── share
    └── cowsay
        ├── cows
        |   ├── DragonAndCow.pm
        |   ├── Example.pm
        |   ├── Frogs.pm
        |   ├── …
        |   ├── vader-koala.cow
        |   ├── vader.cow
        |   └── www.cow
        └── site-cows

5 directories, 58 files
```

... which we can run like this:

```
$ ./result/bin/cowsay howdy
 _______
< howdy >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

So far this isn't very interesting because we can already build the `cowsay` executable from Nixpkgs directly like this:

```
$ nix build 'github:NixOS/nixpkgs/23.11#cowsay'  # Exact same result
```

In fact, we don't even need to create a local copy of the `cowsay` template flake. We could also have run the flake directly from the GitHub repository where it's hosted:

```
$ nix build 'github:Gabriella439/nixos-in-production/0.10?dir=templates/cowsay'
```

This works because flakes support GitHub URIs, so all of the flake operations in this chapter work directly on the GitHub repository without having to clone or template the repository locally. However, for simplicity all of the following examples will still assume you templated the flake locally.

`nix run`

Typically we won't run the command by building it and then running it. Instead, we'll more commonly use `nix run` to do both in one go:

```
$ nix run . -- howdy  # The "." is necessary if the command takes arguments
 _______
< howdy >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

... and if we wanted to we could run `cowsay` directly from Nixpkgs, too:

```
$ nix run 'github:NixOS/nixpkgs/23.11#cowsay' -- howdy  # Exact same result
```

By default, `nix run` will expand out `.` to `.#apps.${system}.default`, falling back to `.#packages.${system}.default` if that's not present. Our flake happens to provide the former attribute path:

```
flake-utils.lib.eachDefaultSystem (system:
  let
    pkgs = nixpkgs.legacyPackages."${system}";

  in
    { packages.default = pkgs.cowsay;

      apps = {
        default = self.apps."${system}".cowsay;

        cowsay = {
          type = "app";

          program = "${self.packages."${system}".default}/bin/cowsay";
        };

        cowthink = {
          type = "app";

          program = "${self.packages."${system}".default}/bin/cowthink";
        };
      };

      …
    }
```

This time our flake has three available apps (`default`, `cowsay`, and `cowthink`) and the `default` app is just a synonym for the `cowsay` app. Each app has to have:

- a field named `type` whose value must be `"app"`

  Nix flakes don't support other types of apps, yet.
- a string containing the desired program to run

  This string cannot contain any command-line options. It can only be a path to an executable.

Notice that flake outputs can reference other flake outputs (via the `self` flake input). All flakes get this `self` flake input for free. We could have also used the `rec` language keyword instead, like this:

```
      apps = rec {
        default = cowsay;

        cowsay = {
          type = "app";

          program = "${self.packages."${system}".default}/bin/cowsay";
        };

        cowthink = {
          type = "app";

          program = "${self.packages."${system}".default}/bin/cowthink";
        };
      };
```

… which would define the `default` attribute to match the `cowsay` attribute within the same record. This works in small cases, but doesn't scale well to more complicated cases; you should prefer using the `self` input to access other attribute paths.

You can use output attributes other than the `default` one by specifying their attribute paths. For example, if we want to use the `cowthink` program then we can run:

```
$ nix run .#cowthink -- howdy
 _______
< howdy >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Apparently, the `cowthink` program produces the exact same result as the `cowsay` program.

Since the `cowthink` app is indistinguishable from the `cowsay` app, let's replace it with a more interesting `kittysay` app that automatically adds the `-f hellokitty` flag. However, we can't do something like this:

```
        apps = {
          …

          kittysay = {
            type = "app";

            program = "${self.packages."${system}".default}/bin/cowsay -f hellokitty";
          };
        };
```

If you try to do that you'll get an error like this one:

```
$ nix run .#kittysay -- howdy
error: unable to execute '/nix/store/…-cowsay-3.7.0/bin/cowsay -f hellokitty': No such file or directo\
ry
```

… because Nix expects the `program` attribute to be an executable path, not including any command-line arguments. If you want to wrap an executable with arguments then you need to do something like this:

```
{ packages = {
    default = pkgs.cowsay;

    kittysay = pkgs.writeScriptBin "kittysay"
      ''
      ${self.packages."${system}".default}/bin/cowsay -f hellokitty "$@";
      '';
  };

  apps = {
    …

    kittysay = {
      type = "app";

      program = "${self.packages."${system}".kittysay}/bin/kittysay";
    };
  };
```

Here we define a `kittysay` package (which wraps `cowsay` with the desired command-line option) and a matching `kittysay` app.

Note that if the name of the app is the same as the default executable for a package then we can just omit the app entirely. In the above `kittysay` example, we could delete the `kittysay` app and the example would still work because Nix will fall back to running `${self.packages.${system}.kittysay}/bin/kittysay`. You can use `nix run --help` to see the full set of rules for how Nix decides what attribute to use and what path to execute.

`nix shell`

If you plan on running the same command (interactively) over and over then you probably don't want to have to type `nix run` before every use of the command. Not only is this less ergonomic but it's also slower since the flake has to be re-evaluated every time you run the command.

The `nix shell` comes in handy for use cases like this where it will take the flake outputs that you specify and add them to your executable search path (e.g. your `$PATH` in `bash`) for ease of repeated use. We can add our `cowsay` to our search path in this way:

```
$ nix shell
$ cowsay howdy
 _______
< howdy >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

`nix shell` creates a temporary subshell providing the desired commands and you can exit from this subshell by entering an `exit` command or typing `Ctrl-D`.

`nix shell` can be useful for pulling in local executables from your flake, but it's even more useful for pulling in executables temporarily from Nixpkgs (since Nixpkgs provides a large array of useful programs). For example, if you wanted to temporarily add `vim` and `tree` to your shell you could run:

```
$ nix shell 'github:NixOS/nixpkgs/23.11#'{vim,tree}
```

Note that the `{vim,tree}` syntax in the previous command is a Bash/Zsh feature. Both shells expand the previous command to:

```
$ nix shell 'github:NixOS/nixpkgs/23.11#vim' 'github:NixOS/nixpkgs/23.11#tree'
```

This feature is a convenient way to avoid having to type out the `github:NixOS/nixpkgs/23.11` flake reference twice when adding multiple programs to your shell environment.

`nix develop`

You can even create a reusable development shell if you find yourself repeatedly using the same temporary executables. Our sample flake illustrates this by providing two shells:

```
devShells = {
  default = self.packages."${system}".default;

  with-dev-tools = pkgs.mkShell {
    inputsFrom = [ self.packages."${system}".default ];

    packages = [
      pkgs.vim
      pkgs.tree
    ];
  };
};
```

You can enter the `default` shell by running:

```
$ nix develop
```

... which will expand out to `.#devShells.${system}.default`, falling back to `.#packages.${system}.default` if the former attribute path is not present. This will give us a development environment for building the `cowsay` executable. You can use these `devShells` to create reusable development environments (for yourself or others) so that you don't have to build up large `nix shell` commands.

You can exit from this development shell by either entering `exit` or typing `Ctrl-D`.

You might wonder what's the difference between `nix develop` and `nix shell`. The difference between the two is that:

- `nix shell` adds the specified programs to your executable search `PATH`
- `nix develop` adds the *development dependencies of the specified programs* to your executable search `PATH`

As a concrete example, when you run:

```
$ nix shell nixpkgs#vim
```

That adds `vim` to your executable search path. In contrast, if you run:

```
$ nix develop nixpkgs#vim
```

That provides a *development environment* necessary to build the `vim` executable (like a C compiler or the `ncurses` package), but does not provide `vim` itself.

In our sample flake, we used the `mkShell` utility:

```
        with-dev-tools = pkgs.mkShell {
          inputsFrom = [ self.packages."${system}".default ];

          packages = [
            pkgs.vim
            pkgs.tree
          ];
        };
```

... which is a convenient way to create a synthetic development environment (appropriate for use with `nix develop`).

The two most common arguments to `mkShell` are:

- `inputsFrom`

  `mkShell` inherits the development dependencies of any package that you list here. Since `self.packages."${system}".default` is just our `cowsay` package then that means that all development dependencies of the `cowsay` package also become development dependencies of `mkShell`.

- `packages`

  All packages listed in the `packages` argument to `mkShell` become development dependencies of `mkShell`. So if we add `vim` and `tree` here then those will be present on the executable search path of our synthetic environment if we call `nix develop` on our `mkShell`.

This means that the `with-dev-tools` shell is essentially the same as the `default` shell, except also extended with the `vim` and `tree` packages added to the executable search path.

`nix flake check`

You can add tests to a flake that the `nix flake check` command will run. These tests go under the `checks` output attribute and our sample flake provides a simple functional test for the `cowsay` package:

```
    checks = {
      default = self.packages."${system}".default;

      diff = pkgs.runCommand "test-cowsay" { }
        ''
        diff <(${self.apps."${system}".default.program} howdy) - <<'EOF'
         _______
        < howdy >
         -------
                \   ^__^
                 \  (oo)\_______
                    (__)\       )\/\
                        ||----w |
                        ||     ||
        EOF

        touch $out
        '';
    };
```

The `default` check is a synonym for our `cowsay` package and just runs `cowsay`'s (non-existent) test suite. The `diff` check is a functional test that compares some sample `cowsay` output against a golden result.

We can run both checks (the `default` check and the `diff` check) using:

```
$ nix flake check
```

The `nix flake check` command also performs other hygiene checks on the given flake and you can learn more about the full set of checks by running:

```
$ nix flake check --help
```

`nix flake init`

You can template a project using `nix flake init`, which we've already used a few times throughout this book (including this chapter). Our `cowsay` flake contains the following `templates` output:

```
  }) // {
    templates.default = {
      path = ./.;

      description = "A tutorial flake wrapping the cowsay package";
    };
  };
```

... so earlier when we ran:

```
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10?dir=templates/cowsay'
```

... that copied the directory pointed to by the `templates.default.path` flake output to our local directory.

Note that this flake output is not system-specific, which is why it's not nested inside the call to `eachDefaultSystems` in our flake. This is because there's nothing system-dependent about templating some text files.

`nix eval`

`nix eval` is another command we've already used a few times throughout this book to query information about our flakes without building anything. For example, if we wanted to query the version of our `cowsay` package, we could run:

```
$ nix eval '.#default.version'
"3.7.0"
```

This is because flake outputs are "just" attribute sets and Nix derivations are also "just" attribute sets, so we can dig into useful information about them by accessing the appropriate attribute path.

However, you might not necessarily know what attribute paths are even available to query, which brings us to the next extremely useful Nix command:

`nix repl`

You can use the `nix repl` command to easily interactively explore what attributes are available using REPL auto-completion. For example, if you run:

```
$ nix repl .  # This requires the `repl-flake` experimental feature
```

That will load all of the flake outputs (e.g. `packages`, `apps`, `checks`, `devShells`, `templates`) as identifiers of the same name into the REPL. Then you can use tab completion to dig further into their available fields:

```
nix-repl> pac<TAB>

nix-repl> packages.<TAB>
packages.x86_64-linux  packages.i686-linux      packages.x86_64-linux
packages.aarch64-linux   packages.x86_64-darwin

nix-repl> packages.x86_64-linux.<TAB>

nix-repl> packages.x86_64-linux.default.<TAB>
packages.x86_64-linux.default.__darwinAllowLocalNetworking
packages.x86_64-linux.default.__ignoreNulls
packages.x86_64-linux.default.__impureHostDeps
…
```

```
packages.x86_64-linux.default.updateScript
packages.x86_64-linux.default.userHook
packages.x86_64-linux.default.version

nix-repl> packages.x86_64-linux.default.version
"3.7.0"
```

Remember that `flake-utils` (specifically the `eachDefaultSystem` function) adds system attributes underneath each of these top-level attributes, so even though we don't explicitly specify system attribute in our `flake.nix` file they're still going to be there when we navigate the flake outputs in the REPL. That's why we have to specify `packages.x86_64-linux.default.version` in the REPL instead of just `packages.default.version`.

However, you can skip having to specify the system if you specify the package you want to load into the REPL. For example, if we load the `default` package output like this:

```
$ nix repl .#default
```

... that's the same as loading `.#packages.${system}.default` into the REPL, meaning that all of the `default` package's attributes are now top-level identifiers in the REPL, including `version`:

```
nix-repl> version
"3.7.0"
```

The `nix repl` command comes in handy if you want to explore Nix code interactively, whereas the `nix eval` command comes more in handy for non-interactive use (e.g. scripting).

`nixos-rebuild`

Last, but not least, the `nixos-rebuild` command also accepts flake outputs that specify the system to deploy. We already saw an example of this in the Deploying to AWS using Terraform chapter where we specified our system to deploy as `.#default` which expands out to the `.#nixosConfigurations.default` flake output.

Similar to the `templates` flake outputs, `nixosConfigurations` are not system-specific. There's no particular good reason why this is the case since NixOS can (in theory) be built for multiple systems (e.g. `x86_64-linux` or `aarch64-linux`), but in practice most NixOS systems are only defined for a single architecture.

Our sample `cowsay` flake doesn't provide any `nixosConfigurations` output, but the flake from our Terraform chapter has an example `nixosConfigurations` output.

11. Integration testing

In Our first web server we covered how to test a server manually and in this chapter we'll go over how to use NixOS to automate this testing process. Specifically, we're going to be authoring a NixOS test, which you can think of as the NixOS-native way of doing integration testing[1]. However, in this chapter we're going to depart from our running "TODO list" example[2] and instead use NixOS tests to automate the Getting Started instructions from an open source tutorial.

Specifically, we're going to be testing the PostgREST tutorial[3]. You can read through the tutorial if you want, but the relevant bits are:

- Install Postgres (the database)

  Nixpkgs will handle this for us because Nixpkgs has already packaged `postgres` (both as a package and as a NixOS module for configuring `postgres`).

- Install PostgREST (the database-to-REST conversion service)

  Nixpkgs also (partially) handles this for us because all Haskell packages (including `postgrest`) are already packaged for us, but (at the time of this writing) there isn't yet a NixOS module for `postgres`. For this particular example, though, that's fine because then the integration testing code will match the tutorial even more closely.

- Set up the database

  ... by running these commands:

```
create schema api;

create table api.todos (
  id serial primary key,
  done boolean not null default false,
  task text not null,
  due timestamptz
);

insert into api.todos (task) values
  ('finish tutorial 0'), ('pat self on back');

create role web_anon nologin;

grant usage on schema api to web_anon;
grant select on api.todos to web_anon;

create role authenticator noinherit login password 'mysecretpassword';
grant web_anon to authenticator;
```

[1] https://en.wikipedia.org/wiki/Integration_testing

[2] The reason why is that writing a (meaningful) test for our TODO list example would require executing JavaScript using something like Selenium, which will significantly increase the size of the example integration test. `postgrest`, on the other hand, is easier to test from the command line.

[3] https://postgrest.org/en/v12/tutorials/tut0.html

- Launch PostgREST

  ... with this configuration file:

```
db-uri = "postgres://authenticator:mysecretpassword@localhost:5432/postgres"
db-schemas = "api"
db-anon-role = "web_anon"
```

- Check that the API works

  ... by verifying that this command:

```
$ curl http://localhost:3000/todos
```

```
[
  {
    "id": 1,
    "done": false,
    "task": "finish tutorial 0",
    "due": null
  },
  {
    "id": 2,
    "done": false,
    "task": "pat self on back",
    "due": null
  }
]
```

NixOS test

You can clone the equivalent NixOS test by running:

```
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10#integration-test'
```

One of the included files is `setup.sql` file which includes the database commands from the tutorial verbatim:

```
create schema api;

create table api.todos (
  id serial primary key,
  done boolean not null default false,
  task text not null,
  due timestamptz
);

insert into api.todos (task) values
  ('finish tutorial 0'), ('pat self on back');

create role web_anon nologin;

grant usage on schema api to web_anon;
```

```
grant select on api.todos to web_anon;

create role authenticator noinherit login password 'mysecretpassword';
grant web_anon to authenticator;
```

Similarly, another file is `tutorial.conf` which includes the PostgREST configuration from the tutorial verbatim:

```
db-uri = "postgres://authenticator:mysecretpassword@localhost:5432/postgres"
db-schemas = "api"
db-anon-role = "web_anon"
```

Now we need to wrap these two into a NixOS module which runs Postgres (with those setup commands) and PostgREST (with that configuration file), which is what `server.nix` does:

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

{ config = {
    networking.firewall.allowedTCPPorts = [ 3000 ];

    services.postgresql = {
      enable = true;

      initialScript = ./setup.sql;
    };

    systemd.services.postgrest = {
      wantedBy = [ "multi-user.target" ];

      after = [ "postgresql.service" ];

      path = [ pkgs.postgrest ];

      script = "postgrest ${./tutorial.conf}";

      serviceConfig.User = "authenticator";
    };

    users = {
      groups.database = { };

      users = {
        authenticator = {
          isSystemUser = true;

          group = "database";
        };
      };
    };
  };
}
```

The main extra thing we do here (that's not mentioned in the tutorial) is that we created an `authenticator` user and `database` group to match the database user of the same name.

Additionally, we open up port 3000 in the firewall, which we're going to need to do to test the PostgREST API (served on port 3000 by default).

We're also going to create a `client.nix` file containing a pretty bare NixOS configuration for our test client machine:

```
{ pkgs, ... }: {
  environment.defaultPackages = [ pkgs.curl ];
}
```

Next, we're going to write a Python script (`script.py`) to orchestrate our integration test:

```
import json

start_all()

expected = [
    {"id": 1, "done": False, "task": "finish tutorial 0", "due": None},
    {"id": 2, "done": False, "task": "pat self on back", "due": None},
]

actual = json.loads(
    client.wait_until_succeeds(
        "curl --fail --silent http://server:3000/todos",
        55,
    )
)

assert expected == actual
```

This Python script logs into the `client` machine to run a `curl` command and compares the JSON output of the command against the expected output from the tutorial.

Finally, we tie this all together in `flake.nix`:

```
{ inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";

    flake-utils.url = "github:numtide/flake-utils/v1.0.0";
  };

  outputs = { flake-utils, nixpkgs, self, ... }:
    flake-utils.lib.eachDefaultSystem (system: {
      packages = {
        inherit (self.checks.${system}.default) driverInteractive;
      };

      checks.default = nixpkgs.legacyPackages."${system}".nixosTest {
        name = "test";

        nodes = {
          server = import ./server.nix;

          client = import ./client.nix;
        };
```

```
        testScript = builtins.readFile ./script.py;
      };
    });
}
```

Here we're using the `nixosTest` function, which is our one-stop shop for integration testing. This function takes two main arguments that we care about:

- `nodes`

  This is an attribute set with one attribute per machine that we want to test and each attribute contains a NixOS configuration for the corresponding machine. Here, we're going to be testing two machines (named "server" and "client") whose NixOS configurations are going to be imported from `server.nix` and `client.nix` respectively. We don't have to store these NixOS configurations in separate files (we could store them inline within this same `flake.nix` file), but the code is a bit easier to follow if we keep things separate.

- `testScript`

  This is our Python script that we're also going to store in a separate file (`script.py`).

Then we can run our NixOS test using:

```
$ nix flake check --print-build-logs
```

If you're on macOS you will need to follow the macOS-specific setup instructions from the Setting up your development environment chapter before you can run the above command. In particular, you will need to have a Linux builder running in order to build the virtual machine image for the above NixOS test.

Interactive testing

You can also interactively run a NixOS test using a Python REPL that has access to the same commands available within our `script.py` test script. To do so, run:

```
$ nix run '.#driverInteractive'
…
>>>
```

This will then open up a Python REPL with autocompletion support and the first thing we're going to do in this REPL is to launch all of the machines associated with our NixOS test (`server` and `client` in this case):

```
>>> start_all()
```

Note that if you do this you won't notice the prompt (because it will be clobbered by the log output from the server), but it's still there. Alternatively, you can prevent that by temporarily silencing the machine's log output like this:

```
>>> serial_stdout_off()
>>> start_all()
start all VMs
client: starting vm
mke2fs 1.47.0 (5-Feb-2023)
client: QEMU running (pid 21160)
server: starting vm
mke2fs 1.47.0 (5-Feb-2023)
server: QEMU running (pid 21169)
(finished: start all VMs, in 0.38 seconds)
>>> serial_stdout_on()
```

These `serial_stdout_{on,off}` functions come in handy if you find the machine log output too noisy.

Once you've started up the machines you can begun running commands that interact with each machine by invoking methods on Python objects of the same name.

For example, you can run a simple `echo "Hello, world!"` command on the server machine like this:

```
>>> server.succeed('echo "Hello, world!"')
server: must succeed: echo "Hello, world!"
server: waiting for the VM to finish booting
server: Guest shell says: b'Spawning backdoor root shell...\n'
server: connected to guest root shell
server: (connecting took 0.00 seconds)
(finished: waiting for the VM to finish booting, in 0.00 seconds)
(finished: must succeed: echo "Hello, world!", in 0.04 seconds)
'Hello, world!\n'
```

... and the `succeed` method will capture the command's output and return it as a string which you can then further process within Python.

Now let's step through the same logic as the original test script so we can see for ourselves the intermediate values computed along the way:

```
>>> response = client.succeed("curl --fail --silent http://server:3000/todos")
client: must succeed: curl --fail --silent http://server:3000/todos
(finished: must succeed: curl --fail --silent http://server:3000/todos, in 0.04 seconds)
>>> response
'[{"id":1,"done":false,"task":"finish tutorial 0","due":null}, \n {"id":2,"done":false,"task":"pat sel\
f on back","due":null}]'

>>> import json
>>> json.loads(response)
[{'id': 1, 'done': False, 'task': 'finish tutorial 0', 'due': None}, {'id': 2, 'done': False, 'task': \
'pat self on back', 'due': None}]
```

This sort of interactive exploration really comes in handy when authoring the test for the first time since it helps you understand the shape of the data and figure out which commands you need to run.

You can consult the NixOS test section of the NixOS manual[4] if you need a full list of available methods that you can invoke on machine objects. Some really common methods are:

- `succeed(command)` - Run a command once and require it to succeed
- `wait_until_succeeds(command, timeout)` - Keep running a command until it succeeds
- `wait_for_open_port(port, address, timeout)` - Wait for a service to open a port
- `wait_for_unit(unit, user, timeout)` - Wait for a Systemd unit to start up

... but there are also some really cool methods you can use like:

- `block()` - Simulate a network disconnect
- `crash()` - Simulate a sudden power failure
- `wait_for_console_text(regex, timeout)` - Wait for the given regex to match against any terminal output
- `get_screen_text()` - Use OCR to capture the current text on the screen

Shared constants

There are several constants that we use repeatedly throughout our integration test, like:

- the PostgREST port
- the database, schema and table
- the username, role, group, and credentials

One advantage of codifying the tutorial as a NixOS test is that we can define constants like these in one place instead of copying them repeatedly and hoping that they remain in sync. For example, we wouldn't want our integration test to break just because we changed the

[4] https://nixos.org/manual/nixos/stable/#ssec-machine-objects

user's password in `setup.sql` and forgot to make the matching change to the password in `tutorial.conf`. Integration tests can often be time consuming to run and debug, so we want our test to break for more meaningful reasons (an actual bug in the system under test[5]) and not because of errors in the test code.

However, we're going to need to restructure things a little bit in order to share constants between the various test files. In particular, we're going to be using NixOS options to store shared constants for reuse throughout the test. To keep this example short, we won't factor out all of the shared constants and we'll focus on a turning a couple of representative constants into NixOS options.

First, we'll factor out the `"authenticator"` username into a shared constant, which we'll store as a `tutorial.username` NixOS option in `server.nix`:

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

{ options = {
    tutorial = {
      user = lib.mkOption {
        type = lib.types.str;
      };
    };
  };

  config = {
    tutorial.user = "authenticator";

    systemd.services.postgrest = {
      …

      serviceConfig.User = config.tutorial.user;
    };

    users = {
      users = {
        "${config.tutorial.user}" = {
          isSystemUser = true;

          group = "database";
        };
      };
    };
  };
}
```

… and that fixes all of the occurrences of the `authenticator` user in `server.nix` but what about `setup.sql` or `tutorial.conf`?

One way to do this is to inline `setup.sql` and `tutorial.conf` into our `server.nix` file so that we can interpolate the NixOS options directly into the generated files, like this:

[5] https://en.wikipedia.org/wiki/System_under_test

```
{ …

  config = {
    services.postgresql = {
      …

      initialScript = pkgs.writeText "setup.sql" ''
        create schema api;

        …

        create role ${config.tutorial.user} noinherit login password 'mysecretpassword';
        grant web_anon to ${config.tutorial.user};
      '';
    };

    systemd.services.postgrest = {
      …

      script =
        let
          configurationFile = pkgs.writeText "tutorial.conf" ''
            db-uri = "postgres://${config.tutorial.user}:mysecretpassword@localhost:5432/postgres"
            db-schemas = "api"
            db-anon-role = "web_anon"
          '';

        in
          "postgrest ${configurationFile}";

      …
    };

    …
  };
}
```

This solution isn't great, though, because it gets cramped pretty quickly and it's harder to edit inline Nix strings than standalone files. For example, when `setup.sql` is a separate file many editors will enable syntax highlighting for those SQL commands, but that syntax highlighting won't work when the SQL commands are instead stored within an inline Nix string.

Alternatively, we can keep the files separate and use the Nixpkgs `substituteAll` utility[6] to interpolate the Nix variables into the file. The way it works is that instead of using `${user}` to interpolate a variable you use `@user@`, like this new `tutorial.conf` file does:

```
db-uri = "postgres://@user@:mysecretpassword@localhost:5432/postgres"
db-schemas = "api"
db-anon-role = "web_anon"
```

Similarly, we change our `setup.sql` file to also substitute in `@user@` where necessary:

[6] https://nixos.org/manual/nixpkgs/stable/#fun-substituteAll

```
create schema api;

…

create role @user@ noinherit login password 'mysecretpassword';
grant web_anon to @user@;
```

Once we've done that we can use the `pkgs.substituteAll` utility to template those files with Nix variables of the same name:

```
{ …

  config = {
    …

    services.postgresql = {
      …

      initialScript = pkgs.substituteAll {
        name = "setup.sql";
        src = ./setup.sql;
        inherit (config.tutorial) user;
      };
    };

    systemd.services.postgrest = {
      …

      script =
        let
          configurationFile = pkgs.substituteAll {
            name = "tutorial.conf";
            src = ./tutorial.conf;
            inherit (config.tutorial) user;
          };

        in
          "postgrest ${configurationFile}";

      …
    };

    …
  };
}
```

The downside to using `pkgs.substituteAll` is that it's easier for there to be a mismatch between the variable names in the template and the variable names in Nix. Even so, this is usually the approach that I would recommend.

We can do something fairly similar to also thread through the PostgREST port everywhere it's needed. The original PostgREST tutorial doesn't specify the port in the `tutorial.conf` file, but we can add it for completeness:

```
db-uri = "postgres://@user@:mysecretpassword@localhost:5432/postgres"
db-schemas = "api"
db-anon-role = "web_anon"
server-port = @port@
```

... and then we can make matching changes to `server.nix` to define and use this port:

```
{ options = {
    tutorial = {
      port = lib.mkOption {
        type = lib.types.port;
      };

      …
    };
  };

  config = {
    tutorial.port = 3000;

    …

    networking.firewall.allowedTCPPorts = [ config.tutorial.port ];

    systemd.services.postgrest = {
      …

      script =
        let
          configurationFile = pkgs.substituteAll {
            name = "tutorial.conf";
            src = ./tutorial.conf;
            inherit (config.tutorial) port user;
          };

        in
          "postgrest ${configurationFile}";

      …
    };

    …
  };
}
```

... but we're not done! We also need to thread this port to `script.py`, which references this same port in the `curl` command.

This might seem trickier because the place where `script.py` is referenced (in `flake.nix`):

```
{ …

  outputs = { flake-utils, nixpkgs, self, ... }:
    flake-utils.lib.eachDefaultSystem (system: {
      …

      checks.default = nixpkgs.legacyPackages."${system}".nixosTest {
        …

        testScript = builtins.readFile ./script.py;
      };
    });
}
```

… is not inside of any NixOS module. So how do we access NixOS option definitions when defining our `testScript`?

The trick is that the `testScript` argument to the `nixosTest` function can be a function:

```
        # I've inlined the test script to simplify things
        testScript = { nodes }:
          let
            inherit (nodes.server.config.tutorial) port;

          in
            ''
            import json

            start_all()

            expected = [
                {"id": 1, "done": False, "task": "finish tutorial 0", "due": None},
                {"id": 2, "done": False, "task": "pat self on back", "due": None},
            ]

            actual = json.loads(
                client.wait_until_succeeds(
                    "curl --fail --silent http://server:${toString port}/todos",
                    7,
                )
            )

            assert expected == actual
            '';
```

This function takes one argument (`nodes`) which is an attribute set containing one attribute for each machine in our integration test (e.g. `server` and `client` for our example). Each of these attributes in turn has all of the output attributes generated by `evalModules`[7], including:

- `options` - the option declarations for the machine
- `config` - the final option definitions for the machine

[7] https://github.com/NixOS/nixpkgs/blob/23.11/lib/modules.nix#L317-L324

This is why we can access the server's `tutorial.port` NixOS option using `nodes.server.config.tutorial.port`.

Moreover, every NixOS configuration also has a `nixpkgs.pkgs` option storing the NixOS package set used by that machine. This means that instead of adding `curl` to our `client` machine's `environment.defaultPackages`, we could instead do something like this:

```
testScript = { nodes }:
  let
    inherit (nodes.client.config.nixpkgs.pkgs) curl;

    …

  in
    ''
    …

    actual = json.loads(
        client.wait_until_succeeds(
            "${curl}/bin/curl --fail --silent http://server:${toString port}/todos",
            7,
        )
    )

    …
    '';
```

12. Containers

The previous chapter on Integration Testing translated the PostgREST tutorial[1] to an equivalent NixOS test, but that translation left out one important detail: the original tutorial asks the user to run Postgres inside of a `docker` container:

> If Docker is not installed, you can get it here. Next, let's pull and start the database image:
>
> ```
> $ sudo docker run --name tutorial -p 5432:5432 \
>                -e POSTGRES_PASSWORD=mysecretpassword \
>                -d postgres
> ```
>
> This will run the Docker instance as a daemon and expose port 5432 to the host system so that it looks like an ordinary PostgreSQL server to the rest of the system.

We don't *have* to use Docker to run Postgres; in fact, I'd normally advise against it and recommend using the Postgres NixOS module instead. However, we can still use this as an illustrative example of how to translate Docker idioms to NixOS.

More generally, in this chapter we're going to cover container management in the context of NixOS and introduce a spectrum of options ranging from more Docker-native to more NixOS-native.

Docker registry

The most Docker-native approach is to fetch a container from the Docker registry and to illustrate that we're going to begin from the previous chapter's example integration test:

```
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10#integration-test'
```

... but this time instead of using the NixOS `postgres` module:

[1] https://postgrest.org/en/v12/tutorials/tut0.html

```
# server.nix

…

    services.postgresql = {
      enable = true;

      initialScript = ./setup.sql;
    };

…
```

… we're going to replace that code with a NixOS configuration that runs the official `postgres` image obtained from the Docker registry in almost the same way as the tutorial:

```
…

    virtualisation.oci-containers = {
      backend = "docker";

      containers = {
        # We really should call this container "postgres" but we're going to
        # call it "tutorial" just for fun to match the original instructions.
        tutorial = {
          image = "postgres:16.2";

          environment.POSTGRES_PASSWORD = "mysecretpassword";

          extraOptions = [ "--network=host" ];
        };
      };
    };

…
```

We've only made a few changes from the tutorial:

- We specify the container tag (`16.2`) explicitly

  By default if we omit the tag we'll get whatever image the `latest` tag points to. We'd rather specify a versioned tag to improve the reproducibility of the example because the `latest` tag is a moving target that points to the most recent published version of the `postgres` image.

- We use host networking instead of publishing port 5432

  This means that our container won't use an isolated network and will instead reuse the host machine's network for communicating between containers. This simplifies the example because at the time of this writing there isn't a stock NixOS option for declaratively managing a Docker network.

NixOS provides a `virtualisation.oci-containers` option hierarchy which lets us declaratively define the same options as `docker run` but it also takes care of several supporting details for us, including:

- installing and running the `docker` service for our NixOS machine

  In particular, the `backend = "docker";` option is what specifies to use Docker as the backend for running our container (instead of the default backend, which is Podman).

- creating a Systemd service to start and stop the container

  This Systemd service runs essentially the same `docker run` command from the tutorial when starting up, and also runs the matching `docker stop` command when shutting down.

We also still need to run the setup commands from `setup.sql` after our container starts up, but we no longer have a convenient `services.postgresql.initialScript` option that we can use for this purpose when going the Docker route. Instead, we're going to create our own "one shot" Systemd service to take care of this setup process for us:

```
…

systemd.services.setup-postgresql =
  let
    uri = "postgresql://postgres:mysecretpassword@localhost";

  in
    { wantedBy = [ "multi-user.target" ];

      path = [ pkgs.docker ];

      preStart = ''
        until docker exec tutorial pg_isready --dbname ${uri}; do
            sleep 1
        done
      '';

      script = ''
        docker exec --interactive tutorial psql ${uri} < ${./setup.sql}
      '';

      serviceConfig = {
        Type = "oneshot";

        RemainAfterExit = "yes";
      };
    };

…
```

We can then sequence our Postgrest service after that one by changing it to be after our setup service:

```
    systemd.services.postgrest = {
      …

-     after = [ "postgresql.service" ];
+     after = [ "setup-postgresql.service" ];

      …
    };
```

… and if we re-run the test it still passes:

```
$ nix flake check
```

The upside of this approach is that it requires the least buy-in to the NixOS ecosystem. However, there's one major downside: it only works if the system has network access to a Docker registry, which can be a non-starter for a few reasons:

- this complicates the your deployment architecture

  … since now your system needs network access at runtime to some sort of Docker registry (either the public one or a private registry you host). This can cause problems when deploying to environments with highly restricted or no network access.

- this adds latency to your system startup

  … because you need to download the image the first time you deploy the system (or upgrade the image). Moreover, if you do any sort of integration testing (like we are) you have to pay that cost every time you run your integration test.

Podman

You might notice that the NixOS option hierarchy for running Docker containers is called `virtualisation.oci-containers` and not `virtualisation.docker-containers`. This is because Docker containers are actually OCI containers (short for "Open Container Initiative") and OCI containers can be run by any OCI-compatible backend.

Moreover, NixOS supports two OCI-compatible backends: Docker and Podman. In fact, you often might prefer to use Podman (the default) instead of Docker for running containers for a few reasons:

- Podman doesn't require a daemon

  Podman is distributed as a `podman` command line tool that encompasses all of the logic needed to run OCI containers. The `podman` command line tool doesn't need to delegate instructions to a daemon running in the background like Docker does.

- Improved security

  Podman's "daemonless" operation also implies "rootless" operation. In other words, you don't need root privileges to run an OCI container using Podman. Docker, on the other hand,

requires elevated privileges by default to run containers (unless you run Docker in rootless mode), which is an enormous security risk. For example, a misconfigured or compromised `Dockerfile` can allow a container to mount the host's root filesystem which in the *best case* corrupts the host's filesystem with the guest container's files and in the worst case enables total compromise of the host by an attacker.

Switching from Docker to Podman is pretty easy: we only need to change the `virtualisation.oci-containers.backend` option from `"docker"` to `"podman"` (or just delete the option, since `"podman"` is the default):

```
    virtualisation.oci-containers = {
-     backend = "docker";
+     backend = "podman";
```

... and then change all command-line references from `docker` to `podman`:

```
-         path = [ pkgs.docker ];
+         path = [ pkgs.podman ];

          preStart = ''
-           until docker exec tutorial pg_isready --dbname ${uri}; do
+           until podman exec tutorial pg_isready --dbname ${uri}; do
              sleep 1
            done
          '';

          script = ''
-           docker exec --interactive tutorial psql ${uri} < ${./setup.sql}
+           podman exec --interactive tutorial psql ${uri} < ${./setup.sql}
          '';
```

This works because the `podman` command-line tool provides the exact same interface as the the `docker` command-line tool, so it's a drop-in replacement.

`streamLayeredImage`

If you're willing to lean more into NixOS, there are even better options at your disposal. For example, you can build the Docker image using NixOS, too! In fact, Docker images built with NixOS tend to be leaner than official Docker images for two main reasons:

- Nix-built Docker images automatically prune build-time dependencies

  This is actually a feature of the Nix language itself, which autodetects which dependencies are runtime dependencies and only includes those in built packages (including Nix-built Docker images). Docker images built using traditional `Dockerfiles` usually have to do a bunch of gymnastics to avoid accidentally pulling in build-time dependencies into the image or any of its layers but in Nix you get this feature for free.

- Nix-built Docker images have more cache-friendly layers

  For more details you can read this post[2] but to summarize: Nix's utilities for building Docker images are smarter than `Dockerfiles` and result in superior layer caching. This means that as you amend the Docker image to add or remove dependencies you get fewer rebuilds and better disk utilization.

Nixpkgs provides several utilities for building Docker images[3] using Nix, but we're only going to concern ourselves with one of those utilities: `pkgs.dockerTools.streamLayeredImage`[4]. This is the most efficient utility at our disposal that will ensure the best caching and least disk churn out of all the available options.

We'll delete the old `postgrest` service and instead use this `streamLayeredImage` utility to build an application container wrapping `postgrest`. We can then reference that container in `virtualisation.oci-containers.containers`, like this:

```
…

virtualisation.oci-containers = {
  backend = "docker";

  containers = {
    …

    postgrest = {
      image = "postgrest:nix";

      imageStream = pkgs.dockerTools.streamLayeredImage {
        name = "postgrest";

        tag = "nix";

        contents = [ pkgs.postgrest ];

        config.Cmd = [ "postgrest" ./tutorial.conf ];
      };

      extraOptions = [ "--network=host" ];
    };
  };
};

…
```

You can also clone an example containing all changes up to this point by running:

```
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10#docker'
```

This creates a new `postgrest` container that doesn't depend on the Docker registry at all. Note that the Docker registry *does host* an official `postgrest` image but we're not going to use that image. Instead, we're using a `postgrest` Docker image built entirely using Nix.

[2] https://grahamc.com/blog/nix-and-layered-docker-images/
[3] https://nixos.org/manual/nixpkgs/stable/#sec-pkgs-dockerTools
[4] https://nixos.org/manual/nixpkgs/stable/#ssec-pkgs-dockerTools-streamLayeredImage

Moreover, this Nix-built docker image integrates efficiently with Nix. If we add or remove dependencies from our Docker image then we'll only build and store what changed (the "diff"), instead of building and storing an entirely new copy of the whole Docker image archive.

Of course, your next thought might be: "if we're using Nix/NixOS to build and consume Docker images, then do we still need Docker?". Can we cut out Docker as an intermediate and still preserve most of the same benefits of containerization?

Yes!

NixOS containers

NixOS actually supports a more NixOS-native alternative to Docker, known as NixOS containers[5]. Under the hood, these use `systemd-nspawn` as the container engine but that's essentially invisible to the end user (you). The user interface for NixOS containers is much simpler than the Docker-based alternatives, so if you don't need Docker specifically but you still want some basic isolation guarantees then this is the way to go.

The easiest way to illustrate how NixOS containers work is to redo our `postgrest` example to put both Postgres and PostgREST in separate NixOS containers. We're going to begin by resetting our example back to the non-container example from the previous chapter:

```
$ nix flake init --template 'github:Gabriella439/nixos-in-production/0.10#integration-test'
```

... and then we'll make two changes. First, instead of running Postgres on the host machine like this:

```
  …

  services.postgresql = {
    enable = true;

    initialScript = ./setup.sql;
  };

  …
```

... we're going to change that code to run it inside of a NixOS container (still named `tutorial`) like this:

[5] https://nixos.org/manual/nixos/stable/#ch-containers

```
…

containers.tutorial = {
  autoStart = true;

  config = {
    services.postgresql = {
      enable = true;

      initialScript = ./setup.sql;
    };
  };
};

…
```

This change illustrates what's neat about NixOS containers: we can configure them using the same NixOS options that we use to configure the host machine. All we have to do is wrap the options inside `containers.${NAME}.config` but otherwise we configure NixOS options the same way whether inside or outside of the container. This is why it's worth trying out NixOS containers if you don't need any Docker-specific functionality but you still want some basic isolation in place. NixOS containers are significantly more ergonomic to use.

We can also wrap our PostgREST service in the exact same way, replacing this:

```
systemd.services.postgrest = {
  wantedBy = [ "multi-user.target" ];

  after = [ "postgresql.service" ];

  path = [ pkgs.postgrest ];

  script = "postgrest ${./tutorial.conf}";

  serviceConfig.User = "authenticator";
};

users = {
  groups.database = { };

  users.authenticator = {
    isSystemUser = true;

    group = "database";
  };
};
```

… with this:

```
containers.postgrest = {
  autoStart = true;

  config = {
    systemd.services.postgrest = {
      wantedBy = [ "multi-user.target" ];

      after = [ "postgresql.service" ];

      path = [ pkgs.postgrest ];

      script = "postgrest ${./tutorial.conf}";

      serviceConfig.User = "authenticator";
    };

    users = {
      groups.database = { };

      users.authenticator = {
        isSystemUser = true;

        group = "database";
      };
    };
  };
};
```

... and that's it! In both cases, we just took our existing NixOS configuration options and wrapped them in something like:

```
containers."${name}" = {
  autoStart = true;

  config = {
    …
  };
};
```

... and we got containerization for free.

Just like the Docker example, these NixOS containers use the host network to connect to one another, meaning that they don't set `privateNetwork = true;` (which creates a private network for the given NixOS container). At the time of this writing there isn't an easy way to network NixOS containers isolated in this way that doesn't involve carefully selecting a bunch of magic strings (IP addresses and network interfaces). This is a poor user experience and not one that I feel comfortable documenting or endorsing at the time of this writing.