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 for sale at http://leanpub.com/nixos-in-production

This version was published on 2023-03-12

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.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

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] Thank you to Adam Gordon Bell for this analogy

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

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[1] 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.

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:

[1] https://nixos.org/download.html

- *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 2.11.0, and you can pin the Nix version like this:

```
$ VERSION='2.11.0'
$ 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[2].

Feel free to use a Nix version newer than 2.11.0 if you want. The above example installation script only pins the version 2.11.0 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='2.11.0'
$ URL="https://releases.nixos.org/nix/nix-${VERSION}/install"
$ sh <(curl --location "${URL}") --help
```

[2] 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='2.11.0'
$ 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[3] 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.builder
```

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

The `nix run nixpkgs#darwin.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#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/f1a49e20e1b4a7eeb43d73d60bae5be84a1e7610";
  };

  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" ];
```

[3] https://nixos.org/manual/nixpkgs/unstable/#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

{ users.users.root.initialPassword = "";
}
```

Obviously, the above NixOS configuration is not secure since it leaves the `root` account wide open. We'll change this to something more secure later.

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

```
$ nix run
…

<<< Welcome to NixOS 22.11.20220918.f1a49e2 (aarch64) - ttyAMA0 >>>

Run 'nixos-help' for the NixOS manual.

nixos login:
```

You can then log into the virtual machine as the `root` user and an empty password. After you successfully log in, shut down the virtual machine by typing `Ctrl-a + c` to open the `qemu` prompt and then type `quit` followed by `Enter` to exit.

If you successfully log into 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

{ users.users.root.initialPassword = "";

  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#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#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.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; }
```

```
  ];

  users.users.root.initialPassword = "";

  system.stateVersion = "22.11";
}
```

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 22.11*

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 21.11 and another machine in your data center that was first deployed using Nixpkgs 22.05. 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.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
         lib.concatMapStrings renderPort config.networking.firewall.allowedTCPPorts
       }
       </ul>
```

[1] http://localhost:8080

```
      </body>
      </html>
    '';
  };
};

networking.firewall.allowedTCPPorts = [ 80 ];

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

users.users.root.initialPassword = "";

system.stateVersion = "22.11";
}
```

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.

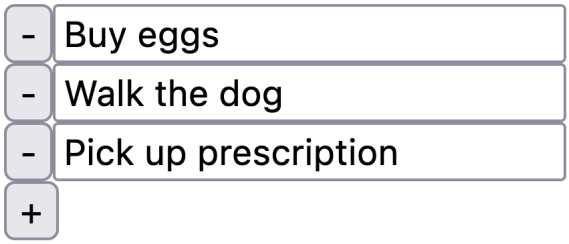
Now save the following NixOS configuration to `module.nix`:

```
# module.nix

{ services.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; }
  ];

  users.users.root.initialPassword = "";

  system.stateVersion = "22.11";
}
```

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/22.05/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/22.05/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 = "22.05";
  };
}
```

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

[3] https://github.com/NixOS/nixpkgs/blob/22.05/nixos/modules/system/activation/top-level.nix

```
$ nix repl github:NixOS/nixpkgs/22.05
…
nix-repl> result = lib.evalModules { modules = [ ./top-level.nix ]; }

nix-repl> :p result.config
{ system = { build = { toplevel = "Fake NixOS - version 22.05"; }; nixos = { release = "22.05"; }; }; }

nix-repl> result.config.system.build.toplevel
"Fake NixOS - version 22.05"
```

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 = "22.05";
  };
}
```

... 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/22.05";

  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 = "22.05";
        };
```

[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 22.05"; }; nixos = { release = "22.05"; }; }; }

$ nix eval ./evalModules#config.system.build.toplevel
"Fake NixOS - version 22.05"
```

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/22.05";

  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 = "22.05";
          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/22.05";

  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 = "22.05";
          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 = "22.05";
      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 = "22.05";
      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 ${"22.05"}"


# Evaluate the string interpolation
= "Fake NixOS - version 22.05"
```

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; }

    { users.users.root.initialPassword = ""; }
  ];
}
```

... 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, ... }: { users.users.root.initialPassword = lib.mkDefault ""; })
  ];
}
```

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/22.05/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; }
    { users.users.root.initialPassword = ""; }
  ];
}
```

... is equivalent to this NixOS module:

```
{ config = {
    services.openssh.enable = true;
    users.users.root.initialPassword = "";
  };
}
```

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; }
    { users.users.root.initialPassword = ""; }
  ];
}
```

... 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 ]; }
    { users.users.root.initialPassword = ""; }
  ];
}
```

... 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 ]; }
    { users.users.root.initialPassword = ""; }
  ];
}
```

... 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 ];
>     users.users.root.initialPassword = "";
>   };
> }
> ```
>
> 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/nixos-22.05/nixos/modules/services/web-servers/nginx/default.nix#L890

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;

      users.users.root.initialPassword = "";
    };
  }
```

If you launch the above NixOS configuration and log in as `root` 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 2022-11-05 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 ];

    users.users.root.initialPassword = "";
  }
```

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 ];

    users.users.root.initialPassword = "";
  }
```

... 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 ];

    users.users.root.initialPassword = "";
  }
```

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/22.11#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#server
```

… 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.

Deploying our configuration

To deploy the Terraform configuration, run the following commands:

```
$ nix shell github:NixOS/nixpkgs/22.11#terraform
$ terraform init
$ terraform apply
```

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

[3] https://docs.aws.amazon.com/general/latest/gr/rande.html

```
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[4].
- The function has outputs
  Terraform calls these output values[5].
- The function does things other than producing output values
  For example, the function might provision a resource[6].
- 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[7] with appropriate function arguments.

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

Input variables

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

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

```
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=…
```

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_key_file" {
}

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_key_file = ssh_key_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 three 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_key_file" {
    filename = "${path.module}/id_ed25519"
    content = tls_private_key.nixos-in-production.private_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[8] is currently not secure because the deployment state is stored locally unencrypted. We can and will fix this by storing the deployment state using the S3 backend[9] but that won't be covered until the next chapter.

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 {
    volume_size = 7
  }
}
```

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:

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

```
module "ami" {
  …;
}

module "nixos" {
  source = "github.com/Gabriella439/terraform-nixos-ng//nixos?ref=d8563d06cc65bc699ffbf1ab8d692b1343ec\
d927"
  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 = "d8563d06cc65bc699ffbf1ab8d692b1343ecd927";
        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_key_file = ssh_key_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=d8563d06cc65bc699ffbf1ab8d692b1343ecd9\
27"
  release = "22.11"
  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=d8563d06cc65bc699ffbf1ab8d692b1343ec\
d927"

  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_key_file.filename}"

  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.

The next chapter will cover how to provision a dedicated builder for this purpose.