

NixOS Manual

Version 20.09

Table of Contents

Preface

I. Installation

1. Obtaining NixOS
2. Installing NixOS
3. Changing the Configuration
4. Upgrading NixOS

II. Configuration

5. Configuration Syntax
6. Package Management
7. User Management
8. File Systems
9. X Window System
10. GPU acceleration
11. Xfce Desktop Environment
12. Networking
13. Linux Kernel
14. Pantheon Desktop
15. Matomo
16. Nextcloud
17. Jitsi Meet
18. Grocy
19. Yggdrasil
20. Prosody
21. Prometheus exporters
22. WeeChat
23. Taskserver
24. Matrix
25. Gitlab
26. Mailman
27. Trezor
28. Emacs
29. Flatpak
30. PostgreSQL

- 31. FoundationDB
- 32. BorgBackup
- 33. Hiding process information
- 34. SSL/TLS Certificates with ACME
- 35. Oh my ZSH
- 36. Plotinus
- 37. Digital Bitbox
- 38. Input Methods
- 39. Profiles
- 40. Kubernetes

III. Administration

- 41. Service Management
- 42. Rebooting and Shutting Down
- 43. User Sessions
- 44. Control Groups
- 45. Logging
- 46. Cleaning the Nix Store
- 47. Container Management
- 48. Troubleshooting

IV. Development

- 49. Getting the Sources
- 50. Writing NixOS Modules
- 51. Building Specific Parts of NixOS
- 52. Writing NixOS Documentation
- 53. Building Your Own NixOS CD
- 54. NixOS Tests
- 55. Testing the Installer
- 56. Releases

A. Configuration Options

B. Release Notes

List of Examples

- 2.1. Example partition schemes for NixOS on `/dev/sda` (MBR)
- 2.2. Example partition schemes for NixOS on `/dev/sda` (UEFI)
- 2.3. Commands for Installing NixOS on `/dev/sda`
- 2.4. NixOS Configuration
- 28.1. Nix expression to build Emacs with packages (`emacs.nix`)
- 28.2. Querying Emacs packages
- 28.3. Custom Emacs in `configuration.nix`

- 28.4. Custom Emacs in `~/.config/nixpkgs/config.nix`
- 28.5. Custom Emacs build
- 28.6. Package initialization in `.emacs`
- 28.7. nXML Schema Configuration (`~/.emacs.d/schemas.xml`)
- 50.1. Structure of NixOS Modules
- 50.2. NixOS Module for the “locate” Service
- 50.3. Extensible type placeholder in the service module
- 50.4. Extending `services.xserver.displayManager.enable` in the `gdm` module
- 50.5. Extending `services.xserver.displayManager.enable` in the `sddm` module
- 50.6. Directly defined submodule
- 50.7. Submodule defined as a reference
- 50.8. Declaration of a list of submodules
- 50.9. Definition of a list of submodules
- 50.10. Declaration of attribute sets of submodules
- 50.11. Declaration of attribute sets of submodules
- 50.12. Adding a type check
- 50.13. Overriding a type check
- 50.14. Freeform submodule
- 50.15. Module with conventional `settings` option
- 50.16. Declaring a type-checked `settings` attribute
- 52.1. Pandoc invocation to convert GitHub-Flavoured Markdown to DocBook 5 XML

Preface

This manual describes how to install, use and extend NixOS, a Linux distribution based on the purely functional package management system Nix, that is composed using modules and packages defined in the Nixpkgs project.

Additional information regarding the Nix package manager and the Nixpkgs project can be found in respectively the Nix manual and the Nixpkgs manual.

If you encounter problems, please report them on the [Discourse](#) or on the `#nixos` channel on Freenode. Bugs should be reported in NixOS’ GitHub issue tracker.

Note

Commands prefixed with `#` have to be run as root, either requiring to login as root user or temporarily switching to it using `sudo` for example.

Part I. Installation

This section describes how to obtain, install, and configure NixOS for first-time use.

Table of Contents

1. Obtaining NixOS
2. Installing NixOS
3. Changing the Configuration

Chapter 1. Obtaining NixOS

NixOS ISO images can be downloaded from the NixOS download page. There are a number of installation options. If you happen to have an optical drive and a spare CD, burning the image to CD and booting from that is probably the easiest option. Most people will need to prepare a USB stick to boot from. Section 2.5.1, “Bootting from a USB Drive” describes the preferred method to prepare a USB stick. A number of alternative methods are presented in the NixOS Wiki.

As an alternative to installing NixOS yourself, you can get a running NixOS system through several other means:

- Using virtual appliances in Open Virtualization Format (OVF) that can be imported into VirtualBox. These are available from the NixOS download page.
- Using AMIs for Amazon’s EC2. To find one for your region and instance type, please refer to the list of most recent AMIs.
- Using NixOps, the NixOS-based cloud deployment tool, which allows you to provision VirtualBox and EC2 NixOS instances from declarative specifications. Check out the NixOps homepage for details.

Chapter 2. Installing NixOS

Table of Contents

- 2.1. Booting the system
- 2.2. Partitioning and formatting
- 2.3. Installing
- 2.4. Installation summary
- 2.5. Additional installation notes

2.1. Booting the system

NixOS can be installed on BIOS or UEFI systems. The procedure for a UEFI installation is by and large the same as a BIOS installation. The differences are mentioned in the steps that follow.

The installation media can be burned to a CD, or now more commonly, ”burned” to a USB drive (see Section 2.5.1, “Bootting from a USB Drive”).

The installation media contains a basic NixOS installation. When it’s finished booting, it should have detected most of your hardware.

The NixOS manual is available by running **nixos-help**.

You are logged-in automatically as **nixos**. The **nixos** user account has an empty password so you can use **sudo** without a password.

If you downloaded the graphical ISO image, you can run **systemctl start display-manager** to start the desktop environment. If you want to continue on the terminal, you can use **loadkeys** to switch to your preferred keyboard layout. (We even provide neo2 via **loadkeys de neo!**)

If the text is too small to be legible, try **setfont ter-v32n** to increase the font size.

2.1.1. Networking in the installer

The boot process should have brought up networking (check **ip a**). Networking is necessary for the installer, since it will download lots of stuff (such as source tarballs or Nixpkgs channel binaries). It’s best if you have a DHCP server on your network. Otherwise configure networking manually using **ifconfig**.

To manually configure the network on the graphical installer, first disable network-manager with **systemctl stop NetworkManager**.

To manually configure the wifi on the minimal installer, run **wpa_supplicant -B -i interface -c <(wpa_passphrase 'SSID' 'key')**.

If you would like to continue the installation from a different machine you can use activated SSH daemon. You need to copy your ssh key to either `/home/nixos/.ssh/authorized_keys` or `/root/.ssh/authorized_keys` (Tip: For installers with a modifiable filesystem such as the sd-card installer image a key can be manually placed by mounting the image on a different machine). Alternatively you must set a password for either **root** or **nixos** with **passwd** to be able to login.

2.2. Partitioning and formatting

The NixOS installer doesn't do any partitioning or formatting, so you need to do that yourself.

The NixOS installer ships with multiple partitioning tools. The examples below use **parted**, but also provides **fdisk**, **gdisk**, **cfdisk**, and **cgdisk**.

The recommended partition scheme differs depending if the computer uses *Legacy Boot* or *UEFI*.

2.2.1. UEFI (GPT)

Here's an example partition scheme for UEFI, using `/dev/sda` as the device.

Note

You can safely ignore **parted**'s informational message about needing to update `/etc/fstab`.

1. Create a *GPT* partition table.

```
# parted /dev/sda -- mklabel gpt
```

2. Add the *root* partition. This will fill the disk except for the end part, where the swap will live, and the space left in front (512MiB) which will be used by the boot partition.

```
1 # parted /dev/sda -- mkpart primary 512MiB -8GiB
```

3. Next, add a *swap* partition. The size required will vary according to needs, here a 8GiB one is created.

```
1 # parted /dev/sda -- mkpart primary linux-swap -8GiB 100%
```

Note

The swap partition size rules are no different than for other Linux distributions.

4. Finally, the *boot* partition. NixOS by default uses the ESP (EFI system partition) as its */boot* partition. It uses the initially reserved 512MiB at the start of the disk.

```
1 # parted /dev/sda -- mkpart ESP fat32 1MiB 512MiB
# parted /dev/sda -- set 3 esp on
```

Once complete, you can follow with Section 2.2.3, "Formatting".

2.2.2. Legacy Boot (MBR)

Here's an example partition scheme for Legacy Boot, using `/dev/sda` as the device.

Note

You can safely ignore **parted**'s informational message about needing to update `/etc/fstab`.

1. Create a *MBR* partition table.

```
# parted /dev/sda -- mklabel msdos
```

2. Add the *root* partition. This will fill the the disk except for the end part, where the swap will live.

```
1 # parted /dev/sda -- mkpart primary 1MiB -8GiB
```

3. Finally, add a *swap* partition. The size required will vary according to needs, here a 8GiB one is created.

```
1 # parted /dev/sda -- mkpart primary linux-swap -8GiB 100%
```

Note

The swap partition size rules are no different than for other Linux distributions.

Once complete, you can follow with Section 2.2.3, “Formatting”.

2.2.3. Formatting

Use the following commands:

- For initialising Ext4 partitions: **mkfs.ext4**. It is recommended that you assign a unique symbolic label to the file system using the option `-L label`, since this makes the file system configuration independent from device changes. For example:

```
# mkfs.ext4 -L nixos /dev/sda1
```

- For creating swap partitions: **mkswap**. Again it's recommended to assign a label to the swap partition: `-L label`. For example:

```
# mkswap -L swap /dev/sda2
```

- **UEFI systems** For creating boot partitions: **mkfs.fat**. Again it's recommended to assign a label to the boot partition: `-n label`. For example:

```
# mkfs.fat -F 32 -n boot /dev/sda3
```

- For creating LVM volumes, the LVM commands, e.g., **pvcreate**, **vgcreate**, and **lvcreate**.
- For creating software RAID devices, use **mdadm**.

2.3. Installing

1. Mount the target file system on which NixOS should be installed on `/mnt`, e.g.

```
# mount /dev/disk/by-label/nixos /mnt
```

2. **UEFI systems** Mount the boot file system on `/mnt/boot`, e.g.

```
# mkdir -p /mnt/boot
2 # mount /dev/disk/by-label/boot /mnt/boot
```

3. If your machine has a limited amount of memory, you may want to activate swap devices now (**swapon device**). The installer (or rather, the build actions that it may spawn) may need quite a bit of RAM, depending on your configuration.

```
# swapon /dev/sda2
```

4. You now need to create a file `/mnt/etc/nixos/configuration.nix` that specifies the intended configuration of the system. This is because NixOS has a *declarative* configuration model: you create or edit a description of the desired configuration of your system, and then NixOS takes care of making it happen. The syntax of the NixOS configuration file is described in Chapter 5, *Configuration Syntax*, while a list of available configuration options appears in Appendix A, *Configuration Options*. A minimal example is shown in Example 2.4, “NixOS Configuration”.

The command **nixos-generate-config** can generate an initial configuration file for you:

```
# nixos-generate-config --root /mnt
```

You should then edit `/mnt/etc/nixos/configuration.nix` to suit your needs:

```
# nano /mnt/etc/nixos/configuration.nix
```

If you're using the graphical ISO image, other editors may be available (such as **vim**). If you have network access, you can also install other editors -- for instance, you can install Emacs by running `nix-env -f <nixpkgs> -iA emacs`.

BIOS systems You *must* set the option `boot.loader.grub.device` to specify on which disk the GRUB boot loader is to be installed. Without it, NixOS cannot boot.

UEFI systems You *must* set the option `boot.loader.systemd-boot.enable` to **true**. **nixos-generate-config** should do this automatically for new configurations when booted in UEFI mode.

You may want to look at the options starting with `boot.loader.efi` and `boot.loader.systemd` as well.

If there are other operating systems running on the machine before installing NixOS, the `boot.loader.grub.useOSProber` option can be set to **true** to automatically add them to the grub menu.

If you need to configure networking for your machine the configuration options are described in Chapter 12, *Networking*. In particular, while wifi is supported on the installation image, it is not enabled by default in the configuration generated by **nixos-generate-config**.

Another critical option is `fileSystems`, specifying the file systems that need to be mounted by NixOS. However, you typically don't need to set it yourself, because **nixos-generate-config** sets it automatically in `/mnt/etc/nixos/hardware-configuration.nix` from your currently mounted file systems. (The configuration file `hardware-configuration.nix` is included from `configuration.nix` and will be overwritten by future invocations of **nixos-generate-config**; thus, you generally should not modify it.) Additionally, you may want to look at Hardware configuration for known-hardware at this point or after installation.

Note

Depending on your hardware configuration or type of file system, you may need to set the option `boot.initrd.kernelModules` to include the kernel modules that are necessary for mounting the root file system, otherwise the installed system will not be able to boot. (If this happens, boot from the installation media again, mount the target file system on `/mnt`, fix `/mnt/etc/nixos/configuration.nix` and rerun `nixos-install`.) In most cases, **nixos-generate-config** will figure out the required modules.

5. Do the installation:

```
# nixos-install
```

This will install your system based on the configuration you provided. If anything fails due to a configuration problem or any other issue (such as a network outage while downloading binaries from the NixOS binary cache), you can re-run **nixos-install** after fixing your `configuration.nix`.

As the last step, **nixos-install** will ask you to set the password for the **root** user, e.g.

```
setting root password...
2 Enter new UNIX password: ***
Retype new UNIX password: ***
```

Note

For unattended installations, it is possible to use **nixos-install --no-root-passwd** in order to disable the password prompt entirely.

6. If everything went well:

```
1 # reboot
```

7. You should now be able to boot into the installed NixOS. The GRUB boot menu shows a list of *available configurations* (initially just one). Every time you change the NixOS configuration (see Changing Configuration), a new item is added to the menu. This allows you to easily roll back to a previous configuration if something goes wrong.

You should log in and change the root password with **passwd**.

You'll probably want to create some user accounts as well, which can be done with **useradd**:

```
$ useradd -c 'Eelco Dolstra' -m eelco
2 $ passwd eelco
```

You may also want to install some software. For instance,

```
$ nix-env -qaP \*
```

shows what packages are available, and

```
1 $ nix-env -f '<nixpkgs>' -iA w3m
```

installs the w3m browser.

2.4. Installation summary

To summarise, Example 2.3, “Commands for Installing NixOS on /dev/sda” shows a typical sequence of commands for installing NixOS on an empty hard drive (here /dev/sda). Example 2.4, “NixOS Configuration” shows a corresponding configuration Nix expression.

Example 2.1. Example partition schemes for NixOS on /dev/sda (MBR)

```
# parted /dev/sda -- mklabel msdos
2 # parted /dev/sda -- mkpart primary 1MiB -8GiB
# parted /dev/sda -- mkpart primary linux-swap -8GiB 100%
```

Example 2.2. Example partition schemes for NixOS on /dev/sda (UEFI)

```
# parted /dev/sda -- mklabel gpt
2 # parted /dev/sda -- mkpart primary 512MiB -8GiB
# parted /dev/sda -- mkpart primary linux-swap -8GiB 100%
4 # parted /dev/sda -- mkpart ESP fat32 1MiB 512MiB
# parted /dev/sda -- set 3 esp on
```

Example 2.3. Commands for Installing NixOS on /dev/sda

With a partitioned disk.

```
# mkfs.ext4 -L nixos /dev/sda1
2 # mkswap -L swap /dev/sda2
# swapon /dev/sda2
4 # mkfs.fat -F 32 -n boot /dev/sda3 # (for UEFI systems only)
# mount /dev/disk/by-label/nixos /mnt
6 # mkdir -p /mnt/boot # (for UEFI systems only)
# mount /dev/disk/by-label/boot /mnt/boot # (for UEFI systems only)
8 # nixos-generate-config --root /mnt
# nano /mnt/etc/nixos/configuration.nix
10 # nixos-install
# reboot
```

Example 2.4. NixOS Configuration

```
1 { config, pkgs, ... }: {
  imports = [
3     # Include the results of the hardware scan.
    ./hardware-configuration.nix
5 ];

7 boot.loader.grub.device = "/dev/sda";    # (for BIOS systems only)
  boot.loader.systemd-boot.enable = true; # (for UEFI systems only)
9
  # Note: setting fileSystems is generally not
11 # necessary, since nixos-generate-config figures them out
  # automatically in hardware-configuration.nix.
13 #fileSystems."/".device = "/dev/disk/by-label/nixos";

15 # Enable the OpenSSH server.
  services.sshd.enable = true;
17 }
```

2.5. Additional installation notes

2.5.1. Booting from a USB Drive

For systems without CD drive, the NixOS live CD can be booted from a USB stick. You can use the **dd** utility to write the image: **dd if=*path-to-image* of=*/dev/sdX***. Be careful about specifying the correct drive; you can use the **lsblk** command to get a list of block devices.

On macOS

```
$ diskutil list
2 [...]
/dev/diskN (external, physical):
4   #:                                TYPE NAME                                SIZE                                IDENTIFIER
  [...]
6 $ diskutil unmountDisk diskN
  Unmount of all volumes on diskN was successful
8 $ sudo dd if=nix.iso of=/dev/rdiskN
```

Using the 'raw' **rdiskN** device instead of **diskN** completes in minutes instead of hours. After **dd** completes, a GUI dialog "The disk you inserted was not readable by this computer" will pop up, which can be ignored.

The **dd** utility will write the image verbatim to the drive, making it the recommended option for both UEFI and non-UEFI installations.

2.5.2. Booting from the "netboot" media (PXE)

Advanced users may wish to install NixOS using an existing PXE or iPXE setup.

These instructions assume that you have an existing PXE or iPXE infrastructure and simply want to add the NixOS installer as another option. To build the necessary files from a recent version of nixpkgs, you can run:

```
nix-build -A netboot.x86_64-linux nixos/release.nix
```

This will create a `result` directory containing: * `bzImage` - the Linux kernel * `initrd` - the initrd file * `netboot.ipxe` - an example ipxe script demonstrating the appropriate kernel command line arguments for this image

If you're using plain PXE, configure your boot loader to use the `bzImage` and `initrd` files and have it provide the same kernel command line arguments found in `netboot.ipxe`.

If you're using iPXE, depending on how your HTTP/FTP/etc. server is configured you may be able to use `netboot.ipxe` unmodified, or you may need to update the paths to the files to match your server's directory layout

In the future we may begin making these files available as build products from hydra at which point we will update this documentation with instructions on how to obtain them either for placing on a dedicated TFTP server or to boot them directly over the internet.

2.5.3. Installing in a VirtualBox guest

Installing NixOS into a VirtualBox guest is convenient for users who want to try NixOS without installing it on bare metal. If you want to use a pre-made VirtualBox appliance, it is available at the downloads page. If you want to set up a VirtualBox guest manually, follow these instructions:

1. Add a New Machine in VirtualBox with OS Type "Linux / Other Linux"
2. Base Memory Size: 768 MB or higher.
3. New Hard Disk of 8 GB or higher.
4. Mount the CD-ROM with the NixOS ISO (by clicking on CD/DVD-ROM)
5. Click on Settings / System / Processor and enable PAE/NX
6. Click on Settings / System / Acceleration and enable "VT-x/AMD-V" acceleration
7. Click on Settings / Display / Screen and select VMSVGA as Graphics Controller
8. Save the settings, start the virtual machine, and continue installation like normal

There are a few modifications you should make in `configuration.nix`. Enable booting:

```
boot.loader.grub.device = "/dev/sda";
```

Also remove the `fsck` that runs at startup. It will always fail to run, stopping your boot until you press `*`.

```
boot.initrd.checkJournalingFS = false;
```

Shared folders can be given a name and a path in the host system in the VirtualBox settings (Machine / Settings / Shared Folders, then click on the "Add" icon). Add the following to the `/etc/nixos/configuration.nix` to auto-mount them. If you do not add `"nofail"`, the system will not boot properly. The same goes for disabling `rngd` which is normally used to get randomness but this does not work in virtual machines.

```
{ config, pkgs, ... } :
2 {
    security.rngd.enable = false; // otherwise vm will not boot
4     ...

6     fileSystems."/virtualboxshare" = {
        fsType = "vboxsf";
8         device = "nameofthesharedfolder";
        options = [ "rw" "nofail" ];
10    };
}
```

The folder will be available directly under the root directory.

2.5.4. Installing from another Linux distribution

Because Nix (the package manager) & Nixpkgs (the Nix packages collection) can both be installed on any (most?) Linux distributions, they can be used to install NixOS in various creative ways. You can, for instance:

1. Install NixOS on another partition, from your existing Linux distribution (without the use of a USB or optical device!)
2. Install NixOS on the same partition (in place!), from your existing non-NixOS Linux distribution using `NIXOS_LUSTRATE`.
3. Install NixOS on your hard drive from the Live CD of any Linux distribution.

The first steps to all these are the same:

1. Install the Nix package manager:

Short version:

```
$ curl -L https://nixos.org/nix/install | sh
2 $ . $HOME/.nix-profile/etc/profile.d/nix.sh # ...or open a fresh shell
```

More details in the Nix manual

2. Switch to the NixOS channel:

If you've just installed Nix on a non-NixOS distribution, you will be on the `nixpkgs` channel by default.

```
$ nix-channel --list
2 nixpkgs https://nixos.org/channels/nixpkgs-unstable
```

As that channel gets released without running the NixOS tests, it will be safer to use the `nixos-*` channels instead:

```
$ nix-channel --add https://nixos.org/channels/nixos-version nixpkgs
```

You may want to throw in a `nix-channel --update` for good measure.

3. Install the NixOS installation tools:

You'll need `nixos-generate-config` and `nixos-install` and we'll throw in some man pages and `nixos-enter` just in case you want to chroot into your NixOS partition. They are installed by default on NixOS, but you don't have NixOS yet..

```
$ nix-env -f '<nixpkgs/nixos>' --arg configuration {} -iA
  ↳ config.system.build.{nixos-generate-config,nixos-install,nixos-enter,manual.manpages}
```

4. **Note**

The following 5 steps are only for installing NixOS to another partition. For installing NixOS in place using `NIXOS_LUSTRATE`, skip ahead.

Prepare your target partition:

At this point it is time to prepare your target partition. Please refer to the partitioning, file-system creation, and mounting steps of Chapter 2, *Installing NixOS*

If you're about to install NixOS in place using `NIXOS_LUSTRATE` there is nothing to do for this step.

5. Generate your NixOS configuration:

```
$ sudo `which nixos-generate-config` --root /mnt
```

You'll probably want to edit the configuration files. Refer to the `nixos-generate-config` step in Chapter 2, *Installing NixOS* for more information.

Consider setting up the NixOS bootloader to give you the ability to boot on your existing Linux partition. For instance, if you're using GRUB and your existing distribution is running Ubuntu, you may want to add something like this to your `configuration.nix`:

```

boot.loader.grub.extraEntries = ''
2   menuentry "Ubuntu" {
      search --set=ubuntu --fs-uuid 3cc3e652-0c1f-4800-8451-033754f68e6e
4     configfile "($ubuntu)/boot/grub/grub.cfg"
      }
6   '';

```

(You can find the appropriate UUID for your partition in `/dev/disk/by-uuid`)

6. Create the `nixbld` group and user on your original distribution:

```

$ sudo groupadd -g 30000 nixbld
2 $ sudo useradd -u 30000 -g nixbld -G nixbld nixbld

```

7. Download/build/install NixOS:

Warning

Once you complete this step, you might no longer be able to boot on existing systems without the help of a rescue USB drive or similar.

```
$ sudo PATH="$PATH" NIX_PATH="$NIX_PATH" `which nixos-install` --root /mnt
```

Again, please refer to the `nixos-install` step in Chapter 2, *Installing NixOS* for more information.

That should be it for installation to another partition!

8. Optionally, you may want to clean up your non-NixOS distribution:

```

$ sudo userdel nixbld
2 $ sudo groupdel nixbld

```

If you do not wish to keep the Nix package manager installed either, run something like `sudo rm -rv ↵ ~/.nix-* /nix` and remove the line that the Nix installer added to your `~/.profile`.

9. **Note**

The following steps are only for installing NixOS in place using `NIXOS_LUSTRATE`:

Generate your NixOS configuration:

```
$ sudo `which nixos-generate-config` --root /
```

Note that this will place the generated configuration files in `/etc/nixos`. You'll probably want to edit the configuration files. Refer to the `nixos-generate-config` step in Chapter 2, *Installing NixOS* for more information.

You'll likely want to set a root password for your first boot using the configuration files because you won't have a chance to enter a password until after you reboot. You can initialize the root password to an empty one with this line: (and of course don't forget to set one once you've rebooted or to lock the account with `sudo passwd -l root` if you use `sudo`)

```
users.users.root.initialHashedPassword = "";
```

10. Build the NixOS closure and install it in the `system` profile:

```

$ nix-env -p /nix/var/nix/profiles/system -f '<nixpkgs/nixos>' -I
  ↵ nixos-config=/etc/nixos/configuration.nix -iA system

```

11. Change ownership of the `/nix` tree to root (since your Nix install was probably single user):

```
$ sudo chown -R 0.0 /nix
```

12. Set up the `/etc/NIXOS` and `/etc/NIXOS_LUSTRATE` files:

`/etc/NIXOS` officializes that this is now a NixOS partition (the bootup scripts require its presence).

`/etc/NIXOS_LUSTRATE` tells the NixOS bootup scripts to move *everything* that's in the root partition to `/old-root`. This will move your existing distribution out of the way in the very early stages of the NixOS bootup. There are exceptions (we do need to keep NixOS there after all), so the NixOS lustrate process will not touch:

- The `/nix` directory
- The `/boot` directory
- Any file or directory listed in `/etc/NIXOS_LUSTRATE` (one per line)

Note

Support for `NIXOS_LUSTRATE` was added in NixOS 16.09. The act of "lustrating" refers to the wiping of the existing distribution. Creating `/etc/NIXOS_LUSTRATE` can also be used on NixOS to remove all mutable files from your root partition (anything that's not in `/nix` or `/boot` gets "lustrated" on the next boot.

`lustrate /l stre t/ verb.`

purify by expiatory sacrifice, ceremonial washing, or some other ritual action.

Let's create the files:

```
$ sudo touch /etc/NIXOS
2 $ sudo touch /etc/NIXOS_LUSTRATE
```

Let's also make sure the NixOS configuration files are kept once we reboot on NixOS:

```
$ echo etc/nixos | sudo tee -a /etc/NIXOS_LUSTRATE
```

13. Finally, move the `/boot` directory of your current distribution out of the way (the lustrate process will take care of the rest once you reboot, but this one must be moved out now because NixOS needs to install its own boot files:

Warning

Once you complete this step, your current distribution will no longer be bootable! If you didn't get all the NixOS configuration right, especially those settings pertaining to boot loading and root partition, NixOS may not be bootable either. Have a USB rescue device ready in case this happens.

```
$ sudo mv -v /boot /boot.bak &&
2 sudo /nix/var/nix/profiles/system/bin/switch-to-configuration boot
```

Cross your fingers, reboot, hopefully you should get a NixOS prompt!

14. If for some reason you want to revert to the old distribution, you'll need to boot on a USB rescue disk and do something along these lines:

```
# mkdir root
2 # mount /dev/sdaX root
# mkdir root/nixos-root
4 # mv -v root/* root/nixos-root/
# mv -v root/nixos-root/old-root/* root/
6 # mv -v root/boot.bak root/boot # We had renamed this by hand earlier
# umount root
8 # reboot
```

This may work as is or you might also need to reinstall the boot loader

And of course, if you're happy with NixOS and no longer need the old distribution:

```
sudo rm -rf /old-root
```

15. It's also worth noting that this whole process can be automated. This is especially useful for Cloud VMs, where provider do not provide NixOS. For instance, `nixos-infect` uses the `lustrate` process to convert Digital Ocean droplets to NixOS from other distributions automatically.

2.5.5. Installing behind a proxy

To install NixOS behind a proxy, do the following before running `nixos-install`.

1. Update proxy configuration in `/mnt/etc/nixos/configuration.nix` to keep the internet accessible after reboot.

```
networking.proxy.default = "http://user:password@proxy:port/";
2 networking.proxy.noProxy = "127.0.0.1,localhost,internal.domain";
```

2. Setup the proxy environment variables in the shell where you are running `nixos-install`.

```
# proxy_url="http://user:password@proxy:port/"
2 # export http_proxy="$proxy_url"
# export HTTP_PROXY="$proxy_url"
4 # export https_proxy="$proxy_url"
# export HTTPS_PROXY="$proxy_url"
```

Note

If you are switching networks with different proxy configurations, use the `specialisation` option in `configuration.nix` to switch proxies at runtime. Refer to Appendix A, *Configuration Options* for more information.

Chapter 3. Changing the Configuration

The file `/etc/nixos/configuration.nix` contains the current configuration of your machine. Whenever you've changed something in that file, you should do

```
# nixos-rebuild switch
```

to build the new configuration, make it the default configuration for booting, and try to realise the configuration in the running system (e.g., by restarting system services).

Warning

This command doesn't start/stop user services automatically. `nixos-rebuild` only runs a `daemon-reload` for each user with running user services.

Warning

These commands must be executed as root, so you should either run them from a root shell or by prefixing them with `sudo -i`.

You can also do

```
# nixos-rebuild test
```

to build the configuration and switch the running system to it, but without making it the boot default. So if (say) the configuration locks up your machine, you can just reboot to get back to a working configuration.

There is also

```
1 # nixos-rebuild boot
```

to build the configuration and make it the boot default, but not switch to it now (so it will only take effect after the next reboot).

You can make your configuration show up in a different submenu of the GRUB 2 boot screen by giving it a different *profile name*, e.g.

```
1 # nixos-rebuild switch -p test
```

which causes the new configuration (and previous ones created using `-p test`) to show up in the GRUB submenu “NixOS - Profile ‘test’”. This can be useful to separate test configurations from “stable” configurations.

Finally, you can do

```
$ nixos-rebuild build
```

to build the configuration but nothing more. This is useful to see whether everything compiles cleanly.

If you have a machine that supports hardware virtualisation, you can also test the new configuration in a sandbox by building and running a QEMU *virtual machine* that contains the desired configuration. Just do

```
1 $ nixos-rebuild build-vm
$ ./result/bin/run-*-vm
```

The VM does not have any data from your host system, so your existing user accounts and home directories will not be available unless you have set `mutableUsers = false`. Another way is to temporarily add the following to your configuration:

```
users.users.your-user.initialHashedPassword = "test";
```

Important: delete the `$hostname.qcow2` file if you have started the virtual machine at least once without the right users, otherwise the changes will not get picked up. You can forward ports on the host to the guest. For instance, the following will forward host port 2222 to guest port 22 (SSH):

```
1 $ QEMU_NET_OPTS="hostfwd=tcp::2222-:22" ./result/bin/run-*-vm
```

allowing you to log in via SSH (assuming you have set the appropriate passwords or SSH authorized keys):

```
1 $ ssh -p 2222 localhost
```

Chapter 4. Upgrading NixOS

Table of Contents

4.1. Automatic Upgrades

The best way to keep your NixOS installation up to date is to use one of the NixOS *channels*. A channel is a Nix mechanism for distributing Nix expressions and associated binaries. The NixOS channels are updated automatically from NixOS’s Git repository after certain tests have passed and all packages have been built. These channels are:

- *Stable channels*, such as `nixos-20.09`. These only get conservative bug fixes and package upgrades. For instance, a channel update may cause the Linux kernel on your system to be upgraded from 4.19.34 to 4.19.38 (a minor bug fix), but not from 4.19.x to 4.20.x (a major change that has the potential to break things). Stable channels are generally maintained until the next stable branch is created.
- The *unstable channel*, `nixos-unstable`. This corresponds to NixOS’s main development branch, and may thus see radical changes between channel updates. It’s not recommended for production systems.
- *Small channels*, such as `nixos-20.09-small` or `nixos-unstable-small`. These are identical to the stable and unstable channels described above, except that they contain fewer binary packages. This means they get updated faster than the regular channels (for instance, when a critical security patch is committed to NixOS’s source tree), but may require more packages to be built from source than usual. They’re mostly intended for server environments and as such contain few GUI applications.

To see what channels are available, go to <https://nixos.org/channels>. (Note that the URIs of the various channels redirect to a directory that contains the channel’s latest version and includes ISO images and VirtualBox appliances.) Please note that during the release process, channels that are not yet released will be present here as well. See the Getting NixOS page <https://nixos.org/nixos/download.html> to find the newest supported stable release.

When you first install NixOS, you're automatically subscribed to the NixOS channel that corresponds to your installation source. For instance, if you installed from a 20.09 ISO, you will be subscribed to the `nixos-20.09` channel. To see which NixOS channel you're subscribed to, run the following as root:

```
# nix-channel --list | grep nixos
2 nixos https://nixos.org/channels/nixos-unstable
```

To switch to a different NixOS channel, do

```
# nix-channel --add https://nixos.org/channels/channel-name nixos
```

(Be sure to include the `nixos` parameter at the end.) For instance, to use the NixOS 20.09 stable channel:

```
# nix-channel --add https://nixos.org/channels/nixos-20.09 nixos
```

If you have a server, you may want to use the “small” channel instead:

```
1 # nix-channel --add https://nixos.org/channels/nixos-20.09-small nixos
```

And if you want to live on the bleeding edge:

```
1 # nix-channel --add https://nixos.org/channels/nixos-unstable nixos
```

You can then upgrade NixOS to the latest version in your chosen channel by running

```
1 # nixos-rebuild switch --upgrade
```

which is equivalent to the more verbose `nix-channel --update nixos; nixos-rebuild switch`.

Note

Channels are set per user. This means that running `nix-channel --add` as a non root user (or without `sudo`) will not affect configuration in `/etc/nixos/configuration.nix`

Warning

It is generally safe to switch back and forth between channels. The only exception is that a newer NixOS may also have a newer Nix version, which may involve an upgrade of Nix's database schema. This cannot be undone easily, so in that case you will not be able to go back to your original channel.

4.1. Automatic Upgrades

You can keep a NixOS system up-to-date automatically by adding the following to `configuration.nix`:

```
system.autoUpgrade.enable = true;
2 system.autoUpgrade.allowReboot = true;
```

This enables a periodically executed systemd service named `nixos-upgrade.service`. If the `allowReboot` option is `false`, it runs `nixos-rebuild switch --upgrade` to upgrade NixOS to the latest version in the current channel. (To see when the service runs, see `systemctl list-timers`.) If `allowReboot` is `true`, then the system will automatically reboot if the new generation contains a different kernel, `initrd` or kernel modules. You can also specify a channel explicitly, e.g.

```
system.autoUpgrade.channel = https://nixos.org/channels/nixos-20.09;
```

Part II. Configuration

This chapter describes how to configure various aspects of a NixOS machine through the configuration file `/etc/nixos/configuration.nix`. As described in Chapter 3, *Changing the Configuration*, changes to this file only take effect after you run `nixos-rebuild`.

Table of Contents

5. Configuration Syntax
6. Package Management
7. User Management
8. File Systems
9. X Window System
10. GPU acceleration
11. Xfce Desktop Environment
12. Networking
13. Linux Kernel
14. Pantheon Desktop
15. Matomo
16. Nextcloud
17. Jitsi Meet
18. Grocy
19. Yggdrasil
20. Prosody
21. Prometheus exporters
22. WeeChat
23. Taskserver
24. Matrix
25. Gitlab
26. Mailman
27. Trezor
28. Emacs
29. Flatpak
30. PostgreSQL
31. FoundationDB
32. BorgBackup
33. Hiding process information
34. SSL/TLS Certificates with ACME
35. Oh my ZSH
36. Plotinus
37. Digital Bitbox
38. Input Methods
39. Profiles
40. Kubernetes

Chapter 5. Configuration Syntax

Table of Contents

5.1. NixOS Configuration File

5.2. Abstractions

5.3. Modularity

5.4. Syntax Summary

The NixOS configuration file `/etc/nixos/configuration.nix` is actually a *Nix expression*, which is the Nix package manager's purely functional language for describing how to build packages and configurations. This means you have all the expressive power of that language at your disposal, including the ability to abstract over common patterns, which is very useful when managing complex systems. The syntax and semantics of the Nix language are fully described in the Nix manual, but here we give a short overview of the most important constructs useful in NixOS configuration files.

5.1. NixOS Configuration File

The NixOS configuration file generally looks like this:

```
1 { config, pkgs, ... }:  
2  
3 { option definitions  
4 }
```

The first line (`{ config, pkgs, ... }:`) denotes that this is actually a function that takes at least the two arguments `config` and `pkgs`. (These are explained later.) The function returns a *set* of option definitions (`{ ... }`). These definitions have the form `name = value`, where *name* is the name of an option and *value* is its value. For example,

```
1 { config, pkgs, ... }:  
2  
3 { services.httpd.enable = true;  
4   services.httpd.adminAddr = "alice@example.org";  
5   services.httpd.virtualHosts.localhost.documentRoot = "/webroot";  
6 }
```

defines a configuration with three option definitions that together enable the Apache HTTP Server with `/webroot` as the document root.

Sets can be nested, and in fact dots in option names are shorthand for defining a set containing another set. For instance, `services.httpd.enable` defines a set named `services` that contains a set named `httpd`, which in turn contains an option definition named `enable` with value `true`. This means that the example above can also be written as:

```
1 { config, pkgs, ... }:  
2  
3 { services = {  
4   httpd = {  
5     enable = true;  
6     adminAddr = "alice@example.org";  
7     virtualHosts = {  
8       localhost = {  
9         documentRoot = "/webroot";  
10      };  
11    };  
12  };  
13 };  
14 }
```

which may be more convenient if you have lots of option definitions that share the same prefix (such as `services.httpd`).

NixOS checks your option definitions for correctness. For instance, if you try to define an option that doesn't exist (that is, doesn't have a corresponding *option declaration*), **nixos-rebuild** will give an error like:

```
The option `services.httpd.enable' defined in `/etc/nixos/configuration.nix' does
↪ not exist.
```

Likewise, values in option definitions must have a correct type. For instance, `services.httpd.enable` must be a Boolean (`true` or `false`). Trying to give it a value of another type, such as a string, will cause an error:

```
The option value `services.httpd.enable' in `/etc/nixos/configuration.nix' is not
↪ a boolean.
```

Options have various types of values. The most important are:

Strings Strings are enclosed in double quotes, e.g.

```
1 networking.hostName = "dexter";
```

Special characters can be escaped by prefixing them with a backslash (e.g. `\`).

Multi-line strings can be enclosed in *double single quotes*, e.g.

```
networking.extraHosts =
2  ''
    127.0.0.2 other-localhost
4    10.0.0.1 server
    '';
```

The main difference is that it strips from each line a number of spaces equal to the minimal indentation of the string as a whole (disregarding the indentation of empty lines), and that characters like `"` and `\` are not special (making it more convenient for including things like shell code). See more info about this in the Nix manual [here](#).

Booleans These can be `true` or `false`, e.g.

```
networking.firewall.enable = true;
2 networking.firewall.allowPing = false;
```

Integers For example,

```
boot.kernel.sysctl."net.ipv4.tcp_keepalive_time" = 60;
```

(Note that here the attribute name `net.ipv4.tcp_keepalive_time` is enclosed in quotes to prevent it from being interpreted as a set named `net` containing a set named `ipv4`, and so on. This is because it's not a NixOS option but the literal name of a Linux kernel setting.)

Sets Sets were introduced above. They are name/value pairs enclosed in braces, as in the option definition

```
fileSystems."/boot" =
2  { device = "/dev/sda1";
    fsType = "ext4";
4    options = [ "rw" "data=ordered" "relatime" ];
    };
```

Lists The important thing to note about lists is that list elements are separated by whitespace, like this:

```
1 boot.kernelModules = [ "fuse" "kvm-intel" "coretemp" ];
```

List elements can be any other type, e.g. sets:

```
1 swapDevices = [ { device = "/dev/disk/by-label/swap"; } ];
```

Packages Usually, the packages you need are already part of the Nix Packages collection, which is a set that can be accessed through the function argument `pkgs`. Typical uses:

```
environment.systemPackages =
2   [ pkgs.thunderbird
      pkgs.emacs
4   ];

6 services.postgresql.package = pkgs.postgresql_10;
```

The latter option definition changes the default PostgreSQL package used by NixOS's PostgreSQL service to 10.x. For more information on packages, including how to add new ones, see Section 6.1.2, “Adding Custom Packages”.

5.2. Abstractions

If you find yourself repeating yourself over and over, it's time to abstract. Take, for instance, this Apache HTTP Server configuration:

```
{
2   services.httpd.virtualHosts =
      { "blog.example.org" = {
4       documentRoot = "/webroot/blog.example.org";
        adminAddr = "alice@example.org";
6       forceSSL = true;
        enableACME = true;
8       enablePHP = true;
      };
10   "wiki.example.org" = {
        documentRoot = "/webroot/wiki.example.org";
12   adminAddr = "alice@example.org";
        forceSSL = true;
14   enableACME = true;
        enablePHP = true;
16   };
18 }
```

It defines two virtual hosts with nearly identical configuration; the only difference is the document root directories. To prevent this duplication, we can use a `let`:

```
let
2   commonConfig =
      { adminAddr = "alice@example.org";
4     forceSSL = true;
      enableACME = true;
6   };
in
8 {
   services.httpd.virtualHosts =
10   { "blog.example.org" = (commonConfig // { documentRoot =
      ↪ "/webroot/blog.example.org"; });
      "wiki.example.org" = (commonConfig // { documentRoot =
12   ↪ "/webroot/wiki.example.com"; });
13 }
```

The `let commonConfig = ...` defines a variable named `commonConfig`. The `//` operator merges two attribute sets, so the configuration of the second virtual host is the set `commonConfig` extended with the document root option.

You can write a `let` wherever an expression is allowed. Thus, you also could have written:

```
{
2  services.httpd.virtualHosts =
    let commonConfig = ...; in
4    { "blog.example.org" = (commonConfig // { ... })
      "wiki.example.org" = (commonConfig // { ... })
6    };
}
```

but not `{ let commonConfig = ...; in ...; }` since attributes (as opposed to attribute values) are not expressions.

Functions provide another method of abstraction. For instance, suppose that we want to generate lots of different virtual hosts, all with identical configuration except for the document root. This can be done as follows:

```
{
2  services.httpd.virtualHosts =
    let
4      makeVirtualHost = webroot:
        { documentRoot = webroot;
          adminAddr = "alice@example.org";
          forceSSL = true;
          enableACME = true;
8        };
10     in
        { "example.org" = (makeVirtualHost "/webroot/example.org");
12        "example.com" = (makeVirtualHost "/webroot/example.com");
          "example.gov" = (makeVirtualHost "/webroot/example.gov");
14        "example.nl" = (makeVirtualHost "/webroot/example.nl");
        };
16 }
```

Here, `makeVirtualHost` is a function that takes a single argument `webroot` and returns the configuration for a virtual host. That function is then called for several names to produce the list of virtual host configurations.

5.3. Modularity

The NixOS configuration mechanism is modular. If your `configuration.nix` becomes too big, you can split it into multiple files. Likewise, if you have multiple NixOS configurations (e.g. for different computers) with some commonality, you can move the common configuration into a shared file.

Modules have exactly the same syntax as `configuration.nix`. In fact, `configuration.nix` is itself a module. You can use other modules by including them from `configuration.nix`, e.g.:

```
{ config, pkgs, ... }:
2
{ imports = [ ./vpn.nix ./kde.nix ];
4  services.httpd.enable = true;
  environment.systemPackages = [ pkgs.emacs ];
6  ...
}
```

Here, we include two modules from the same directory, `vpn.nix` and `kde.nix`. The latter might look like this:

```
{ config, pkgs, ... }:
2
{ services.xserver.enable = true;
4  services.xserver.displayManager.sddm.enable = true;
  services.xserver.desktopManager.plasma5.enable = true;
6  environment.systemPackages = [ pkgs.vim ];
}
```

Note that both `configuration.nix` and `kde.nix` define the option `environment.systemPackages`. When multiple modules define an option, NixOS will try to *merge* the definitions. In the case of `environment.systemPackages`, that's easy: the lists of packages can simply be concatenated. The value in `configuration.nix` is merged last, so for list-type options, it will appear at the end of the merged list. If you want it to appear first, you can use `mkBefore`:

```
boot.kernelModules = mkBefore [ "kvm-intel" ];
```

This causes the `kvm-intel` kernel module to be loaded before any other kernel modules.

For other types of options, a merge may not be possible. For instance, if two modules define `services.httpd.adminAddr`, **nixos-rebuild** will give an error:

```
The unique option `services.httpd.adminAddr' is defined multiple times, in
  ↪ `/etc/nixos/httpd.nix' and `/etc/nixos/configuration.nix'.
```

When that happens, it's possible to force one definition take precedence over the others:

```
1 services.httpd.adminAddr = pkgs.lib.mkForce "bob@example.org";
```

When using multiple modules, you may need to access configuration values defined in other modules. This is what the `config` function argument is for: it contains the complete, merged system configuration. That is, `config` is the result of combining the configurations returned by every module ^[1]. For example, here is a module that adds some packages to `environment.systemPackages` only if `services.xserver.enable` is set to `true` somewhere else:

```
{ config, pkgs, ... }:
2 { environment.systemPackages =
4   if config.services.xserver.enable then
6     [ pkgs.firefox
        pkgs.thunderbird
        ]
8   else
        [ ];
10 }
```

With multiple modules, it may not be obvious what the final value of a configuration option is. The command `nixos-option` allows you to find out:

```
$ nixos-option services.xserver.enable
2 true

4 $ nixos-option boot.kernelModules
[ "tun" "ipv6" "loop" ... ]
```

Interactive exploration of the configuration is possible using **nix repl**, a read-eval-print loop for Nix expressions. A typical use:

```
1 $ nix repl '<nixpkgs/nixos>'

3 nix-repl> config.networking.hostName
"mandark"
5
nix-repl> map (x: x.hostName) config.services.httpd.virtualHosts
7 [ "example.org" "example.gov" ]
```

While abstracting your configuration, you may find it useful to generate modules using code, instead of writing files. The example below would have the same effect as importing a file which sets those options.

```

1 { config, pkgs, ... }:
3 let netConfig = { hostName }: {
    networking.hostName = hostName;
5     networking.useDHCP = false;
    };
7
9 in
{ imports = [ (netConfig "nixos.localdomain") ]; }

```

5.4. Syntax Summary

Below is a summary of the most important syntactic constructs in the Nix expression language. It's not complete. In particular, there are many other built-in functions. See the Nix manual for the rest.

Example	Description
<i>Basic values</i>	
"Hello world"	A string
"\${pkgs.bash}/bin/sh"	A string containing an expression (expands to <code>/nix/store/hash-bash-ve</code>)
true, false	Booleans
123	An integer
./foo.png	A path (relative to the containing Nix expression)
<i>Compound values</i>	
{ x = 1; y = 2; }	A set with attributes named x and y
{ foo.bar = 1; }	A nested set, equivalent to { foo = { bar = 1; }; }
rec { x = "foo"; y = x + "bar"; }	A recursive set, equivalent to { x = "foo"; y = "foobar"; }
["foo" "bar"]	A list with two elements
<i>Operators</i>	
"foo" + "bar"	String concatenation
1 + 2	Integer addition
"foo" == "f" + "oo"	Equality test (evaluates to true)
"foo" != "bar"	Inequality test (evaluates to true)
!true	Boolean negation
{ x = 1; y = 2; }.x	Attribute selection (evaluates to 1)
{ x = 1; y = 2; }.z or 3	Attribute selection with default (evaluates to 3)
{ x = 1; y = 2; } // { z = 3; }	Merge two sets (attributes in the right-hand set taking precedence)
<i>Control structures</i>	
if 1 + 1 == 2 then "yes!" else "no!"	Conditional expression
assert 1 + 1 == 2; "yes!"	Assertion check (evaluates to "yes!"). See Section 50.4, "Warnings and A
let x = "foo"; y = "bar"; in x + y	Variable definition
with pkgs.lib; head [1 2 3]	Add all attributes from the given set to the scope (evaluates to 1)
<i>Functions (lambdas)</i>	
x: x + 1	A function that expects an integer and returns it increased by 1
(x: x + 1)100	A function call (evaluates to 101)
let inc = x: x + 1; in inc (inc (inc 100))	A function bound to a variable and subsequently called by name (evaluate
{ x, y }: x + y	A function that expects a set with required attributes x and y and concat
{ x, y ? "bar" }: x + y	A function that expects a set with required attribute x and optional y, usi
{ x, y, ... }: x + y	A function that expects a set with required attributes x and y and ignores
{ x, y } @ args: x + y	A function that expects a set with required attributes x and y, and binds
<i>Built-in functions</i>	
import ./foo.nix	Load and return Nix expression in given file
map (x: x + x) [1 2 3]	Apply a function to every element of a list (evaluates to [2 4 6])

[1] If you're wondering how it's possible that the (indirect) *result* of a function is passed as an *input* to that same function: that's because Nix is a "lazy" language -- it only computes values when they are needed. This works as long as no individual configuration value depends on itself.

Chapter 6. Package Management

Table of Contents

6.1. Declarative Package Management

6.2. Ad-Hoc Package Management

This section describes how to add additional packages to your system. NixOS has two distinct styles of package management:

- *Declarative*, where you declare what packages you want in your `configuration.nix`. Every time you run `nixos-rebuild`, NixOS will ensure that you get a consistent set of binaries corresponding to your specification.
- *Ad hoc*, where you install, upgrade and uninstall packages via the `nix-env` command. This style allows mixing packages from different Nixpkgs versions. It's the only choice for non-root users.

6.1. Declarative Package Management

With declarative package management, you specify which packages you want on your system by setting the option `environment.systemPackages`. For instance, adding the following line to `configuration.nix` enables the Mozilla Thunderbird email application:

```
environment.systemPackages = [ pkgs.thunderbird ];
```

The effect of this specification is that the Thunderbird package from Nixpkgs will be built or downloaded as part of the system when you run `nixos-rebuild switch`.

Note

Some packages require additional global configuration such as D-Bus or systemd service registration so adding them to `environment.systemPackages` might not be sufficient. You are advised to check the list of options whether a NixOS module for the package does not exist.

You can get a list of the available packages as follows:

```
$ nix-env -qaP '*' --description
2 nixos.firefox    firefox-23.0    Mozilla Firefox - the browser, reloaded
...
```

The first column in the output is the *attribute name*, such as `nixos.thunderbird`.

Note: the `nixos` prefix tells us that we want to get the package from the `nixos` channel and works only in CLI tools. In declarative configuration use `pkgs` prefix (variable).

To "uninstall" a package, simply remove it from `environment.systemPackages` and run `nixos-rebuild switch`.

6.1.1. Customising Packages

Some packages in Nixpkgs have options to enable or disable optional functionality or change other aspects of the package. For instance, the Firefox wrapper package (which provides Firefox with a set of plugins such as the Adobe Flash player) has an option to enable the Google Talk plugin. It can be set in `configuration.nix` as follows: `nixpkgs.config.firefox.enableGoogleTalkPlugin = true;`

Warning

Unfortunately, Nixpkgs currently lacks a way to query available configuration options.

Apart from high-level options, it's possible to tweak a package in almost arbitrary ways, such as changing or disabling dependencies of a package. For instance, the Emacs package in Nixpkgs by default has a dependency on GTK 2. If you want to build it against GTK 3, you can specify that as follows:

```
environment.systemPackages = [ (pkgs.emacs.override { gtk = pkgs.gtk3; }) ];
```

The function `override` performs the call to the Nix function that produces Emacs, with the original arguments amended by the set of arguments specified by you. So here the function argument `gtk` gets the value `pkgs.gtk3`, causing Emacs to depend on GTK 3. (The parentheses are necessary because in Nix, function application binds more weakly than list construction, so without them, `environment.systemPackages` would be a list with two elements.)

Even greater customisation is possible using the function `overrideAttrs`. While the `override` mechanism above overrides the arguments of a package function, `overrideAttrs` allows changing the *attributes* passed to `mkDerivation`. This permits changing any aspect of the package, such as the source code. For instance, if you want to override the source code of Emacs, you can say:

```
environment.systemPackages = [  
2   (pkgs.emacs.overrideAttrs (oldAttrs: {  
    name = "emacs-25.0-pre";  
4     src = /path/to/my/emacs/tree;  
    }))  
6 ];
```

Here, `overrideAttrs` takes the Nix derivation specified by `pkgs.emacs` and produces a new derivation in which the original's `name` and `src` attribute have been replaced by the given values by re-calling `stdenv.mkDerivation`. The original attributes are accessible via the function argument, which is conventionally named `oldAttrs`.

The overrides shown above are not global. They do not affect the original package; other packages in Nixpkgs continue to depend on the original rather than the customised package. This means that if another package in your system depends on the original package, you end up with two instances of the package. If you want to have everything depend on your customised instance, you can apply a *global* override as follows:

```
nixpkgs.config.packageOverrides = pkgs:  
2   { emacs = pkgs.emacs.override { gtk = pkgs.gtk3; };  
     };
```

The effect of this definition is essentially equivalent to modifying the `emacs` attribute in the Nixpkgs source tree. Any package in Nixpkgs that depends on `emacs` will be passed your customised instance. (However, the value `pkgs.emacs` in `nixpkgs.config.packageOverrides` refers to the original rather than overridden instance, to prevent an infinite recursion.)

6.1.2. Adding Custom Packages

It's possible that a package you need is not available in NixOS. In that case, you can do two things. First, you can clone the Nixpkgs repository, add the package to your clone, and (optionally) submit a patch or pull request to have it accepted into the main Nixpkgs repository. This is described in detail in the Nixpkgs manual. In short, you clone Nixpkgs:

```
$ git clone https://github.com/NixOS/nixpkgs  
2 $ cd nixpkgs
```

Then you write and test the package as described in the Nixpkgs manual. Finally, you add it to `environment.systemPackages`, e.g.

```
environment.systemPackages = [ pkgs.my-package ];
```

and you run `nixos-rebuild`, specifying your own Nixpkgs tree:

```
1 # nixos-rebuild switch -I nixpkgs=/path/to/my/nixpkgs
```

The second possibility is to add the package outside of the Nixpkgs tree. For instance, here is how you specify a build of the GNU Hello package directly in `configuration.nix`:

```
environment.systemPackages =
2   let
    my-hello = with pkgs; stdenv.mkDerivation rec {
4     name = "hello-2.8";
    src = fetchurl {
6     url = "mirror://gnu/hello/${name}.tar.gz";
    sha256 = "0wqd8sjmxfskrflaxywc7gqw7sfawrfvdx9skxawzfgyy0pzd6";
8   };
  };
10  in
  [ my-hello ];
```

Of course, you can also move the definition of `my-hello` into a separate Nix expression, e.g.

```
environment.systemPackages = [ (import ./my-hello.nix) ];
```

where `my-hello.nix` contains:

```
with import <nixpkgs> {}; # bring all of Nixpkgs into scope
2
stdenv.mkDerivation rec {
4   name = "hello-2.8";
   src = fetchurl {
6     url = "mirror://gnu/hello/${name}.tar.gz";
    sha256 = "0wqd8sjmxfskrflaxywc7gqw7sfawrfvdx9skxawzfgyy0pzd6";
8   };
}
```

This allows testing the package easily:

```
1 $ nix-build my-hello.nix
$ ./result/bin/hello
3 Hello, world!
```

6.2. Ad-Hoc Package Management

With the command `nix-env`, you can install and uninstall packages from the command line. For instance, to install Mozilla Thunderbird:

```
1 $ nix-env -iA nixos.thunderbird
```

If you invoke this as root, the package is installed in the Nix profile `/nix/var/nix/profiles/default` and visible to all users of the system; otherwise, the package ends up in `/nix/var/nix/profiles/per-user/username/profile` and is not visible to other users. The `-A` flag specifies the package by its attribute name; without it, the package is installed by matching against its package name (e.g. `thunderbird`). The latter is slower because it requires matching against all available Nix packages, and is ambiguous if there are multiple matching packages.

Packages come from the NixOS channel. You typically upgrade a package by updating to the latest version of the NixOS channel:

```
$ nix-channel --update nixos
```

and then running `nix-env -i` again. Other packages in the profile are *not* affected; this is the crucial difference with the declarative style of package management, where running `nixos-rebuild switch` causes all packages to be updated to their current versions in the NixOS channel. You can however upgrade all packages for which there is a newer version by doing:

```
$ nix-env -u '*'
```

A package can be uninstalled using the `-e` flag:

```
$ nix-env -e thunderbird
```

Finally, you can roll back an undesirable **nix-env** action:

```
1 $ nix-env --rollback
```

nix-env has many more flags. For details, see the `nix-env(1)` manpage or the Nix manual.

Chapter 7. User Management

NixOS supports both declarative and imperative styles of user management. In the declarative style, users are specified in `configuration.nix`. For instance, the following states that a user account named **alice** shall exist:

```
users.users.alice = {
2   isNormalUser = true;
   home = "/home/alice";
4   description = "Alice Foobar";
   extraGroups = [ "wheel" "networkmanager" ];
6   openssh.authorizedKeys.keys = [ "ssh-dss AAAAB3Nza... alice@foobar" ];
};
```

Note that **alice** is a member of the **wheel** and **networkmanager** groups, which allows her to use **sudo** to execute commands as **root** and to configure the network, respectively. Also note the SSH public key that allows remote logins with the corresponding private key. Users created in this way do not have a password by default, so they cannot log in via mechanisms that require a password. However, you can use the **passwd** program to set a password, which is retained across invocations of **nixos-rebuild**.

If you set `users.mutableUsers` to false, then the contents of `/etc/passwd` and `/etc/group` will be congruent to your NixOS configuration. For instance, if you remove a user from `users.users` and run `nixos-rebuild`, the user account will cease to exist. Also, imperative commands for managing users and groups, such as `useradd`, are no longer available. Passwords may still be assigned by setting the user's `hashedPassword` option. A hashed password can be generated using **mkpasswd -m sha-512** after installing the **mkpasswd** package.

A user ID (uid) is assigned automatically. You can also specify a uid manually by adding

```
uid = 1000;
```

to the user specification.

Groups can be specified similarly. The following states that a group named **students** shall exist:

```
users.groups.students.gid = 1000;
```

As with users, the group ID (gid) is optional and will be assigned automatically if it's missing.

In the imperative style, users and groups are managed by commands such as **useradd**, **groupmod** and so on. For instance, to create a user account named **alice**:

```
# useradd -m alice
```

To make all nix tools available to this new user use `'su - USER'` which opens a login shell (==shell that loads the profile) for given user. This will create the `~/nix-defexpr` symlink. So run:

```
1 # su - alice -c "true"
```

The flag `-m` causes the creation of a home directory for the new user, which is generally what you want. The user does not have an initial password and therefore cannot log in. A password can be set using the **passwd** utility:

```
# passwd alice
2 Enter new UNIX password: ***
Retype new UNIX password: ***
```

A user can be deleted using **userdel**:

```
1 # userdel -r alice
```

The flag **-r** deletes the user's home directory. Accounts can be modified using **usermod**. Unix groups can be managed using **groupadd**, **groupmod** and **groupdel**.

Chapter 8. File Systems

Table of Contents

8.1. LUKS-Encrypted File Systems

You can define file systems using the **fileSystems** configuration option. For instance, the following definition causes NixOS to mount the Ext4 file system on device **/dev/disk/by-label/data** onto the mount point **/data**:

```
fileSystems."/data" =
2 { device = "/dev/disk/by-label/data";
  fsType = "ext4";
4 };
```

This will create an entry in **/etc/fstab**, which will generate a corresponding **systemd.mount** unit via **systemd-fstab-generator**. The filesystem will be mounted automatically unless **"noauto"** is present in options. **"noauto"** filesystems can be mounted explicitly using **systemctl** e.g. **systemctl start data.mount**. Mount points are created automatically if they don't already exist. For device, it's best to use the topology-independent device aliases in **/dev/disk/by-label** and **/dev/disk/by-uuid**, as these don't change if the topology changes (e.g. if a disk is moved to another IDE controller).

You can usually omit the file system type (**fsType**), since **mount** can usually detect the type and load the necessary kernel module automatically. However, if the file system is needed at early boot (in the initial ramdisk) and is not **ext2**, **ext3** or **ext4**, then it's best to specify **fsType** to ensure that the kernel module is available.

Note

System startup will fail if any of the filesystems fails to mount, dropping you to the emergency shell. You can make a mount asynchronous and non-critical by adding **options = ["nofail"]**;

8.1. LUKS-Encrypted File Systems

NixOS supports file systems that are encrypted using **LUKS** (Linux Unified Key Setup). For example, here is how you create an encrypted Ext4 file system on the device **/dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d**:

```
# cryptsetup luksFormat /dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d
2 WARNING!
4 =====
This will overwrite data on /dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d
  ↪ irrevocably.
6 Are you sure? (Type uppercase yes): YES
8 Enter LUKS passphrase: ***
Verify passphrase: ***
10 # cryptsetup luksOpen /dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d
  ↪ crypted
12 Enter passphrase for /dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d: ***
14 # mkfs.ext4 /dev/mapper/crypted
```

To ensure that this file system is automatically mounted at boot time as **/**, add the following to **configuration.nix**:

```
boot.initrd.luks.devices.crypted.device =
    ↪ "/dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d";
2 fileSystems."/".device = "/dev/mapper/crypted";
```

Should grub be used as bootloader, and /boot is located on an encrypted partition, it is necessary to add the following grub option:

```
boot.loader.grub.enableCryptodisk = true;
```

8.1.1. FIDO2

NixOS also supports unlocking your LUKS-Encrypted file system using a FIDO2 compatible token. In the following example, we will create a new FIDO2 credential and add it as a new key to our existing device `/dev/sda2`:

```
# export FIDO2_LABEL="/dev/sda2 @ $HOSTNAME"
2 # fido2luks credential "$FIDO2_LABEL"
  f1d00200108b9d6e849a8b388da457688e3dd653b4e53770012d8f28e5d3b269865038c346802f36f3da7278b13ad
4
# fido2luks -i add-key /dev/sda2
  ↪ f1d00200108b9d6e849a8b388da457688e3dd653b4e53770012d8f28e5d3b269865038c346802f36f3da7278b13ad
6 Password:
  Password (again):
8 Old password:
  Old password (again):
10 Added to key to device /dev/sda2, slot: 2
```

To ensure that this file system is decrypted using the FIDO2 compatible key, add the following to `configuration.nix`:

```
boot.initrd.luks.fido2Support = true;
2 boot.initrd.luks.devices."/dev/sda2".fido2.credential =
  ↪ "f1d00200108b9d6e849a8b388da457688e3dd653b4e53770012d8f28e5d3b269865038c346802f36f3da7278b13ad"
```

You can also use the FIDO2 passwordless setup, but for security reasons, you might want to enable it only when your device is PIN protected, such as Trezor.

```
boot.initrd.luks.devices."/dev/sda2".fido2.passwordLess = true;
```

Chapter 9. X Window System

The X Window System (X11) provides the basis of NixOS' graphical user interface. It can be enabled as follows:

```
1 services.xserver.enable = true;
```

The X server will automatically detect and use the appropriate video driver from a set of X.org drivers (such as `vesa` and `intel`). You can also specify a driver manually, e.g.

```
services.xserver.videoDrivers = [ "r128" ];
```

to enable X.org's `xf86-video-r128` driver.

You also need to enable at least one desktop or window manager. Otherwise, you can only log into a plain undecorated `xterm` window. Thus you should pick one or more of the following lines:

```
services.xserver.desktopManager.plasma5.enable = true;
2 services.xserver.desktopManager.xfce.enable = true;
  services.xserver.desktopManager.gnome3.enable = true;
4 services.xserver.desktopManager.mate.enable = true;
  services.xserver.windowManager.xmonad.enable = true;
6 services.xserver.windowManager.twm.enable = true;
```

```

services.xserver.windowManager.icewm.enable = true;
8 services.xserver.windowManager.i3.enable = true;
services.xserver.windowManager.herbstluftwm.enable = true;

```

NixOS's default *display manager* (the program that provides a graphical login prompt and manages the X server) is LightDM. You can select an alternative one by picking one of the following lines:

```

1 services.xserver.displayManager.sddm.enable = true;
services.xserver.displayManager.gdm.enable = true;

```

You can set the keyboard layout (and optionally the layout variant):

```

services.xserver.layout = "de";
2 services.xserver.xkbVariant = "neo";

```

The X server is started automatically at boot time. If you don't want this to happen, you can set:

```

services.xserver.autorun = false;

```

The X server can then be started manually:

```

1 # systemctl start display-manager.service

```

On 64-bit systems, if you want OpenGL for 32-bit programs such as in Wine, you should also set the following:

```

1 hardware.opengl.driSupport32Bit = true;

```

Auto-login

The x11 login screen can be skipped entirely, automatically logging you into your window manager and desktop environment when you boot your computer.

This is especially helpful if you have disk encryption enabled. Since you already have to provide a password to decrypt your disk, entering a second password to login can be redundant.

To enable auto-login, you need to define your default window manager and desktop environment. If you wanted no desktop environment and i3 as your window manager, you'd define:

```

1 services.xserver.displayManager.defaultSession = "none+i3";

```

Every display manager in NixOS supports auto-login, here is an example using lightdm for a user **alice**:

```

services.xserver.displayManager.lightdm.enable = true;
2 services.xserver.displayManager.autoLogin.enable = true;
services.xserver.displayManager.autoLogin.user = "alice";

```

Intel Graphics drivers

There are two choices for Intel Graphics drivers in X.org: **modesetting** (included in the xorg-server itself) and **intel** (provided by the package `xf86-video-intel`).

The default and recommended is **modesetting**. It is a generic driver which uses the kernel mode setting (KMS) mechanism. It supports Glamor (2D graphics acceleration via OpenGL) and is actively maintained but may perform worse in some cases (like in old chipsets).

The second driver, **intel**, is specific to Intel GPUs, but not recommended by most distributions: it lacks several modern features (for example, it doesn't support Glamor) and the package hasn't been officially updated since 2015.

The results vary depending on the hardware, so you may have to try both drivers. Use the option `services.xserver.videoDrivers` to set one. The recommended configuration for modern systems is:

```

services.xserver.videoDrivers = [ "modesetting" ];
2 services.xserver.useGlamor = true;

```

If you experience screen tearing no matter what, this configuration was reported to resolve the issue:

```
1 services.xserver.videoDrivers = [ "intel" ];
2 services.xserver.deviceSection = ''
   Option "DRI" "2"
4   Option "TearFree" "true"
   '';
```

Note that this will likely downgrade the performance compared to `modesetting` or `intel` with DRI 3 (default).

Proprietary NVIDIA drivers

NVIDIA provides a proprietary driver for its graphics cards that has better 3D performance than the X.org drivers. It is not enabled by default because it's not free software. You can enable it as follows:

```
services.xserver.videoDrivers = [ "nvidia" ];
```

Or if you have an older card, you may have to use one of the legacy drivers:

```
1 services.xserver.videoDrivers = [ "nvidiaLegacy390" ];
services.xserver.videoDrivers = [ "nvidiaLegacy340" ];
3 services.xserver.videoDrivers = [ "nvidiaLegacy304" ];
services.xserver.videoDrivers = [ "nvidiaLegacy173" ];
```

You may need to reboot after enabling this driver to prevent a clash with other kernel modules.

Proprietary AMD drivers

AMD provides a proprietary driver for its graphics cards that has better 3D performance than the X.org drivers. It is not enabled by default because it's not free software. You can enable it as follows:

```
services.xserver.videoDrivers = [ "ati_unfree" ];
```

You will need to reboot after enabling this driver to prevent a clash with other kernel modules.

Note

For recent AMD GPUs you most likely want to keep either the defaults or `"amdgpu"` (both free).

Touchpads

Support for Synaptics touchpads (found in many laptops such as the Dell Latitude series) can be enabled as follows:

```
services.xserver.libinput.enable = true;
```

The driver has many options (see Appendix A, *Configuration Options*). For instance, the following disables tap-to-click behavior:

```
1 services.xserver.libinput.tapping = false;
```

Note: the use of `services.xserver.synaptics` is deprecated since NixOS 17.09.

GTK/Qt themes

GTK themes can be installed either to user profile or system-wide (via `environment.systemPackages`). To make Qt 5 applications look similar to GTK2 ones, you can install `qt5.qtbase.gtk` package into your system environment. It should work for all Qt 5 library versions.

Custom XKB layouts

It is possible to install custom XKB keyboard layouts using the option `services.xserver.extraLayouts`. As a first example, we are going to create a layout based on the basic US layout, with an additional layer to type some greek symbols by pressing the right-alt key.

To do this we are going to create a `us-greek` file with a `xkb_symbols` section.

```
xkb_symbols "us-greek"
2 {
    include "us(basic)"           // includes the base US keys
4   include "level3(ralt_switch)" // configures right alt as a third level switch

6   key <LatA> { [ a, A, Greek_alpha ] };
    key <LatB> { [ b, B, Greek_beta  ] };
8   key <LatG> { [ g, G, Greek_gamma ] };
    key <LatD> { [ d, D, Greek_delta ] };
10  key <LatZ> { [ z, Z, Greek_zeta  ] };
};
```

To install the layout, the filepath, a description and the list of languages must be given:

```
1 services.xserver.extraLayouts.us-greek = {
    description = "US layout with alt-gr greek";
3   languages   = [ "eng" ];
    symbolsFile = /path/to/us-greek;
5 }
```

Note

The name should match the one given to the `xkb_symbols` block.

The layout should now be installed and ready to use: try it by running `setxkbmap us-greek` and type `<alt>+a`. To change the default the usual `services.xserver.layout` option can still be used.

A layout can have several other components besides `xkb_symbols`, for example we will define new keycodes for some multimedia key and bind these to some symbol.

Use the `xev` utility from `pkgs.xorg.xev` to find the codes of the keys of interest, then create a `media-key` file to hold the keycodes definitions

```
xkb_keycodes "media"
2 {
    <volUp>    = 123;
4   <volDown> = 456;
};
```

Now use the newly define keycodes in `media-sym`:

```
xkb_symbols "media"
2 {
    key.type = "ONE_LEVEL";
4   key <volUp>    { [ XF86AudioLowerVolume ] };
    key <volDown> { [ XF86AudioRaiseVolume ] };
6 }
```

As before, to install the layout do

```
services.xserver.extraLayouts.media = {
2   description = "Multimedia keys remapping";
    languages   = [ "eng" ];
4   symbolsFile = /path/to/media-key;
```

```
keycodesFile = /path/to/media-sym;  
6 };
```

Note

The function `pkgs.writeText <filename> <content>` can be useful if you prefer to keep the layout definitions inside the NixOS configuration.

Unfortunately, the Xorg server does not (currently) support setting a keymap directly but relies instead on XKB rules to select the matching components (keycodes, types, ...) of a layout. This means that components other than symbols won't be loaded by default. As a workaround, you can set the keymap using `setxkbmap` at the start of the session with:

```
services.xserver.displayManager.sessionCommands = "setxkbmap -keycodes media";
```

If you are manually starting the X server, you should set the argument `-xkbdir /etc/X11/xkb`, otherwise X won't find your layout files. For example with `xinit` run

```
$ xinit -- -xkbdir /etc/X11/xkb
```

To learn how to write layouts take a look at the XKB documentation . More example layouts can also be found here .

Chapter 10. GPU acceleration

Table of Contents

10.1. OpenCL

10.2. Vulkan

10.3. Common issues

NixOS provides various APIs that benefit from GPU hardware acceleration, such as VA-API and VDPAU for video playback; OpenGL and Vulkan for 3D graphics; and OpenCL for general-purpose computing. This chapter describes how to set up GPU hardware acceleration (as far as this is not done automatically) and how to verify that hardware acceleration is indeed used.

Most of the aforementioned APIs are agnostic with regards to which display server is used. Consequently, these instructions should apply both to the X Window System and Wayland compositors.

10.1. OpenCL

OpenCL is a general compute API. It is used by various applications such as Blender and Darktable to accelerate certain operations.

OpenCL applications load drivers through the *Installable Client Driver* (ICD) mechanism. In this mechanism, an ICD file specifies the path to the OpenCL driver for a particular GPU family. In NixOS, there are two ways to make ICD files visible to the ICD loader. The first is through the `OCL_ICD_VENDORS` environment variable. This variable can contain a directory which is scanned by the ICD loader for ICD files. For example:

```
$ export \  
2 OCL_ICD_VENDORS=`nix-build '<nixpkgs>' --no-out-link -A  
  ↪ rocm-opencl-icd`/etc/OpenCL/vendors/
```

The second mechanism is to add the OpenCL driver package to `hardware.opengl.extraPackages`. This links the ICD file under `/run/opengl-driver`, where it will be visible to the ICD loader.

The proper installation of OpenCL drivers can be verified through the `clinfo` command of the `clinfo` package. This command will report the number of hardware devices that is found and give detailed information for each device:

```
$ clinfo | head -n3
2 Number of platforms      1
  Platform Name            AMD Accelerated Parallel Processing
4 Platform Vendor          Advanced Micro Devices, Inc.
```

10.1.1. AMD

Modern AMD Graphics Core Next (GCN) GPUs are supported through the rocm-opengl-icd package. Adding this package to `hardware.opengl.extraPackages` enables OpenCL support:

```
hardware.opengl.extraPackages = [
2   rocm-opengl-icd
];
```

10.1.2. Intel

Intel Gen8 and later GPUs are supported by the Intel NEO OpenCL runtime that is provided by the intel-compute-runtime package. For Gen7 GPUs, the deprecated Beignet runtime can be used, which is provided by the beignet package. The proprietary Intel OpenCL runtime, in the intel-ocl package, is an alternative for Gen7 GPUs.

The intel-compute-runtime, beignet, or intel-ocl package can be added to `hardware.opengl.extraPackages` to enable OpenCL support. For example, for Gen8 and later GPUs, the following configuration can be used:

```
hardware.opengl.extraPackages = [
2   intel-compute-runtime
];
```

10.2. Vulkan

Vulkan is a graphics and compute API for GPUs. It is used directly by games or indirectly through compatibility layers like DXVK.

By default, if `hardware.opengl.driSupport` is enabled, mesa is installed and provides Vulkan for supported hardware.

Similar to OpenCL, Vulkan drivers are loaded through the *Installable Client Driver* (ICD) mechanism. ICD files for Vulkan are JSON files that specify the path to the driver library and the supported Vulkan version. All successfully loaded drivers are exposed to the application as different GPUs. In NixOS, there are two ways to make ICD files visible to Vulkan applications: an environment variable and a module option.

The first option is through the `VK_ICD_FILENAMES` environment variable. This variable can contain multiple JSON files, separated by `..`. For example:

```
$ export \
2   VK_ICD_FILENAMES=`nix-build '<nixpkgs>' --no-out-link -A
   ↳ amdvlk`/share/vulkan/icd.d/amd_icd64.json
```

The second mechanism is to add the Vulkan driver package to `hardware.opengl.extraPackages`. This links the ICD file under `/run/opengl-driver`, where it will be visible to the ICD loader.

The proper installation of Vulkan drivers can be verified through the **vulkaninfo** command of the vulkan-tools package. This command will report the hardware devices and drivers found, in this example output amdvlk and radv:

```
$ vulkaninfo | grep GPU
2   GPU id   : 0 (Unknown AMD GPU)
   GPU id   : 1 (AMD RADV NAVI10 (LLVM 9.0.1))
4   ...
GPU0:
6   deviceType = PHYSICAL_DEVICE_TYPE_DISCRETE_GPU
   deviceName  = Unknown AMD GPU
```

```
8 GPU1:
    deviceType      = PHYSICAL_DEVICE_TYPE_DISCRETE_GPU
```

A simple graphical application that uses Vulkan is **vkcube** from the **vulkan-tools** package.

10.2.1. AMD

Modern AMD Graphics Core Next (GCN) GPUs are supported through either **radv**, which is part of **mesa**, or the **amdvk** package. Adding the **amdvk** package to **hardware.opengl.extraPackages** makes both drivers available for applications and lets them choose. A specific driver can be forced as follows:

```
hardware.opengl.extraPackages = [
2   amdvlk
];
4
# For amdvlk
6 environment.variables.VK_ICD_FILENAMES =
   "/run/opengl-driver/share/vulkan/icd.d/amd_icd64.json";
8 # For radv
   environment.variables.VK_ICD_FILENAMES =
10  "/run/opengl-driver/share/vulkan/icd.d/radeon_icd.x86_64.json";
```

10.3. Common issues

10.3.1. User permissions

Except where noted explicitly, it should not be necessary to adjust user permissions to use these acceleration APIs. In the default configuration, GPU devices have world-read/write permissions (**/dev/dri/renderD***) or are tagged as **uaccess** (**/dev/dri/card***). The access control lists of devices with the **uaccess** tag will be updated automatically when a user logs in through **systemd-logind**. For example, if the user *jane* is logged in, the access control list should look as follows:

```
$ getfacl /dev/dri/card0
2 # file: dev/dri/card0
  # owner: root
  # group: video
  user::rw-
6 user:jane:rw-
  group::rw-
8 mask::rw-
  other::---
```

If you disabled (this functionality of) **systemd-logind**, you may need to add the user to the **video** group and log in again.

10.3.2. Mixing different versions of nixpkgs

The *Installable Client Driver* (ICD) mechanism used by OpenCL and Vulkan loads runtimes into its address space using **dlopen**. Mixing an ICD loader mechanism and runtimes from different version of **nixpkgs** may not work. For example, if the ICD loader uses an older version of **glibc** than the runtime, the runtime may not be loadable due to missing symbols. Unfortunately, the loader will generally be quiet about such issues.

If you suspect that you are running into library version mismatches between an ICD loader and a runtime, you could run an application with the **LD_DEBUG** variable set to get more diagnostic information. For example, OpenCL can be tested with **LD_DEBUG=files cinfo**, which should report missing symbols.

Chapter 11. Xfce Desktop Environment

To enable the Xfce Desktop Environment, set

```
services.xserver.desktopManager.xfce.enable = true;
2 services.xserver.displayManager.defaultSession = "xfce";
```

Optionally, *picom* can be enabled for nice graphical effects, some example settings:

```
services.picom = {
2   enable = true;
   fade = true;
4   inactiveOpacity = 0.9;
   shadow = true;
6   fadeDelta = 4;
};
```

Some Xfce programs are not installed automatically. To install them manually (system wide), put them into your `environment.systemPackages` from `pkgs.xfce`.

Thunar Plugins

If you'd like to add extra plugins to Thunar, add them to `services.xserver.desktopManager.xfce.thunarPlugins`. You shouldn't just add them to `environment.systemPackages`.

Troubleshooting

Even after enabling `udisks2`, volume management might not work. Thunar and/or the desktop takes time to show up. Thunar will spit out this kind of message on start (look at `journalctl --user -b`).

```
Thunar:2410): GVFS-RemoteVolumeMonitor-WARNING **: remote volume monitor with dbus
↳ name org.gtk.Private.UDisks2VolumeMonitor is not supported
```

This is caused by some needed GNOME services not running. This is all fixed by enabling "Launch GNOME services on startup" in the Advanced tab of the Session and Startup settings panel. Alternatively, you can run this command to do the same thing.

```
1 $ xfconf-query -c xfce4-session -p /compat/LaunchGNOME -s true
```

A log-out and re-log will be needed for this to take effect.

Chapter 12. Networking

Table of Contents

- 12.1. NetworkManager
- 12.2. Secure Shell Access
- 12.3. IPv4 Configuration
- 12.4. IPv6 Configuration
- 12.5. Firewall
- 12.6. Wireless Networks
- 12.7. Ad-Hoc Configuration

This section describes how to configure networking components on your NixOS machine.

12.1. NetworkManager

To facilitate network configuration, some desktop environments use NetworkManager. You can enable NetworkManager by setting:

```
1 networking.networkmanager.enable = true;
```

some desktop managers (e.g., GNOME) enable NetworkManager automatically for you.

All users that should have permission to change network settings must belong to the `networkmanager` group:

```
users.users.alice.extraGroups = [ "networkmanager" ];
```

NetworkManager is controlled using either `nmcli` or `nmtui` (curses-based terminal user interface). See their manual pages for details on their usage. Some desktop environments (GNOME, KDE) have their own configuration tools for NetworkManager. On XFCE, there is no configuration tool for NetworkManager by default: by enabling `programs.nm-applet.enable`, the graphical applet will be installed and will launch automatically when the graphical session is started.

Note

`networking.networkmanager` and `networking.wireless` (WPA Supplicant) can be used together if desired. To do this you need to instruct NetworkManager to ignore those interfaces like:

```
networking.networkmanager.unmanaged = [  
2  "*" "except:type:wwan" "except:type:gsm"  
  ];
```

Refer to the option description for the exact syntax and references to external documentation.

12.2. Secure Shell Access

Secure shell (SSH) access to your machine can be enabled by setting:

```
1 services.openssh.enable = true;
```

By default, root logins using a password are disallowed. They can be disabled entirely by setting `services.openssh.permitRootLogin` to "no".

You can declaratively specify authorised RSA/DSA public keys for a user as follows:

```
users.users.alice.openssh.authorizedKeys.keys =  
2  [ "ssh-dss AAAAB3NzaC1kc3MAAACBAPIkGWVEt4..." ];
```

12.3. IPv4 Configuration

By default, NixOS uses DHCP (specifically, `dhcpcd`) to automatically configure network interfaces. However, you can configure an interface manually as follows:

```
networking.interfaces.eth0.ipv4.addresses = [ {  
2  address = "192.168.1.2";  
  prefixLength = 24;  
4 } ];
```

Typically you'll also want to set a default gateway and set of name servers:

```
networking.defaultGateway = "192.168.1.1";  
2 networking.nameservers = [ "8.8.8.8" ];
```

Note

Statically configured interfaces are set up by the `systemd` service `interface-name-cfg.service`. The default gateway and name server configuration is performed by `network-setup.service`.

The host name is set using `networking.hostName`:

```
networking.hostName = "cartman";
```

The default host name is `nixos`. Set it to the empty string ("") to allow the DHCP server to provide the host name.

12.4. IPv6 Configuration

IPv6 is enabled by default. Stateless address autoconfiguration is used to automatically assign IPv6 addresses to all interfaces. You can disable IPv6 support globally by setting:

```
networking.enableIPv6 = false;
```

You can disable IPv6 on a single interface using a normal `sysctl` (in this example, we use interface `eth0`):

```
boot.kernel.sysctl."net.ipv6.conf.eth0.disable_ipv6" = true;
```

As with IPv4 networking interfaces are automatically configured via DHCPv6. You can configure an interface manually:

```
1 networking.interfaces.eth0.ipv6.addresses = [ {  
    address = "fe00:aa:bb:cc::2";  
3     prefixLength = 64;  
    } ];
```

For configuring a gateway, optionally with explicitly specified interface:

```
networking.defaultGateway6 = {  
2     address = "fe00::1";  
    interface = "enp0s3";  
4 };
```

See Section 12.3, “IPv4 Configuration” for similar examples and additional information.

12.5. Firewall

NixOS has a simple stateful firewall that blocks incoming connections and other unexpected packets. The firewall applies to both IPv4 and IPv6 traffic. It is enabled by default. It can be disabled as follows:

```
networking.firewall.enable = false;
```

If the firewall is enabled, you can open specific TCP ports to the outside world:

```
1 networking.firewall.allowedTCPPorts = [ 80 443 ];
```

Note that TCP port 22 (ssh) is opened automatically if the SSH daemon is enabled (`services.openssh.enable = true`). UDP ports can be opened through `networking.firewall.allowedUDPPorts`.

To open ranges of TCP ports:

```
networking.firewall.allowedTCPPortRanges = [  
2     { from = 4000; to = 4007; }  
    { from = 8000; to = 8010; }  
4 ];
```

Similarly, UDP port ranges can be opened through `networking.firewall.allowedUDPPortRanges`.

12.6. Wireless Networks

For a desktop installation using NetworkManager (e.g., GNOME), you just have to make sure the user is in the `networkmanager` group and you can skip the rest of this section on wireless networks.

NixOS will start `wpa_supplicant` for you if you enable this setting:

```
networking.wireless.enable = true;
```

NixOS lets you specify networks for `wpa_supplicant` declaratively:

```

1 networking.wireless.networks = {
    echelon = {                # SSID with no spaces or special characters
3      psk = "abcdefgh";
    };
5    "echelon's AP" = {        # SSID with spaces and/or special characters
      psk = "ijklmnop";
7    };
    echelon = {                # Hidden SSID
8      hidden = true;
      psk = "qrstuvwxyz";
11   };
    free.wifi = {};            # Public wireless network
13 };

```

Be aware that keys will be written to the nix store in plaintext! When no networks are set, it will default to using a configuration file at `/etc/wpa_supplicant.conf`. You should edit this file yourself to define wireless networks, WPA keys and so on (see `wpa_supplicant.conf(5)`).

If you are using WPA2 you can generate pskRaw key using `wpa_passphrase`:

```

$ wpa_passphrase ESSID PSK
2 network={
    ssid="echelon"
4    #psk="abcdefgh"
    psk=dca6d6ed41f4ab5a984c9f55f6f66d4efdc720ebf66959810f4329bb391c5435
6 }

```

```

networking.wireless.networks = {
2   echelon = {
      pskRaw = "dca6d6ed41f4ab5a984c9f55f6f66d4efdc720ebf66959810f4329bb391c5435";
4   };
}

```

or you can use it to directly generate the `wpa_supplicant.conf`:

```
# wpa_passphrase ESSID PSK > /etc/wpa_supplicant.conf
```

After you have edited the `wpa_supplicant.conf`, you need to restart the `wpa_supplicant` service.

```
# systemctl restart wpa_supplicant.service
```

12.7. Ad-Hoc Configuration

You can use `networking.localCommands` to specify shell commands to be run at the end of `network-setup.service`. This is useful for doing network configuration not covered by the existing NixOS modules. For instance, to statically configure an IPv6 address:

```

networking.localCommands =
2   ''
      ip -6 addr add 2001:610:685:1::1/64 dev eth0
4   '';

```

Chapter 13. Linux Kernel

Table of Contents

13.1. Customize your kernel

13.2. Developing kernel modules

You can override the Linux kernel and associated packages using the option `boot.kernelPackages`. For instance, this selects the Linux 3.10 kernel:

```
boot.kernelPackages = pkgs.linuxPackages_3_10;
```

Note that this not only replaces the kernel, but also packages that are specific to the kernel version, such as the NVIDIA video drivers. This ensures that driver packages are consistent with the kernel.

The default Linux kernel configuration should be fine for most users. You can see the configuration of your current kernel with the following command:

```
1 zcat /proc/config.gz
```

If you want to change the kernel configuration, you can use the `packageOverrides` feature (see Section 6.1.1, “Customising Packages”). For instance, to enable support for the kernel debugger KGDB:

```
nixpkgs.config.packageOverrides = pkgs:
2   { linux_3_4 = pkgs.linux_3_4.override {
      extraConfig =
4       ''
          KGDB y
6       ''
    };
8   };
```

`extraConfig` takes a list of Linux kernel configuration options, one per line. The name of the option should not include the prefix `CONFIG_`. The option value is typically `y`, `n` or `m` (to build something as a kernel module).

Kernel modules for hardware devices are generally loaded automatically by **udev**. You can force a module to be loaded via `boot.kernelModules`, e.g.

```
boot.kernelModules = [ "fuse" "kvm-intel" "coretemp" ];
```

If the module is required early during the boot (e.g. to mount the root file system), you can use `boot.initrd.kernelModules`:

```
boot.initrd.kernelModules = [ "cifs" ];
```

This causes the specified modules and their dependencies to be added to the initial ramdisk.

Kernel runtime parameters can be set through `boot.kernel.sysctl`, e.g.

```
boot.kernel.sysctl."net.ipv4.tcp_keepalive_time" = 120;
```

sets the kernel’s TCP keepalive time to 120 seconds. To see the available parameters, run **sysctl -a**.

13.1. Customize your kernel

The first step before compiling the kernel is to generate an appropriate `.config` configuration. Either you pass your own config via the `configfile` setting of `linuxManualConfig`:

```
custom-kernel = super.linuxManualConfig {
2   inherit (super) stdenv hostPlatform;
   inherit (linux_4_9) src;
4   version = "${linux_4_9.version}-custom";

6   configfile = /home/me/my_kernel_config;
   allowImportFromDerivation = true;
8   };
```

You can edit the config with this snippet (by default **make menuconfig** won’t work out of the box on nixos):

```

1  nix-shell -E 'with import <nixpkgs> {}; kernelToOverride.overrideAttrs (o:
    ↪ {nativeBuildInputs=o.nativeBuildInputs ++ [ pkgconfig ncurses ];})'

```

or you can let nixpkgs generate the configuration. Nixpkgs generates it via answering the interactive kernel utility **make config**. The answers depend on parameters passed to `pkgs/os-specific/linux/kernel/generic.nix` (which you can influence by overriding `extraConfig`, `autoModules`, `modDirVersion`, `preferBuiltin`, `↪ extraConfig`).

```

    mptcp93.override ({
2      name="mptcp-local";

4      ignoreConfigErrors = true;
      autoModules = false;
6      kernelPreferBuiltin = true;

8      enableParallelBuilding = true;

10     extraConfig = ''
        DEBUG_KERNEL y
12     FRAME_POINTER y
        KGDB y
14     KGDB_SERIAL_CONSOLE y
        DEBUG_INFO y
16     '';
    });

```

13.2. Developing kernel modules

When developing kernel modules it's often convenient to run edit-compile-run loop as quickly as possible. See below snippet as an example of developing `mellanox` drivers.

```

$ nix-build '<nixpkgs>' -A linuxPackages.kernel.dev
2 $ nix-shell '<nixpkgs>' -A linuxPackages.kernel
$ unpackPhase
4 $ cd linux-*
$ make -C $dev/lib/modules/*/build M=$(pwd)/drivers/net/ethernet/mellanox modules
6 # insmod ./drivers/net/ethernet/mellanox/mlx5/core/mlx5_core.ko

```

Chapter 14. Pantheon Desktop

Table of Contents

14.1. Enabling Pantheon

14.2. Wingpanel and Switchboard plugins

14.3. FAQ

Pantheon is the desktop environment created for the elementary OS distribution. It is written from scratch in Vala, utilizing GNOME technologies with GTK 3 and Granite.

14.1. Enabling Pantheon

All of Pantheon is working in NixOS and the applications should be available, aside from a few exceptions. To enable Pantheon, set

```
services.xserver.desktopManager.pantheon.enable = true;
```

This automatically enables LightDM and Pantheon's LightDM greeter. If you'd like to disable this, set

```
1 services.xserver.displayManager.lightdm.greeters.pantheon.enable = false;
  services.xserver.displayManager.lightdm.enable = false;
```

but please be aware using Pantheon without LightDM as a display manager will break screenlocking from the UI. The NixOS module for Pantheon installs all of Pantheon's default applications. If you'd like to not install Pantheon's apps, set

```
services.pantheon.apps.enable = false;
```

You can also use `environment.pantheon.excludePackages` to remove any other app (like geary).

14.2. Wingpanel and Switchboard plugins

Wingpanel and Switchboard work differently than they do in other distributions, as far as using plugins. You cannot install a plugin globally (like with `environment.systemPackages`) to start using it. You should instead be using the following options:

- `services.xserver.desktopManager.pantheon.extraWingpanelIndicators`
- `services.xserver.desktopManager.pantheon.extraSwitchboardPlugs`

to configure the programs with plugs or indicators.

The difference in NixOS is both these programs are patched to load plugins from a directory that is the value of an environment variable. All of which is controlled in Nix. If you need to configure the particular packages manually you can override the packages like:

```
wingpanel-with-indicators.override {
2   indicators = [
      pkgs.some-special-indicator
4   ];
};
6
switchboard-with-plugs.override {
8   plugs = [
      pkgs.some-special-plug
10  ];
};
```

please note that, like how the NixOS options describe these as extra plugins, this would only add to the default plugins included with the programs. If for some reason you'd like to configure which plugins to use exactly, both packages have an argument for this:

```
1 wingpanel-with-indicators.override {
    useDefaultIndicators = false;
3   indicators = specialListOfIndicators;
};
5
switchboard-with-plugs.override {
7   useDefaultPlugs = false;
    plugs = specialListOfPlugs;
9 };
```

this could be most useful for testing a particular plug-in in isolation.

14.3. FAQ

I have switched from a different desktop and Pantheon's theming looks messed up. Open Switchboard and go to: Administration → About → Restore Default Settings → Restore Settings. This will reset any dconf settings to their Pantheon defaults. Note this could reset certain GNOME specific preferences if that desktop was used prior.

I cannot enable both GNOME 3 and Pantheon. This is a known issue and there is no known workaround.

Does AppCenter work, or is it available? AppCenter has been available since 20.03, but it is of little use. This is because there is no functioning PackageKit backend for Nix 2.0. In the near future you will be able to install Flatpak applications from AppCenter on NixOS. See this issue.

Chapter 15. Matomo

Table of Contents

15.1. Database Setup

15.2. Archive Processing

15.3. Backup

15.4. Issues

15.5. Using other Web Servers than nginx

Matomo is a real-time web analytics application. This module configures php-fpm as backend for Matomo, optionally configuring an nginx vhost as well.

An automatic setup is not supported by Matomo, so you need to configure Matomo itself in the browser-based Matomo setup.

15.1. Database Setup

You also need to configure a MariaDB or MySQL database and -user for Matomo yourself, and enter those credentials in your browser. You can use passwordless database authentication via the UNIX_SOCKET authentication plugin with the following SQL commands:

```
1 # For MariaDB
  INSTALL PLUGIN unix_socket SONAME 'auth_socket';
3 CREATE DATABASE matomo;
  CREATE USER 'matomo'@'localhost' IDENTIFIED WITH unix_socket;
5 GRANT ALL PRIVILEGES ON matomo.* TO 'matomo'@'localhost';

7 # For MySQL
  INSTALL PLUGIN auth_socket SONAME 'auth_socket.so';
9 CREATE DATABASE matomo;
  CREATE USER 'matomo'@'localhost' IDENTIFIED WITH auth_socket;
11 GRANT ALL PRIVILEGES ON matomo.* TO 'matomo'@'localhost';
```

Then fill in `matomo` as database user and database name, and leave the password field blank. This authentication works by allowing only the `matomo` unix user to authenticate as the `matomo` database user (without needing a password), but no other users. For more information on passwordless login, see https://mariadb.com/kb/en/mariadb/unix_socket-authentication-plugin/.

Of course, you can use password based authentication as well, e.g. when the database is not on the same host.

15.2. Archive Processing

This module comes with the systemd service `matomo-archive-processing.service` and a timer that automatically triggers archive processing every hour. This means that you can safely disable browser triggers for Matomo archiving at **Administration > System > General Settings**.

With automatic archive processing, you can now also enable to delete old visitor logs at **Administration > System > Privacy**, but make sure that you run `systemctl start matomo-archive-processing.service` at least once without errors if you have already collected data before, so that the reports get archived before the source data gets deleted.

15.3. Backup

You only need to take backups of your MySQL database and the `/var/lib/matomo/config/config.ini.php` file. Use a user in the `matomo` group or `root` to access the file. For more information, see https://matomo.org/faq/how-to-install/faq_138/.

15.4. Issues

- Matomo will warn you that the JavaScript tracker is not writable. This is because it's located in the read-only nix store. You can safely ignore this, unless you need a plugin that needs JavaScript tracker access.

15.5. Using other Web Servers than nginx

You can use other web servers by forwarding calls for `index.php` and `piwik.php` to the `services.phpfpm.pools.<name>.socket` fastcgi unix socket. You can use the nginx configuration in the module code as a reference to what else should be configured.

Chapter 16. Nextcloud

Table of Contents

16.1. Basic usage

16.2. Pitfalls

16.3. Using an alternative webserver as reverse-proxy (e.g. `httpd`)

16.4. Maintainer information

Nextcloud is an open-source, self-hostable cloud platform. The server setup can be automated using `services.nextcloud`. A desktop client is packaged at `pkgs.nextcloud-client`.

The current default by NixOS is `nextcloud19` though it's recommended to upgrade to the latest version, `nextcloud21`. Please note that it's necessary to install `nextcloud20` first!

16.1. Basic usage

Nextcloud is a PHP-based application which requires an HTTP server (`services.nextcloud` optionally supports `services.nginx`) and a database (it's recommended to use `services.postgresql`).

A very basic configuration may look like this:

```
1 { pkgs, ... }:
2 {
3     services.nextcloud = {
4         enable = true;
5         hostName = "nextcloud.tld";
6         config = {
7             dbtype = "pgsql";
8             dbuser = "nextcloud";
9             dbhost = "/run/postgresql"; # nextcloud will add /.s.PGSQL.5432 by itself
10            dbname = "nextcloud";
11            adminpassFile = "/path/to/admin-pass-file";
12            adminuser = "root";
13        };
14    };
15
16    services.postgresql = {
17        enable = true;
18        ensureDatabases = [ "nextcloud" ];
19        ensureUsers = [
```

```

20     { name = "nextcloud";
      ensurePermissions."DATABASE nextcloud" = "ALL PRIVILEGES";
22     }
    ];
24 };

26 # ensure that postgres is running before running the setup
systemd.services."nextcloud-setup" = {
28     requires = ["postgresql.service"];
    after = ["postgresql.service"];
30 };

32 networking.firewall.allowedTCPPorts = [ 80 443 ];
}

```

The `hostName` option is used internally to configure an HTTP server using PHP-FPM and `nginx`. The `config` attribute set is used by the imperative installer and all values are written to an additional file to ensure that changes can be applied by changing the module's options.

In case the application serves multiple domains (those are checked with `$_SERVER['HTTP_HOST']`) it's needed to add them to `services.nextcloud.config.extraTrustedDomains`.

Auto updates for Nextcloud apps can be enabled using `services.nextcloud.autoUpdateApps`.

16.2. Pitfalls

Unfortunately Nextcloud appears to be very stateful when it comes to managing its own configuration. The config file lives in the home directory of the `nextcloud` user (by default `/var/lib/nextcloud/config/config.php`) and is also used to track several states of the application (e.g. whether installed or not).

All configuration parameters are also stored in `/var/lib/nextcloud/config/override.config.php` which is generated by the module and linked from the store to ensure that all values from `config.php` can be modified by the module. However `config.php` manages the application's state and shouldn't be touched manually because of that.

Warning

Don't delete `config.php`! This file tracks the application's state and a deletion can cause unwanted side-effects!

Warning

Don't rerun `nextcloud-occ maintenance:install`! This command tries to install the application and can cause unwanted side-effects!

Nextcloud doesn't allow to move more than one major-version forward. If you're e.g. on `v16`, you cannot upgrade to `v18`, you need to upgrade to `v17` first. This is ensured automatically as long as the `stateVersion` is declared properly. In that case the oldest version available (one major behind the one from the previous NixOS release) will be selected by default and the module will generate a warning that reminds the user to upgrade to latest Nextcloud *after* that deploy.

16.3. Using an alternative webserver as reverse-proxy (e.g. `httpd`)

By default, `nginx` is used as reverse-proxy for nextcloud. However, it's possible to use e.g. `httpd` by explicitly disabling `nginx` using `services.nginx.enable` and fixing the settings `listen.owner` & `listen.group` in the corresponding `phpfpm` pool.

An exemplary configuration may look like this:

```

{ config, lib, pkgs, ... }: {
2   services.nginx.enable = false;
   services.nextcloud = {
4     enable = true;
     hostName = "localhost";

```

```

6      /* further, required options */
8  };
  services.phpfpm.pools.nextcloud.settings = {
10    "listen.owner" = config.services.httpd.user;
    "listen.group" = config.services.httpd.group;
12  };
  services.httpd = {
14    enable = true;
    adminAddr = "webmaster@localhost";
16    extraModules = [ "proxy_fcgi" ];
    virtualHosts."localhost" = {
18      documentRoot = config.services.nextcloud.package;
      extraConfig = ''
20        <Directory "${config.services.nextcloud.package}">
          <FilesMatch "\.php$">
22            <If "-f %{REQUEST_FILENAME}">
              SetHandler
                ↪ "proxy:unix:${config.services.phpfpm.pools.nextcloud.socket}|fcgi://local
24            </If>
          </FilesMatch>
26          <IfModule mod_rewrite.c>
            RewriteEngine On
28            RewriteBase /
            RewriteRule ^index\.php$ - [L]
30            RewriteCond %{REQUEST_FILENAME} !-f
            RewriteCond %{REQUEST_FILENAME} !-d
32            RewriteRule . /index.php [L]
          </IfModule>
34          DirectoryIndex index.php
          Require all granted
36          Options +FollowSymLinks
        </Directory>
38      '';
    };
40  };
}

```

16.4. Maintainer information

As stated in the previous paragraph, we must provide a clean upgrade-path for Nextcloud since it cannot move more than one major version forward on a single upgrade. This chapter adds some notes how Nextcloud updates should be rolled out in the future.

While minor and patch-level updates are no problem and can be done directly in the package-expression (and should be backported to supported stable branches after that), major-releases should be added in a new attribute (e.g. Nextcloud v19.0.0 should be available in `nixpkgs` as `pkgs.nextcloud19`). To provide simple upgrade paths it's generally useful to backport those as well to stable branches. As long as the package-default isn't altered, this won't break existing setups. After that, the versioning-warning in the `nextcloud`-module should be updated to make sure that the package-option selects the latest version on fresh setups.

If major-releases will be abandoned by upstream, we should check first if those are needed in NixOS for a safe upgrade-path before removing those. In that case we should keep those packages, but mark them as insecure in an expression like this (in `<nixpkgs/pkgs/servers/nextcloud/default.nix>`):

```

/* ... */
2 {
  nextcloud17 = generic {

```

```

4     version = "17.0.x";
      sha256 = "0000000000000000000000000000000000000000000000000000000000000000";
6     eol = true;
    };
8 }

```

Ideally we should make sure that it's possible to jump two NixOS versions forward: i.e. the warnings and the logic in the module should guard a user to upgrade from a Nextcloud on e.g. 19.09 to a Nextcloud on 20.09.

Chapter 17. Jitsi Meet

Table of Contents

17.1. Basic usage

17.2. Configuration

With Jitsi Meet on NixOS you can quickly configure a complete, private, self-hosted video conferencing solution.

17.1. Basic usage

A minimal configuration using Let's Encrypt for TLS certificates looks like this:

```

{
2  services.jitsi-meet = {
      enable = true;
4      hostname = "jitsi.example.com";
    };
6  services.jitsi-videobridge.openFirewall = true;
  networking.firewall.allowedTCPPorts = [ 80 443 ];
8  security.acme.email = "me@example.com";
  security.acme.acceptTerms = true;
10 }

```

17.2. Configuration

Here is the minimal configuration with additional configurations:

```

{
2  services.jitsi-meet = {
      enable = true;
4      hostname = "jitsi.example.com";
      config = {
6        enableWelcomePage = false;
        prejoinPageEnabled = true;
8        defaultLang = "fi";
      };
10   interfaceConfig = {
        SHOW_JITSI_WATERMARK = false;
12     SHOW_WATERMARK_FOR_GUESTS = false;
    };
14 };
  services.jitsi-videobridge.openFirewall = true;
16  networking.firewall.allowedTCPPorts = [ 80 443 ];
  security.acme.email = "me@example.com";
18  security.acme.acceptTerms = true;
}

```

Chapter 18. Grocy

Table of Contents

18.1. Basic usage

18.2. Settings

Grocy is a web-based self-hosted groceries & household management solution for your home.

18.1. Basic usage

A very basic configuration may look like this:

```
1 { pkgs, ... }:  
  {  
3   services.grocy = {  
     enable = true;  
5     hostname = "grocy.tld";  
   };  
7 }
```

This configures a simple vhost using nginx which listens to `grocy.tld` with fully configured ACME/LE (this can be disabled by setting `services.grocy.nginx.enableSSL` to `false`). After the initial setup the credentials `admin:admin` can be used to login.

The application's state is persisted at `/var/lib/grocy/grocy.db` in a sqlite3 database. The migration is applied when requesting the `/`-route of the application.

18.2. Settings

The configuration for `grocy` is located at `/etc/grocy/config.php`. By default, the following settings can be defined in the NixOS-configuration:

```
{ pkgs, ... }:  
2 {  
  services.grocy.settings = {  
4    # The default currency in the system for invoices etc.  
    # Please note that exchange rates aren't taken into account, this  
6    # is just the setting for what's shown in the frontend.  
    currency = "EUR";  
8  
    # The display language (and locale configuration) for grocy.  
10   culture = "de";  
  
12   calendar = {  
    # Whether or not to show the week-numbers  
14    # in the calendar.  
    showWeekNumber = true;  
16  
    # Index of the first day to be shown in the calendar (0=Sunday, 1=Monday,  
18    # 2=Tuesday and so on).  
    firstDayOfWeek = 2;  
20   };  
  };  
22 }
```

If you want to alter the configuration file on your own, you can do this manually with an expression like this:

```
{ lib, ... }:  
2 {
```

```

environment.etc."grocy/config.php".text = lib.mkAfter ''
4   // Arbitrary PHP code in grocy's configuration file
   '';
6 }

```

Chapter 19. Yggdrasil

Table of Contents

19.1. Configuration

Source: modules/services/networking/yggdrasil/default.nix

Upstream documentation: <https://yggdrasil-network.github.io/>

Yggdrasil is an early-stage implementation of a fully end-to-end encrypted, self-arranging IPv6 network.

19.1. Configuration

19.1.1. Simple ephemeral node

An annotated example of a simple configuration:

```

{
2   services.yggdrasil = {
      enable = true;
4     persistentKeys = false;
      # The NixOS module will generate new keys and a new IPv6 address each time
6     # it is started if persistentKeys is not enabled.

8     config = {
      Peers = [
10      # Yggdrasil will automatically connect and "peer" with other nodes it
      # discovers via link-local multicast announcements. Unless this is the
12      # case (it probably isn't) a node needs peers within the existing
      # network that it can tunnel to.
14      "tcp://1.2.3.4:1024"
      "tcp://1.2.3.5:1024"
16      # Public peers can be found at
      # https://github.com/yggdrasil-network/public-peers
18    ];
    };
20  };
}

```

19.1.2. Persistent node with prefix

A node with a fixed address that announces a prefix:

```

1 let
    address = "210:5217:69c0:9afc:1b95:b9f:8718:c3d2";
3   prefix = "310:5217:69c0:9afc";
    # taken from the output of "yggdrasilctl getself".
5 in {

7   services.yggdrasil = {
      enable = true;
9     persistentKeys = true; # Maintain a fixed public key and IPv6 address.
      config = {

```

```

11     Peers = [ "tcp://1.2.3.4:1024" "tcp://1.2.3.5:1024" ];
    NodeInfo = {
13         # This information is visible to the network.
        name = config.networking.hostName;
15         location = "The North Pole";
    };
17 };
};

19
boot.kernel.sysctl."net.ipv6.conf.all.forwarding" = 1;
21     # Forward traffic under the prefix.

23     networking.interfaces.${eth0}.ipv6.addresses = [{
        # Set a 300::/8 address on the local physical device.
25         address = prefix + "::1";
        prefixLength = 64;
27     }];

29     services.radvd = {
        # Announce the 300::/8 prefix to eth0.
31         enable = true;
        config = ''
33         interface eth0
        {
35             AdvSendAdvert on;
            AdvDefaultLifetime 0;
37             prefix ${prefix}::/64 {
                AdvOnLink on;
39                 AdvAutonomous on;
            };
41             route 200::/8 {};
        };
43     '';
    };
45 }

```

19.1.3. Yggdrasil attached Container

A NixOS container attached to the Yggdrasil network via a node running on the host:

```

1 let
    yggPrefix64 = "310:5217:69c0:9afc";
3     # Again, taken from the output of "yggdrasilctl getself".
in
5 {
    boot.kernel.sysctl."net.ipv6.conf.all.forwarding" = 1;
7     # Enable IPv6 forwarding.

9     networking = {
        bridges.br0.interfaces = [ ];
11         # A bridge only to containers...

13         interfaces.br0 = {
            # ... configured with a prefix address.
15             ipv6.addresses = [{
                address = "${yggPrefix64}::1";
17             prefixLength = 64;
            }];
        };
    };
}

```

```

    }];
19 };
   };
21
   containers.foo = {
23     autoStart = true;
       privateNetwork = true;
25     hostBridge = "br0";
       # Attach the container to the bridge only.
27     config = { config, pkgs, ... }: {
       networking.interfaces.eth0.ipv6 = {
29         addresses = [{
           # Configure a prefix address.
31           address = "${yggPrefix64}::2";
           prefixLength = 64;
33         }];
       routes = [{
35         # Configure the prefix route.
           address = "200::";
37         prefixLength = 7;
           via = "${yggPrefix64}::1";
39       }];
       };
41
       services.httpd.enable = true;
43       networking.firewall.allowedTCPPorts = [ 80 ];
       };
45 };
47 }

```

Chapter 20. Prosody

Table of Contents

20.1. Basic usage

20.2. Let's Encrypt Configuration

Prosody is an open-source, modern XMPP server.

20.1. Basic usage

A common struggle for most XMPP newcomers is to find the right set of XMPP Extensions (XEPs) to setup. Forget to activate a few of those and your XMPP experience might turn into a nightmare!

The XMPP community tackles this problem by creating a meta-XEP listing a decent set of XEPs you should implement. This meta-XEP is issued every year, the 2020 edition being XEP-0423.

The NixOS Prosody module will implement most of these recommendend XEPs out of the box. That being said, two components still require some manual configuration: the Multi User Chat (MUC) and the HTTP File Upload ones. You'll need to create a DNS subdomain for each of those. The current convention is to name your MUC endpoint `conference.example.org` and your HTTP upload domain `upload.example.org`.

A good configuration to start with, including a Multi User Chat (MUC) endpoint as well as a HTTP File Upload endpoint will look like this:

```

services.prosody = {
2   enable = true;

```

```

admins = [ "root@example.org" ];
4  ssl.cert = "/var/lib/acme/example.org/fullchain.pem";
  ssl.key = "/var/lib/acme/example.org/key.pem";
6  virtualHosts."example.org" = {
    enabled = true;
8    domain = "example.org";
    ssl.cert = "/var/lib/acme/example.org/fullchain.pem";
10   ssl.key = "/var/lib/acme/example.org/key.pem";
  };
12  muc = [ {
    domain = "conference.example.org";
14  } ];
  uploadHttp = {
16    domain = "upload.example.org";
  };
18 };

```

20.2. Let's Encrypt Configuration

As you can see in the code snippet from the previous section, you'll need a single TLS certificate covering your main endpoint, the MUC one as well as the HTTP Upload one. We can generate such a certificate by leveraging the ACME `extraDomainNames` module option.

Provided the setup detailed in the previous section, you'll need the following acme configuration to generate a TLS certificate for the three endpoints:

```

security.acme = {
2  email = "root@example.org";
  acceptTerms = true;
4  certs = {
    "example.org" = {
6    webroot = "/var/www/example.org";
    email = "root@example.org";
8    extraDomainNames = [ "conference.example.org" "upload.example.org" ];
    };
10 };
};

```

Chapter 21. Prometheus exporters

Table of Contents

21.1. Configuration

21.2. Adding a new exporter

21.3. Updating an exporter module

Prometheus exporters provide metrics for the prometheus monitoring system.

21.1. Configuration

One of the most common exporters is the node exporter, it provides hardware and OS metrics from the host it's running on. The exporter could be configured as follows:

```

1  services.prometheus.exporters.node = {
    enable = true;
3    enabledCollectors = [
        "logind"
    ]
  };

```

```

5     "systemd"
    ];
7     disabledCollectors = [
        "textfile"
9     ];
    openFirewall = true;
11    firewallFilter = "-i br0 -p tcp -m tcp --dport 9100";
};

```

It should now serve all metrics from the collectors that are explicitly enabled and the ones that are enabled by default, via http under `/metrics`. In this example the firewall should just allow incoming connections to the exporter's port on the bridge interface `br0` (this would have to be configured separately of course). For more information about configuration see `man configuration.nix` or search through the available options.

21.2. Adding a new exporter

To add a new exporter, it has to be packaged first (see `nixpkgs/pkg/servers/monitoring/prometheus/` for examples), then a module can be added. The postfix exporter is used in this example:

- Some default options for all exporters are provided by `nixpkgs/nixos/modules/services/monitoring/prometheus/exporters/`
- - `enable`
 - `port`
 - `listenAddress`
 - `extraFlags`
 - `openFirewall`
 - `firewallFilter`
 - `user`
 - `group`
- As there is already a package available, the module can now be added. This is accomplished by adding a new file to the `nixos/modules/services/monitoring/prometheus/exporters/` directory, which will be called `postfix.nix` and contains all exporter specific options and configuration:

```

# nixpkgs/nixos/modules/services/monitoring/prometheus/exporters/postfix.nix
2 { config, lib, pkgs, options }:
3
4 with lib;
5
6 let
7     # for convenience we define cfg here
8     cfg = config.services.prometheus.exporters.postfix;
9 in
10 {
11     port = 9154; # The postfix exporter listens on this port by default
12
13     # `extraOpts` is an attribute set which contains additional options
14     # (and optional overrides for default options).
15     # Note that this attribute is optional.
16     extraOpts = {
17         telemetryPath = mkOption {
18             type = types.str;
19             default = "/metrics";
20             description = ''
                Path under which to expose metrics.
            '';

```

```

22     '';
23 };
24 logfilePath = mkOption {
25     type = types.path;
26     default = /var/log/postfix_exporter_input.log;
27     example = /var/log/mail.log;
28     description = ''
29         Path where Postfix writes log entries.
30         This file will be truncated by this exporter!
31     '';
32 };
33 showqPath = mkOption {
34     type = types.path;
35     default = /var/spool/postfix/public/showq;
36     example = /var/lib/postfix/queue/public/showq;
37     description = ''
38         Path at which Postfix places its showq socket.
39     '';
40 };
41 };
42
43 # `serviceOpts` is an attribute set which contains configuration
44 # for the exporter's systemd service. One of
45 # `serviceOpts.script` and `serviceOpts.serviceConfig.ExecStart`
46 # has to be specified here. This will be merged with the default
47 # service configuration.
48 # Note that by default 'DynamicUser' is 'true'.
49 serviceOpts = {
50     serviceConfig = {
51         DynamicUser = false;
52         ExecStart = ''
53             ${pkgs.prometheus-postfix-exporter}/bin/postfix_exporter \
54             --web.listen-address ${cfg.listenAddress}:${toString cfg.port} \
55             --web.telemetry-path ${cfg.telemetryPath} \
56             ${concatStringsSep " \\\n " cfg.extraFlags}
57         '';
58     };
59 };
60 }

```

- This should already be enough for the postfix exporter. Additionally one could now add assertions and conditional default values. This can be done in the 'meta-module' that combines all exporter definitions and generates the submodules: `nixpkgs/nixos/modules/services/prometheus/exporters.nix`

21.3. Updating an exporter module

Should an exporter option change at some point, it is possible to add information about the change to the exporter definition similar to `nixpkgs/nixos/modules/rename.nix`:

```

{ config, lib, pkgs, options }:
2
with lib;
4
let
6     cfg = config.services.prometheus.exporters.nginx;
in
8 {

```

```

10  port = 9113;
    extraOpts = {
        # additional module options
12      # ...
    };
14  serviceOpts = {
        # service configuration
16      # ...
    };
18  imports = [
        # 'services.prometheus.exporters.nginx.telemetryEndpoint' ->
        ↪ 'services.prometheus.exporters.nginx.telemetryPath'
20      (mkRenamedOptionModule [ "telemetryEndpoint" ] [ "telemetryPath" ])

        # removed option 'services.prometheus.exporters.nginx.insecure'
        (mkRemovedOptionModule [ "insecure" ] ''
24          This option was replaced by 'prometheus.exporters.nginx.sslVerify' which
            ↪ defaults to true.
        '')
26      ({ options.warnings = options.warnings; })
    ];
28 }

```

Chapter 22. WeeChat

Table of Contents

22.1. Basic Usage

22.2. Re-attaching to WeeChat

WeeChat is a fast and extensible IRC client.

22.1. Basic Usage

By default, the module creates a **systemd** unit which runs the chat client in a detached **screen** session.

This can be done by enabling the **weechat** service:

```

{ ... }:
2
{
4   services.weechat.enable = true;
}

```

The service is managed by a dedicated user named **weechat** in the state directory **/var/lib/weechat**.

22.2. Re-attaching to WeeChat

WeeChat runs in a screen session owned by a dedicated user. To explicitly allow your another user to attach to this session, the **screenrc** needs to be tweaked by adding multiuser support:

```

{
2   programs.screen.screenrc = ''
        multiuser on
        acladd normal_user
4       '';
6 }

```

Now, the session can be re-attached like this:

```
screen -x weechat/weechat-screen
```

The session name can be changed using `services.weechat.sessionName`.

Chapter 23. Taskserver

Table of Contents

23.1. Configuration

23.2. The `nixos-taskserver` tool

23.3. Declarative/automatic CA management

23.4. Manual CA management

Taskserver is the server component of Taskwarrior, a free and open source todo list application.

Upstream documentation: <https://taskwarrior.org/docs/#taskd>

23.1. Configuration

Taskserver does all of its authentication via TLS using client certificates, so you either need to roll your own CA or purchase a certificate from a known CA, which allows creation of client certificates. These certificates are usually advertised as “server certificates”.

So in order to make it easier to handle your own CA, there is a helper tool called **nixos-taskserver** which manages the custom CA along with Taskserver organisations, users and groups.

While the client certificates in Taskserver only authenticate whether a user is allowed to connect, every user has its own UUID which identifies it as an entity.

With **nixos-taskserver** the client certificate is created along with the UUID of the user, so it handles all of the credentials needed in order to setup the Taskwarrior client to work with a Taskserver.

23.2. The `nixos-taskserver` tool

Because Taskserver by default only provides scripts to setup users imperatively, the **nixos-taskserver** tool is used for addition and deletion of organisations along with users and groups defined by `services.taskserver.organisations` and as well for imperative set up.

The tool is designed to not interfere if the command is used to manually set up some organisations, users or groups.

For example if you add a new organisation using **nixos-taskserver org add foo**, the organisation is not modified and deleted no matter what you define in `services.taskserver.organisations`, even if you’re adding the same organisation in that option.

The tool is modelled to imitate the official **taskd** command, documentation for each subcommand can be shown by using the `--help` switch.

23.3. Declarative/automatic CA management

Everything is done according to what you specify in the module options, however in order to set up a Taskwarrior client for synchronisation with a Taskserver instance, you have to transfer the keys and certificates to the client machine.

This is done using **nixos-taskserver user export \$orgname \$username** which is printing a shell script fragment to stdout which can either be used verbatim or adjusted to import the user on the client machine.

For example, let’s say you have the following configuration:

```

{
2  services.taskserver.enable = true;
   services.taskserver.fqdn = "server";
4  services.taskserver.listenHost = ":::~";
   services.taskserver.organisations.my-company.users = [ "alice" ];
6 }

```

This creates an organisation called `my-company` with the user `alice`.

Now in order to import the `alice` user to another machine `alicebox`, all we need to do is something like this:

```
$ ssh server nixos-taskserver user export my-company alice | sh
```

Of course, if no SSH daemon is available on the server you can also copy & paste it directly into a shell.

After this step the user should be set up and you can start synchronising your tasks for the first time with **task sync init** on `alicebox`.

Subsequent synchronisation requests merely require the command **task sync** after that stage.

23.4. Manual CA management

If you set any options within `service.taskserver.pki.manual.*`, **nixos-taskserver** won't issue certificates, but you can still use it for adding or removing user accounts.

Chapter 24. Matrix

Table of Contents

24.1. Synapse Homeserver

24.2. Element (formerly known as Riot) Web Client

Matrix is an open standard for interoperable, decentralised, real-time communication over IP. It can be used to power Instant Messaging, VoIP/WebRTC signalling, Internet of Things communication - or anywhere you need a standard HTTP API for publishing and subscribing to data whilst tracking the conversation history.

This chapter will show you how to set up your own, self-hosted Matrix homeserver using the Synapse reference homeserver, and how to serve your own copy of the Element web client. See the Try Matrix Now! overview page for links to Element Apps for Android and iOS, desktop clients, as well as bridges to other networks and other projects around Matrix.

24.1. Synapse Homeserver

Synapse is the reference homeserver implementation of Matrix from the core development team at matrix.org. The following configuration example will set up a synapse server for the **example.org** domain, served from the host `myhostname.example.org`. For more information, please refer to the installation instructions of Synapse .

```

{ pkgs, ... }:
2 let
   fqdn =
4     let
       join = hostName: domain: hostName + optionalString (domain != null)
         ↪ ".${domain}";
6     in join config.networking.hostName config.networking.domain;
in {
8   networking = {
       hostName = "myhostname";
10    domain = "example.org";
   };
12   networking.firewall.allowedTCPPorts = [ 80 443 ];

```

```

14 services.postgresql.enable = true;
15 services.postgresql.initialScript = pkgs.writeText "synapse-init.sql" ''
16     CREATE ROLE "matrix-synapse" WITH LOGIN PASSWORD 'synapse';
17     CREATE DATABASE "matrix-synapse" WITH OWNER "matrix-synapse"
18         TEMPLATE template0
19         LC_COLLATE = "C"
20         LC_CTYPE = "C";
21 '';
22
23 services.nginx = {
24     enable = true;
25     # only recommendedProxySettings and recommendedGzipSettings are strictly
26     #   ↪ required,
27     # but the rest make sense as well
28     recommendedTlsSettings = true;
29     recommendedOptimisation = true;
30     recommendedGzipSettings = true;
31     recommendedProxySettings = true;
32
33     virtualHosts = {
34         # This host section can be placed on a different host than the rest,
35         # i.e. to delegate from the host being accessible as
36         #   ↪ ${config.networking.domain}
37         # to another host actually running the Matrix homeserver.
38         "${config.networking.domain}" = {
39             locations."= /.well-known/matrix/server".extraConfig =
40             let
41                 # use 443 instead of the default 8448 port to unite
42                 # the client-server and server-server port for simplicity
43                 server = { "m.server" = "${fqdn}:443"; };
44             in ''
45                 add_header Content-Type application/json;
46                 return 200 '${builtins.toJSON server}';
47             '';
48             locations."= /.well-known/matrix/client".extraConfig =
49             let
50                 client = {
51                     "m.homeserver" = { "base_url" = "https://${fqdn}"; };
52                     "m.identity_server" = { "base_url" = "https://vector.im"; };
53                 };
54                 # ACA0 required to allow element-web on any URL to request this json file
55                 in ''
56                     add_header Content-Type application/json;
57                     add_header Access-Control-Allow-Origin *;
58                     return 200 '${builtins.toJSON client}';
59                 '';
60             };
61
62             # Reverse proxy for Matrix client-server and server-server communication
63             ${fqdn} = {
64                 enableACME = true;
65                 forceSSL = true;
66
67                 # Or do a redirect instead of the 404, or whatever is appropriate for you.
68                 # But do not put a Matrix Web client here! See the Element web section

```

```

        ↪ below.
        locations."/".extraConfig = ''
68         return 404;
        '';

70
        # forward all Matrix API calls to the synapse Matrix homeserver
72         locations."/_matrix" = {
            proxyPass = "http://[::1]:8008"; # without a trailing /
74         };
        };
76     };
};
78 services.matrix-synapse = {
    enable = true;
80     server_name = config.networking.domain;
    listeners = [
82         {
            port = 8008;
84             bind_address = "::1";
            type = "http";
86             tls = false;
            x_forwarded = true;
88             resources = [
                {
90                 names = [ "client" "federation" ];
                    compress = false;
92                 }
            ];
94         }
    ];
96 };
};

```

If the A and AAAA DNS records on `example.org` do not point on the same host as the records for `myhostname.example.org`, you can easily move the `/.well-known` virtualHost section of the code to the host that is serving `example.org`, while the rest stays on `myhostname.example.org` with no other changes required. This pattern also allows to seamlessly move the homeserver from `myhostname.example.org` to `myotherhost.example.org` by only changing the `/.well-known` redirection target.

If you want to run a server with public registration by anybody, you can then enable `services.matrix-synapse.enable_registration = true;`. Otherwise, or you can generate a registration secret with `pwgen -s 64 1` and set it with `services.matrix-synapse.registration_shared_secret`. To create a new user or admin, run the following after you have set the secret and have rebuilt NixOS:

```

$ nix run nixpkgs.matrix-synapse
2 $ register_new_matrix_user -k your-registration-shared-secret http://localhost:8008
New user localpart: your-username
4 Password:
Confirm password:
6 Make admin [no]:
Success!

```

In the example, this would create a user with the Matrix Identifier `@your-username:example.org`. Note that the registration secret ends up in the nix store and therefore is world-readable by any user on your machine, so it makes sense to only temporarily activate the `registration_shared_secret` option until a better solution for NixOS is in place.

24.2. Element (formerly known as Riot) Web Client

Element Web is the reference web client for Matrix and developed by the core team at matrix.org. Element was formerly known as Riot.im, see the Element introductory blog post for more information. The following snippet can be optionally added to the code before to complete the synapse installation with a web client served at <https://element.myhostname.example.org> and <https://element.example.org>. Alternatively, you can use the hosted copy at <https://app.element.io/>, or use other web clients or native client applications. Due to the `.well-known` urls set up done above, many clients should fill in the required connection details automatically when you enter your Matrix Identifier. See Try Matrix Now! for a list of existing clients and their supported featureset.

```
{
2  services.nginx.virtualHosts."element.${fqdn}" = {
    enableACME = true;
4    forceSSL = true;
    serverAliases = [
6      "element.${config.networking.domain}"
    ];
8
    root = pkgs.element-web.override {
10     conf = {
        default_server_config."m.homeserver" = {
12         "base_url" = "${config.networking.domain}";
        "server_name" = "${fqdn}";
14     };
    };
16 };
18 }
```

Note that the Element developers do not recommend running Element and your Matrix homeserver on the same fully-qualified domain name for security reasons. In the example, this means that you should not reuse the `myhostname.example.org` virtualHost to also serve Element, but instead serve it on a different subdomain, like `element.example.org` in the example. See the Element Important Security Notes for more information on this subject.

Chapter 25. Gitlab

Table of Contents

25.1. Prerequisites

25.2. Configuring

25.3. Maintenance

Gitlab is a feature-rich git hosting service.

25.1. Prerequisites

The gitlab service exposes only an Unix socket at `/run/gitlab/gitlab-workhorse.socket`. You need to configure a webserver to proxy HTTP requests to the socket.

For instance, the following configuration could be used to use nginx as frontend proxy:

```
services.nginx = {
2  enable = true;
    recommendedGzipSettings = true;
4  recommendedOptimisation = true;
    recommendedProxySettings = true;
```

```

6   recommendedTlsSettings = true;
   virtualHosts."git.example.com" = {
8       enableACME = true;
       forceSSL = true;
10      locations."/.proxyPass = "http://unix:/run/gitlab/gitlab-workhorse.socket";
   };
12 };

```

25.2. Configuring

Gitlab depends on both PostgreSQL and Redis and will automatically enable both services. In the case of PostgreSQL, a database and a role will be created.

The default state dir is `/var/gitlab/state`. This is where all data like the repositories and uploads will be stored.

A basic configuration with some custom settings could look like this:

```

services.gitlab = {
2   enable = true;
   databasePasswordFile = "/var/keys/gitlab/db_password";
4   initialRootPasswordFile = "/var/keys/gitlab/root_password";
   https = true;
6   host = "git.example.com";
   port = 443;
8   user = "git";
   group = "git";
10  smtp = {
       enable = true;
12      address = "localhost";
       port = 25;
14  };
   secrets = {
16      dbFile = "/var/keys/gitlab/db";
       secretFile = "/var/keys/gitlab/secret";
18      otpFile = "/var/keys/gitlab/otp";
       jwsFile = "/var/keys/gitlab/jws";
20  };
   extraConfig = {
22      gitlab = {
           email_from = "gitlab-no-reply@example.com";
24          email_display_name = "Example GitLab";
           email_reply_to = "gitlab-no-reply@example.com";
26          default_projects_features = { builds = false; };
       };
28  };
};

```

If you're setting up a new Gitlab instance, generate new secrets. You for instance use `tr -dc A-Za-z0-9 < /dev/urandom | head -c 128 > /var/keys/gitlab/db` to generate a new db secret. Make sure the files can be read by, and only by, the user specified by `services.gitlab.user`. Gitlab encrypts sensitive data stored in the database. If you're restoring an existing Gitlab instance, you must specify the secrets secret from `config/secrets.yml` located in your Gitlab state folder.

When `icoming_mail.enabled` is set to `true` in `extraConfig` an additional service called `gitlab-mailroom` is enabled for fetching incoming mail.

Refer to Appendix A, *Configuration Options* for all available configuration options for the `services.gitlab` module.

25.3. Maintenance

You can run Gitlab's rake tasks with `gitlab-rake` which will be available on the system when gitlab is enabled. You will have to run the command as the user that you configured to run gitlab with.

For example, to backup a Gitlab instance:

```
$ sudo -u git -H gitlab-rake gitlab:backup:create
```

A list of all available rake tasks can be obtained by running:

```
1 $ sudo -u git -H gitlab-rake -T
```

Chapter 26. Mailman

Table of Contents

26.1. Basic usage

Mailman is free software for managing electronic mail discussion and e-newsletter lists. Mailman and its web interface can be configured using the corresponding NixOS module. Note that this service is best used with an existing, securely configured Postfix setup, as it does not automatically configure this.

26.1. Basic usage

For a basic configuration, the following settings are suggested:

```
1 { config, ... }: {
2     services.postfix = {
3         enable = true;
4         relayDomains = ["hash:/var/lib/mailman/data/postfix_domains"];
5         sslCert = config.security.acme.certs."lists.example.org".directory +
6             ↪ "/full.pem";
7         sslKey = config.security.acme.certs."lists.example.org".directory + "/key.pem";
8         config = {
9             transport_maps = ["hash:/var/lib/mailman/data/postfix_lmtp"];
10            local_recipient_maps = ["hash:/var/lib/mailman/data/postfix_lmtp"];
11        };
12    };
13    services.mailman = {
14        enable = true;
15        serve.enable = true;
16        hyperkitty.enable = true;
17        webHosts = ["lists.example.org"];
18        siteOwner = "mailman@example.org";
19    };
20    services.nginx.virtualHosts."lists.example.org".enableACME = true;
21    networking.firewall.allowedTCPPorts = [ 25 80 443 ];
22 }
```

DNS records will also be required:

- AAAA and A records pointing to the host in question, in order for browsers to be able to discover the address of the web server;
- An MX record pointing to a domain name at which the host is reachable, in order for other mail servers to be able to deliver emails to the mailing lists it hosts.

After this has been done and appropriate DNS records have been set up, the Postorius mailing list manager and the Hyperkitty archive browser will be available at <https://lists.example.org/>. Note that this setup is not sufficient to deliver emails to most email providers nor to avoid spam -- a number of additional measures for authenticating

incoming and outgoing mails, such as SPF, DMARC and DKIM are necessary, but outside the scope of the Mailman module.

Chapter 27. Trezor

Trezor is an open-source cryptocurrency hardware wallet and security token allowing secure storage of private keys. It offers advanced features such U2F two-factor authorization, SSH login through Trezor SSH agent, GPG and a password manager. For more information, guides and documentation, see <https://wiki.trezor.io>.

To enable Trezor support, add the following to your `configuration.nix`:

```
services.trezord.enable = true;
```

This will add all necessary udev rules and start Trezor Bridge.

Chapter 28. Emacs

Table of Contents

28.1. Installing Emacs

28.2. Running Emacs as a Service

28.3. Configuring Emacs

Emacs is an extensible, customizable, self-documenting real-time display editor -- and more. At its core is an interpreter for Emacs Lisp, a dialect of the Lisp programming language with extensions to support text editing.

Emacs runs within a graphical desktop environment using the X Window System, but works equally well on a text terminal. Under macOS, a "Mac port" edition is available, which uses Apple's native GUI frameworks.

Nixpkgs provides a superior environment for running Emacs. It's simple to create custom builds by overriding the default packages. Chaotic collections of Emacs Lisp code and extensions can be brought under control using declarative package management. NixOS even provides a **systemd** user service for automatically starting the Emacs daemon.

28.1. Installing Emacs

Emacs can be installed in the normal way for Nix (see Chapter 6, *Package Management*). In addition, a NixOS *service* can be enabled.

28.1.1. The Different Releases of Emacs

Nixpkgs defines several basic Emacs packages. The following are attributes belonging to the `pkgs` set:

emacs , **emacs** The latest stable version of Emacs using the GTK 2 widget toolkit.

emacs-nox Emacs built without any dependency on X11 libraries.

emacsMacport , **emacsMacport** Emacs with the "Mac port" patches, providing a more native look and feel under macOS.

If those aren't suitable, then the following imitation Emacs editors are also available in Nixpkgs: Zile, mg, Yi, jmacs.

28.1.2. Adding Packages to Emacs

Emacs includes an entire ecosystem of functionality beyond text editing, including a project planner, mail and news reader, debugger interface, calendar, and more.

Most extensions are gotten with the Emacs packaging system (`package.el`) from Emacs Lisp Package Archive (ELPA), MELPA, MELPA Stable, and Org ELPA. Nixpkgs is regularly updated to mirror all these archives.

Under NixOS, you can continue to use `package-list-packages` and `package-install` to install packages. You can also declare the set of Emacs packages you need using the derivations from Nixpkgs. The rest of this section discusses declarative installation of Emacs packages through nixpkgs.

The first step to declare the list of packages you want in your Emacs installation is to create a dedicated derivation. This can be done in a dedicated `emacs.nix` file such as:

Example 28.1. Nix expression to build Emacs with packages (`emacs.nix`)

```
/*
2 This is a nix expression to build Emacs and some Emacs packages I like
  from source on any distribution where Nix is installed. This will install
4 all the dependencies from the nixpkgs repository and build the binary files
  without interfering with the host distribution.
6
  To build the project, type the following from the current directory:
8
  $ nix-build emacs.nix
10
  To run the newly compiled executable:
12
  $ ./result/bin/emacs
14 */
{ pkgs ? import <nixpkgs> {} }:
16
let
18   myEmacs = pkgs.emacs;
   emacsWithPackages = (pkgs.emacsPackagesGen myEmacs).emacsWithPackages;
20 in
   emacsWithPackages (epkgs: (with epkgs.melpaStablePackages; [
22     magit           # ; Integrate git <C-x g>
     zerodark-theme  # ; Nicolas' theme
24   ]) ++ (with epkgs.melpaPackages; [
     undo-tree       # ; <C-x u> to show the undo tree
26     zoom-frm       # ; increase/decrease font size for all buffers %lt;C-x C-+>
   ]) ++ (with epkgs.elpaPackages; [
28     auctex          # ; LaTeX mode
     beacon          # ; highlight my cursor when scrolling
30     nameless        # ; hide current package name everywhere in elisp code
   ]) ++ [
32     pkgs.notmuch    # From main packages set
   ])
}
```

-
- 1 The first non-comment line in this file (`{ pkgs ? ... }`) indicates that the whole file represents a function.
 - 2 The `let` expression below defines a `myEmacs` binding pointing to the current stable version of Emacs. This binding is here to
 - 3 This generates an `emacsWithPackages` function. It takes a single argument: a function from a package set to a list of packa
 - 4 The rest of the file specifies the list of packages to install. In the example, two packages (`magit` and `zerodark-theme`) are
 - 5 Two packages (`undo-tree` and `zoom-frm`) are taken from MELPA.
 - 6 Three packages are taken from GNU ELPA.
 - 7 `notmuch` is taken from a nixpkgs derivation which contains an Emacs mode.
-

The result of this configuration will be an `emacs` command which launches Emacs with all of your chosen packages in the `load-path`.

You can check that it works by executing this in a terminal:

```
$ nix-build emacs.nix
2 $ ./result/bin/emacs -q
```

and then typing `M-x package-initialize`. Check that you can use all the packages you want in this Emacs instance. For example, try switching to the `zerodark` theme through `M-x load-theme <RET> zerodark <RET> y`.

Tip

A few popular extensions worth checking out are: `auctex`, `company`, `edit-server`, `flycheck`, `helm`, `iedit`, `magit`, `multiple-cursors`, `projectile`, and `yasnippet`.

The list of available packages in the various ELPA repositories can be seen with the following commands:

Example 28.2. Querying Emacs packages

```
nix-env -f "<nixpkgs>" -qaP -A emacsPackages.elpaPackages
2 nix-env -f "<nixpkgs>" -qaP -A emacsPackages.melpaPackages
  nix-env -f "<nixpkgs>" -qaP -A emacsPackages.melpaStablePackages
4 nix-env -f "<nixpkgs>" -qaP -A emacsPackages.orgPackages
```

If you are on NixOS, you can install this particular Emacs for all users by adding it to the list of system packages (see Section 6.1, “Declarative Package Management”). Simply modify your file `configuration.nix` to make it contain:

Example 28.3. Custom Emacs in `configuration.nix`

```
{
2  environment.systemPackages = [
    # [...]
4    (import /path/to/emacs.nix { inherit pkgs; })
  ];
6 }
```

In this case, the next `nixos-rebuild switch` will take care of adding your `emacs` to the `PATH` environment variable (see Chapter 3, *Changing the Configuration*).

If you are not on NixOS or want to install this particular Emacs only for yourself, you can do so by adding it to your `~/.config/nixpkgs/config.nix` (see Nixpkgs manual):

Example 28.4. Custom Emacs in `~/.config/nixpkgs/config.nix`

```
{
2  packageOverrides = super: let self = super.pkgs; in {
    myemacs = import /path/to/emacs.nix { pkgs = self; };
4  };
}
```

In this case, the next `nix-env -f '<nixpkgs>' -iA myemacs` will take care of adding your emacs to the `PATH` environment variable.

28.1.3. Advanced Emacs Configuration

If you want, you can tweak the Emacs package itself from your `emacs.nix`. For example, if you want to have a GTK 3-based Emacs instead of the default GTK 2-based binary and remove the automatically generated `emacs.desktop` (useful if you only use `emacsclient`), you can change your file `emacs.nix` in this way:

Example 28.5. Custom Emacs build

```
{ pkgs ? import <nixpkgs> {} }:  
2 let  
    myEmacs = (pkgs.emacs.override {  
4       # Use gtk3 instead of the default gtk2  
        withGTK3 = true;  
6       withGTK2 = false;  
    }).overrideAttrs (attrs: {  
8       # I don't want emacs.desktop file because I only use  
        # emacsclient.  
10      postInstall = (attrs.postInstall or "") + '  
        rm $out/share/applications/emacs.desktop  
12      '';  
    });  
14 in [...]
```

After building this file as shown in Example 28.1, “Nix expression to build Emacs with packages (`emacs.nix`)”, you will get an GTK 3-based Emacs binary pre-loaded with your favorite packages.

28.2. Running Emacs as a Service

NixOS provides an optional **systemd** service which launches Emacs daemon with the user’s login session.

Source: `modules/services/editors/emacs.nix`

28.2.1. Enabling the Service

To install and enable the **systemd** user service for Emacs daemon, add the following to your `configuration.nix`:

```
services.emacs.enable = true;  
2 services.emacs.package = import /home/cassou/.emacs.d { pkgs = pkgs; };
```

The `services.emacs.package` option allows a custom derivation to be used, for example, one created by `emacsWithPackages`.

Ensure that the Emacs server is enabled for your user’s Emacs configuration, either by customizing the `server-mode` variable, or by adding `(server-start)` to `~/.emacs.d/init.el`.

To start the daemon, execute the following:

```
$ nixos-rebuild switch # to activate the new configuration.nix  
2 $ systemctl --user daemon-reload # to force systemd reload  
$ systemctl --user start emacs.service # to start the Emacs daemon
```

The server should now be ready to serve Emacs clients.

28.2.2. Starting the client

Ensure that the emacs server is enabled, either by customizing the `server-mode` variable, or by adding `(server-start)` to `~/.emacs`.

To connect to the emacs daemon, run one of the following:

```
emacsclient FILENAME  
2 emacsclient --create-frame # opens a new frame (window)  
emacsclient --create-frame --tty # opens a new frame on the current terminal
```

28.2.3. Configuring the EDITOR variable

If `services.emacs.defaultEditor` is `true`, the `EDITOR` variable will be set to a wrapper script which launches `emacsclient`.

Any setting of `EDITOR` in the shell config files will override `services.emacs.defaultEditor`. To make sure `EDITOR` refers to the Emacs wrapper script, remove any existing `EDITOR` assignment from `.profile`, `.bashrc`, `.zshenv` or any other shell config file.

If you have formed certain bad habits when editing files, these can be corrected with a shell alias to the wrapper script:

```
alias vi=$EDITOR
```

28.2.4. Per-User Enabling of the Service

In general, **systemd** user services are globally enabled by symlinks in `/etc/systemd/user`. In the case where Emacs daemon is not wanted for all users, it is possible to install the service but not globally enable it:

```
services.emacs.enable = false;  
2 services.emacs.install = true;
```

To enable the **systemd** user service for just the currently logged in user, run:

```
systemctl --user enable emacs
```

This will add the symlink `~/.config/systemd/user/emacs.service`.

28.3. Configuring Emacs

The Emacs init file should be changed to load the extension packages at startup:

Example 28.6. Package initialization in `.emacs`

```
(require 'package)  
2  
;; optional. makes unpure packages archives unavailable  
4 (setq package-archives nil)  
6 (setq package-enable-at-startup nil)  
(package-initialize)
```

After the declarative emacs package configuration has been tested, previously downloaded packages can be cleaned up by removing `~/.emacs.d/elpa` (do make a backup first, in case you forgot a package).

28.3.1. A Major Mode for Nix Expressions

Of interest may be `melpaPackages.nix-mode`, which provides syntax highlighting for the Nix language. This is particularly convenient if you regularly edit Nix files.

28.3.2. Accessing man pages

You can use `woman` to get completion of all available man pages. For example, type `M-x woman <RET>`
`↵ nixos-rebuild <RET>`.

28.3.3. Editing DocBook 5 XML Documents

Emacs includes nXML, a major-mode for validating and editing XML documents. When editing DocBook 5.0 documents, such as this one, nXML needs to be configured with the relevant schema, which is not included.

To install the DocBook 5.0 schemas, either add `pkgs.docbook5` to `environment.systemPackages` (NixOS), or run `nix-env -f '<nixpkgs>' -iA docbook5` (Nix).

Then customize the variable `rng-schema-locating-files` to include `~/.emacs.d/schemas.xml` and put the following text into that file:

Example 28.7. nXML Schema Configuration (`~/.emacs.d/schemas.xml`)

```
<?xml version="1.0"?>
2 <!--
   To let emacs find this file, evaluate:
4   (add-to-list 'rng-schema-locating-files "~/.emacs.d/schemas.xml")
   -->
6 <locatingRules xmlns="http://thaiopensource.com/ns/locating-rules/1.0">
   <!--
8     Use this variation if pkgs.docbook5 is added to environment.systemPackages
   -->
10  <namespace ns="http://docbook.org/ns/docbook"
      uri="/run/current-system/sw/share/xml/docbook-5.0/rng/docbookxi.rnc"/>
12  <!--
      Use this variation if installing schema with "nix-env -iA pkgs.docbook5".
14  <namespace ns="http://docbook.org/ns/docbook"
      uri="../.nix-profile/share/xml/docbook-5.0/rng/docbookxi.rnc"/>
16  -->
</locatingRules>
```

Chapter 29. Flatpak

Source: `modules/services/desktop/flatpak.nix`

Upstream documentation: <https://github.com/flatpak/flatpak/wiki>

Flatpak is a system for building, distributing, and running sandboxed desktop applications on Linux.

To enable Flatpak, add the following to your `configuration.nix`:

```
services.flatpak.enable = true;
```

For the sandboxed apps to work correctly, desktop integration portals need to be installed. If you run GNOME, this will be handled automatically for you; in other cases, you will need to add something like the following to your `configuration.nix`:

```
xdg.portal.extraPortals = [ pkgs.xdg-desktop-portal-gtk ];
```

Then, you will need to add a repository, for example, Flathub, either using the following commands:

```
1 $ flatpak remote-add --if-not-exists flathub
   ↪ https://flathub.org/repo/flathub.flatpakrepo
$ flatpak update
```

or by opening the repository file in GNOME Software.

Finally, you can search and install programs:

```
$ flatpak search bustle
2 $ flatpak install flathub org.freedesktop.Bustle
$ flatpak run org.freedesktop.Bustle
```

Again, GNOME Software offers graphical interface for these tasks.

Chapter 30. PostgreSQL

Table of Contents

30.1. Configuring

30.2. Upgrading

30.3. Options

30.4. Plugins

Source: `modules/services/databases/postgresql.nix`

Upstream documentation: <http://www.postgresql.org/docs/>

PostgreSQL is an advanced, free relational database.

30.1. Configuring

To enable PostgreSQL, add the following to your `configuration.nix`:

```
services.postgresql.enable = true;
2 services.postgresql.package = pkgs.postgresql_11;
```

Note that you are required to specify the desired version of PostgreSQL (e.g. `pkgs.postgresql_11`). Since upgrading your PostgreSQL version requires a database dump and reload (see below), NixOS cannot provide a default value for `services.postgresql.package` such as the most recent release of PostgreSQL.

By default, PostgreSQL stores its databases in `/var/lib/postgresql/$psqlSchema`. You can override this using `services.postgresql.dataDir`, e.g.

```
services.postgresql.dataDir = "/data/postgresql";
```

30.2. Upgrading

Major PostgreSQL upgrade requires PostgreSQL downtime and a few imperative steps to be called. To simplify this process, use the following NixOS module:

```
1  containers.temp-pg.config.services.postgresql = {
    enable = true;
3  package = pkgs.postgresql_12;
    ## set a custom new dataDir
5  # dataDir = "/some/data/dir";
  };
7  environment.systemPackages =
    let newpg = config.containers.temp-pg.config.services.postgresql;
    in [
        (pkgs.writeScriptBin "upgrade-pg-cluster" ''
11     set -x
        export OLDDATA="${config.services.postgresql.dataDir}"
13     export NEWDATA="${newpg.dataDir}"
        export OLDBIN="${config.services.postgresql.package}/bin"
15     export NEWBIN="${newpg.package}/bin"
```

```

17     install -d -m 0700 -o postgres -g postgres "$NEWDATA"
18     cd "$NEWDATA"
19     sudo -u postgres $NEWBIN/initdb -D "$NEWDATA"

21     systemctl stop postgresql      # old one

23     sudo -u postgres $NEWBIN/pg_upgrade \
24         --old-datadir "$OLDDATA" --new-datadir "$NEWDATA" \
25         --old-bindir $OLDBIN --new-bindir $NEWBIN \
26         "$@"
27     '')
    ];

```

The upgrade process is:

1. Rebuild nixos configuration with the configuration above added to your `configuration.nix`. Alternatively, add that into separate file and reference it in `imports` list.
2. Login as root (`sudo su -`)
3. Run `upgrade-pg-cluster`. It will stop old postgresql, initialize new one and migrate old one to new one. You may supply arguments like `--jobs 4` and `--link` to speedup migration process. See <https://www.postgresql.org/docs/current/pgupgrade.html> for details.
4. Change postgresql package in NixOS configuration to the one you were upgrading to, and change `dataDir` to the one you have migrated to. Rebuild NixOS. This should start new postgres using upgraded data directory.
5. After upgrade you may want to `ANALYZE` new db.

30.3. Options

A complete list of options for the PostgreSQL module may be found [here](#).

30.4. Plugins

Plugins collection for each PostgreSQL version can be accessed with `.pkgs`. For example, for `pkgs.postgresql_11` package, its plugin collection is accessed by `pkgs.postgresql_11.pkgs`:

```

$ nix repl '<nixpkgs>'
2
Loading '<nixpkgs>'...
4 Added 10574 variables.

6 nix-repl> postgresql_11.pkgs.<TAB><TAB>
  postgresql_11.pkgs.cstore_fdw      postgresql_11.pkgs.pg_repack
  postgresql_11.pkgs.pg_auto_failover postgresql_11.pkgs.pg_safeupdate
  postgresql_11.pkgs.pg_bigm          postgresql_11.pkgs.pg_similarity
  postgresql_11.pkgs.pg_cron          postgresql_11.pkgs.pg_topn
  postgresql_11.pkgs.pg_hll           postgresql_11.pkgs.pgjwt
  postgresql_11.pkgs.pg_partman       postgresql_11.pkgs.pgroonga
  ...

```

To add plugins via NixOS configuration, set `services.postgresql.extraPlugins`:

```

services.postgresql.package = pkgs.postgresql_11;
2 services.postgresql.extraPlugins = with pkgs.postgresql_11.pkgs; [
  pg_repack
4  postgis
];

```

You can build custom PostgreSQL-with-plugins (to be used outside of NixOS) using function `.withPackages`. For example, creating a custom PostgreSQL package in an overlay can look like:

```
self: super: {
2   postgresql_custom = self.postgresql_11.withPackages (ps: [
      ps.pg_repack
4     ps.postgis
    ]);
6 }
```

Here's a recipe on how to override a particular plugin through an overlay:

```
self: super: {
2   postgresql_11 = super.postgresql_11.override { this = self.postgresql_11; } // {
      pkgs = super.postgresql_11.pkgs // {
4       pg_repack = super.postgresql_11.pkgs.pg_repack.overrideAttrs (_: {
          name = "pg_repack-v20181024";
6         src = self.fetchzip {
              url =
                ↪ "https://github.com/reorg/pg_repack/archive/923fa2f3c709a506e111cc963034bf2fd
8             sha256 = "17k6hq9xaax87yz79j773qyigm4fwk8z4zh5cyp6z0sxnwfqxxw5";
          };
10      });
    };
12 };
}
```

Chapter 31. FoundationDB

Table of Contents

- 31.1. Configuring and basic setup
- 31.2. Scaling processes and backup agents
- 31.3. Clustering
- 31.4. Client connectivity
- 31.5. Client authorization and TLS
- 31.6. Backups and Disaster Recovery
- 31.7. Known limitations
- 31.8. Options
- 31.9. Full documentation

Source: `modules/services/databases/foundationdb.nix`

Upstream documentation: <https://apple.github.io/foundationdb/>

Maintainer: Austin Seipp

Available version(s): 5.1.x, 5.2.x, 6.0.x

FoundationDB (or "FDB") is an open source, distributed, transactional key-value store.

31.1. Configuring and basic setup

To enable FoundationDB, add the following to your `configuration.nix`:

```

services.foundationdb.enable = true;
2 services.foundationdb.package = pkgs.foundationdb52; # FoundationDB 5.2.x

```

The `services.foundationdb.package` option is required, and must always be specified. Due to the fact FoundationDB network protocols and on-disk storage formats may change between (major) versions, and upgrades must be explicitly handled by the user, you must always manually specify this yourself so that the NixOS module will use the proper version. Note that minor, bugfix releases are always compatible.

After running **nixos-rebuild**, you can verify whether FoundationDB is running by executing **fdbcli** (which is added to `environment.systemPackages`):

```

$ sudo -u foundationdb fdbcli
2 Using cluster file `/etc/foundationdb/fdb.cluster'.

4 The database is available.

6 Welcome to the fdbcli. For help, type `help'.
fdb> status
8
Using cluster file `/etc/foundationdb/fdb.cluster'.
10
Configuration:
12   Redundancy mode          - single
   Storage engine          - memory
14   Coordinators            - 1

16 Cluster:
   FoundationDB processes - 1
18   Machines                - 1
   Memory availability     - 5.4 GB per process on machine with least available
20   Fault Tolerance         - 0 machines
   Server time             - 04/20/18 15:21:14
22
...
24
fdb>

```

You can also write programs using the available client libraries. For example, the following Python program can be run in order to grab the cluster status, as a quick example. (This example uses **nix-shell** shebang support to automatically supply the necessary Python modules).

```

1 a@link> cat fdb-status.py
  #! /usr/bin/env nix-shell
3  #! nix-shell -i python -p python pythonPackages.foundationdb52

5  import fdb
  import json
7
  def main():
8      fdb.api_version(520)
      db = fdb.open()
11
      @fdb.transactional
13      def get_status(tr):
          return str(tr['\xff\xff/status/json'])
15
      obj = json.loads(get_status(db))
17      print('FoundationDB available: %s' %
          ↪ obj['client']['database_status']['available'])

```

```

19 if __name__ == "__main__":
    main()
21 a@link> chmod +x fdb-status.py
    a@link> ./fdb-status.py
23 FoundationDB available: True
    a@link>

```

FoundationDB is run under the **foundationdb** user and group by default, but this may be changed in the NixOS configuration. The systemd unit **foundationdb.service** controls the **fdbmonitor** process.

By default, the NixOS module for FoundationDB creates a single SSD-storage based database for development and basic usage. This storage engine is designed for SSDs and will perform poorly on HDDs; however it can handle far more data than the alternative "memory" engine and is a better default choice for most deployments. (Note that you can change the storage backend on-the-fly for a given FoundationDB cluster using **fdbcli**.)

Furthermore, only 1 server process and 1 backup agent are started in the default configuration. See below for more on scaling to increase this.

FoundationDB stores all data for all server processes under `/var/lib/foundationdb`. You can override this using `services.foundationdb.dataDir`, e.g.

```
services.foundationdb.dataDir = "/data/fdb";
```

Similarly, logs are stored under `/var/log/foundationdb` by default, and there is a corresponding `services.foundationdb.logDir` as well.

31.2. Scaling processes and backup agents

Scaling the number of server processes is quite easy; simply specify `services.foundationdb.serverProcesses` to be the number of FoundationDB worker processes that should be started on the machine.

FoundationDB worker processes typically require 4GB of RAM per-process at minimum for good performance, so this option is set to 1 by default since the maximum amount of RAM is unknown. You're advised to abide by this restriction, so pick a number of processes so that each has 4GB or more.

A similar option exists in order to scale backup agent processes, `services.foundationdb.backupProcesses`. Backup agents are not as performance/RAM sensitive, so feel free to experiment with the number of available backup processes.

31.3. Clustering

FoundationDB on NixOS works similarly to other Linux systems, so this section will be brief. Please refer to the full FoundationDB documentation for more on clustering.

FoundationDB organizes clusters using a set of *coordinators*, which are just specially-designated worker processes. By default, every installation of FoundationDB on NixOS will start as its own individual cluster, with a single coordinator: the first worker process on **localhost**.

Coordinators are specified globally using the `/etc/foundationdb/fdb.cluster` file, which all servers and client applications will use to find and join coordinators. Note that this file *can not* be managed by NixOS so easily: FoundationDB is designed so that it will rewrite the file at runtime for all clients and nodes when cluster coordinators change, with clients transparently handling this without intervention. It is fundamentally a mutable file, and you should not try to manage it in any way in NixOS.

When dealing with a cluster, there are two main things you want to do:

- Add a node to the cluster for storage/compute.
- Promote an ordinary worker to a coordinator.

A node must already be a member of the cluster in order to properly be promoted to a coordinator, so you must always add it first if you wish to promote it.

To add a machine to a FoundationDB cluster:

- Choose one of the servers to start as the initial coordinator.
- Copy the `/etc/foundationdb/fdb.cluster` file from this server to all the other servers. Restart FoundationDB on all of these other servers, so they join the cluster.
- All of these servers are now connected and working together in the cluster, under the chosen coordinator.

At this point, you can add as many nodes as you want by just repeating the above steps. By default there will still be a single coordinator: you can use `fdbcli` to change this and add new coordinators.

As a convenience, FoundationDB can automatically assign coordinators based on the redundancy mode you wish to achieve for the cluster. Once all the nodes have been joined, simply set the replication policy, and then issue the `coordinators auto` command

For example, assuming we have 3 nodes available, we can enable double redundancy mode, then auto-select coordinators. For double redundancy, 3 coordinators is ideal: therefore FoundationDB will make *every* node a coordinator automatically:

```
fdbcli> configure double ssd
2 fdbcli> coordinators auto
```

This will transparently update all the servers within seconds, and appropriately rewrite the `fdb.cluster` file, as well as informing all client processes to do the same.

31.4. Client connectivity

By default, all clients must use the current `fdb.cluster` file to access a given FoundationDB cluster. This file is located by default in `/etc/foundationdb/fdb.cluster` on all machines with the FoundationDB service enabled, so you may copy the active one from your cluster to a new node in order to connect, if it is not part of the cluster.

31.5. Client authorization and TLS

By default, any user who can connect to a FoundationDB process with the correct cluster configuration can access anything. FoundationDB uses a pluggable design to transport security, and out of the box it supports a LibreSSL-based plugin for TLS support. This plugin not only does in-flight encryption, but also performs client authorization based on the given endpoint's certificate chain. For example, a FoundationDB server may be configured to only accept client connections over TLS, where the client TLS certificate is from organization *Acme Co* in the *Research and Development* unit.

Configuring TLS with FoundationDB is done using the `services.foundationdb.tls` options in order to control the peer verification string, as well as the certificate and its private key.

Note that the certificate and its private key must be accessible to the FoundationDB user account that the server runs under. These files are also NOT managed by NixOS, as putting them into the store may reveal private information.

After you have a key and certificate file in place, it is not enough to simply set the NixOS module options -- you must also configure the `fdb.cluster` file to specify that a given set of coordinators use TLS. This is as simple as adding the suffix `:tls` to your cluster coordinator configuration, after the port number. For example, assuming you have a coordinator on localhost with the default configuration, simply specifying:

```
XXXXXX:XXXXXX@127.0.0.1:4500:tls
```

will configure all clients and server processes to use TLS from now on.

31.6. Backups and Disaster Recovery

The usual rules for doing FoundationDB backups apply on NixOS as written in the FoundationDB manual. However, one important difference is the security profile for NixOS: by default, the `foundationdb` systemd unit uses *Linux namespaces* to restrict write access to the system, except for the log directory, data directory, and the `/etc/foundationdb/` directory. This is enforced by default and cannot be disabled.

However, a side effect of this is that the **fdbbackup** command doesn't work properly for local filesystem backups: FoundationDB uses a server process alongside the database processes to perform backups and copy the backups to the filesystem. As a result, this process is put under the restricted namespaces above: the backup process can only write to a limited number of paths.

In order to allow flexible backup locations on local disks, the FoundationDB NixOS module supports a `services.foundationdb.extraReadWritePaths` option. This option takes a list of paths, and adds them to the systemd unit, allowing the processes inside the service to write (and read) the specified directories.

For example, to create backups in `/opt/fdb-backups`, first set up the paths in the module options:

```
services.foundationdb.extraReadWritePaths = [ "/opt/fdb-backups" ];
```

Restart the FoundationDB service, and it will now be able to write to this directory (even if it does not yet exist.) Note: this path *must* exist before restarting the unit. Otherwise, systemd will not include it in the private FoundationDB namespace (and it will not add it dynamically at runtime).

You can now perform a backup:

```
1 $ sudo -u foundationdb fdbbackup start -t default -d file:///opt/fdb-backups
$ sudo -u foundationdb fdbbackup status -t default
```

31.7. Known limitations

The FoundationDB setup for NixOS should currently be considered beta. FoundationDB is not new software, but the NixOS compilation and integration has only undergone fairly basic testing of all the available functionality.

- There is no way to specify individual parameters for individual **fdbserver** processes. Currently, all server processes inherit all the global **fdbmonitor** settings.
- Ruby bindings are not currently installed.
- Go bindings are not currently installed.

31.8. Options

NixOS's FoundationDB module allows you to configure all of the most relevant configuration options for **fdbmonitor**, matching it quite closely. A complete list of options for the FoundationDB module may be found here. You should also read the FoundationDB documentation as well.

31.9. Full documentation

FoundationDB is a complex piece of software, and requires careful administration to properly use. Full documentation for administration can be found here: <https://apple.github.io/foundationdb/>.

Chapter 32. BorgBackup

Table of Contents

- 32.1. Configuring
- 32.2. Basic usage for a local backup
- 32.3. Create a borg backup server
- 32.4. Backup to the borg repository server
- 32.5. Backup to a hosting service
- 32.6. Vorta backup client for the desktop

Source: `modules/services/backup/borgbackup.nix`

Upstream documentation: <https://borgbackup.readthedocs.io/>

BorgBackup (short: Borg) is a deduplicating backup program. Optionally, it supports compression and authenticated encryption.

The main goal of Borg is to provide an efficient and secure way to backup data. The data deduplication technique used makes Borg suitable for daily backups since only changes are stored. The authenticated encryption technique makes it suitable for backups to not fully trusted targets.

32.1. Configuring

A complete list of options for the Borgbase module may be found [here](#).

32.2. Basic usage for a local backup

A very basic configuration for backing up to a locally accessible directory is:

```
{
2   opt.services.borgbackup.jobs = {
      { rootBackup = {
4         paths = "/";
          exclude = [ "/nix" "/path/to/local/repo" ];
6         repo = "/path/to/local/repo";
          doInit = true;
8         encryption = {
              mode = "repokey";
10            passphrase = "secret";
          };
12        compression = "auto,lzma";
          startAt = "weekly";
14      };
    };
16  };
}
```

Warning

If you do not want the passphrase to be stored in the world-readable Nix store, use `passCommand`. You find an example below.

32.3. Create a borg backup server

You should use a different SSH key for each repository you write to, because the specified keys are restricted to running `borg serve` and can only access this single repository. You need the output of the `generate pub` file.

```
1 # sudo ssh-keygen -N '' -t ed25519 -f /run/keys/id_ed25519_my_borg_repo
# cat /run/keys/id_ed25519_my_borg_repo
3 ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAID78zm0yA+5uPG40t0hfAy+sLDPU1L4AiIoRYEIVbbQ/
   ↪ root@nixos
```

Add the following snippet to your NixOS configuration:

```
1 {
  services.borgbackup.repos = {
3    my_borg_repo = {
      authorizedKeys = [
5        "ssh-ed25519
          ↪ AAAAC3NzaC1lZDI1NTE5AAAAID78zm0yA+5uPG40t0hfAy+sLDPU1L4AiIoRYEIVbbQ/
          ↪ root@nixos"
      ] ;
7    path = "/var/lib/my_borg_repo" ;
  };
}
```

```

9  };
  }

```

32.4. Backup to the borg repository server

The following NixOS snippet creates an hourly backup to the service (on the host nixos) as created in the section above. We assume that you have stored a secret passphrase in the file `/run/keys/borgbackup_passphrase`, which should be only accessible by root

```

{
2  services.borgbackup.jobs = {
    backupToLocalServer = {
4      paths = [ "/etc/nixos" ];
      doInit = true;
6      repo = "borg@nixos:." ;
      encryption = {
8        mode = "repokey-blake2";
        passCommand = "cat /run/keys/borgbackup_passphrase";
10     };
      environment = { BORGRSH = "ssh -i /run/keys/id_ed25519_my_borg_repo"; };
12     compression = "auto,lzma";
      startAt = "hourly";
14   };
  };
16 };

```

The following few commands (run as root) let you test your backup.

```

> nixos-rebuild switch
2 ...restarting the following units: polkit.service
> systemctl restart borgbackup-job-backupToLocalServer
4 > sleep 10
> systemctl restart borgbackup-job-backupToLocalServer
6 > export BORGPASSPHRASE=topSecret
> borg list --rsh='ssh -i /run/keys/id_ed25519_my_borg_repo' borg@nixos:.
8 nixos-backupToLocalServer-2020-03-30T21:46:17 Mon, 2020-03-30 21:46:19
   ↪ [84feb97710954931ca384182f5f3cb90665f35cef214760abd7350fb064786ac]
nixos-backupToLocalServer-2020-03-30T21:46:30 Mon, 2020-03-30 21:46:32
   ↪ [e77321694ecd160ca2228611747c6ad1be177d6e0d894538898de7a2621b6e68]

```

32.5. Backup to a hosting service

Several companies offer (paid) hosting services for Borg repositories.

To backup your home directory to borgbase you have to:

- Generate a SSH key without a password, to access the remote server. E.g.

```

1 sudo ssh-keygen -N '' -t ed25519 -f /run/keys/id_ed25519_borgbase

```

- Create the repository on the server by following the instructions for your hosting server.
- Initialize the repository on the server. Eg.

```

1 sudo borg init --encryption=repokey-blake2 \
   -rsh "ssh -i /run/keys/id_ed25519_borgbase" \
3   zzz2aaaaa@zzz2aaaaa.repo.borgbase.com:repo

```

- Add it to your NixOS configuration, e.g.

```

1 {
    services.borgbackup.jobs = {
3     my_Remote_Backup = {
        paths = [ "/" ];
5         exclude = [ "/nix" "'**/.cache'" ];
        repo = "zzz2aaaaa@zzz2aaaaa.repo.borgbase.com:repo";
7         encryption = {
            mode = "repokey-blake2";
9             passCommand = "cat /run/keys/borgbackup_passphrase";
        };
11        BORG_RSH = "ssh -i /run/keys/id_ed25519_borgbase";
        compression = "auto,lzma";
13        startAt = "daily";
    };
15 };
}}

```

32.6. Vorta backup client for the desktop

Vorta is a backup client for macOS and Linux desktops. It integrates the mighty BorgBackup with your desktop environment to protect your data from disk failure, ransomware and theft.

It can be installed in NixOS e.g. by adding `pkgs.vorta` to `environment.systemPackages`.

Details about using Vorta can be found under <https://vorta.borgbase.com>.

Chapter 33. Hiding process information

Setting

```
security.hideProcessInformation = true;
```

ensures that access to process information is restricted to the owning user. This implies, among other things, that command-line arguments remain private. Unless your deployment relies on unprivileged users being able to inspect the process information of other users, this option should be safe to enable.

Members of the `proc` group are exempt from process information hiding.

To allow a service `foo` to run without process information hiding, set

```
systemd.services.foo.serviceConfig.SupplementaryGroups = [ "proc" ];
```

Chapter 34. SSL/TLS Certificates with ACME

Table of Contents

- 34.1. Prerequisites
- 34.2. Using ACME certificates in Nginx
- 34.3. Using ACME certificates in Apache/httpd
- 34.4. Manual configuration of HTTP-01 validation
- 34.5. Configuring ACME for DNS validation
- 34.6. Regenerating certificates
- 34.7. Fixing JWS Verification error

NixOS supports automatic domain validation & certificate retrieval and renewal using the ACME protocol. Any provider can be used, but by default NixOS uses Let's Encrypt. The alternative ACME client `lego` is used under the hood.

Automatic cert validation and configuration for Apache and Nginx virtual hosts is included in NixOS, however if you would like to generate a wildcard cert or you are not using a web server you will have to configure DNS based validation.

34.1. Prerequisites

To use the ACME module, you must accept the provider's terms of service by setting `security.acme.acceptTerms` to `true`. The Let's Encrypt ToS can be found [here](#).

You must also set an email address to be used when creating accounts with Let's Encrypt. You can set this for all certs with `security.acme.email` and/or on a per-cert basis with `security.acme.certs.<name>.email`. This address is only used for registration and renewal reminders, and cannot be used to administer the certificates in any way.

Alternatively, you can use a different ACME server by changing the `security.acme.server` option to a provider of your choosing, or just change the server for one cert with `security.acme.certs.<name>.server`.

You will need an HTTP server or DNS server for verification. For HTTP, the server must have a webroot defined that can serve `.well-known/acme-challenge`. This directory must be writeable by the user that will run the ACME client. For DNS, you must set up credentials with your provider/server for use with `lego`.

34.2. Using ACME certificates in Nginx

NixOS supports fetching ACME certificates for you by setting `enableACME = true;` in a virtualHost config. We first create self-signed placeholder certificates in place of the real ACME certs. The placeholder certs are overwritten when the ACME certs arrive. For `foo.example.com` the config would look like.

```
security.acme.acceptTerms = true;
2 security.acme.email = "admin+acme@example.com";
services.nginx = {
4   enable = true;
   virtualHosts = {
6     "foo.example.com" = {
        forceSSL = true;
8       enableACME = true;
        # All serverAliases will be added as extra domain names on the certificate.
10      serverAliases = [ "bar.example.com" ];
        locations."/\" = {
12        root = "/var/www";
        };
14    };

16    # We can also add a different vhost and reuse the same certificate
    # but we have to append extraDomainNames manually.
18    security.acme.certs."foo.example.com".extraDomainNames = [ "baz.example.com" ];
    "baz.example.com" = {
20      forceSSL = true;
      useACMEHost = "foo.example.com";
22      locations."/\" = {
        root = "/var/www";
24      };
    };
26  };
}
```

34.3. Using ACME certificates in Apache/httpd

Using ACME certificates with Apache virtual hosts is identical to using them with Nginx. The attribute names are all the same, just replace "nginx" with "httpd" where appropriate.

34.4. Manual configuration of HTTP-01 validation

First off you will need to set up a virtual host to serve the challenges. This example uses a vhost called `certs.example.com`, with the intent that you will generate certs for all your vhosts and redirect everyone to HTTPS.

```
security.acme.acceptTerms = true;
2 security.acme.email = "admin+acme@example.com";
services.nginx = {
4   enable = true;
   virtualHosts = {
6     "acmechallenge.example.com" = {
       # Catchall vhost, will redirect users to HTTPS for all vhosts
8       serverAliases = [ "*.example.com" ];
       # /var/lib/acme/.challenges must be writable by the ACME user
10      # and readable by the Nginx user.
       # By default, this is the case.
12      locations["/.well-known/acme-challenge"] = {
        root = "/var/lib/acme/.challenges";
14      };
      locations["/"] = {
16        return = "301 https://$host$request_uri";
      };
18    };
  };
20 }
# Alternative config for Apache
22 services.httpd = {
   enable = true;
24   virtualHosts = {
     "acmechallenge.example.com" = {
26       # Catchall vhost, will redirect users to HTTPS for all vhosts
       serverAliases = [ "*.example.com" ];
28       # /var/lib/acme/.challenges must be writable by the ACME user and readable
       ↪ by the Apache user.
       # By default, this is the case.
30       documentRoot = "/var/lib/acme/.challenges";
       extraConfig = ''
32         RewriteEngine On
         RewriteCond %{HTTPS} off
34         RewriteCond %{REQUEST_URI} !^/\..well-known/acme-challenge [NC]
         RewriteRule (.*?) https://%{HTTP_HOST}%{REQUEST_URI} [R=301]
36       '';
     };
38   };
}
```

Now you need to configure ACME to generate a certificate.

```
1 security.acme.certs."foo.example.com" = {
   webroot = "/var/lib/acme/.challenges";
3   email = "foo@example.com";
   # Ensure that the web server you use can read the generated certs
```

```

5  # Take a look at the group option for the web server you choose.
   group = "nginx";
7  # Since we have a wildcard vhost to handle port 80,
   # we can generate certs for anything!
9  # Just make sure your DNS resolves them.
   extraDomainNames = [ "mail.example.com" ];
11 };

```

The private key `key.pem` and certificate `fullchain.pem` will be put into `/var/lib/acme/foo.example.com`.

Refer to Appendix A, *Configuration Options* for all available configuration options for the `security.acme` module.

34.5. Configuring ACME for DNS validation

This is useful if you want to generate a wildcard certificate, since ACME servers will only hand out wildcard certs over DNS validation. There a number of supported DNS providers and servers you can utilise, see the `lego` docs for provider/server specific configuration values. For the sake of these docs, we will provide a fully self-hosted example using `bind`.

```

services.bind = {
2  enable = true;
   extraConfig = ''
4  include "/var/lib/secrets/dnskeys.conf";
   '';
6  zones = [
     rec {
8      name = "example.com";
       file = "/var/db/bind/${name}";
10     master = true;
       extraConfig = "allow-update { key rfc2136key.example.com.; }";
12     }
   ];
14 }

16 # Now we can configure ACME
security.acme.acceptTerms = true;
18 security.acme.email = "admin+acme@example.com";
security.acme.certs."example.com" = {
20  domain = "*.example.com";
   dnsProvider = "rfc2136";
22  credentialsFile = "/var/lib/secrets/certs.secret";
   # We don't need to wait for propagation since this is a local DNS server
24  dnsPropagationCheck = false;
};

```

The `dnskeys.conf` and `certs.secret` must be kept secure and thus you should not keep their contents in your Nix config. Instead, generate them one time with these commands:

```

mkdir -p /var/lib/secrets
2  tsig-keygen rfc2136key.example.com > /var/lib/secrets/dnskeys.conf
   chown named:root /var/lib/secrets/dnskeys.conf
4  chmod 400 /var/lib/secrets/dnskeys.conf

6  # Copy the secret value from the dnskeys.conf, and put it in
   # RFC2136_TSIG_SECRET below
8
   cat > /var/lib/secrets/certs.secret << EOF
10 RFC2136_NAMESERVER='127.0.0.1:53'

```

```

RFC2136_TSIG_ALGORITHM='hmac-sha256.'
12 RFC2136_TSIG_KEY='rfc2136key.example.com'
RFC2136_TSIG_SECRET='your secret key'
14 EOF
chmod 400 /var/lib/secrets/certs.secret

```

Now you're all set to generate certs! You should monitor the first invocation by running `systemctl start ↪ acme-example.com.service & journalctl -fu acme-example.com.service` and watching its log output.

34.6. Regenerating certificates

Should you need to regenerate a particular certificate in a hurry, such as when a vulnerability is found in Let's Encrypt, there is now a convenient mechanism for doing so. Running `systemctl clean --what=state ↪ acme-example.com.service` will remove all certificate files and the account data for the given domain, allowing you to then `systemctl start acme-example.com.service` to generate fresh ones.

34.7. Fixing JWS Verification error

It is possible that your account credentials file may become corrupt and need to be regenerated. In this scenario `lego` will produce the error `JWS verification error`. The solution is to simply delete the associated accounts file and re-run the affected service(s).

```

# Find the accounts folder for the certificate
2 systemctl cat acme-example.com.service | grep -Po 'accounts/[^:]*'
export accountdir="$(!)"
4 # Move this folder to some place else
mv /var/lib/acme/.lego/$accountdir{,.bak}
6 # Recreate the folder using systemd-tmpfiles
systemd-tmpfiles --create
8 # Get a new account and reissue certificates
# Note: Do this for all certs that share the same account email address
10 systemctl start acme-example.com.service

```

Chapter 35. Oh my ZSH

Table of Contents

- 35.1. Basic usage
- 35.2. Custom additions
- 35.3. Custom environments
- 35.4. Package your own customizations

`oh-my-zsh` is a framework to manage your ZSH configuration including completion scripts for several CLI tools or custom prompt themes.

35.1. Basic usage

The module uses the `oh-my-zsh` package with all available features. The initial setup using Nix expressions is fairly similar to the configuration format of `oh-my-zsh`.

```

{
2   programs.zsh.ohMyZsh = {
      enable = true;
4     plugins = [ "git" "python" "man" ];
      theme = "agnoster";
6   };
}

```

For a detailed explanation of these arguments please refer to the `oh-my-zsh` docs.

The expression generates the needed configuration and writes it into your `/etc/zshrc`.

35.2. Custom additions

Sometimes third-party or custom scripts such as a modified theme may be needed. `oh-my-zsh` provides the `ZSH_CUSTOM` environment variable for this which points to a directory with additional scripts.

The module can do this as well:

```
2 {
  programs.zsh.ohMyZsh.custom = "~/path/to/custom/scripts";
}
```

35.3. Custom environments

There are several extensions for `oh-my-zsh` packaged in `nixpkgs`. One of them is `nix-zsh-completions` which bundles completion scripts and a plugin for `oh-my-zsh`.

Rather than using a single mutable path for `ZSH_CUSTOM`, it's also possible to generate this path from a list of Nix packages:

```
2 { pkgs, ... }:
3 {
4   programs.zsh.ohMyZsh.customPkgs = [
5     pkgs.nix-zsh-completions
6     # and even more...
7   ];
8 }
```

Internally a single store path will be created using `buildEnv`. Please refer to the docs of `buildEnv` for further reference.

Please keep in mind that this is not compatible with `programs.zsh.ohMyZsh.custom` as it requires an immutable store path while `custom` shall remain mutable! An evaluation failure will be thrown if both `custom` and `customPkgs` are set.

35.4. Package your own customizations

If third-party customizations (e.g. new themes) are supposed to be added to `oh-my-zsh` there are several pitfalls to keep in mind:

- To comply with the default structure of ZSH the entire output needs to be written to `$out/share/zsh`.
- Completion scripts are supposed to be stored at `$out/share/zsh/site-functions`. This directory is part of the `fpath` and the package should be compatible with pure ZSH setups. The module will automatically link the contents of `site-functions` to completions directory in the proper store path.
- The `plugins` directory needs the structure `pluginname/pluginname.plugin.zsh` as structured in the upstream repo.

A derivation for `oh-my-zsh` may look like this:

```
2 { stdenv, fetchFromGitHub }:
3
4 stdenv.mkDerivation rec {
5   name = "exemplary-zsh-customization-${version}";
6   version = "1.0.0";
7   src = fetchFromGitHub {
8     # path to the upstream repository
9   };
10 }
```

```

10  dontBuild = true;
    installPhase = ''
12      mkdir -p $out/share/zsh/site-functions
      cp {themes,plugins} $out/share/zsh
14      cp completions $out/share/zsh/site-functions
      '';
16 }

```

Chapter 36. Plotinus

Source: `modules/programs/plotinus.nix`

Upstream documentation: <https://github.com/p-e-w/plotinus>

Plotinus is a searchable command palette in every modern GTK application.

When in a GTK 3 application and Plotinus is enabled, you can press **Ctrl+Shift+P** to open the command palette. The command palette provides a searchable list of all menu items in the application.

To enable Plotinus, add the following to your `configuration.nix`:

```
programs.plotinus.enable = true;
```

Chapter 37. Digital Bitbox

Table of Contents

37.1. Package

37.2. Hardware

Digital Bitbox is a hardware wallet and second-factor authenticator.

The `digitalbitbox` programs module may be installed by setting `programs.digitalbitbox` to `true` in a manner similar to

```
programs.digitalbitbox.enable = true;
```

and bundles the `digitalbitbox` package (see Section 37.1, “Package”), which contains the `dbb-app` and `dbb-cli` binaries, along with the hardware module (see Section 37.2, “Hardware”) which sets up the necessary udev rules to access the device.

Enabling the `digitalbitbox` module is pretty much the easiest way to get a Digital Bitbox device working on your system.

For more information, see https://digitalbitbox.com/start_linux.

37.1. Package

The binaries, `dbb-app` (a GUI tool) and `dbb-cli` (a CLI tool), are available through the `digitalbitbox` package which could be installed as follows:

```

2  environment.systemPackages = [
    pkgs.digitalbitbox
  ];

```

37.2. Hardware

The digitalbitbox hardware package enables the udev rules for Digital Bitbox devices and may be installed as follows:

```
1 hardware.digitalbitbox.enable = true;
```

In order to alter the udev rules, one may provide different values for the `udevRule51` and `udevRule52` attributes by means of overriding as follows:

```
programs.digitalbitbox = {
2   enable = true;
   package = pkgs.digitalbitbox.override {
4     udevRule51 = "something else";
   };
6 };
```

Chapter 38. Input Methods

Table of Contents

38.1. IBus

38.2. Fcitx

38.3. Nabi

38.4. Uim

Input methods are an operating system component that allows any data, such as keyboard strokes or mouse movements, to be received as input. In this way users can enter characters and symbols not found on their input devices. Using an input method is obligatory for any language that has more graphemes than there are keys on the keyboard.

The following input methods are available in NixOS:

- IBus: The intelligent input bus.
- Fcitx: A customizable lightweight input method.
- Nabi: A Korean input method based on XIM.
- Uim: The universal input method, is a library with a XIM bridge.

38.1. IBus

IBus is an Intelligent Input Bus. It provides full featured and user friendly input method user interface.

The following snippet can be used to configure IBus:

```
i18n.inputMethod = {
2   enabled = "ibus";
   ibus.engines = with pkgs.ibus-engines; [ anthy hangul mozc ];
4 };
```

`i18n.inputMethod.ibus.engines` is optional and can be used to add extra IBus engines.

Available extra IBus engines are:

- Anthy (`ibus-engines.anthy`): Anthy is a system for Japanese input method. It converts Hiragana text to Kana Kanji mixed text.
- Hangul (`ibus-engines.hangul`): Korean input method.

- `m17n (ibus-engines.m17n)`: `m17n` is an input method that uses input methods and corresponding icons in the `m17n` database.
- `mozc (ibus-engines.mozc)`: A Japanese input method from Google.
- `Table (ibus-engines.table)`: An input method that load tables of input methods.
- `table-others (ibus-engines.table-others)`: Various table-based input methods. To use this, and any other table-based input methods, it must appear in the list of engines along with `table`. For example:

```
ibus.engines = with pkgs.ibus-engines; [ table table-others ];
```

To use any input method, the package must be added in the configuration, as shown above, and also (after running `nixos-rebuild`) the input method must be added from IBus' preference dialog.

Troubleshooting

If IBus works in some applications but not others, a likely cause of this is that IBus is depending on a different version of `glib` to what the applications are depending on. This can be checked by running `nix-store -q --requisites ↵ <path> | grep glib`, where `<path>` is the path of either IBus or an application in the Nix store. The `glib` packages must match exactly. If they do not, uninstalling and reinstalling the application is a likely fix.

38.2. Fcix

Fcix is an input method framework with extension support. It has three built-in Input Method Engine, Pinyin, QuWei and Table-based input methods.

The following snippet can be used to configure Fcix:

```
i18n.inputMethod = {
2   enabled = "fcitx";
   fcitx.engines = with pkgs.fcitx-engines; [ mozc hangul m17n ];
4 };
```

`i18n.inputMethod.fcitx.engines` is optional and can be used to add extra Fcix engines.

Available extra Fcix engines are:

- `Anthy (fcitx-engines.anthy)`: Anthy is a system for Japanese input method. It converts Hiragana text to Kana Kanji mixed text.
- `Chewing (fcitx-engines.chewing)`: Chewing is an intelligent Zhuyin input method. It is one of the most popular input methods among Traditional Chinese Unix users.
- `Hangul (fcitx-engines.hangul)`: Korean input method.
- `Unikey (fcitx-engines.unikey)`: Vietnamese input method.
- `m17n (fcitx-engines.m17n)`: `m17n` is an input method that uses input methods and corresponding icons in the `m17n` database.
- `mozc (fcitx-engines.mozc)`: A Japanese input method from Google.
- `table-others (fcitx-engines.table-others)`: Various table-based input methods.

38.3. Nabi

Nabi is an easy to use Korean X input method. It allows you to enter phonetic Korean characters (`hangul`) and pictographic Korean characters (`hanja`).

The following snippet can be used to configure Nabi:

```
i18n.inputMethod = {
2   enabled = "nabi";
};
```

38.4. Uim

Uim (short for "universal input method") is a multilingual input method framework. Applications can use it through so-called bridges.

The following snippet can be used to configure uim:

```
1 i18n.inputMethod = {  
    enabled = "uim";  
3 };
```

Note: The `i18n.inputMethod.uim.toolbar` option can be used to choose uim toolbar.

Chapter 39. Profiles

Table of Contents

- 39.1. All Hardware
- 39.2. Base
- 39.3. Clone Config
- 39.4. Demo
- 39.5. Docker Container
- 39.6. Graphical
- 39.7. Hardened
- 39.8. Headless
- 39.9. Installation Device
- 39.10. Minimal
- 39.11. QEMU Guest

In some cases, it may be desirable to take advantage of commonly-used, predefined configurations provided by `nixpkgs`, but different from those that come as default. This is a role fulfilled by NixOS's Profiles, which come as files living in `<nixpkgs/nixos/modules/profiles>`. That is to say, expected usage is to add them to the imports list of your `/etc/configuration.nix` as such:

```
1 imports = [  
2   <nixpkgs/nixos/modules/profiles/profile-name.nix>  
   ];
```

Even if some of these profiles seem only useful in the context of install media, many are actually intended to be used in real installs.

What follows is a brief explanation on the purpose and use-case for each profile. Detailing each option configured by each one is out of scope.

39.1. All Hardware

Enables all hardware supported by NixOS: i.e., all firmware is included, and all devices from which one may boot are enabled in the `initrd`. Its primary use is in the NixOS installation CDs.

The enabled kernel modules include support for SATA and PATA, SCSI (partially), USB, Firewire (untested), Virtio (QEMU, KVM, etc.), VMware, and Hyper-V. Additionally, `hardware.enableAllFirmware` is enabled, and the firmware for the ZyDAS ZD1211 chipset is specifically installed.

39.2. Base

Defines the software packages included in the "minimal" installation CD. It installs several utilities useful in a simple recovery or install media, such as a text-mode web browser, and tools for manipulating block devices, networking, hardware diagnostics, and filesystems (with their respective kernel modules).

39.3. Clone Config

This profile is used in installer images. It provides an editable `configuration.nix` that imports all the modules that were also used when creating the image in the first place. As a result it allows users to edit and rebuild the live-system.

On images where the installation media also becomes an installation target, copying over `configuration.nix` should be disabled by setting `installer.cloneConfig` to `false`. For example, this is done in `sd-image-aarch64.nix`.

39.4. Demo

This profile just enables a `demo` user, with password `demo`, uid 1000, `wheel` group and autologin in the SDDM display manager.

39.5. Docker Container

This is the profile from which the Docker images are generated. It prepares a working system by importing the Minimal and Clone Config profiles, and setting appropriate configuration options that are useful inside a container context, like `boot.isContainer`.

39.6. Graphical

Defines a NixOS configuration with the Plasma 5 desktop. It's used by the graphical installation CD.

It sets `services.xserver.enable`, `services.xserver.displayManager.sddm.enable`, `services.xserver.desktopManager.plasma5.enable` and `services.xserver.libinput.enable` to `true`. It also includes `glxinfo` and `firefox` in the system packages list.

39.7. Hardened

A profile with most (vanilla) hardening options enabled by default, potentially at the cost of stability, features and performance.

This includes a hardened kernel, and limiting the system information available to processes through the `/sys` and `/proc` filesystems. It also disables the User Namespaces feature of the kernel, which stops Nix from being able to build anything (this particular setting can be overridden via `security.allowUserNamespaces`). See the `profile` source for further detail on which settings are altered.

Warning

This profile enables options that are known to affect system stability. If you experience any stability issues when using the profile, try disabling it. If you report an issue and use this profile, always mention that you do.

39.8. Headless

Common configuration for headless machines (e.g., Amazon EC2 instances).

Disables sound, vesa, serial consoles, emergency mode, grub splash images and configures the kernel to reboot automatically on panic.

39.9. Installation Device

Provides a basic configuration for installation devices like CDs. This enables redistributable firmware, includes the Clone Config profile and a copy of the Nixpkgs channel, so `nixos-install` works out of the box.

Documentation for Nixpkgs and NixOS are forcefully enabled (to override the Minimal profile preference); the NixOS manual is shown automatically on TTY 8, `udisks` is disabled. Autologin is enabled as `nixos` user, while

passwordless login as both `root` and `nixos` is possible. Passwordless `sudo` is enabled too. `wpa_supplicant` is enabled, but configured to not autostart.

It is explained how to login, start the ssh server, and if available, how to start the display manager.

Several settings are tweaked so that the installer has a better chance of succeeding under low-memory environments.

39.10. Minimal

This profile defines a small NixOS configuration. It does not contain any graphical stuff. It's a very short file that enables `noXlibs`, sets `i18n.supportedLocales` to only support the user-selected locale, disables packages' documentation, and disables sound.

39.11. QEMU Guest

This profile contains common configuration for virtual machines running under QEMU (using `virtio`).

It makes `virtio` modules available on the `initrd`, sets the system time from the hardware clock to work around a bug in `qemu-kvm`, and enables `rngd`.

Chapter 40. Kubernetes

The NixOS Kubernetes module is a collective term for a handful of individual submodules implementing the Kubernetes cluster components.

There are generally two ways of enabling Kubernetes on NixOS. One way is to enable and configure cluster components appropriately by hand:

```
services.kubernetes = {
2   apiserver.enable = true;
   controllerManager.enable = true;
4   scheduler.enable = true;
   addonManager.enable = true;
6   proxy.enable = true;
   flannel.enable = true;
8 };
```

Another way is to assign cluster roles ("master" and/or "node") to the host. This enables `apiserver`, `controllerManager`, `scheduler`, `addonManager`, `kube-proxy` and `etcd`:

```
services.kubernetes.roles = [ "master" ];
```

While this will enable the `kubelet` and `kube-proxy` only:

```
1 services.kubernetes.roles = [ "node" ];
```

Assigning both the master and node roles is usable if you want a single node Kubernetes cluster for dev or testing purposes:

```
1 services.kubernetes.roles = [ "master" "node" ];
```

Note: Assigning either role will also default both `services.kubernetes.flannel.enable` and `services.kubernetes.easyCerts` to true. This sets up flannel as CNI and activates automatic PKI bootstrapping.

As of kubernetes 1.10.X it has been deprecated to open non-tls-enabled ports on kubernetes components. Thus, from NixOS 19.03 all plain HTTP ports have been disabled by default. While opening insecure ports is still possible, it is recommended not to bind these to other interfaces than loopback. To re-enable the insecure port on the `apiserver`, see options: `services.kubernetes.apiserver.insecurePort` and `services.kubernetes.apiserver.insecureBindAddress`

Note

As of NixOS 19.03, it is mandatory to configure: `services.kubernetes.masterAddress`. The `masterAddress` must be resolveable and routeable by all cluster nodes. In single node clusters, this can be set to `localhost`.

Role-based access control (RBAC) authorization mode is enabled by default. This means that anonymous requests to the apiserver secure port will expectedly cause a permission denied error. All cluster components must therefore be configured with x509 certificates for two-way tls communication. The x509 certificate subject section determines the roles and permissions granted by the apiserver to perform clusterwide or namespaced operations. See also: Using RBAC Authorization.

The NixOS `kubernetes` module provides an option for automatic certificate bootstrapping and configuration, `services.kubernetes.easyCerts`. The PKI bootstrapping process involves setting up a certificate authority (CA) daemon (`cfssl`) on the kubernetes master node. `cfssl` generates a CA-cert for the cluster, and uses the CA-cert for signing subordinate certs issued to each of the cluster components. Subsequently, the `certmgr` daemon monitors active certificates and renews them when needed. For single node Kubernetes clusters, setting `services.kubernetes.easyCerts = true` is sufficient and no further action is required. For joining extra node machines to an existing cluster on the other hand, establishing initial trust is mandatory.

To add new nodes to the cluster: On any (non-master) cluster node where `services.kubernetes.easyCerts` is enabled, the helper script `nixos-kubernetes-node-join` is available on `PATH`. Given a token on `stdin`, it will copy the token to the kubernetes secrets directory and restart the `certmgr` service. As requested certificates are issued, the script will restart kubernetes cluster components as needed for them to pick up new keypairs.

Note

Multi-master (HA) clusters are not supported by the `easyCerts` module.

In order to interact with an RBAC-enabled cluster as an administrator, one needs to have cluster-admin privileges. By default, when `easyCerts` is enabled, a cluster-admin kubeconfig file is generated and linked into `/etc/kubernetes/cluster-admin.kubeconfig` as determined by `services.kubernetes.pki.etcClusterAdminKubeconfig`. `export KUBECONFIG=/etc/kubernetes/cluster-admin.kubeconfig` will make `kubectl` use this kubeconfig to access and authenticate the cluster. The cluster-admin kubeconfig references an auto-generated keypair owned by `root`. Thus, only `root` on the kubernetes master may obtain cluster-admin rights by means of this file.

Part III. Administration

This chapter describes various aspects of managing a running NixOS system, such as how to use the `systemd` service manager.

Table of Contents

- 41. Service Management
- 42. Rebooting and Shutting Down
- 43. User Sessions
- 44. Control Groups
- 45. Logging
- 46. Cleaning the Nix Store
- 47. Container Management
- 48. Troubleshooting

Chapter 41. Service Management

In NixOS, all system services are started and monitored using the `systemd` program. `Systemd` is the “init” process of the system (i.e. PID 1), the parent of all other processes. It manages a set of so-called “units”, which can be things like system services (programs), but also mount points, swap files, devices, targets (groups of units) and more. Units can have complex dependencies; for instance, one unit can require that another unit must be successfully started before the first unit can be started. When the system boots, it starts a unit named `default.target`;

the dependencies of this unit cause all system services to be started, file systems to be mounted, swap files to be activated, and so on.

The command **systemctl** is the main way to interact with **systemd**. Without any arguments, it shows the status of active units:

```
$ systemctl
2 -.mount          loaded active mounted   /
  swapfile.swap    loaded active active     /swapfile
4 sshd.service     loaded active running   SSH Daemon
  graphical.target loaded active active     Graphical Interface
6 ...
```

You can ask for detailed status information about a unit, for instance, the PostgreSQL database service:

```
$ systemctl status postgresql.service
2 postgresql.service - PostgreSQL Server
   Loaded: loaded
           ↪ (/nix/store/pn3q73mvh75gsrl8w7fdlfk3fq5qm5mw-unit/postgresql.service)
4   Active: active (running) since Mon, 2013-01-07 15:55:57 CET; 9h ago
   Main PID: 2390 (postgres)
6   CGroup: name=systemd:/system/postgresql.service
           2390 postgres
           2418 postgres: writer process
           2419 postgres: wal writer process
10          2420 postgres: autovacuum launcher process
           2421 postgres: stats collector process
12          2498 postgres: zabbix zabbix [local] idle

14 Jan 07 15:55:55 hagbard postgres[2394]: [1-1] LOG:  database system was shut down
           ↪ at 2013-01-07 15:55:05 CET
   Jan 07 15:55:57 hagbard postgres[2390]: [1-1] LOG:  database system is ready to
           ↪ accept connections
16 Jan 07 15:55:57 hagbard postgres[2420]: [1-1] LOG:  autovacuum launcher started
   Jan 07 15:55:57 hagbard systemd[1]: Started PostgreSQL Server.
```

Note that this shows the status of the unit (active and running), all the processes belonging to the service, as well as the most recent log messages from the service.

Units can be stopped, started or restarted:

```
1 # systemctl stop postgresql.service
  # systemctl start postgresql.service
3 # systemctl restart postgresql.service
```

These operations are synchronous: they wait until the service has finished starting or stopping (or has failed). Starting a unit will cause the dependencies of that unit to be started as well (if necessary).

Chapter 42. Rebooting and Shutting Down

The system can be shut down (and automatically powered off) by doing:

```
1 # shutdown
```

This is equivalent to running **systemctl poweroff**.

To reboot the system, run

```
1 # reboot
```

which is equivalent to **systemctl reboot**. Alternatively, you can quickly reboot the system using **kexec**, which bypasses the BIOS by directly loading the new kernel into memory:

```
# systemctl kexec
```

The machine can be suspended to RAM (if supported) using **systemctl suspend**, and suspended to disk using **systemctl hibernate**.

These commands can be run by any user who is logged in locally, i.e. on a virtual console or in X11; otherwise, the user is asked for authentication.

Chapter 43. User Sessions

Systemd keeps track of all users who are logged into the system (e.g. on a virtual console or remotely via SSH). The command **loginctl** allows querying and manipulating user sessions. For instance, to list all user sessions:

```
1 $ loginctl
   SESSION      UID USER      SEAT
3      c1        500 eelco      seat0
      c3         0 root       seat0
5      c4        500 alice
```

This shows that two users are logged in locally, while another is logged in remotely. (“Seats” are essentially the combinations of displays and input devices attached to the system; usually, there is only one seat.) To get information about a session:

```
1 $ loginctl session-status c3
c3 - root (0)
3      Since: Tue, 2013-01-08 01:17:56 CET; 4min 42s ago
      Leader: 2536 (login)
5      Seat: seat0; vc3
      TTY: /dev/tty3
7      Service: login; type tty; class user
      State: online
9      CGroup: name=systemd:/user/root/c3
              2536
              ↪ /nix/store/10mn4xip9n7y9bxqwnsx7xwx2v2g34xn-shadow-4.1.5.1/bin/login
              ↪ --
11      10339 -bash
              10355 w3m nixos.org
```

This shows that the user is logged in on virtual console 3. It also lists the processes belonging to this session. Since systemd keeps track of this, you can terminate a session in a way that ensures that all the session’s processes are gone:

```
# loginctl terminate-session c3
```

Chapter 44. Control Groups

To keep track of the processes in a running system, systemd uses *control groups* (cgroups). A control group is a set of processes used to allocate resources such as CPU, memory or I/O bandwidth. There can be multiple control group hierarchies, allowing each kind of resource to be managed independently.

The command **systemd-cgls** lists all control groups in the **systemd** hierarchy, which is what systemd uses to keep track of the processes belonging to each service or user session:

```
$ systemd-cgls
2 user
  eelco
```

```

4      c1
      2567 -:0
6      2682 kdeinit4: kdeinit4 Running...
      ...
8      10851 sh -c less -R
system
10     httpd.service
      2444 httpd -f /nix/store/3pyacby5cpr55a03qwbndizpciwq161-httpd.conf
      ↪ -DNO_DETACH
12     ...
      dhcpd.service
14     2376 dhcpd --config /nix/store/f8dif8dsi2yaa70n03xir8r653776ka6-dhcpd.conf
      ...

```

Similarly, **systemd-cgls cpu** shows the cgroups in the CPU hierarchy, which allows per-cgroup CPU scheduling priorities. By default, every systemd service gets its own CPU cgroup, while all user sessions are in the top-level CPU cgroup. This ensures, for instance, that a thousand run-away processes in the **httpd.service** cgroup cannot starve the CPU for one process in the **postgresql.service** cgroup. (By contrast, if they were in the same cgroup, then the PostgreSQL process would get 1/1001 of the cgroup's CPU time.) You can limit a service's CPU share in **configuration.nix**:

```
systemd.services.httpd.serviceConfig.CPUShares = 512;
```

By default, every cgroup has 1024 CPU shares, so this will halve the CPU allocation of the **httpd.service** cgroup.

There also is a **memory** hierarchy that controls memory allocation limits; by default, all processes are in the top-level cgroup, so any service or session can exhaust all available memory. Per-cgroup memory limits can be specified in **configuration.nix**; for instance, to limit **httpd.service** to 512 MiB of RAM (excluding swap):

```
systemd.services.httpd.serviceConfig.MemoryLimit = "512M";
```

The command **systemd-cgtop** shows a continuously updated list of all cgroups with their CPU and memory usage.

Chapter 45. Logging

System-wide logging is provided by systemd's *journal*, which subsumes traditional logging daemons such as syslogd and klogd. Log entries are kept in binary files in **/var/log/journal/**. The command **journalctl** allows you to see the contents of the journal. For example,

```
$ journalctl -b
```

shows all journal entries since the last reboot. (The output of **journalctl** is piped into **less** by default.) You can use various options and match operators to restrict output to messages of interest. For instance, to get all messages from PostgreSQL:

```

1 $ journalctl -u postgresql.service
  -- Logs begin at Mon, 2013-01-07 13:28:01 CET, end at Tue, 2013-01-08 01:09:57
  ↪ CET. --
3 ...
Jan 07 15:44:14 hagbard postgres[2681]: [2-1] LOG:  database system is shut down
5 -- Reboot --
Jan 07 15:45:10 hagbard postgres[2532]: [1-1] LOG:  database system was shut down
  ↪ at 2013-01-07 15:44:14 CET
7 Jan 07 15:45:13 hagbard postgres[2500]: [1-1] LOG:  database system is ready to
  ↪ accept connections

```

Or to get all messages since the last reboot that have at least a “critical” severity level:

```

1 $ journalctl -b -p crit
   Dec 17 21:08:06 mandark sudo[3673]: pam_unix(sudo:auth): auth could not identify
     ↪ password for [alice]
3  Dec 29 01:30:22 mandark kernel[6131]: [1053513.909444] CPU6: Core temperature
     ↪ above threshold, cpu clock throttled (total events = 1)

```

The system journal is readable by root and by users in the `wheel` and `systemd-journal` groups. All users have a private journal that can be read using `journalctl`.

Chapter 46. Cleaning the Nix Store

Table of Contents

46.1. NixOS Boot Entries

Nix has a purely functional model, meaning that packages are never upgraded in place. Instead new versions of packages end up in a different location in the Nix store (`/nix/store`). You should periodically run Nix's *garbage collector* to remove old, unreferenced packages. This is easy:

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

Alternatively, you can use a systemd unit that does the same in the background:

```
1 # systemctl start nix-gc.service
```

You can tell NixOS in `configuration.nix` to run this unit automatically at certain points in time, for instance, every night at 03:15:

```

nix.gc.automatic = true;
2 nix.gc.dates = "03:15";

```

The commands above do not remove garbage collector roots, such as old system configurations. Thus they do not remove the ability to roll back to previous configurations. The following command deletes old roots, removing the ability to roll back to them:

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

You can also do this for specific profiles, e.g.

```
1 $ nix-env -p /nix/var/nix/profiles/per-user/eelco/profile --delete-generations old
```

Note that NixOS system configurations are stored in the profile `/nix/var/nix/profiles/system`.

Another way to reclaim disk space (often as much as 40% of the size of the Nix store) is to run Nix's store optimiser, which seeks out identical files in the store and replaces them with hard links to a single copy.

```
$ nix-store --optimise
```

Since this command needs to read the entire Nix store, it can take quite a while to finish.

46.1. NixOS Boot Entries

If your `/boot` partition runs out of space, after clearing old profiles you must rebuild your system with `nixos-rebuild` to update the `/boot` partition and clear space.

Chapter 47. Container Management

Table of Contents

47.1. Imperative Container Management

47.2. Declarative Container Specification

47.3. Container Networking

NixOS allows you to easily run other NixOS instances as *containers*. Containers are a light-weight approach to virtualisation that runs software in the container at the same speed as in the host system. NixOS containers share the Nix store of the host, making container creation very efficient.

Warning

Currently, NixOS containers are not perfectly isolated from the host system. This means that a user with root access to the container can do things that affect the host. So you should not give container root access to untrusted users.

NixOS containers can be created in two ways: imperatively, using the command **nixos-container**, and declaratively, by specifying them in your **configuration.nix**. The declarative approach implies that containers get upgraded along with your host system when you run **nixos-rebuild**, which is often not what you want. By contrast, in the imperative approach, containers are configured and updated independently from the host system.

47.1. Imperative Container Management

We'll cover imperative container management using **nixos-container** first. Be aware that container management is currently only possible as **root**.

You create a container with identifier **foo** as follows:

```
# nixos-container create foo
```

This creates the container's root directory in `/var/lib/containers/foo` and a small configuration file in `/etc/containers/foo.conf`. It also builds the container's initial system configuration and stores it in `/nix/var/nix/profiles/per-container/foo/system`. You can modify the initial configuration of the container on the command line. For instance, to create a container that has **sshd** running, with the given public key for **root**:

```
# nixos-container create foo --config '
2  services.openssh.enable = true;
  users.users.root.openssh.authorizedKeys.keys = ["ssh-dss AAAAB3N..."];
4  '
```

By default the next free address in the 10.233.0.0/16 subnet will be chosen as container IP. This behavior can be altered by setting **--host-address** and **--local-address**:

```
# nixos-container create test --config-file test-container.nix \
2  --local-address 10.235.1.2 --host-address 10.235.1.1
```

Creating a container does not start it. To start the container, run:

```
# nixos-container start foo
```

This command will return as soon as the container has booted and has reached **multi-user.target**. On the host, the container runs within a systemd unit called **container@container-name.service**. Thus, if something went wrong, you can get status info using **systemctl**:

```
# systemctl status container@foo
```

If the container has started successfully, you can log in as root using the **root-login** operation:

```
1 # nixos-container root-login foo
  [root@foo:~]#
```

Note that only root on the host can do this (since there is no authentication). You can also get a regular login prompt using the **login** operation, which is available to all users on the host:

```
# nixos-container login foo
2 foo login: alice
Password: ***
```

With **nixos-container run**, you can execute arbitrary commands in the container:

```
1 # nixos-container run foo -- uname -a
Linux foo 3.4.82 #1-NixOS SMP Thu Mar 20 14:44:05 UTC 2014 x86_64 GNU/Linux
```

There are several ways to change the configuration of the container. First, on the host, you can edit `/var/lib/container/name/etc/nixos/configuration.nix`, and run

```
# nixos-container update foo
```

This will build and activate the new configuration. You can also specify a new configuration on the command line:

```
1 # nixos-container update foo --config '
    services.httpd.enable = true;
3   services.httpd.adminAddr = "foo@example.org";
    networking.firewall.allowedTCPPorts = [ 80 ];
5 '
7 # curl http://$(nixos-container show-ip foo)/
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">...
```

However, note that this will overwrite the container's `/etc/nixos/configuration.nix`.

Alternatively, you can change the configuration from within the container itself by running **nixos-rebuild switch** inside the container. Note that the container by default does not have a copy of the NixOS channel, so you should run **nix-channel --update** first.

Containers can be stopped and started using **nixos-container stop** and **nixos-container start**, respectively, or by using **systemctl** on the container's service unit. To destroy a container, including its file system, do

```
# nixos-container destroy foo
```

47.2. Declarative Container Specification

You can also specify containers and their configuration in the host's `configuration.nix`. For example, the following specifies that there shall be a container named **database** running PostgreSQL:

```
containers.database =
2 { config =
    { config, pkgs, ... }:
4     { services.postgresql.enable = true;
      services.postgresql.package = pkgs.postgresql_9_6;
6     };
};
```

If you run **nixos-rebuild switch**, the container will be built. If the container was already running, it will be updated in place, without rebooting. The container can be configured to start automatically by setting `containers.database.autoStart = true` in its configuration.

By default, declarative containers share the network namespace of the host, meaning that they can listen on (privileged) ports. However, they cannot change the network configuration. You can give a container its own network as follows:

```
containers.database = {
2   privateNetwork = true;
   hostAddress = "192.168.100.10";
4   localAddress = "192.168.100.11";
};
```

This gives the container a private virtual Ethernet interface with IP address 192.168.100.11, which is hooked up to a virtual Ethernet interface on the host with IP address 192.168.100.10. (See the next section for details on container networking.)

To disable the container, just remove it from `configuration.nix` and run `nixos-rebuild switch`. Note that this will not delete the root directory of the container in `/var/lib/containers`. Containers can be destroyed using the imperative method: `nixos-container destroy foo`.

Declarative containers can be started and stopped using the corresponding systemd service, e.g. `systemctl start ↪ container@database`.

47.3. Container Networking

When you create a container using `nixos-container create`, it gets its own private IPv4 address in the range 10.233.0.0/16. You can get the container's IPv4 address as follows:

```
# nixos-container show-ip foo
2 10.233.4.2

4 $ ping -c1 10.233.4.2
64 bytes from 10.233.4.2: icmp_seq=1 ttl=64 time=0.106 ms
```

Networking is implemented using a pair of virtual Ethernet devices. The network interface in the container is called `eth0`, while the matching interface in the host is called `ve-container-name` (e.g., `ve-foo`). The container has its own network namespace and the `CAP_NET_ADMIN` capability, so it can perform arbitrary network configuration such as setting up firewall rules, without affecting or having access to the host's network.

By default, containers cannot talk to the outside network. If you want that, you should set up Network Address Translation (NAT) rules on the host to rewrite container traffic to use your external IP address. This can be accomplished using the following configuration on the host:

```
networking.nat.enable = true;
2 networking.nat.internalInterfaces = ["ve-+"];
networking.nat.externalInterface = "eth0";
```

where `eth0` should be replaced with the desired external interface. Note that `ve-+` is a wildcard that matches all container interfaces.

If you are using Network Manager, you need to explicitly prevent it from managing container interfaces:

```
networking.networkmanager.unmanaged = [ "interface-name:ve-*" ];
```

You may need to restart your system for the changes to take effect.

Chapter 48. Troubleshooting

Table of Contents

- 48.1. Boot Problems
- 48.2. Maintenance Mode
- 48.3. Rolling Back Configuration Changes
- 48.4. Nix Store Corruption
- 48.5. Network Problems

This chapter describes solutions to common problems you might encounter when you manage your NixOS system.

48.1. Boot Problems

If NixOS fails to boot, there are a number of kernel command line parameters that may help you to identify or fix the issue. You can add these parameters in the GRUB boot menu by pressing “e” to modify the selected boot entry and editing the line starting with **linux**. The following are some useful kernel command line parameters that are recognised by the NixOS boot scripts or by systemd:

- boot.shell_on_fail** Allows the user to start a root shell if something goes wrong in stage 1 of the boot process (the initial ramdisk). This is disabled by default because there is no authentication for the root shell.
- boot.debug1** Start an interactive shell in stage 1 before anything useful has been done. That is, no modules have been loaded and no file systems have been mounted, except for **/proc** and **/sys**.
- boot.debug1devices** Like **boot.debug1**, but runs stage1 until kernel modules are loaded and device nodes are created. This may help with e.g. making the keyboard work.
- boot.debug1mounts** Like **boot.debug1** or **boot.debug1devices**, but runs stage1 until all filesystems that are mounted during **initrd** are mounted (see **neededForBoot**). As a motivating example, this could be useful if you’ve forgotten to set **neededForBoot** on a file system.
- boot.trace** Print every shell command executed by the stage 1 and 2 boot scripts.
- single** Boot into rescue mode (a.k.a. single user mode). This will cause systemd to start nothing but the unit **rescue.target**, which runs **sulogin** to prompt for the root password and start a root login shell. Exiting the shell causes the system to continue with the normal boot process.
- systemd.log_level=debug systemd.log_target=console** Make systemd very verbose and send log messages to the console instead of the journal.

For more parameters recognised by systemd, see `systemd(1)`.

Notice that for **boot.shell_on_fail**, **boot.debug1**, **boot.debug1devices**, and **boot.debug1mounts**, if you did *not* select “start the new shell as pid 1”, and you **exit** from the new shell, boot will proceed normally from the point where it failed, as if you’d chosen “ignore the error and continue”.

If no login prompts or X11 login screens appear (e.g. due to hanging dependencies), you can press **Alt+ArrowUp**. If you’re lucky, this will start rescue mode (described above). (Also note that since most units have a 90-second timeout before systemd gives up on them, the **agetty** login prompts should appear eventually unless something is very wrong.)

48.2. Maintenance Mode

You can enter rescue mode by running:

```
# systemctl rescue
```

This will eventually give you a single-user root shell. Systemd will stop (almost) all system services. To get out of maintenance mode, just exit from the rescue shell.

48.3. Rolling Back Configuration Changes

After running **nixos-rebuild** to switch to a new configuration, you may find that the new configuration doesn’t work very well. In that case, there are several ways to return to a previous configuration.

First, the GRUB boot manager allows you to boot into any previous configuration that hasn’t been garbage-collected. These configurations can be found under the GRUB submenu “NixOS - All configurations”. This is especially useful if the new configuration fails to boot. After the system has booted, you can make the selected configuration the default for subsequent boots:

```
1 # /run/current-system/bin/switch-to-configuration boot
```

Second, you can switch to the previous configuration in a running system:

```
1 # nixos-rebuild switch --rollback
```

This is equivalent to running:

```
1 # /nix/var/nix/profiles/system-N-link/bin/switch-to-configuration switch
```

where *N* is the number of the NixOS system configuration. To get a list of the available configurations, do:

```
$ ls -l /nix/var/nix/profiles/system-*-link
2 ...
lrwxrwxrwx 1 root root 78 Aug 12 13:54 /nix/var/nix/profiles/system-268-link ->
  ↪ /nix/store/202b...-nixos-13.07pre4932_5a676e4-4be1055
```

48.4. Nix Store Corruption

After a system crash, it's possible for files in the Nix store to become corrupted. (For instance, the Ext4 file system has the tendency to replace un-synced files with zero bytes.) NixOS tries hard to prevent this from happening: it performs a **sync** before switching to a new configuration, and Nix's database is fully transactional. If corruption still occurs, you may be able to fix it automatically.

If the corruption is in a path in the closure of the NixOS system configuration, you can fix it by doing

```
1 # nixos-rebuild switch --repair
```

This will cause Nix to check every path in the closure, and if its cryptographic hash differs from the hash recorded in Nix's database, the path is rebuilt or redownloaded.

You can also scan the entire Nix store for corrupt paths:

```
1 # nix-store --verify --check-contents --repair
```

Any corrupt paths will be redownloaded if they're available in a binary cache; otherwise, they cannot be repaired.

48.5. Network Problems

Nix uses a so-called *binary cache* to optimise building a package from source into downloading it as a pre-built binary. That is, whenever a command like **nixos-rebuild** needs a path in the Nix store, Nix will try to download that path from the Internet rather than build it from source. The default binary cache is <https://cache.nixos.org/>. If this cache is unreachable, Nix operations may take a long time due to HTTP connection timeouts. You can disable the use of the binary cache by adding `--option use-binary-caches false`, e.g.

```
# nixos-rebuild switch --option use-binary-caches false
```

If you have an alternative binary cache at your disposal, you can use it instead:

```
1 # nixos-rebuild switch --option binary-caches http://my-cache.example.org/
```

Part IV. Development

This chapter describes how you can modify and extend NixOS.

Table of Contents

- 49. Getting the Sources
- 50. Writing NixOS Modules
- 51. Building Specific Parts of NixOS
- 52. Writing NixOS Documentation
- 53. Building Your Own NixOS CD
- 54. NixOS Tests
- 55. Testing the Installer
- 56. Releases

Chapter 49. Getting the Sources

By default, NixOS's **nixos-rebuild** command uses the NixOS and Nixpkgs sources provided by the **nixos** channel (kept in `/nix/var/nix/profiles/per-user/root/channels/nixos`). To modify NixOS, however, you should check out the latest sources from Git. This is as follows:

```
$ git clone https://github.com/NixOS/nixpkgs
2 $ cd nixpkgs
$ git remote update origin
```

This will check out the latest Nixpkgs sources to `./nixpkgs` the NixOS sources to `./nixpkgs/nixos`. (The NixOS source tree lives in a subdirectory of the Nixpkgs repository.) The **nixpkgs** repository has branches that correspond to each Nixpkgs/NixOS channel (see Chapter 4, *Upgrading NixOS* for more information about channels). Thus, the Git branch `origin/nixos-17.03` will contain the latest built and tested version available in the **nixos-17.03** channel.

It's often inconvenient to develop directly on the master branch, since if somebody has just committed (say) a change to GCC, then the binary cache may not have caught up yet and you'll have to rebuild everything from source. So you may want to create a local branch based on your current NixOS version:

```
$ nixos-version
2 17.09pre104379.6e0b727 (Hummingbird)
4 $ git checkout -b local 6e0b727
```

Or, to base your local branch on the latest version available in a NixOS channel:

```
$ git remote update origin
2 $ git checkout -b local origin/nixos-17.03
```

(Replace **nixos-17.03** with the name of the channel you want to use.) You can use **git merge** or **git rebase** to keep your local branch in sync with the channel, e.g.

```
$ git remote update origin
2 $ git merge origin/nixos-17.03
```

You can use **git cherry-pick** to copy commits from your local branch to the upstream branch.

If you want to rebuild your system using your (modified) sources, you need to tell **nixos-rebuild** about them using the **-I** flag:

```
# nixos-rebuild switch -I nixpkgs=/my/sources/nixpkgs
```

If you want **nix-env** to use the expressions in `/my/sources`, use **nix-env -f /my/sources/nixpkgs**, or change the default by adding a symlink in `~/.nix-defexpr`:

```
$ ln -s /my/sources/nixpkgs ~/.nix-defexpr/nixpkgs
```

You may want to delete the symlink `~/.nix-defexpr/channels_root` to prevent root's NixOS channel from clashing with your own tree (this may break the `command-not-found` utility though). If you want to go back to the default state, you may just remove the `~/.nix-defexpr` directory completely, log out and log in again and it should have been recreated with a link to the root channels.

Chapter 50. Writing NixOS Modules

Table of Contents

- 50.1. Option Declarations
- 50.2. Options Types
- 50.3. Option Definitions

50.4. Warnings and Assertions

50.5. Meta Attributes

50.6. Importing Modules

50.7. Replace Modules

50.8. Freeform modules

50.9. Options for Program Settings

NixOS has a modular system for declarative configuration. This system combines multiple *modules* to produce the full system configuration. One of the modules that constitute the configuration is `/etc/nixos/configuration.nix`. Most of the others live in the `nixos/modules` subdirectory of the Nixpkgs tree.

Each NixOS module is a file that handles one logical aspect of the configuration, such as a specific kind of hardware, a service, or network settings. A module configuration does not have to handle everything from scratch; it can use the functionality provided by other modules for its implementation. Thus a module can *declare* options that can be used by other modules, and conversely can *define* options provided by other modules in its own implementation. For example, the module `pam.nix` declares the option `security.pam.services` that allows other modules (e.g. `sshd.nix`) to define PAM services; and it defines the option `environment.etc` (declared by `etc.nix`) to cause files to be created in `/etc/pam.d`.

In Chapter 5, *Configuration Syntax*, we saw the following structure of NixOS modules:

```
1 { config, pkgs, ... }:  
2  
3 { option definitions  
4 }
```

This is actually an *abbreviated* form of module that only defines options, but does not declare any. The structure of full NixOS modules is shown in Example 50.1, “Structure of NixOS Modules”.

Example 50.1. Structure of NixOS Modules

```
1 { config, pkgs, ... }:  
2  
3 {  
4   imports =  
5     [ paths of other modules  
6   ];  
7  
8   options = {  
9     option declarations  
10  };  
11  
12  config = {  
13    option definitions  
14  };  
15 }
```

The meaning of each part is as follows.

-
- | | |
|---|---|
| 1 | This line makes the current Nix expression a function. The variable <code>pkgs</code> contains Nixpkgs, while <code>config</code> contains the full |
| 2 | This list enumerates the paths to other NixOS modules that should be included in the evaluation of the system configuration |
| 3 | The attribute <code>options</code> is a nested set of <i>option declarations</i> (described below). |
| 4 | The attribute <code>config</code> is a nested set of <i>option definitions</i> (also described below). |
-

Example 50.2, “NixOS Module for the “locate” Service” shows a module that handles the regular update of the “locate” database, an index of all files in the file system. This module declares two options that can be defined by other modules (typically the user’s `configuration.nix`): `services.locate.enable` (whether the database should be updated) and `services.locate.interval` (when the update should be done). It implements its functionality by defining two options declared by other modules: `systemd.services` (the set of all systemd services) and `systemd.timers` (the list of commands to be executed periodically by `systemd`).

Example 50.2. NixOS Module for the “locate” Service

```

1 { config, lib, pkgs, ... }:
2
3 with lib;
4
5 let
6   cfg = config.services.locate;
7 in {
8   options.services.locate = {
9     enable = mkOption {
10       type = types.bool;
11       default = false;
12       description = ''
13         If enabled, NixOS will periodically update the database of
14         files used by the locate command.
15       '';
16     };
17
18     interval = mkOption {
19       type = types.str;
20       default = "02:15";
21       example = "hourly";
22       description = ''
23         Update the locate database at this interval. Updates by
24         default at 2:15 AM every day.
25
26         The format is described in
27         systemd.time(7).
28       '';
29     };
30
31     # Other options omitted for documentation
32   };
33
34   config = {
35     systemd.services.update-locatedb =
36       { description = "Update Locate Database";
37         path = [ pkgs.su ];
38         script =
39           ''
40             mkdir -m 0755 -p $(dirname ${toString cfg.output})
41             exec updatedb \
42               --localuser=${cfg.localuser} \
43               ${optionalString (!cfg.includeStore) "--prunepaths='/nix/store'"} \
44               --output=${toString cfg.output} ${concatStringsSep " "
45                 ↪ cfg.extraFlags}
46           '';
47     };
48
49     systemd.timers.update-locatedb = mkIf cfg.enable

```

```

50     { description = "Update timer for locate database";
      partOf      = [ "update-locatedb.service" ];
      wantedBy    = [ "timers.target" ];
52     timerConfig.OnCalendar = cfg.interval;
      };
54 };
}

```

50.1. Option Declarations

An option declaration specifies the name, type and description of a NixOS configuration option. It is invalid to define an option that hasn't been declared in any module. An option declaration generally looks like this:

```

1 options = {
  name = mkOption {
3    type = type specification;
    default = default value;
5    example = example value;
    description = "Description for use in the NixOS manual.";
7  };
};

```

The attribute names within the *name* attribute path must be camel cased in general but should, as an exception, match the package attribute name when referencing a Nixpkgs package. For example, the option `services.nix-serve.bindAddress` references the `nix-serve` Nixpkgs package.

The function `mkOption` accepts the following arguments.

type The type of the option (see Section 50.2, “Options Types”). It may be omitted, but that's not advisable since it may lead to errors that are hard to diagnose.

default The default value used if no value is defined by any module. A default is not required; but if a default is not given, then users of the module will have to define the value of the option, otherwise an error will be thrown.

example An example value that will be shown in the NixOS manual.

description A textual description of the option, in DocBook format, that will be included in the NixOS manual.

50.1.1. Extensible Option Types

Extensible option types is a feature that allow to extend certain types declaration through multiple module files. This feature only work with a restricted set of types, namely `enum` and `submodules` and any composed forms of them.

Extensible option types can be used for `enum` options that affects multiple modules, or as an alternative to related `enable` options.

As an example, we will take the case of display managers. There is a central display manager module for generic display manager options and a module file per display manager backend (`sddm`, `gdm` ...).

There are two approach to this module structure:

- Managing the display managers independently by adding an enable option to every display manager module backend. (NixOS)
- Managing the display managers in the central module by adding an option to select which display manager backend to use.

Both approaches have problems.

Making backends independent can quickly become hard to manage. For display managers, there can be only one enabled at a time, but the type system can not enforce this restriction as there is no relation between each backend `enable` option. As a result, this restriction has to be done explicitly by adding assertions in each display manager backend module.

On the other hand, managing the display managers backends in the central module will require to change the central module option every time a new backend is added or removed.

By using extensible option types, it is possible to create a placeholder option in the central module (Example 50.3, “Extensible type placeholder in the service module”), and to extend it in each backend module (Example 50.4, “Extending `services.xserver.displayManager.enable` in the `gdm` module”, Example 50.5, “Extending `services.xserver.displayManager.enable` in the `sddm` module”).

As a result, `displayManager.enable` option values can be added without changing the main service module file and the type system automatically enforce that there can only be a single display manager enabled.

Example 50.3. Extensible type placeholder in the service module

```
services.xserver.displayManager.enable = mkOption {  
2   description = "Display manager to use";  
    type = with types; nullOr (enum [ ]);  
4 };
```

Example 50.4. Extending `services.xserver.displayManager.enable` in the `gdm` module

```
services.xserver.displayManager.enable = mkOption {  
2   type = with types; nullOr (enum [ "gdm" ]);  
};
```

Example 50.5. Extending `services.xserver.displayManager.enable` in the `sddm` module

```
services.xserver.displayManager.enable = mkOption {  
2   type = with types; nullOr (enum [ "sddm" ]);  
};
```

The placeholder declaration is a standard `mkOption` declaration, but it is important that extensible option declarations only use the `type` argument.

Extensible option types work with any of the composed variants of `enum` such as `with types; nullOr (enum [↪ "foo" "bar"])` or `with types; listOf (enum ["foo" "bar"])`.

50.2. Options Types

Option types are a way to put constraints on the values a module option can take. Types are also responsible of how values are merged in case of multiple value definitions.

50.2.1. Basic Types

Basic types are the simplest available types in the module system. Basic types include multiple string types that mainly differ in how definition merging is handled.

types.attrs A free-form attribute set.

types.bool A boolean, its values can be `true` or `false`.

types.path A filesystem path, defined as anything that when coerced to a string starts with a slash. Even if derivations can be considered as path, the more specific **types.package** should be preferred.

types.package A derivation or a store path.

Integer-related types:

types.int A signed integer.

types.ints.{s8, s16, s32} Signed integers with a fixed length (8, 16 or 32 bits). They go from $-2^n/2$ to $2^n/2-1$ respectively (e.g. -128 to 127 for 8 bits).

types.ints.unsigned An unsigned integer (that is ≥ 0).

types.ints.{u8, u16, u32} Unsigned integers with a fixed length (8, 16 or 32 bits). They go from 0 to 2^n-1 respectively (e.g. 0 to 255 for 8 bits).

types.ints.positive A positive integer (that is > 0).

types.port A port number. This type is an alias to **types.ints.u16**.

String-related types:

types.str A string. Multiple definitions cannot be merged.

types.lines A string. Multiple definitions are concatenated with a new line `"\n"`.

types.commas A string. Multiple definitions are concatenated with a comma `","`.

types.envVar A string. Multiple definitions are concatenated with a collon `":"`.

types.strMatching A string matching a specific regular expression. Multiple definitions cannot be merged. The regular expression is processed using **builtins.match**.

50.2.2. Value Types

Value types are types that take a value parameter.

types.enum l One element of the list *l*, e.g. **types.enum** [`"left"` `"right"`]. Multiple definitions cannot be merged.

types.separatedString sep A string with a custom separator *sep*, e.g. **types.separatedString** `"|"`.

types.ints.between lowest highest An integer between *lowest* and *highest* (both inclusive). Useful for creating types like **types.port**.

types.submodule o A set of sub options *o*. *o* can be an attribute set, a function returning an attribute set, or a path to a file containing such a value. Submodules are used in composed types to create modular options. This is equivalent to **types.submoduleWith** { *modules* = `toList o`; *shorthandOnlyDefinesConfig* = `true`; \hookrightarrow }. Submodules are detailed in Section 50.2.4, “Submodule”.

types.submoduleWith { modules, specialArgs ? {}, shorthandOnlyDefinesConfig ? false } Like **types.submodule**, but more flexible and with better defaults. It has parameters

- *modules* A list of modules to use by default for this submodule type. This gets combined with all option definitions to build the final list of modules that will be included.

Note

Only options defined with this argument are included in rendered documentation.

- *specialArgs* An attribute set of extra arguments to be passed to the module functions. The option `_module.args` should be used instead for most arguments since it allows overriding. *specialArgs* should only be used for arguments that can't go through the module fixed-point, because of infinite recursion or other problems. An example is overriding the `lib` argument, because `lib` itself is used to define `_module.args`, which makes using `_module.args` to define it impossible.

- *shorthandOnlyDefinesConfig* Whether definitions of this type should default to the `config` section of a module (see Example 50.1, “Structure of NixOS Modules”) if it is an attribute set. Enabling this only has a benefit when the submodule defines an option named `config` or `options`. In such a case it would allow the option to be set with `the-submodule.config = "value"` instead of requiring `the-submodule.config.config = "value"`. This is because only when modules *don't* set the `config` or `options` keys, all keys are interpreted as option definitions in the `config` section. Enabling this option implicitly puts all attributes in the `config` section.

With this option enabled, defining a non-`config` section requires using a function: `the-submodule = ↪ { ... } : { options = { ... }; }.`

50.2.3. Composed Types

Composed types are types that take a type as parameter. `listOf int` and `either int str` are examples of composed types.

types.listOf *t* A list of *t* type, e.g. `types.listOf int`. Multiple definitions are merged with list concatenation.

types.attrsOf *t* An attribute set of where all the values are of *t* type. Multiple definitions result in the joined attribute set.

Note

This type is *strict* in its values, which in turn means attributes cannot depend on other attributes. See `types.lazyAttrsOf` for a lazy version.

types.lazyAttrsOf *t* An attribute set of where all the values are of *t* type. Multiple definitions result in the joined attribute set. This is the lazy version of `types.attrsOf`, allowing attributes to depend on each other.

Warning

This version does not fully support conditional definitions! With an option `foo` of this type and a definition `foo.attr = lib.mkIf false 10`, evaluating `foo ? attr` will return `true` even though it should be false. Accessing the value will then throw an error. For types *t* that have an `emptyValue` defined, that value will be returned instead of throwing an error. So if the type of `foo.attr` was `lazyAttrsOf (nullOr int)`, `null` would be returned instead for the same `mkIf false` definition.

types.nullOr *t* `null` or type *t*. Multiple definitions are merged according to type *t*.

types.uniq *t* Ensures that type *t* cannot be merged. It is used to ensure option definitions are declared only once.

types.either *t1 t2* Type *t1* or type *t2*, e.g. with `types; either int str`. Multiple definitions cannot be merged.

types.oneOf [*t1 t2 ...*] Type *t1* or type *t2* and so forth, e.g. with `types; oneOf [int str bool]`. Multiple definitions cannot be merged.

types.coercedTo from *f* to *to* Type *to* or type *from* which will be coerced to type *to* using function *f* which takes an argument of type *from* and return a value of type *to*. Can be used to preserve backwards compatibility of an option if its type was changed.

50.2.4. Submodule

`submodule` is a very powerful type that defines a set of sub-options that are handled like a separate module.

It takes a parameter *o*, that should be a set, or a function returning a set with an `options` key defining the sub-options. Submodule option definitions are type-checked accordingly to the `options` declarations. Of course, you can nest submodule option definitions for even higher modularity.

The option set can be defined directly (Example 50.6, “Directly defined submodule”) or as reference (Example 50.7, “Submodule defined as a reference”).

Example 50.6. Directly defined submodule

```

options.mod = mkOption {
2  description = "submodule example";
  type = with types; submodule {
4    options = {
      foo = mkOption {
6        type = int;
      };
      bar = mkOption {
8        type = str;
10       };
    };
12  };
};

```

Example 50.7. Submodule defined as a reference

```

1 let
  modOptions = {
3    options = {
      foo = mkOption {
5        type = int;
      };
      bar = mkOption {
7        type = int;
9      };
    };
11  };
in
13 options.mod = mkOption {
  description = "submodule example";
15  type = with types; submodule modOptions;
};

```

The submodule type is especially interesting when used with composed types like `attrsOf` or `listOf`. When composed with `listOf` (Example 50.8, “Declaration of a list of submodules”), `submodule` allows multiple definitions of the submodule option set (Example 50.9, “Definition of a list of submodules”).

Example 50.8. Declaration of a list of submodules

```

options.mod = mkOption {
2  description = "submodule example";
  type = with types; listOf (submodule {
4    options = {
      foo = mkOption {
6        type = int;
      };
      bar = mkOption {
8        type = str;
10       };
    };
12  });
};

```

Example 50.9. Definition of a list of submodules

```
1 config.mod = [  
  { foo = 1; bar = "one"; }  
3  { foo = 2; bar = "two"; }  
  ];
```

When composed with `attrsOf` (Example 50.10, “Declaration of attribute sets of submodules”), `submodule` allows multiple named definitions of the submodule option set (Example 50.11, “Declaration of attribute sets of submodules”).

Example 50.10. Declaration of attribute sets of submodules

```
options.mod = mkOption {  
2  description = "submodule example";  
  type = with types; attrsOf (submodule {  
4    options = {  
      foo = mkOption {  
6        type = int;  
      };  
8      bar = mkOption {  
        type = str;  
10     };  
    };  
12  });  
};
```

Example 50.11. Declaration of attribute sets of submodules

```
1 config.mod.one = { foo = 1; bar = "one"; };  
  config.mod.two = { foo = 2; bar = "two"; };
```

50.2.5. Extending types

Types are mainly characterized by their `check` and `merge` functions.

check The function to type check the value. Takes a value as parameter and return a boolean. It is possible to extend a type check with the `addCheck` function (Example 50.12, “Adding a type check”), or to fully override the check function (Example 50.13, “Overriding a type check”).

Example 50.12. Adding a type check

```
byte = mkOption {  
2  description = "An integer between 0 and 255.";  
  type = types.addCheck types.int (x: x >= 0 && x <= 255);  
4  };
```

Example 50.13. Overriding a type check

```

nixThings = mkOption {
2   description = "words that start with 'nix'";
    type = types.str // {
4     check = (x: lib.hasPrefix "nix" x)
    };
6 };

```

merge Function to merge the options values when multiple values are set. The function takes two parameters, **loc** the option path as a list of strings, and **defs** the list of defined values as a list. It is possible to override a type merge function for custom needs.

50.2.6. Custom Types

Custom types can be created with the `mkOptionType` function. As type creation includes some more complex topics such as submodule handling, it is recommended to get familiar with `types.nix` code before creating a new type.

The only required parameter is **name**.

name A string representation of the type function name.

definition Description of the type used in documentation. Give information of the type and any of its arguments.

check A function to type check the definition value. Takes the definition value as a parameter and returns a boolean indicating the type check result, **true** for success and **false** for failure.

merge A function to merge multiple definitions values. Takes two parameters:

loc The option path as a list of strings, e.g. `["boot" "loader" "grub" "enable"]`.

defs The list of sets of defined value and file where the value was defined, e.g. `[{ file =
 ↪ "/foo.nix"; value = 1; } { file = "/bar.nix"; value = 2 }]`. The **merge** function should return the merged value or throw an error in case the values are impossible or not meant to be merged.

getSubOptions For composed types that can take a submodule as type parameter, this function generate sub-options documentation. It takes the current option prefix as a list and return the set of sub-options. Usually defined in a recursive manner by adding a term to the prefix, e.g. `prefix: elemType.getSubOptions
 ↪ (prefix ++ ["prefix"])` where *"prefix"* is the newly added prefix.

getSubModules For composed types that can take a submodule as type parameter, this function should return the type parameters submodules. If the type parameter is called `elemType`, the function should just recursively look into submodules by returning `elemType.getSubModules`.

substSubModules For composed types that can take a submodule as type parameter, this function can be used to substitute the parameter of a submodule type. It takes a module as parameter and return the type with the submodule options substituted. It is usually defined as a type function call with a recursive call to `substSubModules`, e.g for a type `composedType` that take an `elemType` type parameter, this function should be defined as `m: composedType (elemType.substSubModules m)`.

typeMerge A function to merge multiple type declarations. Takes the type to merge **functor** as parameter. A null return value means that type cannot be merged.

f The type to merge **functor**.

Note: There is a generic `defaultTypeMerge` that work with most of value and composed types.

functor An attribute set representing the type. It is used for type operations and has the following keys:

type The type function.

wrapped Holds the type parameter for composed types.

payload Holds the value parameter for value types. The types that have a **payload** are the `enum`, `separatedString` and `submodule` types.

binOp A binary operation that can merge the payloads of two same types. Defined as a function that take two payloads as parameters and return the payloads merged.

50.3. Option Definitions

Option definitions are generally straight-forward bindings of values to option names, like

```
config = {  
2   services.httpd.enable = true;  
};
```

However, sometimes you need to wrap an option definition or set of option definitions in a *property* to achieve certain effects:

Delaying Conditionals

If a set of option definitions is conditional on the value of another option, you may need to use **mkIf**. Consider, for instance:

```
config = if config.services.httpd.enable then {  
2   environment.systemPackages = [ ... ];  
   ...  
4 } else {};
```

This definition will cause Nix to fail with an “infinite recursion” error. Why? Because the value of `config.services.httpd.enable` depends on the value being constructed here. After all, you could also write the clearly circular and contradictory:

```
config = if config.services.httpd.enable then {  
2   services.httpd.enable = false;  
} else {  
4   services.httpd.enable = true;  
};
```

The solution is to write:

```
1 config = mkIf config.services.httpd.enable {  
   environment.systemPackages = [ ... ];  
3   ...  
};
```

The special function **mkIf** causes the evaluation of the conditional to be “pushed down” into the individual definitions, as if you had written:

```
config = {  
2   environment.systemPackages = if config.services.httpd.enable then [ ... ] else  
   ↪ [ ];  
   ...  
4 };
```

Setting Priorities

A module can override the definitions of an option in other modules by setting a *priority*. All option definitions that do not have the lowest priority value are discarded. By default, option definitions have priority 1000. You can specify an explicit priority by using **mkOverride**, e.g.

```
services.openssh.enable = mkOverride 10 false;
```

This definition causes all other definitions with priorities above 10 to be discarded. The function **mkForce** is equal to **mkOverride 50**.

Merging Configurations

In conjunction with `mkIf`, it is sometimes useful for a module to return multiple sets of option definitions, to be merged together as if they were declared in separate modules. This can be done using `mkMerge`:

```
config = mkMerge
2  [ # Unconditional stuff.
    { environment.systemPackages = [ ... ];
4    }
    # Conditional stuff.
6    (mkIf config.services.bla.enable {
        environment.systemPackages = [ ... ];
8    })
];
```

50.4. Warnings and Assertions

When configuration problems are detectable in a module, it is a good idea to write an assertion or warning. Doing so provides clear feedback to the user and prevents errors after the build.

Although Nix has the `abort` and `builtins.trace` functions to perform such tasks, they are not ideally suited for NixOS modules. Instead of these functions, you can declare your warnings and assertions using the NixOS module system.

50.4.1. Warnings

This is an example of using `warnings`.

```
{ config, lib, ... }:
2 {
    config = lib.mkIf config.services.foo.enable {
4        warnings =
            if config.services.foo.bar
6            then [ ''You have enabled the bar feature of the foo service.
                    This is known to cause some specific problems in certain situations.
8                    '' ]
            else [];
10 }
}
```

50.4.2. Assertions

This example, extracted from the `syslogd` module shows how to use `assertions`. Since there can only be one active syslog daemon at a time, an assertion is useful to prevent such a broken system from being built.

```
{ config, lib, ... }:
2 {
    config = lib.mkIf config.services.syslogd.enable {
4        assertions =
            [ { assertion = !config.services.rsyslogd.enable;
6              message = "rsyslogd conflicts with syslogd";
              }
8        ];
    }
10 }
```

50.5. Meta Attributes

Like Nix packages, NixOS modules can declare meta-attributes to provide extra information. Module meta attributes are defined in the `meta.nix` special module.

`meta` is a top level attribute like `options` and `config`. Available meta-attributes are `maintainers` and `doc`.

Each of the meta-attributes must be defined at most once per module file.

```
{ config, lib, pkgs, ... }:  
2 {  
    options = {  
4     ...  
    };  
6  
    config = {  
8     ...  
    };  
10  
    meta = {  
12     maintainers = with lib.maintainers; [ ericsagnes ];  
        doc = ./default.xml;  
14    };  
}
```

1	<code>maintainers</code> contains a list of the module maintainers.
2	<code>doc</code> points to a valid DocBook file containing the module documentation. Its contents is automatically added to Part II, “Configuration”. Changes to a module documentation have to be checked to not break building the NixOS manual:

```
$ nix-build nixos/release.nix -A manual
```

50.6. Importing Modules

Sometimes NixOS modules need to be used in configuration but exist outside of Nixpkgs. These modules can be imported:

```
1 { config, lib, pkgs, ... }:  
  
3 {  
    imports =  
5     [ # Use a locally-available module definition in  
        # ./example-module/default.nix  
7       ./example-module  
    ];  
9  
    services.exampleModule.enable = true;  
11 }
```

The environment variable `NIXOS_EXTRA_MODULE_PATH` is an absolute path to a NixOS module that is included alongside the Nixpkgs NixOS modules. Like any NixOS module, this module can import additional modules:

```
# ./module-list/default.nix  
2 [  
    ./example-module1  
4    ./example-module2  
]
```

```

1 # ./extra-module/default.nix
  { imports = import ./module-list.nix; }

# NIXOS_EXTRA_MODULE_PATH=/absolute/path/to/extra-module
2 { config, lib, pkgs, ... }:

4 {
  # No `imports` needed
6
  services.exampleModule1.enable = true;
8 }

```

50.7. Replace Modules

Modules that are imported can also be disabled. The option declarations, config implementation and the imports of a disabled module will be ignored, allowing another to take its place. This can be used to import a set of modules from another channel while keeping the rest of the system on a stable release.

`disabledModules` is a top level attribute like `imports`, `options` and `config`. It contains a list of modules that will be disabled. This can either be the full path to the module or a string with the filename relative to the modules path (eg. `<nixpkgs/nixos/modules>` for `nixos`).

This example will replace the existing `postgresql` module with the version defined in the `nixos-unstable` channel while keeping the rest of the modules and packages from the original `nixos` channel. This only overrides the module definition, this won't use `postgresql` from `nixos-unstable` unless explicitly configured to do so.

```

{ config, lib, pkgs, ... }:
2
{
4   disabledModules = [ "services/databases/postgresql.nix" ];

6   imports =
      [ # Use postgresql service from nixos-unstable channel.
8       # sudo nix-channel --add https://nixos.org/channels/nixos-unstable
        ↪ nixos-unstable
        <nixos-unstable/nixos/modules/services/databases/postgresql.nix>
10    ];

12   services.postgresql.enable = true;
}

```

This example shows how to define a custom module as a replacement for an existing module. Importing this module will disable the original module without having to know its implementation details.

```

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

3 with lib;

5 let
  cfg = config.programs.man;
7 in

9 {
  disabledModules = [ "services/programs/man.nix" ];
11
  options = {
13    programs.man.enable = mkOption {
      type = types.bool;

```

```

15     default = true;
      description = "Whether to enable manual pages.";
17   };
18 };
19
20 config = mkIf cfg.enabled {
21   warnings = [ "disabled manpages for production deployments." ];
22 };
23 }

```

50.8. Freeform modules

Freeform modules allow you to define values for option paths that have not been declared explicitly. This can be used to add attribute-specific types to what would otherwise have to be `attrsOf` options in order to accept all attribute names.

This feature can be enabled by using the attribute `freeformType` to define a freeform type. By doing this, all assignments without an associated option will be merged using the freeform type and combined into the resulting `config` set. Since this feature nullifies name checking for entire option trees, it is only recommended for use in submodules.

Example 50.14. Freeform submodule

The following shows a submodule assigning a freeform type that allows arbitrary attributes with `str` values below `settings`, but also declares an option for the `settings.port` attribute to have it type-checked and assign a default value. See Example 50.16, “Declaring a type-checked `settings` attribute” for a more complete example.

```

{ lib, config, ... }: {
2
  options.settings = lib.mkOption {
4    type = lib.types.submodule {
6
      freeformType = with lib.types; attrsOf str;
8
      # We want this attribute to be checked for the correct type
      options.port = lib.mkOption {
10       type = lib.types.port;
        # Declaring the option also allows defining a default value
12       default = 8080;
      };
14
    };
16  };
}

```

And the following shows what such a module then allows

```

{
2  # Not a declared option, but the freeform type allows this
  settings.logLevel = "debug";
4
  # Not allowed because the the freeform type only allows strings
6  # settings.enable = true;
8
  # Allowed because there is a port option declared
  settings.port = 80;
10
  # Not allowed because the port option doesn't allow strings
12  # settings.port = "443";
}

```

```
}
```

Note

Freeform attributes cannot depend on other attributes of the same set without infinite recursion:

```
{
2  # This throws infinite recursion encountered
  settings.logLevel = lib.mkIf (config.settings.port == 80) "debug";
4 }
```

To prevent this, declare options for all attributes that need to depend on others. For above example this means to declare `logLevel` to be an option.

50.9. Options for Program Settings

Many programs have configuration files where program-specific settings can be declared. File formats can be separated into two categories:

- Nix-representable ones: These can trivially be mapped to a subset of Nix syntax. E.g. JSON is an example, since its values like `{"foo":{"bar":10}}` can be mapped directly to Nix: `{ foo = { bar = 10; }; }`. Other examples are INI, YAML and TOML. The following section explains the convention for these settings.
- Non-nix-representable ones: These can't be trivially mapped to a subset of Nix syntax. Most generic programming languages are in this group, e.g. bash, since the statement `if true; then echo hi; fi` doesn't have a trivial representation in Nix.

Currently there are no fixed conventions for these, but it is common to have a `configFile` option for setting the configuration file path directly. The default value of `configFile` can be an auto-generated file, with convenient options for controlling the contents. For example an option of type `attrsOf str` can be used for representing environment variables which generates a section like `export FOO="foo"`. Often it can also be useful to also include an `extraConfig` option of type `lines` to allow arbitrary text after the autogenerated part of the file.

50.9.1. Nix-representable Formats (JSON, YAML, TOML, INI, ...)

By convention, formats like this are handled with a generic `settings` option, representing the full program configuration as a Nix value. The type of this option should represent the format. The most common formats have a predefined type and string generator already declared under `pkgs.formats`:

`pkgs.formats.json { }` A function taking an empty attribute set (for future extensibility) and returning a set with JSON-specific attributes `type` and `generate` as specified below.

`pkgs.formats.yaml { }` A function taking an empty attribute set (for future extensibility) and returning a set with YAML-specific attributes `type` and `generate` as specified below.

`pkgs.formats.ini { listsAsDuplicateKeys ? false, ... }` A function taking an attribute set with values

`listsAsDuplicateKeys` A boolean for controlling whether list values can be used to represent duplicate INI keys

It returns a set with INI-specific attributes `type` and `generate` as specified below.

`pkgs.formats.toml { }` A function taking an empty attribute set (for future extensibility) and returning a set with TOML-specific attributes `type` and `generate` as specified below.

These functions all return an attribute set with these values:

`type` A module system type representing a value of the format

generate filename jsonValue A function that can render a value of the format to a file. Returns a file path.

Note

This function puts the value contents in the Nix store. So this should be avoided for secrets.

Example 50.15. Module with conventional settings option

The following shows a module for an example program that uses a JSON configuration file. It demonstrates how above values can be used, along with some other related best practices. See the comments for explanations.

```
{ options, config, lib, pkgs, ... }:
2 let
    cfg = config.services.foo;
4   # Define the settings format used for this program
    settingsFormat = pkgs.formats.json {};
6 in {

8   options.services.foo = {
        enable = lib.mkEnableOption "foo service";
10
        settings = lib.mkOption {
12          # Setting this type allows for correct merging behavior
          type = settingsFormat.type;
14          default = {};
          description = ''
16            Configuration for foo, see
            <link xlink:href="https://example.com/docs/foo"/>
18            for supported settings.
            '';
20        };
    };
22
    config = lib.mkIf cfg.enable {
24      # We can assign some default settings here to make the service work by just
      # enabling it. We use `mkDefault` for values that can be changed without
26      # problems
      services.foo.settings = {
28        # Fails at runtime without any value set
        log_level = lib.mkDefault "WARN";
30
        # We assume systemd's `StateDirectory` is used, so we require this value,
32        # therefore no mkDefault
        data_path = "/var/lib/foo";
34
        # Since we use this to create a user we need to know the default value at
36        # eval time
        user = lib.mkDefault "foo";
38      };

40      environment.etc."foo.json".source =
        # The formats generator function takes a filename and the Nix value
42        # representing the format value and produces a filepath with that value
        # rendered in the format
        settingsFormat.generate "foo-config.json" cfg.settings;

46      # We know that the `user` attribute exists because we set a default value
      # for it above, allowing us to use it without worries here
      users.users.${cfg.settings.user} = {};
```

```

50     # ...
    };
52 }

```

50.9.1.1. Option declarations for attributes Some `settings` attributes may deserve some extra care. They may need a different type, default or merging behavior, or they are essential options that should show their documentation in the manual. This can be done using Section 50.8, “Freeform modules”.

Example 50.16. Declaring a type-checked `settings` attribute

We extend above example using freeform modules to declare an option for the port, which will enforce it to be a valid integer and make it show up in the manual.

```

settings = lib.mkOption {
2   type = lib.types.submodule {

4       freeformType = settingsFormat.type;

6       # Declare an option for the port such that the type is checked and this option
        # is shown in the manual.
8       options.port = lib.mkOption {
            type = lib.types.port;
10            default = 8080;
            description = ''
12                Which port this service should listen on.
                '';
14        };

16    };
    default = {};
18    description = ''
        Configuration for Foo, see
20        <link xlink:href="https://example.com/docs/foo"/>
        for supported values.
22    '';
};

```

Chapter 51. Building Specific Parts of NixOS

With the command `nix-build`, you can build specific parts of your NixOS configuration. This is done as follows:

```

1 $ cd /path/to/nixpkgs/nixos
$ nix-build -A config.option

```

where *option* is a NixOS option with type “derivation” (i.e. something that can be built). Attributes of interest include:

system.build.toplevel The top-level option that builds the entire NixOS system. Everything else in your configuration is indirectly pulled in by this option. This is what `nixos-rebuild` builds and what `/run/current-system` points to afterwards.

A shortcut to build this is:

```
$ nix-build -A system
```

system.build.manual.manualHTML The NixOS manual.

system.build.etc A tree of symlinks that form the static parts of `/etc`.

system.build.initialRamdisk , **system.build.kernel** The initial ramdisk and kernel of the system. This allows a quick way to test whether the kernel and the initial ramdisk boot correctly, by using QEMU's `-kernel` and `-initrd` options:

```
$ nix-build -A config.system.build.initialRamdisk -o initrd
2 $ nix-build -A config.system.build.kernel -o kernel
$ qemu-system-x86_64 -kernel ./kernel/bzImage -initrd ./initrd/initrd -hda
  ↪ /dev/null
```

system.build.nixos-rebuild , **system.build.nixos-install** , **system.build.nixos-generate-config**
These build the corresponding NixOS commands.

systemd.units.unit-name.unit This builds the unit with the specified name. Note that since unit names contain dots (e.g. `httpd.service`), you need to put them between quotes, like this:

```
$ nix-build -A 'config.systemd.units."httpd.service".unit'
```

You can also test individual units, without rebuilding the whole system, by putting them in `/run/systemd/system`:

```
$ cp $(nix-build -A 'config.systemd.units."httpd.service".unit')/httpd.service \
2   /run/systemd/system/tmp-httpd.service
# systemctl daemon-reload
4 # systemctl start tmp-httpd.service
```

Note that the unit must not have the same name as any unit in `/etc/systemd/system` since those take precedence over `/run/systemd/system`. That's why the unit is installed as `tmp-httpd.service` here.

Chapter 52. Writing NixOS Documentation

Table of Contents

52.1. Building the Manual

52.2. Editing DocBook XML

52.3. Creating a Topic

52.4. Adding a Topic to the Book

As NixOS grows, so too does the need for a catalogue and explanation of its extensive functionality. Collecting pertinent information from disparate sources and presenting it in an accessible style would be a worthy contribution to the project.

52.1. Building the Manual

The DocBook sources of the NixOS Manual are in the `nixos/doc/manual` subdirectory of the Nixpkgs repository.

You can quickly validate your edits with **make**:

```
$ cd /path/to/nixpkgs/nixos/doc/manual
2 $ make
```

Once you are done making modifications to the manual, it's important to build it before committing. You can do that as follows:

```
nix-build nixos/release.nix -A manual.x86_64-linux
```

When this command successfully finishes, it will tell you where the manual got generated. The HTML will be accessible through the **result** symlink at `./result/share/doc/nixos/index.html`.

52.2. Editing DocBook XML

For general information on how to write in DocBook, see DocBook 5: The Definitive Guide.

Emacs nXML Mode is very helpful for editing DocBook XML because it validates the document as you write, and precisely locates errors. To use it, see Section 28.3.3, “Editing DocBook 5 XML Documents”.

Pandoc can generate DocBook XML from a multitude of formats, which makes a good starting point.

Example 52.1. Pandoc invocation to convert GitHub-Flavoured Markdown to DocBook 5 XML

```
pandoc -f markdown_github -t docbook5 docs.md -o my-section.md
```

Pandoc can also quickly convert a single `section.xml` to HTML, which is helpful when drafting.

Sometimes writing valid DocBook is simply too difficult. In this case, submit your documentation updates in a GitHub Issue and someone will handle the conversion to XML for you.

52.3. Creating a Topic

You can use an existing topic as a basis for the new topic or create a topic from scratch.

Keep the following guidelines in mind when you create and add a topic:

- The NixOS book element is in `nixos/doc/manual/manual.xml`. It includes several `parts` which are in subdirectories.
- Store the topic file in the same directory as the `part` to which it belongs. If your topic is about configuring a NixOS module, then the XML file can be stored alongside the module definition `nix` file.
- If you include multiple words in the file name, separate the words with a dash. For example: `ipv6-config.xml`.
- Make sure that the `xml:id` value is unique. You can use abbreviations if the ID is too long. For example: `nixos-config`.
- Determine whether your topic is a chapter or a section. If you are unsure, open an existing topic file and check whether the main element is chapter or section.

52.4. Adding a Topic to the Book

Open the parent XML file and add an `xi:include` element to the list of chapters with the file name of the topic that you created. If you created a `section`, you add the file to the `chapter` file. If you created a `chapter`, you add the file to the `part` file.

If the topic is about configuring a NixOS module, it can be automatically included in the manual by using the `meta.doc` attribute. See Section 50.5, “Meta Attributes” for an explanation.

Chapter 53. Building Your Own NixOS CD

Building a NixOS CD is as easy as configuring your own computer. The idea is to use another module which will replace your `configuration.nix` to configure the system that would be installed on the CD.

Default CD/DVD configurations are available inside `nixos/modules/installer/cd-dvd`.

```
$ git clone https://github.com/NixOS/nixpkgs.git
2 $ cd nixpkgs/nixos
$ nix-build -A config.system.build.isoImage -I
  ↪ nixos-config=modules/installer/cd-dvd/installation-cd-minimal.nix default.nix
```

Before burning your CD/DVD, you can check the content of the image by mounting anywhere like suggested by the following command:

```
1 # mount -o loop -t iso9660 ./result/iso/cd.iso /mnt/iso
```

Chapter 54. NixOS Tests

Table of Contents

54.1. Writing Tests

54.2. Running Tests

54.3. Running Tests interactively

When you add some feature to NixOS, you should write a test for it. NixOS tests are kept in the directory `nixos/tests`, and are executed (using Nix) by a testing framework that automatically starts one or more virtual machines containing the NixOS system(s) required for the test.

54.1. Writing Tests

A NixOS test is a Nix expression that has the following structure:

```
import ./make-test-python.nix {
2
  # Either the configuration of a single machine:
4  machine =
    { config, pkgs, ... }:
6    { configuration...
    };
8
  # Or a set of machines:
10 nodes =
    { machine1 =
12      { config, pkgs, ... }: { ... };
      machine2 =
14      { config, pkgs, ... }: { ... };
      ...
16    };
18 testScript =
    ''
20     Python code...
    '';
22 }
```

The attribute `testScript` is a bit of Python code that executes the test (described below). During the test, it will start one or more virtual machines, the configuration of which is described by the attribute `machine` (if you need only one machine in your test) or by the attribute `nodes` (if you need multiple machines). For instance, `login.nix` only needs a single machine to test whether users can log in on the virtual console, whether device ownership is correctly maintained when switching between consoles, and so on. On the other hand, `nfs.nix`, which tests NFS client and server functionality in the Linux kernel (including whether locks are maintained across server crashes), requires three machines: a server and two clients.

There are a few special NixOS configuration options for test VMs:

`virtualisation.memorySize` The memory of the VM in megabytes.

`virtualisation.vlans` The virtual networks to which the VM is connected. See `nat.nix` for an example.

`virtualisation.writableStore` By default, the Nix store in the VM is not writable. If you enable this option, a writable union file system is mounted on top of the Nix store to make it appear writable. This is necessary for tests that run Nix operations that modify the store.

For more options, see the module `qemu-vm.nix`.

The test script is a sequence of Python statements that perform various actions, such as starting VMs, executing commands in the VMs, and so on. Each virtual machine is represented as an object stored in the variable `name` if

this is also the identifier of the machine in the declarative config. If you didn't specify multiple machines using the `nodes` attribute, it is just `machine`. The following example starts the machine, waits until it has finished booting, then executes a command and checks that the output is more-or-less correct:

```
machine.start()
2 machine.wait_for_unit("default.target")
  if not "Linux" in machine.succeed("uname"):
4   raise Exception("Wrong OS")
```

The first line is actually unnecessary; machines are implicitly started when you first execute an action on them (such as `wait_for_unit` or `succeed`). If you have multiple machines, you can speed up the test by starting them in parallel:

```
start_all()
```

The following methods are available on machine objects:

start Start the virtual machine. This method is asynchronous -- it does not wait for the machine to finish booting.

shutdown Shut down the machine, waiting for the VM to exit.

crash Simulate a sudden power failure, by telling the VM to exit immediately.

block Simulate unplugging the Ethernet cable that connects the machine to the other machines.

unblock Undo the effect of `block`.

screenshot Take a picture of the display of the virtual machine, in PNG format. The screenshot is linked from the HTML log.

get_screen_text Return a textual representation of what is currently visible on the machine's screen using optical character recognition.

Note

This requires passing `enableOCR` to the test attribute set.

send_monitor_command Send a command to the QEMU monitor. This is rarely used, but allows doing stuff such as attaching virtual USB disks to a running machine.

send_key Simulate pressing keys on the virtual keyboard, e.g., `send_key("ctrl-alt-delete")`.

send_chars Simulate typing a sequence of characters on the virtual keyboard, e.g., `send_chars("foobar\n")` will type the string `foobar` followed by the Enter key.

execute Execute a shell command, returning a list (`status`, `stdout`).

succeed Execute a shell command, raising an exception if the exit status is not zero, otherwise returning the standard output.

fail Like `succeed`, but raising an exception if the command returns a zero status.

wait_until_succeeds Repeat a shell command with 1-second intervals until it succeeds.

wait_until_fails Repeat a shell command with 1-second intervals until it fails.

wait_for_unit Wait until the specified systemd unit has reached the "active" state.

wait_for_file Wait until the specified file exists.

wait_for_open_port Wait until a process is listening on the given TCP port (on `localhost`, at least).

wait_for_closed_port Wait until nobody is listening on the given TCP port.

wait_for_x Wait until the X11 server is accepting connections.

wait_for_text Wait until the supplied regular expressions matches the textual contents of the screen by using optical character recognition (see `get_screen_text`).

Note

This requires passing `enableOCR` to the test attribute set.

wait_for_console_text Wait until the supplied regular expressions match a line of the serial console output. This method is useful when OCR is not possible or accurate enough.

wait_for_window Wait until an X11 window has appeared whose name matches the given regular expression, e.g., `wait_for_window("Terminal")`.

copy_from_host Copies a file from host to machine, e.g., `copy_from_host("myfile", "/etc/my/important/file")`.

The first argument is the file on the host. The file needs to be accessible while building the nix derivation. The second argument is the location of the file on the machine.

systemctl Runs `systemctl` commands with optional support for `systemctl --user`

```
machine.systemctl("list-jobs --no-pager") # runs `systemctl list-jobs --no-pager`
2 machine.systemctl("list-jobs --no-pager", "any-user") # spawns a shell for
  ↳ `any-user` and runs `systemctl --user list-jobs --no-pager`
```

To test user units declared by `systemd.user.services` the optional `user` argument can be used:

```
machine.start()
2 machine.wait_for_x()
machine.wait_for_unit("xautolock.service", "x-session-user")
```

This applies to `systemctl`, `get_unit_info`, `wait_for_unit`, `start_job` and `stop_job`.

For faster dev cycles it's also possible to disable the code-linters (this shouldn't be committed though):

```
import ./make-test-python.nix {
2   skipLint = true;
   machine =
4     { config, pkgs, ... }:
      { configuration...
6   };

8   testScript =
    ''
10     Python code...
    '';
12 }
```

54.2. Running Tests

You can run tests using **nix-build**. For example, to run the test `login.nix`, you just do:

```
$ nix-build '<nixpkgs/nixos/tests/login.nix>'
```

or, if you don't want to rely on `NIX_PATH`:

```
$ cd /my/nixpkgs/nixos/tests
2 $ nix-build login.nix
...
4 running the VM test script
machine: QEMU running (pid 8841)
6 ...
6 out of 6 tests succeeded
```

After building/downloading all required dependencies, this will perform a build that starts a QEMU/KVM virtual machine containing a NixOS system. The virtual machine mounts the Nix store of the host; this makes VM creation very fast, as no disk image needs to be created. Afterwards, you can view a pretty-printed log of the test:

```
1 $ firefox result/log.html
```

54.3. Running Tests interactively

The test itself can be run interactively. This is particularly useful when developing or debugging a test:

```
1 $ nix-build nixos/tests/login.nix -A driver
$ ./result/bin/nixos-test-driver
3 starting VDE switch for network 1
>
```

You can then take any Python statement, e.g.

```
> start_all()
2 > test_script()
> machine.succeed("touch /tmp/foo")
4 > print(machine.succeed("pwd")) # Show stdout of command
```

The function **test_script** executes the entire test script and drops you back into the test driver command line upon its completion. This allows you to inspect the state of the VMs after the test (e.g. to debug the test script).

To just start and experiment with the VMs, run:

```
$ nix-build nixos/tests/login.nix -A driver
2 $ ./result/bin/nixos-run-vm
```

The script **nixos-run-vm** starts the virtual machines defined by test.

You can re-use the VM states coming from a previous run by setting the **--keep-vm-state** flag.

```
$ ./result/bin/nixos-run-vm --keep-vm-state
```

The machine state is stored in the `$TMPDIR/vm-state-machinename` directory.

Chapter 55. Testing the Installer

Building, burning, and booting from an installation CD is rather tedious, so here is a quick way to see if the installer works properly:

```
# mount -t tmpfs none /mnt
2 # nixos-generate-config --root /mnt
$ nix-build '<nixpkgs/nixos>' -A config.system.build.nixos-install
4 # ./result/bin/nixos-install
```

To start a login shell in the new NixOS installation in `/mnt`:

```
$ nix-build '<nixpkgs/nixos>' -A config.system.build.nixos-enter
2 # ./result/bin/nixos-enter
```

Chapter 56. Releases

Table of Contents

56.1. Release process

56.2. Release Management Team

56.3. Release schedule

56.1. Release process

Going through an example of releasing NixOS 19.09:

56.1.1. One month before the beta

- Create an announcement on Discourse as a warning about upcoming beta “feature freeze” in a month. See this post as an example.
- Discuss with Eelco Dolstra and the community (via IRC, ML) about what will reach the deadline. Any issue or Pull Request targeting the release should be included in the release milestone.
- Remove attributes that we know we will not be able to support, especially if there is a stable alternative. E.g. Check that our Linux kernels’ projected end-of-life are after our release projected end-of-life.

56.1.2. At beta release time

1. From the master branch run:

```
git checkout -b release-19.09
```

2. Bump the `system.defaultChannel` attribute in `nixos/modules/misc/version.nix`
3. Update `versionSuffix` in `nixos/release.nix`

To get the commit count, use the following command:

```
git rev-list --count release-19.09
```

1. Edit changelog at `nixos/doc/manual/release-notes/rl-1909.xml`.

- Get all new NixOS modules:

```
git diff release-19.03..release-19.09 nixos/modules/module-list.nix | grep ^+
```

- Note `systemd`, `kernel`, `glibc`, `desktop environment`, and `Nix upgrades`.

2. Tag the release:

```
1 git tag --annotate --message="Release 19.09-beta" 19.09-beta
git push upstream 19.09-beta
```

3. On the `master` branch, increment the `.version` file

```
echo -n "20.03" > .version
```

4. Update `codeName` in `lib/trivial.nix` This will be the name for the next release.
5. Create a new release notes file for the upcoming release + 1, in our case this is `rl-2003.xml`.
6. Contact the infrastructure team to create the necessary Hydra Jobsets.
7. Create a channel at <https://nixos.org/channels> by creating a PR to `nixos-org-configurations`, changing `channels.nix`
8. Get all Hydra jobsets for the release to have their first evaluation.
9. Create an issue for tracking Zero Hydra Failures progress. ZHF is an effort to get build failures down to zero.

56.1.3. During Beta

- Monitor the master branch for bugfixes and minor updates and cherry-pick them to the release branch.

56.1.4. Before the final release

- Re-check that the release notes are complete.
- Release Nix (currently only Eelco Dolstra can do that). Make sure fallback is updated.
- Update README.md with new stable NixOS version information.
- Change `stableBranch` to `true` in Hydra and wait for the channel to update.

56.1.5. At final release time

1. Update Chapter 4, *Upgrading NixOS* section of the manual to match new stable release version.
2. Update `rl-1909.xml` with the release date.
3. Tag the final release

```
git tag --annotate --message="Release 19.09" 19.09
2 git push upstream 19.09
```

4. Update nixos-homepage for the release.
 - a. Update `NIXOS_SERIES` in the Makefile.
 - b. Update `nixos-release.tt` with the new NixOS version.
 - c. Update the `flake.nix` input `released-nixpkgs` to 19.09.
 - d. Run `./update.sh` (this updates `flake.lock` to updated channel).
 - e. Add a compressed version of the NixOS logo for 19.09.
 - f. Compose a news item for the website RSS feed.
5. Create a new topic on the Discourse instance to announce the release.

You should include the following information:

- Number of commits for the release:

```
bash git log release-19.03..release-19.09 --format=%an | wc -l
```

- Commits by contributor:

```
1 git shortlog --summary --numbered release-19.03..release-19.09
```

Best to check how the previous post was formulated to see what needs to be included.

56.2. Release Management Team

For each release there are two release managers. After each release the release manager having managed two releases steps down and the release management team of the last release appoints a new release manager.

This makes sure a release management team always consists of one release manager who already has managed one release and one release manager being introduced to their role, making it easier to pass on knowledge and experience.

Release managers for the current NixOS release are tracked by GitHub team `@NixOS/nixos-release-managers`.

A release manager's role and responsibilities are:

- manage the release process
- start discussions about features and changes for a given release
- create a roadmap
- release in cooperation with Eelco Dolstra
- decide which bug fixes, features, etc... get backported after a release

56.3. Release schedule

Date	Event
2016-07-25	Send email to nix-dev about upcoming branch-off
2016-09-01	release-16.09 branch and corresponding jobsets are created, change freeze
2016-09-30	NixOS 16.09 released