



NixOS Manual

Version 24.05

Table of Contents

[Preface](#)

[Installation](#)

[Obtaining NixOS](#)

[Installing NixOS](#)

[Changing the Configuration](#)

[Upgrading NixOS](#)

[Building a NixOS \(Live\) ISO](#)

[Building Images via `systemd-repart`](#)

[Configuration](#)

[Configuration Syntax](#)

[Package Management](#)

[User Management](#)

[File Systems](#)

[X Window System](#)

[Wayland](#)

v: unstable -

[GPU acceleration](#)

[Xfce Desktop Environment](#)

[Networking](#)

[Linux Kernel](#)

[Subversion](#)

[Pantheon Desktop](#)

[GNOME Desktop](#)

[External Bootloader Backends](#)

[Clevis](#)

[Garage](#)

[Suwayomi-Server](#)

[Plausible](#)

[Pict-rs](#)

[Nextcloud](#)

[Matomo](#)

[Lemmy](#)

[Keycloak](#)

[Jitsi Meet](#)

[Honk](#)

[Grocy](#)

[GoToSocial](#)

[Discourse](#)[c2FmZQ](#)[Akkoma](#)[systemd-lock-handler](#)[Meilisearch](#)[Yggdrasil](#)[Prosody](#)[Pleroma](#)[Netbird](#)[Mosquitto](#)[GNS3 Server](#)[Firefox Sync server](#)[Dnsmasq](#)[Litestream](#)[Prometheus exporters](#)[parsedmarc](#)[OCS Inventory Agent](#)[Goss](#)[Cert Spotter](#)[WeeChat](#)[Taskserver](#)

[Sourcehut](#)

[GitLab](#)

[Forgejo](#)

[Apache Kafka](#)

[Anki Sync Server](#)

[Matrix](#)

[Mjolnir \(Matrix Moderation Tool\)](#)

[Maubot](#)

[Mailman](#)

[Trezor](#)

[Emacs](#)

[Livebook](#)

[Blackfire profiler](#)

[Athens](#)

[Flatpak](#)

[TigerBeetle](#)

[PostgreSQL](#)

[FoundationDB](#)

[BorgBackup](#)

[Castopod](#)

[SSL/TLS Certificates with ACME](#)

[Oh my ZSH](#)

[Plotinus](#)

[Digital Bitbox](#)

[Input Methods](#)

[Profiles](#)

[Kubernetes](#)

[Administration](#)

[Service Management](#)

[Rebooting and Shutting Down](#)

[User Sessions](#)

[Control Groups](#)

[Logging](#)

[Necessary system state](#)

[Cleaning the Nix Store](#)

[Container Management](#)

[Troubleshooting](#)

[Development](#)

[Getting the Sources](#)

[Writing NixOS Modules](#)

[Building Specific Parts of NixOS](#)

[Experimental feature: Bootspec](#)

[What happens during a system switch?](#)

[Writing NixOS Documentation](#)

[NixOS Tests](#)

[Developing the NixOS Test Driver](#)

[Testing the Installer](#)

[Contributing to this manual](#)

[A. Configuration Options](#)

[B. Release Notes](#)

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](#), the [Matrix room](#), or on the [#nixos channel](#) on [Libera.Chat](#). Alternatively, consider [contributing to this manual](#). 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.

Installation

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

Table of Contents

[Obtaining NixOS](#)

[Installing NixOS](#)

[Changing the Configuration](#)

[Upgrading NixOS](#)

[Building a NixOS \(Live\) ISO](#)

[Building Images via `systemd-repart`](#)

Obtaining NixOS

NixOS ISO images can be downloaded from the [NixOS download page](#). Follow the instructions in [the section called “Booting from a USB flash drive”](#) to create a bootable USB flash drive.

If you have a very old system that can't boot from USB, you can burn the image to an empty CD. NixOS might not work very well on such systems.

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, please refer to the [download page](#).
- 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.

Installing NixOS

v: unstable -

Table of Contents

[Booting from the install medium](#)

[Graphical Installation](#)

[Manual Installation](#)

[Additional installation notes](#)

Booting from the install medium

To begin the installation, you have to boot your computer from the install drive.

1. Plug in the install drive. Then turn on or restart your computer.
2. Open the boot menu by pressing the appropriate key, which is usually shown on the display on early boot. Select the USB flash drive (the option usually contains the word “USB”). If you choose the incorrect drive, your computer will likely continue to boot as normal. In that case restart your computer and pick a different drive.

Note

The key to open the boot menu is different across computer brands and even models. It can be **F12**, but also **F1**, **F9**, **F10**, **Enter**, **Del**, **Esc** or another function key. If you are unsure and don't see it on the early boot screen, you can search online for your computers brand, model followed by “boot from usb”. The computer might not even have that feature, so you have to go into the BIOS/UEFI settings to change the boot order. Again, search online for details about your specific computer model.

For Apple computers with Intel processors press and hold the **⌥** (Option or Alt) key until you see the boot menu. On Apple silicon press and hold the power button.

Note

If your computer supports both BIOS and UEFI boot, choose the UEFI option.

v: unstable -

Note

If you use a CD for the installation, the computer will probably boot from it automatically. If not, choose the option containing the word “CD” from the boot menu.

3. Shortly after selecting the appropriate boot drive, you should be presented with a menu with different installer options. Leave the default and wait (or press **Enter** to speed up).
4. The graphical images will start their corresponding desktop environment and the graphical installer, which can take some time. The minimal images will boot to a command line. You have to follow the instructions in [the section called “Manual Installation”](#) there.

Graphical Installation

The graphical installer is recommended for desktop users and will guide you through the installation.

1. In the “Welcome” screen, you can select the language of the Installer and the installed system.

Tip

Leaving the language as “American English” will make it easier to search for error messages in a search engine or to report an issue.

2. Next you should choose your location to have the timezone set correctly. You can actually click on the map!

Note

The installer will use an online service to guess your location based on your public IP address.

3. Then you can select the keyboard layout. The default keyboard model should work well with most desktop keyboards. If you have a special keyboard or notebook, your model might be in the list. Select the language you are most comfortable typing in.
4. On the “Users” screen, you have to type in your display name, login name and pass\ v: unstable -

also enable an option to automatically login to the desktop.

5. Then you have the option to choose a desktop environment. If you want to create a custom setup with a window manager, you can select “No desktop”.

Tip

If you don't have a favorite desktop and don't know which one to choose, you can stick to either GNOME or Plasma. They have a quite different design, so you should choose whichever you like better. They are both popular choices and well tested on NixOS.

6. You have the option to allow unfree software in the next screen.

7. The easiest option in the “Partitioning” screen is “Erase disk”, which will delete all data from the selected disk and install the system on it. Also select “Swap (with Hibernation)” in the dropdown below it. You have the option to encrypt the whole disk with LUKS.

Note

At the top left you see if the Installer was booted with BIOS or UEFI. If you know your system supports UEFI and it shows “BIOS”, reboot with the correct option.

Warning

Make sure you have selected the correct disk at the top and that no valuable data is still on the disk! It will be deleted when formatting the disk.

8. Check the choices you made in the “Summary” and click “Install”.

Note

The installation takes about 15 minutes. The time varies based on the selected desktop environment, internet connection speed and disk write speed.

9. When the install is complete, remove the USB flash drive and reboot into your new system!

Manual Installation

NixOS can be installed on BIOS or UEFI systems. The procedure for a UEFI installation is broadly the same as for a BIOS installation. The differences are mentioned in the following steps.

The NixOS manual is available by running `nixos-help` in the command line or from the application menu in the desktop environment.

To have access to the command line on the graphical images, open Terminal (GNOME) or Konsole (Plasma) from the application menu.

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

```
$ sudo -i
```

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.

To install over a serial port connect with `115200n8` (e.g. `picocom -b 115200 /dev/ttyUSB0`). When the bootloader lists boot entries, select the serial console boot entry.

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`.

On the graphical installer, you can configure the network, wifi included, through NetworkManager. Using the `nmtui` program, you can do so even in a non-graphical session. If you prefer to configure the network manually, disable NetworkManager with `systemctl stop NetworkManager`.

On the minimal installer, NetworkManager is not available, so configuration must be performed manually. To configure the wifi, first start `wpa_supplicant` with `sudo systemctl start v:unstable -`

wpa_supplicant, then run `wpa_cli`. For most home networks, you need to type in the following commands:

```
> add_network
0
> set_network 0 ssid "myhomenetwork"
OK
> set_network 0 psk "mypassword"
OK
> set_network 0 key_mgmt WPA-PSK
OK
> enable_network 0
OK
```

For enterprise networks, for example `eduroam`, instead do:

```
> add_network
0
> set_network 0 ssid "eduroam"
OK
> set_network 0 identity "myname@example.com"
OK
> set_network 0 password "mypassword"
OK
> set_network 0 key_mgmt WPA-EAP
OK
> enable_network 0
OK
```

When successfully connected, you should see a line such as this one

```
<3>CTRL-EVENT-CONNECTED - Connection to 32:85:ab:ef:24:5c completed [id=0]
```

you can now leave `wpa_cli` by typing `quit`.

If you would like to continue the installation from a different machine you can use `activat` v: unstable -

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.

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*.

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.

```
# parted /dev/sda -- mkpart root ext4 512MB -8GB
```

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

```
# parted /dev/sda -- mkpart swap linux-swap -8GB 100%
```

v: unstable -

Note

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

- 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.

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

Once complete, you can follow with [the section called “Formatting”](#).

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*.

- Create a *MBR* partition table.

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

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

```
# parted /dev/sda -- mkpart primary 1MB -8GB
```

- Set the root partition's boot flag to on. This allows the disk to be booted from.

```
# parted /dev/sda -- set 1 boot on
```

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

v: unstable -

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

Note

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

Once complete, you can follow with [the section called “Formatting”](#).

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., `pvccreate`, `vgcreate`, and `lvcreate`.
- For creating software RAID devices, use `mdadm`.

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
# 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 [Configuration Syntax](#), while a list of available configuration options appears in [Appendix A](#). A minimal example is shown in [Example: 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.

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.

UEFI systems

You must select a boot-loader, either systemd-boot or GRUB. The recommended option is systemd-boot: 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-boot` as well.

If you want to use GRUB, set `boot.loader.grub.device` to `nodev` and `boot.loader.grub.efiSupport` to `true`.

With systemd-boot, you should not need any special configuration to detect other installed systems. With GRUB, set `boot.loader.grub.useOSProber` to `true`, but this will only detect windows partitions, not other Linux distributions. If you dual boot another Linux distribution, use systemd-boot instead.

If you need to configure networking for your machine the configuration options are described in [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 located in `/etc/nixos`.)

v: unstable -

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...
New password: ***
Retype new 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:

v: unstable -

```
# 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  
$ passwd eelco
```

You may also want to install some software. This will be covered in [Package Management](#).

Installation summary

To summarise, [Example: 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: NixOS Configuration](#) shows a corresponding configuration Nix expression.

Example 1. Example partition schemes for NixOS on /dev/sda (MBR)

```
# parted /dev/sda -- mklabel msdos  
# parted /dev/sda -- mkpart primary 1MB -8GB  
# parted /dev/sda -- mkpart primary linux-swap -8GB 100%
```

Example 2. Example partition schemes for NixOS on /dev/sda (UEFI)

```
# parted /dev/sda -- mklabel gpt  
# parted /dev/sda -- mkpart root ext4 512MB -8GB
```

v: unstable -

```
# parted /dev/sda -- mkpart swap linux-swap -8GB 100%
# parted /dev/sda -- mkpart ESP fat32 1MB 512MB
# parted /dev/sda -- set 3 esp on
```

Example 3. Commands for Installing NixOS on /dev/sda

With a partitioned disk.

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

Example 4. Example: NixOS Configuration

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

  boot.loader.grub.device = "/dev/sda";    # (for BIOS systems only)
  boot.loader.systemd-boot.enable = true; # (for UEFI systems only)

  # Note: setting fileSystems is generally not
```

v: unstable -

```
# necessary, since nixos-generate-config figures them out
# automatically in hardware-configuration.nix.
#fileSystems."/".device = "/dev/disk/by-label/nixos";

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

Additional installation notes

Booting from a USB flash drive

The image has to be written verbatim to the USB flash drive for it to be bootable on UEFI and BIOS systems. Here are the recommended tools to do that.

Creating bootable USB flash drive with a graphical tool

Etcher is a popular and user-friendly tool. It works on Linux, Windows and macOS.

Download it from [balena.io](https://www.balena.io/etcher/), start the program, select the downloaded NixOS ISO, then select the USB flash drive and flash it.

Warning

Etcher reports errors and usage statistics by default, which can be disabled in the settings.

An alternative is [USBImager](https://github.com/balena-io-teambuilders/usb-imager), which is very simple and does not connect to the internet. Download the version with write-only (wo) interface for your system. Start the program, select the image, select the USB flash drive and click “Write”.

Creating bootable USB flash drive from a Terminal on Linux

1. Plug in the USB flash drive.
2. Find the corresponding device with `lsblk`. You can distinguish them by their size.

v: unstable -

3. Make sure all partitions on the device are properly unmounted. Replace `sdX` with your device (e.g. `sdb`).

```
sudo umount /dev/sdX*
```

4. Then use the `dd` utility to write the image to the USB flash drive.

```
sudo dd bs=4M conv=fsync oflag=direct status=progress if=<path-to-image> of=/dev/sdX
```

Creating bootable USB flash drive from a Terminal on macOS

1. Plug in the USB flash drive.
2. Find the corresponding device with `diskutil list`. You can distinguish them by their size.
3. Make sure all partitions on the device are properly unmounted. Replace `diskX` with your device (e.g. `disk1`).

```
diskutil unmountDisk diskX
```

4. Then use the `dd` utility to write the image to the USB flash drive.

```
sudo dd if=<path-to-image> of=/dev/rdiskX bs=4m
```

After `dd` completes, a GUI dialog “The disk you inserted was not readable by this computer” will pop up, which can be ignored.

Note

Using the ‘raw’ `rdiskX` device instead of `diskX` with `dd` completes in minutes instead of hours.

5. Eject the disk when it is finished.

```
diskutil eject /dev/diskX
```

v: unstable -

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 want to add the NixOS installer as another option. To build the necessary files from your current version of nixpkgs, you can run:

```
nix-build -A netboot.x86_64-linux '<nixpkgs/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.

“Booting” into NixOS via kexec

In some cases, your system might already be booted into/preinstalled with another Linux distribution, and booting NixOS by attaching an installation image is quite a manual process.

This is particularly useful for (cloud) providers where you can’t boot a custom image, but get some Debian or Ubuntu installation.

In these cases, it might be easier to use `kexec` to “jump into NixOS” from the running system, which only assumes `bash` and `kexec` to be installed on the machine.

Note that `kexec` may not work correctly on some hardware, as devices are not fully re-init

process. In practice, this however is rarely the case.

To build the necessary files from your current version of nixpkgs, you can run:

```
nix-build -A kexec.x86_64-linux '<nixpkgs/nixos/release.nix>'
```

This will create a `result` directory containing the following:

- `bzImage` (the Linux kernel)
- `initrd` (the initrd file)
- `kexec-boot` (a shellscript invoking `kexec`)

These three files are meant to be copied over to the other already running Linux Distribution.

Note its symlinks pointing elsewhere, so `cd` in, and use `scp * root@$destination` to copy it over, rather than `rsync`.

Once you finished copying, execute `kexec-boot` on the destination, and after some seconds, the machine should be booting into an (ephemeral) NixOS installation medium.

In case you want to describe your own system closure to `kexec` into, instead of the default installer image, you can build your own `configuration.nix`:

```
{ modulesPath, ... }: {
  imports = [
    (modulesPath + "/installer/netboot/netboot-minimal.nix")
  ];

  services.openssh.enable = true;
  users.users.root.openssh.authorizedKeys.keys = [
    "my-ssh-pubkey"
  ];
}
```

```
nix-build '<nixpkgs/nixos>' \
```

v: unstable -

```
--arg configuration ./configuration.nix  
--attr config.system.build.kexecTree
```

Make sure your `configuration.nix` does still import `netboot-minimal.nix` (or `netboot-base.nix`).

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 setti

v: unstable -

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.

```
{ config, pkgs, ...} :  
{  
  fileSystems."/virtualboxshare" = {  
    fsType = "vboxsf";  
    device = "nameofthesharedfolder";  
    options = [ "rw" "nofail" ];  
  };  
}
```

The folder will be available directly under the root directory.

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_LUSTRE`.
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  
$ . $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  
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`, but this also makes some man pages and `nixos-enter` available, just in case you want to chroot into your NixOS partition. NixOS installs these by default, but you don't have NixOS yet...

```
$ nix-env -f '<nixpkgs>' -iA nixos-install-tools
```

4. **Note**

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

Prepare your target partition:

v: unstable -

At this point it is time to prepare your target partition. Please refer to the partitioning, file-system creation, and mounting steps of [Installing NixOS](#)

If you're about to install NixOS in place using `NIXOS_LUSTRE` 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 [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 = ''  
  menuentry "Ubuntu" {  
    search --set=ubuntu --fs-uuid 3cc3e652-0c1f-4800-8451-033754f68e6  
    configfile "($ubuntu)/boot/grub/grub.cfg"  
  }  
'';
```

(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  
$ 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 sy

v: unstable -

without the help of a rescue USB drive or similar.

Note

On some distributions there are separate PATHS for programs intended only for root. In order for the installation to succeed, you might have to use `PATH="$PATH:/usr/sbin:/sbin"` in the following command.

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

Again, please refer to the `nixos-install` step in [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  
$ 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_LUSTRE`:

Generate your NixOS configuration:

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

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 [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 nix
```

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 no

or `/boot` gets “lustrated” on the next boot.

lustrate /'lʌstreɪt/ verb.

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

Let's create the files:

```
$ sudo touch /etc/NIXOS  
$ 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 &&  
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:

v: unstable -

```
# mkdir root
# mount /dev/sdaX root
# mkdir root/nixos-root
# mv -v root/* root/nixos-root/
# mv -v root/nixos-root/old-root/* root/
# mv -v root/boot.bak root/boot # We had renamed this by hand earlier
# umount root
# 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 providers do not provide NixOS. For instance, [nixos-infect](#) uses the `lustrate` process to convert Digital Ocean droplets to NixOS from other distributions automatically.

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/";
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/"
# export http_proxy="$proxy_url"
# export HTTP_PROXY="$proxy_url"
# export https_proxy="$proxy_url"
```

v: unstable -

```
# 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](#) for more information.

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 wor v: unstable -

configuration.

There is also

```
# 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.

```
# 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.

A repl, or read-eval-print loop, is also available. You can inspect your configuration and use the Nix language with

```
# nixos-rebuild repl
```

Your configuration is loaded into the `config` variable. Use tab for autocompletion, use the `:r` command to reload the configuration files. See `:?` or [nix repl](#) in the Nix manual to learn more.

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

v: unstable -

```
$ 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):

```
$ QEMU_NET_OPTS="hostfwd=tcp:127.0.0.1:2222-:22" ./result/bin/run-*-vm
```

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

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

Such port forwardings connect via the VM's virtual network interface. Thus they cannot connect to ports that are only bound to the VM's loopback interface (`127.0.0.1`), and the VM's NixOS firewall must be configured to allow these connections.

Upgrading NixOS

Table of Contents

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 `channel` command provides a convenient way to manage channels and update the system.

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-23.11](#). 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-23.11-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://channels.nixos.org>. (Note that the URLs 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 23.11 ISO, you will be subscribed to the [nixos-23.11](#) channel. To see which NixOS channel you're subscribed to, run the following as root:

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

To switch to a different NixOS channel, do

v: unstable -

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

(Be sure to include the `nixos` parameter at the end.) For instance, to use the NixOS 23.11 stable channel:

```
# nix-channel --add https://channels.nixos.org/nixos-23.11 nixos
```

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

```
# nix-channel --add https://channels.nixos.org/nixos-23.11-small nixos
```

And if you want to live on the bleeding edge:

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

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

```
# 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.

Automatic Upgrades

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

```
system.autoUpgrade.enable = true;  
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://channels.nixos.org/nixos-23.11";
```

Building a NixOS (Live) ISO

Table of Contents

[Practical Instructions](#)

[Additional drivers or firmware](#)

[Technical Notes](#)

Default live installer configurations are available inside `nixos/modules/installer/cd-dvd`. For building other system images, [nixos-generators](#) is a good place to start looking at.

You have two options:

- Use any of those default configurations as is
- Combine them with (any of) your host config(s)

System images, such as the live installer ones, know how to enforce configuration settings on which they immediately depend in order to work correctly.

However, if you are confident, you can opt to override those enforced values with `mkForce`.

Practical Instructions

To build an ISO image for the channel `nixos-unstable`:

```
$ git clone https://github.com/NixOS/nixpkgs.git
$ cd nixpkgs/nixos
$ git switch nixos-unstable
$ nix-build -A config.system.build.isoImage -I nixos-config=modules/installer/cd-dvd/installation-cd-graphical-nixos.nix
```

To check the content of an ISO image, mount it like so:

```
# mount -o loop -t iso9660 ./result/iso/cd.iso /mnt/iso
```

Additional drivers or firmware

If you need additional (non-distributable) drivers or firmware in the installer, you might want to extend these configurations.

For example, to build the GNOME graphical installer ISO, but with support for certain WiFi adapters present in some MacBooks, you can create the following file at `modules/installer/cd-dvd/installation-cd-graphical-gnome-macbook.nix`:

v: unstable -

```
{ config, ... }:

{

  imports = [ ./installation-cd-graphical-gnome.nix ];

  boot.initrd.kernelModules = [ "wl" ];

  boot.kernelModules = [ "kvm-intel" "wl" ];
  boot.extraModulePackages = [ config.boot.kernelPackages.broadcom_sta ];
}
```

Then build it like in the example above:

```
$ git clone https://github.com/NixOS/nixpkgs.git
$ cd nixpkgs/nixos
$ export NIXPKGS_ALLOW_UNFREE=1
$ nix-build -A config.system.build.isoImage -I nixos-config=modules/insta
```

Technical Notes

The config value enforcement is implemented via `mkImageMediaOverride = mkOverride 60;` and therefore primes over simple value assignments, but also yields to `mkForce`.

This property allows image designers to implement in semantically correct ways those configuration values upon which the correct functioning of the image depends.

For example, the iso base image overrides those file systems which it needs at a minimum for correct functioning, while the installer base image overrides the entire file system layout because there can't be any other guarantees on a live medium than those given by the live medium itself. The latter is especially true before formatting the target block device(s). On the other hand, the netboot iso only overrides its minimum dependencies since netboot images are always made-to-target.

Building Images via `systemd-repart`

Table of Contents

v: unstable -

Nix Store Partition

Appliance Image

You can build disk images in NixOS with the `image.repart` option provided by the module [image/repart.nix](#). This module uses `systemd-repart` to build the images and exposes its entire interface via the `repartConfig` option.

An example of how to build an image:

```
{ config, modulesPath, ... }: {  
  
  imports = [ "${modulesPath}/image/repart.nix" ];  
  
  image.repart = {  
    name = "image";  
    partitions = {  
      "esp" = {  
        contents = {  
          ...  
        };  
        repartConfig = {  
          Type = "esp";  
          ...  
        };  
      };  
      "root" = {  
        storePaths = [ config.system.build.toplevel ];  
        repartConfig = {  
          Type = "root";  
          Label = "nixos";  
          ...  
        };  
      };  
    };  
  };  
};
```

v: unstable -

{

Nix Store Partition

You can define a partition that only contains the Nix store and then mount it under `/nix/store`.

Because the `/nix/store` part of the paths is already determined by the mount point, you have to set `stripNixStorePrefix = true;` so that the prefix is stripped from the paths before copying them into the image.

```
fileSystems."/nix/store".device = "/dev/disk/by-partlabel/nix-store"

image.repart.partitions = {
  "store" = {
    storePaths = [ config.system.build.toplevel ];
    stripNixStorePrefix = true;
    repartConfig = {
      Type = "linux-generic";
      Label = "nix-store";
      ...
    };
  };
};
```

Appliance Image

The `image/repart.nix` module can also be used to build self-contained [software appliances](#).

The generation based update mechanism of NixOS is not suited for appliances. Updates of appliances are usually either performed by replacing the entire image with a new one or by updating partitions via an A/B scheme. See the [Chrome OS update process](#) for an example of how to achieve this. The appliance image built in the following example does not contain a `configuration.nix` and thus you will not be able to call `nixos-rebuild` from this system.

let

v: unstable -

```
pkgs = import <nixpkgs> { };
efiArch = pkgs.stdenv.hostPlatform.efiArch;
in
(pkgs.nixos [
 ({ config, lib, pkgs, modulesPath, ... }: {
   imports = [ "${modulesPath}/image/repart.nix" ];
   boot.loader.grub.enable = false;
   fileSystems."/".device = "/dev/disk/by-label/nixos";
   image.repart = {
     name = "image";
     partitions = {
       "esp" = {
         contents = {
           "/EFI/BOOT/BOOT${lib.toUpperCase efiArch}.EFI".source =
           "${pkgs.systemd}/lib/systemd/boot/efi/systemd-boot${efiArch}/
             "/loader/entries/nixos.conf".source = pkgs.writeText "nixos.conf"
               title NixOS
               linux /EFI/nixos/kernel.efd
               initrd /EFI/nixos/initrd.efd
               options init=${config.system.build.toplevel}/init ${toString config.system.boot}
             '';
           "/EFI/nixos/kernel.efd".source =
           "${config.boot.kernelPackages.kernel}/${config.system.boot}.
             "/EFI/nixos/initrd.efd".source =
           "${config.system.build.initialRamdisk}/${config.system.boot.
         };
       repartConfig = {
         Type = "esp";
         Format = "vfat";
         SizeMinBytes = "96M";
       };
     };
   };
 });
});
```

v: unstable -

```
};

"root" = {
    storePaths = [ config.system.build.toplevel ];
    repartConfig = {
        Type = "root";
        Format = "ext4";
        Label = "nixos";
        Minimize = "guess";
    };
};

});

]).image
```

Configuration

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

Table of Contents

[Configuration Syntax](#)

[Package Management](#)

[User Management](#)

[File Systems](#)

[X Window System](#)

[Wayland](#)

[GPU acceleration](#)

v: unstable -

[Xfce Desktop Environment](#)[Networking](#)[Linux Kernel](#)[Subversion](#)[Pantheon Desktop](#)[GNOME Desktop](#)[External Bootloader Backends](#)[Clevis](#)[Garage](#)[Suwayomi-Server](#)[Plausible](#)[Pict-rs](#)[Nextcloud](#)[Matomo](#)[Lemmy](#)[Keycloak](#)[Jitsi Meet](#)[Honk](#)[Grocy](#)[GoToSocial](#)[Discourse](#)

[c2FmZQ](#)

[Akkoma](#)

[systemd-lock-handler](#)

[Meilisearch](#)

[Yggdrasil](#)

[Prosody](#)

[Pleroma](#)

[Netbird](#)

[Mosquitto](#)

[GNS3 Server](#)

[Firefox Sync server](#)

[Dnsmasq](#)

[Litestream](#)

[Prometheus exporters](#)

[parsedmarc](#)

[OCS Inventory Agent](#)

[Goss](#)

[Cert Spotter](#)

[WeeChat](#)

[Taskserver](#)

[Sourcehut](#)

[GitLab](#)[Forgejo](#)[Apache Kafka](#)[Anki Sync Server](#)[Matrix](#)[Mjolnir \(Matrix Moderation Tool\)](#)[Maubot](#)[Mailman](#)[Trezor](#)[Emacs](#)[Livebook](#)[Blackfire profiler](#)[Athens](#)[Flatpak](#)[TigerBeetle](#)[PostgreSQL](#)[FoundationDB](#)[BorgBackup](#)[Castopod](#)[SSL/TLS Certificates with ACME](#)[Oh my ZSH](#)

[Plotinus](#)[Digital Bitbox](#)[Input Methods](#)[Profiles](#)[Kubernetes](#)

Configuration Syntax

Table of Contents

[NixOS Configuration File](#)[Abstractions](#)[Modularity](#)

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.

NixOS Configuration File

The NixOS configuration file generally looks like this:

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

{ option definitions
}
```

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, in chapter [Writing NixOS Modules](#)) 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,

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

{ services.httpd.enable = true;
  services.httpd.adminAddr = "alice@example.org";
  services.httpd.virtualHosts.localhost.documentRoot = "/webroot";
}
```

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:

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

{ services = {
    httpd = {
      enable = true;
      adminAddr = "alice@example.org";
      virtualHosts = {
        localhost = {
          documentRoot = "/webroot";
        };
      };
    };
}
```

v: unstable -

```
    };
  };
};

}
```

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'
```

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'
```

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

Strings

Strings are enclosed in double quotes, e.g.

```
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 =
  ''
  127.0.0.2 other-localhost
  10.0.0.1 server
```

v: unstable -

```
'';
```

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;
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" =
{ device = "/dev/sda1";
  fsType = "ext4";
  options = [ "rw" "data=ordered" "relatime" ];
};
```

Lists

The important thing to note about lists is that list elements are separated by whites

v: unstable -

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

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

```
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 =
[ pkgs.thunderbird
  pkgs.emacs
];

services.postgresql.package = pkgs.postgresql_14;
```

The latter option definition changes the default PostgreSQL package used by NixOS's PostgreSQL service to 14.x. For more information on packages, including how to add new ones, see [the section called “Adding Custom Packages”](#).

Abstractions

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

```
{
  services.httpd.virtualHosts =
  { "blog.example.org" = {
      documentRoot = "/webroot/blog.example.org";
      adminAddr = "alice@example.org";
      forceSSL = true;
      enableACME = true;
      enablePHP = true;
```

v: unstable -

```
};

"wiki.example.org" = {
  documentRoot = "/webroot/wiki.example.org";
  adminAddr = "alice@example.org";
  forceSSL = true;
  enableACME = true;
  enablePHP = true;
};

};

}
```

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
  commonConfig =
    { adminAddr = "alice@example.org";
      forceSSL = true;
      enableACME = true;
    };
in
{
  services.httpd.virtualHosts =
    { "blog.example.org" = (commonConfig // { documentRoot = "/webroot/blog";
      "wiki.example.org" = (commonConfig // { documentRoot = "/webroot/wik
    );
}
}
```

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:

```
{
  services.httpd.virtualHosts =
```

v: unstable -

```
let commonConfig = ...; in
{ "blog.example.org" = (commonConfig // { ... })
  "wiki.example.org" = (commonConfig // { ... })
};
```

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:

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

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.

Modularity

v: unstable -

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, ... }:

{ imports = [ ./vpn.nix ./kde.nix ];
  services.httpd.enable = true;
  environment.systemPackages = [ pkgs.emacs ];
  ...
}
```

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

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

{ services.xserver.enable = true;
  services.xserver.displayManager.sddm.enable = true;
  services.xserver.desktopManager.plasma5.enable = true;
  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` the lists of packages will 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 module. v: unstable -

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, i
```

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

```
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. (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.)

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, ... }:

{ environment.systemPackages =
  if config.services.xserver.enable then
    [ pkgs.firefox
      pkgs.thunderbird
    ]
  else
    [ ];
}
```

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
```

v: unstable -

```
true
```

```
$ 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:

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

nix-repl> config.networking.hostName
"mandark"

nix-repl> map (x: x.hostName) config.services.httpd.virtualHosts
[ "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.

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

let netConfig = hostName: {
    networking.hostName = hostName;
    networking.useDHCP = false;
};

in

{ imports = [ (netConfig "nixos.locaLdomain") ]; }
```

Package Management

Table of Contents

v: unstable -

[Declarative Package Management](#)

[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.

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
```

v: unstable -

```
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, remove it from [`environment.systemPackages`](#) and run `nixos-rebuild switch`.

Customising Packages

Some packages in Nixpkgs have options to enable or disable optional functionality or change other aspects of the package.

Warning

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

Note

For example, many packages come with extensions one might add. Examples include:

- [`passExtensions.pass-otp`](#)
- [`python310Packages.requests`](#)

You can use them like this:

```
environment.systemPackages = with pkgs; [
  sl
  (pass.withExtensions (subpkgs: with subpkgs; [
    pass-audit
    pass-otp
    pass-genphrase
  ]))
  (python3.withPackages (subpkgs: with subpkgs; [
    v: unstable -
```

```
    requests
  ])
  cowsay
];
```

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

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 `oldAttrs`.

v: unstable -

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:  
{ 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.)

Adding Custom Packages

It's possible that a package you need is not available in NixOS. In that case, you can do two things. Either you can package it with Nix, or you can try to use prebuilt packages from upstream. Due to the peculiarities of NixOS, it is important to note that building software from source is often easier than using pre-built executables.

Building with Nix

This can be done either in-tree or out-of-tree. For an in-tree build, 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  
$ cd nixpkgs
```

Then you write and test the package as described in the Nixpkgs manual. Finally, you add v: unstable -

[environment.systemPackages](#), e.g.

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

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

```
# 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 =
let
  my-hello = with pkgs; stdenv.mkDerivation rec {
    name = "hello-2.8";
    src = fetchurl {
      url = "mirror://gnu/hello/${name}.tar.gz";
      hash = "sha256-5rd/gffPfa761Kn1tl3myunD8TuM+66oy107XqVGDXM=";
    };
  };
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

stdenv.mkDerivation rec {
  name = "hello-2.8";
  src = fetchurl {
```

v: unstable -

```
url = "mirror://gnu/hello/${name}.tar.gz";
hash = "sha256-5rd/gffPfa761Kn1tl3myunD8TuM+66oy107XqVGDXM=";
};

}
```

This allows testing the package easily:

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

Using pre-built executables

Most pre-built executables will not work on NixOS. There are two notable exceptions: flatpaks and AppImages. For flatpaks see the [dedicated section](#). AppImages will not run “as-is” on NixOS. First you need to install `appimage-run`: add to `/etc/nixos/configuration.nix`

```
environment.systemPackages = [ pkgs.appimage-run ];
```

Then instead of running the AppImage “as-is”, run `appimage-run foo.appimage`.

To make other pre-built executables work on NixOS, you need to package them with Nix and special helpers like `autoPatchelfHook` or `buildFHSEnv`. See the [Nixpkgs manual](#) for details. This is complex and often doing a source build is easier.

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:

```
$ 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/v: unstable -`

`/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:

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

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

User Management

Table of Contents

[Create users and groups with `systemd-sysusers`](#)

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 = {  
    isNormalUser = true;  
    home = "/home/alice";  
    description = "Alice Foobar";  
    extraGroups = [ "wheel" "networkmanager" ];  
    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`.

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;
```

v: unstable -

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:

```
# 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  
Enter new UNIX password: ***  
Retype new UNIX password: ***
```

A user can be deleted using `userdel`:

```
# 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`.

Create users and groups with `systemd-sysusers`

Note

This is experimental.

Instead of using a custom perl script to create users and groups, you can use `systemd-sysusers`:

```
systemd.sysusers.enable = true;
```

The primary benefit of this is to remove a dependency on perl.

File Systems

Table of Contents

[LUKS-Encrypted File Systems](#)

[SSHFS 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" =
{ device = "/dev/disk/by-label/data";
  fsType = "ext4";
};
```

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.

v: unstable -

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"];`.

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
```

WARNING!
=====

```
This will overwrite data on /dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d.
```

Are you sure? (Type uppercase yes): YES
Enter LUKS passphrase: ***
Verify passphrase: ***

```
# cryptsetup luksOpen /dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d
```

```
Enter passphrase for /dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d:
```

```
# mkfs.ext4 /dev/mapper/crypted
```

The LUKS volume should be automatically picked up by `nixos-generate-config`, but you might want to verify that your `hardware-configuration.nix` looks correct. To manually ensure that the 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";
fileSystems."/".device = "/dev/mapper/crypted";
```

Should grub be used as bootloader, and `/boot` is located on an encrypted partition, it is important to set the `grub.cryptDevice` option in `configuration.nix` to the path of the decrypted `/boot` device.

v: unstable -

add the following grub option:

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

FIDO2

NixOS also supports unlocking your LUKS-Encrypted file system using a FIDO2 compatible token.

Without systemd in initrd

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 @ $HOSTNAME"
# fido2luks credential "$FIDO2_LABEL"
f1d00200108b9d6e849a8b388da457688e3dd653b4e53770012d8f28e5d3b269865038c34f

# fido2luks -i add-key /dev/sda2 f1d00200108b9d6e849a8b388da457688e3dd653b4e53770012d8f28e5d3b269865038c34f
Password:
Password (again):
Old password:
Old password (again):
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;
boot.initrd.luks.devices."/dev/sda2".fido2.credential = "f1d00200108b9d6e849a8b388da457688e3dd653b4e53770012d8f28e5d3b269865038c34f"
```

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
```

v: unstable -

systemd Stage 1

If systemd stage 1 is enabled, it handles unlocking of LUKS-encrypted volumes during boot. The following example enables systemd stage1 and adds support for unlocking the existing LUKS2 volume `root` using any enrolled FIDO2 compatible tokens.

```
boot.initrd = {  
    luks.devices.root = {  
        crypttabExtraOpts = [ "fido2-device=auto" ];  
        device = "/dev/sda2";  
    };  
    systemd.enable = true;  
};
```

All tokens that should be used for unlocking the LUKS2-encrypted volume must first be enrolled using [systemd-cryptenroll](#). In the following example, a new key slot for the first discovered token is added to the LUKS volume.

```
# systemd-cryptenroll --fido2-device=auto /dev/sda2
```

Existing key slots are left intact, unless `--wipe-slot=` is specified. It is recommended to add a recovery key that should be stored in a secure physical location and can be entered wherever a password would be entered.

```
# systemd-cryptenroll --recovery-key /dev/sda2
```

SSHFS File Systems

[SSHFS](#) is a [FUSE](#) filesystem that allows easy access to directories on a remote machine using the SSH File Transfer Protocol (SFTP). It means that if you have SSH access to a machine, no additional setup is needed to mount a directory.

Interactive mounting

v: unstable -

In NixOS, SSHFS is packaged as `sshfs`. Once installed, mounting a directory interactively is simple as running:

```
$ sshfs my-user@example.com:/my-dir /mnt/my-dir
```

Like any other FUSE file system, the directory is unmounted using:

```
$ fusermount -u /mnt/my-dir
```

Non-interactive mounting

Mounting non-interactively requires some precautions because `sshfs` will run at boot and under a different user (root). For obvious reason, you can't input a password, so public key authentication using an unencrypted key is needed. To create a new key without a passphrase you can do:

```
$ ssh-keygen -t ed25519 -P '' -f example-key
Generating public/private ed25519 key pair.
Your identification has been saved in test-key
Your public key has been saved in test-key.pub
The key fingerprint is:
SHA256:yjxl3UbTn31fLWeyLYTAKYJPRmzknjQZoyG8gSNEoIE my-user@workstation
```

To keep the key safe, change the ownership to `root:root` and make sure the permissions are `600`: OpenSSH normally refuses to use the key if it's not well-protected.

The file system can be configured in NixOS via the usual [fileSystems](#) option. Here's a typical setup:

```
{
  fileSystems."/mnt/my-dir" = {
    device = "my-user@example.com:/my-dir/";
    fsType = "sshfs";
    options =
      [ # Filesystem options
        "allow_other"          # for non-root access
        "_netdev"              # this is a network fs
```

v: unstable -

```
"x-systemd.automount" # mount on demand

# SSH options
"reconnect"           # handle connection drops
"ServerAliveInterval=15" # keep connections alive
"IdentityFile=/var/secrets/example-key"
];
};

}
```

More options from `ssh_config(5)` can be given as well, for example you can change the default SSH port or specify a jump proxy:

```
{
  options =
  [
    "ProxyJump=bastion@example.com"
    "Port=22"
  ];
}
```

It's also possible to change the `ssh` command used by SSHFS to connect to the server. For example:

```
{
  options =
  [
    (builtins.replaceStrings [" "] ["\\040"]
      "ssh_command=${pkgs.openssh}/bin/ssh -v -L 8080:localhost:80")
  ];
}
```

Note

The escaping of spaces is needed because every option is written to the `/etc/fstab` file, which is a space-separated table.

Troubleshooting

If you're having a hard time figuring out why mounting is failing, you can add the option "debug". This enables a verbose log in SSHFS that you can access via:

```
$ journalctl -u $(systemd-escape -p /mnt/my-dir/).mount
Jun 22 11:41:18 workstation mount[87790]: SSHFS version 3.7.1
Jun 22 11:41:18 workstation mount[87793]: executing <ssh> <-x> <-a> <-oCle...
Jun 22 11:41:19 workstation mount[87793]: my-user@example.com: Permission ...
Jun 22 11:41:19 workstation mount[87790]: read: Connection reset by peer
Jun 22 11:41:19 workstation systemd[1]: mnt-my\x2ddir.mount: Mount process...
Jun 22 11:41:19 workstation systemd[1]: mnt-my\x2ddir.mount: Failed with ...
Jun 22 11:41:19 workstation systemd[1]: Failed to mount /mnt/my-dir.
Jun 22 11:41:19 workstation systemd[1]: mnt-my\x2ddir.mount: Consumed 54ms
```

Note

If the mount point contains special characters it needs to be escaped using `systemd-escape`. This is due to the way `systemd` converts paths into unit names.

X Window System

Table of Contents

[Auto-login](#)

[Intel Graphics drivers](#)

[Proprietary NVIDIA drivers](#)

[Proprietary AMD drivers](#)

[Touchpads](#)

[GTK/Qt themes](#)

[Custom XKB layouts](#)

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

```
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;
services.xserver.desktopManager.xfce.enable = true;
services.xserver.desktopManager.gnome.enable = true;
services.xserver.desktopManager.mate.enable = true;
services.xserver.windowManager.xmonad.enable = true;
services.xserver.windowManager.twm.enable = true;
services.xserver.windowManager.icewm.enable = true;
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:

```
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.xkb.layout = "de";
```

v: unstable -

```
services.xserver.xkb.variant = "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:

```
# 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:

```
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:

```
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;
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" ];
```

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

```
services.xserver.videoDrivers = [ "intel" ];
services.xserver.deviceSection = ''
  Option "DRI" "2"
  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:

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

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 is not enabled by default because it's not Free Software, is often broken in nixpkgs and as of this writing doesn't offer more features or performance. If you still want to use it anyway, you need to explicitly set:

```
services.xserver.videoDrivers = [ "amdgpu-pro" ];
```

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

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](#)). For instance, the following disables tap-to-click behavior:

v: unstable -

```
services.xserver.libinput.touchpad.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 GTK ones, you can use the following configuration:

```
qt.enable = true;
qt.platformTheme = "gtk2";
qt.style = "gtk2";
```

Custom XKB layouts

It is possible to install custom [XKB](#) keyboard layouts using the option `services.xserver.xkb.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.

Create a file called `us-greek` with the following content (under a directory called `symbols`; it's an XKB peculiarity that will help with testing):

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

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

v: unstable -

```
};
```

A minimal layout specification must include the following:

```
services.xserver.xkb.extraLayouts.us-greek = {  
    description = "US layout with alt-gr greek";  
    languages   = [ "eng" ];  
    symbolsFile = /yourpath/symbols/us-greek;  
};
```

Note

The name (after `extraLayouts.`) should match the one given to the `xkb_symbols` block.

Applying this customization requires rebuilding several packages, and a broken XKB file can lead to the X session crashing at login. Therefore, you're strongly advised to **test your layout before applying it**:

```
$ nix-shell -p xorg.xkbcomp  
$ setxkbmap -I/yourpath us-greek -print | xkbcomp -I/yourpath - $DISPLAY
```

You can inspect the predefined XKB files for examples:

```
$ echo "$(nix-build --no-out-link '<nixpkgs>' -A xorg.xkeyboardconfig)/etc/xkb/layouts/us-greek.us-greek.xkb"
```

Once the configuration is applied, and you did a logout/login cycle, the layout should be ready to use. You can try it by e.g. running `setxkbmap us-greek` and then type `<alt>+a` (it may not get applied in your terminal straight away). To change the default, the usual `services.xserver.xkb.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

v: unstable -

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

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

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

As before, to install the layout do

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

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:

v: unstable -

```
services.xserver.displayManager.sessionCommands = "setxkbmap -keycodes me
```

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

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

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

Wayland

While X11 (see [X Window System](#)) is still the primary display technology on NixOS, Wayland support is steadily improving. Where X11 separates the X Server and the window manager, on Wayland those are combined: a Wayland Compositor is like an X11 window manager, but also embeds the Wayland 'Server' functionality. This means it is sufficient to install a Wayland Compositor such as sway without separately enabling a Wayland server:

```
programs.sway.enable = true;
```

This installs the sway compositor along with some essential utilities. Now you can start sway from the TTY console.

If you are using a wlroots-based compositor, like sway, and want to be able to share your screen, you might want to activate this option:

```
xdg.portal.wlr.enable = true;
```

and configure Pipewire using [`services.pipewire.enable`](#) and related options.

GPU acceleration

Table of Contents

v: unstable -

[OpenCL](#)[Vulkan](#)[VA-API](#)[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.

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 ICL loader for ICD files. For example:

```
$ export \
OCL_ICD_VENDORS=`nix-build '<nixpkgs>' --no-out-link -A rocmPackages.cl`
```

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

package. This command will report the number of hardware devices that is found and give detailed information for each device:

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

AMD

Modern AMD [Graphics Core Next](#) (GCN) GPUs are supported through the `rocmPackages.clr.icd` package. Adding this package to [`hardware.opengl.extraPackages`](#) enables OpenCL support:

```
hardware.opengl.extraPackages = [
  rocmPackages.clr.icd
];
```

Intel

[Intel Gen8 and later GPUs](#) are supported by the Intel NEO OpenCL runtime that is provided by the `intel-compute-runtime` package. The proprietary Intel OpenCL runtime, in the `intel-ocl` package, is an alternative for Gen7 GPUs.

The `intel-compute-runtime` 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 = [
  intel-compute-runtime
];
```

Vulkan

[Vulkan](#) is a graphics and compute API for GPUs. It is used directly by games or indirectly via the `v:unstable` channel.

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 \
VK_ICD_FILenames=`nix-build '<nixpkgs>' --no-out-link -A amdvdk`/share/`
```

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 amdvdk and radv:

```
$ vulkaninfo | grep GPU
    GPU id : 0 (Unknown AMD GPU)
    GPU id : 1 (AMD RADV NAVI10 (LLVM 9.0.1))
    ...
GPU0:
    deviceType      = PHYSICAL_DEVICE_TYPE_DISCRETE_GPU
    deviceName      = Unknown AMD GPU
GPU1:
    deviceType      = PHYSICAL_DEVICE_TYPE_DISCRETE_GPU
```

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

v: unstable -

AMD

Modern AMD [Graphics Core Next \(GCN\)](#) GPUs are supported through either radv, which is part of mesa, or the amdvlk package. Adding the amdvlk package to [hardware.opengl.extraPackages](#) makes amdvlk the default driver and hides radv and lavapipe from the device list. A specific driver can be forced as follows:

```
hardware.opengl.extraPackages = [
  pkgs.amdvlk
];

# To enable Vulkan support for 32-bit applications, also add:
hardware.opengl.extraPackages32 = [
  pkgs.driversi686Linux.amdvlk
];

# Force radv
environment.variables.AMD_VULKAN_ICD = "RADV";
# Or
environment.variables.VK_ICD_FILenames =
  "/run/opengl-driver/share/vulkan/icd.d/radeon_icd.x86_64.json";
```

VA-API

[VA-API \(Video Acceleration API\)](#) is an open-source library and API specification, which provides access to graphics hardware acceleration capabilities for video processing.

VA-API drivers are loaded by `libva`. The version in nixpkgs is built to search the opengl driver path, so drivers can be installed in [hardware.opengl.extraPackages](#).

VA-API can be tested using:

```
$ nix-shell -p libva-utils --run vainfo
```

Intel

v: unstable -

Modern Intel GPUs use the iHD driver, which can be installed with:

```
hardware.opengl.extraPackages = [  
    intel-media-driver  
];
```

Older Intel GPUs use the i965 driver, which can be installed with:

```
hardware.opengl.extraPackages = [  
    intel-vaapi-driver  
];
```

Common issues

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 `alice` is logged in, the access control list should look as follows:

```
$ getfacl /dev/dri/card0  
# file: dev/dri/card0  
# owner: root  
# group: video  
user::rw-  
user:alice:rw-  
group::rw-  
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.

v: unstable -

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 ICL 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 clinfo`, which should report missing symbols.

Xfce Desktop Environment

Table of Contents

[Thunar](#)

[Troubleshooting](#)

To enable the Xfce Desktop Environment, set

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

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

```
services.picom = {  
    enable = true;  
    fade = true;  
    inactiveOpacity = 0.9;  
    shadow = true;  
    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

Thunar (the Xfce file manager) is automatically enabled when Xfce is enabled. To enable Thunar without enabling Xfce, use the configuration option [programs.thunar.enable](#) instead of adding `pkgs.xfce.thunar` to [environment.systemPackages](#).

If you'd like to add extra plugins to Thunar, add them to [programs.thunar.plugins](#). 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
```

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.

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

v: unstable -

It is necessary to log out and log in again for this to take effect.

Networking

Table of Contents

[NetworkManager](#)

[Secure Shell Access](#)

[IPv4 Configuration](#)

[IPv6 Configuration](#)

[Firewall](#)

[Wireless Networks](#)

[Ad-Hoc Configuration](#)

[Renaming network interfaces](#)

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

NetworkManager

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

```
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:

v: unstable -

```
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 = [  
    "*" "except:type:wwan" "except:type:gsm"  
];
```

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

Secure Shell Access

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

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

By default, root logins using a password are disallowed. They can be disabled entirely by setting [`services.openssh.settings.PermitRootLogin`](#) to "no".

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

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

v: unstable -

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 = [ {  
    address = "192.168.1.2";  
    prefixLength = 24;  
} ];
```

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

```
networking.defaultGateway = "192.168.1.1";  
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.

IPv6 Configuration

IPv6 is enabled by default. Stateless address autoconfiguration is used to automatically assign IPv6 addresses to all interfaces, and Privacy Extensions (RFC 4946) are enabled by default. You can adjust the default for this by setting `networking.tempAddresses`. This option may be overridden on a per-interface basis by `networking.interfaces.<name>.tempAddress`. You can disable this feature by setting `networking.disableStatelessAddressAutoConfiguration` to `true`.

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:

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

For configuring a gateway, optionally with explicitly specified interface:

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

See [the section called “IPv4 Configuration”](#) for similar examples and additional information.

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:

v: unstable -

```
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 = [
  { from = 4000; to = 4007; }
  { from = 8000; to = 8010; }
];
```

Similarly, UDP port ranges can be opened through

[`networking.firewall.allowedUDPPortRanges`](#).

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:

```
networking.wireless.networks = {
  echelon = {                                     # SSID with no spaces or special characters
    psk = "abcdefgh";
  };
  "echelon's AP" = {                            # SSID with spaces and/or special characters
    psk = "ijklmnop";
  };
  echelon = {                                     # Hidden SSID
};
```

v: unstable -

```
    hidden = true;
    psk = "qrstuvwx";
};

free.wifi = {};
# Public wireless network
};
```

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
network={
    ssid="echelon"
    #psk="abcdefgh"
    psk=dca6d6ed41f4ab5a984c9f55f6f66d4efdc720ebf66959810f4329bb391c54
}
```

```
networking.wireless.networks = {
    echelon = {
        pskRaw = "dca6d6ed41f4ab5a984c9f55f6f66d4efdc720ebf66959810f4329bb391c54
    };
};
```

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
```

Ad-Hoc Configuration

v: unstable -

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 =  
  ''  
    ip -6 addr add 2001:610:685:1::1/64 dev eth0  
  '';
```

Renaming network interfaces

NixOS uses the udev [predictable naming scheme](#) to assign names to network interfaces. This means that by default cards are not given the traditional names like `eth0` or `eth1`, whose order can change unpredictably across reboots. Instead, relying on physical locations and firmware information, the scheme produces names like `ens1`, `enp2s0`, etc.

These names are predictable but less memorable and not necessarily stable: for example installing new hardware or changing firmware settings can result in a [name change](#). If this is undesirable, for example if you have a single ethernet card, you can revert to the traditional scheme by setting [`networking.usePredictableInterfaceNames`](#) to `false`.

Assigning custom names

In case there are multiple interfaces of the same type, it's better to assign custom names based on the device hardware address. For example, we assign the name `wan` to the interface with MAC address `52:54:00:12:01:01` using a netword link unit:

```
systemd.network.links."10-wan" = {  
  matchConfig.PermanentMACAddress = "52:54:00:12:01:01";  
  linkConfig.Name = "wan";  
};
```

Note that links are directly read by udev, *not* `networkd`, and will work even if `networkd` is disabled.

Alternatively, we can use a plain old udev rule:

v: unstable -

```
boot.initrd.services.udev.rules = ''  
  SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", \  
  ATTR{address}=="52:54:00:12:01:01", KERNEL=="eth*", NAME="wan"  
'';
```

Warning

The rule must be installed in the initrd using `boot.initrd.services.udev.rules`, not the usual `services.udev.extraRules` option. This is to avoid race conditions with other programs controlling the interface.

Linux Kernel

Table of Contents

[Building a custom kernel](#)

[Rust](#)

[Developing kernel modules](#)

[ZFS](#)

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.linuxKernel.packages.linux_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.

While `pkgs.linuxKernel.packages` contains all available kernel packages, you may want to use one of the unversioned `pkgs.linuxPackages_*` aliases such as `pkgs.linuxPackages_latest`, that are kept up to date with new versions.

v: unstable -

Please note that the current convention in NixOS is to only keep actively maintained kernel versions on both unstable and the currently supported stable release(s) of NixOS. This means that a non-longterm kernel will be removed after it's abandoned by the kernel developers, even on stable NixOS versions. If you pin your kernel onto a non-longterm version, expect your evaluation to fail as soon as the version is out of maintenance.

Longterm versions of kernels will be removed before the next stable NixOS that will exceed the maintenance period of the kernel version.

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

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

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

```
nixpkgs.config.packageOverrides = pkgs: pkgs.lib.recursiveUpdate pkgs {  
    linuxKernel.kernels.linux_5_10 = pkgs.linuxKernel.kernels.linux_5_10.Overrides;  
    extraConfig = ''  
        KGDB y  
    '';  
};  
};
```

`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.kernelModules` in `configuration.nix`:

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`.

Building a custom kernel

Please refer to the Nixpkgs manual for the various ways of [building a custom kernel](#).

To use your custom kernel package in your NixOS configuration, set

```
boot.kernelPackages = pkgs.linuxPackagesFor yourCustomKernel;
```

Rust

The Linux kernel does not have Rust language support enabled by default. For kernel versions 6.7 or newer, experimental Rust support can be enabled. In a NixOS configuration, set:

```
boot.kernelPatches = [
  {
    name = "Rust Support";
    patch = null;
    features = {
      rust = true;
    };
  }
];
```

v: unstable -

Developing kernel modules

This section was moved to the [Nixpkgs manual](#).

ZFS

It's a common issue that the latest stable version of ZFS doesn't support the latest available Linux kernel. It is recommended to use the latest available LTS that's compatible with ZFS. Usually this is the default kernel provided by nixpkgs (i.e. `pkgs.linuxPackages`).

Alternatively, it's possible to pin the system to the latest available kernel version *that is supported by ZFS* like this:

```
{  
  boot.kernelPackages = pkgs.zfs.latestCompatibleLinuxPackages;  
}
```

Please note that the version this attribute points to isn't monotonic because the latest kernel version only refers to kernel versions supported by the Linux developers. In other words, the latest kernel version that ZFS is compatible with may decrease over time.

An example: the latest version ZFS is compatible with is 5.19 which is a non-longterm version. When 5.19 is out of maintenance, the latest supported kernel version is 5.15 because it's longterm and the versions 5.16, 5.17 and 5.18 are already out of maintenance because they're non-longterm.

Subversion

Table of Contents

[Subversion inside Apache HTTP](#)

[Subversion](#) is a centralized version-control system. It can use a [variety of protocols](#) for communication between client and server.

Subversion inside Apache HTTP

This section focuses on configuring a web-based server on top of the Apache HTTP server, which uses [WebDAV/DeltaV](#) for communication.

For more information on the general setup, please refer to the [the appropriate section of the Subversion book](#).

To configure, include in `/etc/nixos/configuration.nix` code to activate Apache HTTP, setting [`services.httpd.adminAddr`](#) appropriately:

```
services.httpd.enable = true;
services.httpd.adminAddr = ...;
networking.firewall.allowedTCPPorts = [ 80 443 ];
```

For a simple Subversion server with basic authentication, configure the Subversion module for Apache as follows, setting `hostName` and `documentRoot` appropriately, and `SVNParentPath` to the parent directory of the repositories, `AuthzSVNAccessFile` to the location of the `.authz` file describing access permission, and `AuthUserFile` to the password file.

```
services.httpd.extraModules = [
    # note that order is *super* important here
    { name = "dav_svn"; path = "${pkgs.apacheHttpdPackages.subversion}/modules/mod_dav_svn.so" },
    { name = "authz_svn"; path = "${pkgs.apacheHttpdPackages.subversion}/modules/mod_authz_svn.so" },
];
services.httpd.virtualHosts = {
    "svn" = {
        hostName = HOSTNAME;
        documentRoot = DOCUMENTROOT;
        locations."/svn".extraConfig = ''
            DAV svn
            SVNParentPath REPO_PARENT
            AuthzSVNAccessFile ACCESS_FILE
            AuthName "SVN Repositories"
            AuthType Basic
            AuthUserFile PASSWORD_FILE
    }
};
```

v: unstable -

```
        Require valid-user  
        '';  
    }
```

The key "svn" is just a symbolic name identifying the virtual host. The "/svn" in `locations."/svn".extraConfig` is the path underneath which the repositories will be served.

[This page](#) explains how to set up the Subversion configuration itself. This boils down to the following:

Underneath REPO_PARENT repositories can be set up as follows:

```
$ svn create REPO_NAME
```

Repository files need to be accessible by wwwrun:

```
$ chown -R wwwrun:wwwrun REPO_PARENT
```

The password file PASSWORD_FILE can be created as follows:

```
$ htpasswd -cs PASSWORD_FILE USER_NAME
```

Additional users can be set up similarly, omitting the c flag:

```
$ htpasswd -s PASSWORD_FILE USER_NAME
```

The file describing access permissions ACCESS_FILE will look something like the following:

```
[/]  
* = r  
  
[REPO_NAME:/]  
USER_NAME = rw
```

The Subversion repositories will be accessible as `http://HOSTNAME/svn/REPO_NAME` v:unstable -

Pantheon Desktop

Table of Contents

[Enabling Pantheon](#)

[Wingpanel and Switchboard plugins](#)

[FAQ](#)

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

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

```
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 `elementary-mail`).

v: unstable -

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 {  
    indicators = [  
        pkgs.some-special-indicator  
    ];  
};
```

```
switchboard-with-plugs.override {  
    plugs = [  
        pkgs.some-special-plug  
    ];  
};
```

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:

```
wingpanel-with-indicators.override {  
    useDefaultIndicators = false;  
    indicators = specialListOfIndicators;  
};
```

v: unstable -

```
switchboard-with-plugs.override {  
    useDefaultPlugs = false;  
    plugs = specialListOfPlugs;  
};
```

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

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 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. Starting from 21.11, the Flatpak backend should work so you can install some Flatpak applications using it. However, due to missing appstream metadata, the Packagekit backend does not function currently. See this [issue](#).

If you are using Pantheon, AppCenter should be installed by default if you have [Flatpak support](#) enabled. If you also wish to add the `appcenter` Flatpak remote:

```
$ flatpak remote-add --if-not-exists appcenter https://flatpak.elemen...
```

GNOME Desktop

Table of Contents

[Enabling GNOME](#)

[Enabling GNOME Flashback](#)

[Icons and GTK Themes](#)

v: unstable -

[Shell Extensions](#)[GSettings Overrides](#)[Frequently Asked Questions](#)

GNOME provides a simple, yet full-featured desktop environment with a focus on productivity. Its Mutter compositor supports both Wayland and X server, and the GNOME Shell user interface is fully customizable by extensions.

Enabling GNOME

All of the core apps, optional apps, games, and core developer tools from GNOME are available.

To enable the GNOME desktop use:

```
services.xserver.desktopManager.gnome.enable = true;  
services.xserver.displayManager.gdm.enable = true;
```

Note

While it is not strictly necessary to use GDM as the display manager with GNOME, it is recommended, as some features such as screen lock might not work without it.

The default applications used in NixOS are very minimal, inspired by the defaults used in [gnome-build-meta](#).

GNOME without the apps

If you'd like to only use the GNOME desktop and not the apps, you can disable them with:

```
services.gnome.core-utilities.enable = false;
```

and none of them will be installed.

v: unstable -

If you'd only like to omit a subset of the core utilities, you can use [environment.gnome.excludePackages](#). Note that this mechanism can only exclude core utilities, games and core developer tools.

Disabling GNOME services

It is also possible to disable many of the [core services](#). For example, if you do not need indexing files, you can disable Tracker with:

```
services.gnome.tracker-miners.enable = false;  
services.gnome.tracker.enable = false;
```

Note, however, that doing so is not supported and might break some applications. Notably, GNOME Music cannot work without Tracker.

GNOME games

You can install all of the GNOME games with:

```
services.gnome.games.enable = true;
```

GNOME core developer tools

You can install GNOME core developer tools with:

```
services.gnome.core-developer-tools.enable = true;
```

Enabling GNOME Flashback

GNOME Flashback provides a desktop environment based on the classic GNOME 2 architecture. You can enable the default GNOME Flashback session, which uses the Metacity window manager, with:

```
services.xserver.desktopManager.gnome.flashback.enableMetacity
```

v: unstable -

It is also possible to create custom sessions that replace Metacity with a different window manager using `services.xserver.desktopManager.gnome.flashback.customSessions`.

The following example uses `xmonad` window manager:

```
services.xserver.desktopManager.gnome.flashback.customSessions = [
{
  wmName = "xmonad";
  wmLabel = "XMonad";
  wmCommand = "${pkgs.haskellPackages.xmonad}/bin/xmonad";
  enableGnomePanel = false;
}
];
```

Icons and GTK Themes

Icon themes and GTK themes don't require any special option to install in NixOS.

You can add them to `environment.systemPackages` and switch to them with GNOME Tweaks. If you'd like to do this manually in dconf, change the values of the following keys:

```
/org/gnome/desktop/interface/gtk-theme  
/org/gnome/desktop/interface/icon-theme
```

in `dconf-editor`

Shell Extensions

Most Shell extensions are packaged under the `gnomeExtensions` attribute. Some packages that include Shell extensions, like `gnome.gpaste`, don't have their extension decoupled under this attribute.

You can install them like any other package:

```
environment.systemPackages = [
  gnomeExtensions.dash-to-dock
  gnomeExtensions.gsconnect
```

v: unstable -

```
gnomeExtensions.mpris-indicator-button  
];
```

Unfortunately, we lack a way for these to be managed in a completely declarative way. So you have to enable them manually with an Extensions application. It is possible to use a [GSettings override](#) for this on `org.gnome.shell.enabled-extensions`, but that will only influence the default value.

GSettings Overrides

Majority of software building on the GNOME platform use GLib's [GSettings](#) system to manage runtime configuration. For our purposes, the system consists of XML schemas describing the individual configuration options, stored in the package, and a settings backend, where the values of the settings are stored. On NixOS, like on most Linux distributions, dconf database is used as the backend.

[GSettings vendor overrides](#) can be used to adjust the default values for settings of the GNOME desktop and apps by replacing the default values specified in the XML schemas. Using overrides will allow you to pre-seed user settings before you even start the session.

Warning

Overrides really only change the default values for GSettings keys so if you or an application changes the setting value, the value set by the override will be ignored. Until [NixOS's dconf module implements changing values](#), you will either need to keep that in mind and clear the setting from the backend using `dconf reset` command when that happens, or use the module from `home-manager`.

You can override the default GSettings values using the [`services.xserver.desktopManager.gnome.extraGSettingsOverrides`](#) option.

Take note that whatever packages you want to override GSettings for, you need to add them to [`services.xserver.desktopManager.gnome.extraGSettingsOverridePackages`](#).

You can use `dconf-editor` tool to explore which GSettings you can set.

Example

v: unstable -

```
services.xserver.desktopManager.gnome = {
  extraGSettingsOverrides = ''
    # Change default background
    [org.gnome.desktop.background]
    picture-uri='file://${pkgs.nixos-artwork.wallpapers.mosaic-blue.gnome}'

    # Favorite apps in gnome-shell
    [org.gnome.shell]
    favorite-apps=['org.gnome.Console.desktop', 'org.gnome.Nautilus.desktop'];
};

extraGSettingsOverridePackages = [
  pkgs.gsettings-desktop-schemas # for org.gnome.desktop
  pkgs.gnome.gnome-shell # for org.gnome.shell
];
};
```

Frequently Asked Questions

Can I use LightDM with GNOME?

Yes you can, and any other display-manager in NixOS.

However, it doesn't work correctly for the Wayland session of GNOME Shell yet, and won't be able to lock your screen.

See [this issue](#).

External Bootloader Backends

Table of Contents

[Developing Custom Bootloader Backends](#)

NixOS has support for several bootloader backends by default: systemd-boot, grub, uboot, etc. The built-in bootloader backend support is generic and supports most use cases. Some users may prefer to create advanced workflows around managing the bootloader and bootable entries.

You can replace the built-in bootloader support with your own tooling using the “external” bootloader option.

Imagine you have created a new package called FooBoot. FooBoot provides a program at `pkgs.fooboot/bin/fooboot-install` which takes the system closure’s path as its only argument and configures the system’s bootloader.

You can enable FooBoot like this:

```
{ pkgs, ... }: {  
    boot.loader.external = {  
        enable = true;  
        installHook = "${pkgs.fooboot}/bin/fooboot-install";  
    };  
}
```

Developing Custom Bootloader Backends

Bootloaders should use [RFC-0125](#)’s Bootspec format and synthesis tools to identify the key properties for bootable system generations.

Clevis

Table of Contents

[Create a JWE file containing your secret](#)

[Activate unattended decryption of a resource at boot](#)

[Clevis](#) is a framework for automated decryption of resources. Clevis allows for secure unattended disk decryption during boot, using decryption policies that must be satisfied for the data to de v: unstable -

Create a JWE file containing your secret

The first step is to embed your secret in a [JWE](#) file. JWE files have to be created through the clevis command line. 3 types of policies are supported:

1. TPM policies

Secrets are pinned against the presence of a TPM2 device, for example:

```
echo -n hi | clevis encrypt tpm2 '{ }' > hi.jwe
```

2. Tang policies

Secrets are pinned against the presence of a Tang server, for example:

```
echo -n hi | clevis encrypt tang '{"url": "http://tang.local"}' > hi.jwe
```

3. Shamir Secret Sharing

Using Shamir's Secret Sharing ([sss](#)), secrets are pinned using a combination of the two preceding policies. For example:

```
echo -n hi | clevis encrypt sss \  
'{"t": 2, "pins": {"tpm2": {"pcr_ids": "0"}, "tang": {"url": "http://tang.local"}}, "secret": "mysecret"}' > hi.jwe
```

For more complete documentation on how to generate a secret with clevis, see the [clevis documentation](#).

Activate unattended decryption of a resource at boot

In order to activate unattended decryption of a resource at boot, enable the `clevis` module:

```
boot.initrd.clevis.enable = true;
```

Then, specify the device you want to decrypt using a given clevis secret. Clevis will automatically try to decrypt the device at boot and will fallback to interactive unlocking if the decryption policy is not fulfilled.

```
boot.initrd.clevis.devices."/dev/nvme0n1p1".secretFile = ./nvme0n1p1.jwe;
```

Only `bcachefs`, `zfs` and `luks` encrypted devices are supported at this time.

Garage

Table of Contents

[General considerations on upgrades](#)

[Advanced upgrades \(minor/major version upgrades\)](#)

[Maintainer information](#)

[Garage](#) is an open-source, self-hostable S3 store, simpler than MinIO, for geodistributed stores. The server setup can be automated using [services.garage](#). A client configured to your local Garage instance is available in the global environment as `garage-manage`.

The current default by NixOS is `garage_0_8` which is also the latest major version available.

General considerations on upgrades

Garage provides a cookbook documentation on how to upgrade: <https://garagehq.deuxfleurs.fr/documentation/cookbook/upgrading/>

Warning

Garage has two types of upgrades: patch-level upgrades and minor/major version upgrades.

In all cases, you should read the changelog and ideally test the upgrade on a staging cluster.

Checking the health of your cluster can be achieved using `garage-manage repair`.

Warning

Until 1.0 is released, patch-level upgrades are considered as minor version upgrades. Minor version upgrades are considered as major version upgrades. i.e. 0.6 to 0.7 is a major version upgrade.

- **Straightforward upgrades (patch-level upgrades).** Upgrades must be performed one by one, i.e. for each node, stop it, upgrade it : change `stateVersion` or `services.garage.package`, restart it if it was not already by switching.
- **Multiple version upgrades.** Garage do not provide any guarantee on moving more than one major-version forward. E.g., if you're on `0.7`, you cannot upgrade to `0.9`. You need to upgrade to `0.8` first. As long as `stateVersion` is declared properly, this is enforced automatically. The module will issue a warning to remind the user to upgrade to latest Garage after that deploy.

Advanced upgrades (minor/major version upgrades)

Here are some baseline instructions to handle advanced upgrades in Garage, when in doubt, please refer to upstream instructions.

- Disable API and web access to Garage.
- Perform `garage-manage repair --all-nodes --yes tables` and `garage-manage repair --all-nodes --yes blocks`.
- Verify the resulting logs and check that data is synced properly between all nodes. If you have time, do additional checks (`scrub`, `block_refs`, etc.).
- Check if queues are empty by `garage-manage stats` or through monitoring tools.

- Run `systemctl stop garage` to stop the actual Garage version.
- Backup the metadata folder of ALL your nodes, e.g. for a metadata directory (the default one) in `/var/lib/garage/meta`, you can run `pushd /var/lib/garage; tar -acf meta-v0.7.tar.zst meta/; popd`.
- Run the offline migration: `nix-shell -p garage_0_8 --run "garage offline-repair --yes"`, this can take some time depending on how many objects are stored in your cluster.
- Bump Garage version in your NixOS configuration, either by changing [stateVersion](#) or bumping [services.garage.package](#), this should restart Garage automatically.
- Perform `garage-manage repair --all-nodes --yes tables` and `garage-manage repair --all-nodes --yes blocks`.
- Wait for a full table sync to run.

Your upgraded cluster should be in a working state, re-enable API and web access.

Maintainer information

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

While 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. Garage `v0.8.0` should be available in `nixpkgs` as `pkgs.garage_0_8_0`). 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 `garage`-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/tools/filesystem/garage/default.nix>`):

v: unstable -

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 Garage on e.g. 22.11 to a Garage on 23.11.

Suwayomi-Server

Table of Contents

Basic usage

Basic authentication

Extra configuration

A free and open source manga reader server that runs extensions built for Tachiyomi.

Basic usage

By default, the module will execute Suwayomi-Server backend and web UI:

```
{ ... }:  
  
{  
    services.suwayomi-server = {  
        enable = true;  
    };  
};  
v: unstable -
```

}

It runs in the systemd service named `suwayomi-server` in the data directory `/var/lib/suwayomi-server`.

You can change the default parameters with some other parameters:

```
{ ... }:

{

  services.suwayomi-server = {
    enable = true;

    dataDir = "/var/lib/suwayomi"; # Default is "/var/lib/suwayomi-server"
    openFirewall = true;

    settings = {
      server.port = 4567;
    };
  };
}
```

If you want to create a desktop icon, you can activate the system tray option:

```
{ ... }:

{

  services.suwayomi-server = {
    enable = true;

    dataDir = "/var/lib/suwayomi"; # Default is "/var/lib/suwayomi-server"
    openFirewall = true;

    settings = {
      server.port = 4567;
      server.enableSystemTray = true;
    };
}
```

v: unstable -

```
    };
};

}
```

Basic authentication

You can configure a basic authentication to the web interface with:

```
{ ... }:

{
  services.suwayomi-server = {
    enable = true;

    openFirewall = true;

    settings = {
      server.port = 4567;
      server = {
        basicAuthEnabled = true;
        basicAuthUsername = "username";

        # NOTE: this is not a real upstream option
        basicAuthPasswordFile = ./path/to/the/password/file;
      };
    };
  };
}
```

Extra configuration

Not all the configuration options are available directly in this module, but you can add the other options of suwayomi-server with:

```
{ ... }:
```

v: unstable -

```
{  
  services.suwayomi-server = {  
    enable = true;  
  
    openFirewall = true;  
  
    settings = {  
      server = {  
        port = 4567;  
        autoDownloadNewChapters = false;  
        maxSourcesInParallel" = 6;  
      };  
    };  
  };  
}
```

Plausible

Table of Contents

Basic Usage

Plausible is a privacy-friendly alternative to Google analytics.

Basic Usage

At first, a secret key is needed to be generated. This can be done with e.g.

```
$ openssl rand -base64 64
```

After that, `plausible` can be deployed like this:

```
{
```

v: unstable -

```
services.plausible = {
    enable = true;
    adminUser = {
        # activate is used to skip the email verification of the admin-user
        # automatically created by plausible. This is only supported if
        # postgresql is configured by the module. This is done by default, it
        # can be turned off with services.plausible.database.postgres.setup
        activate = true;
        email = "admin@localhost";
        passwordFile = "/run/secrets/plausible-admin-pwd";
    };
    server = {
        baseUrl = "http://analytics.example.org";
        # secretKeybaseFile is a path to the file which contains the secret
        # with openssl as described above.
        secretKeybaseFile = "/run/secrets/plausible-secret-key-base";
    };
};
```

Pict-rs

Table of Contents

[Quickstart](#)

[Usage](#)

[Missing](#)

pict-rs is a simple image hosting service.

Quickstart

the minimum to start pict-rs is

v: unstable -

```
services.pict-rs.enable = true;
```

this will start the http server on port 8080 by default.

Usage

pict-rs offers the following endpoints:

- **POST /image** for uploading an image. Uploaded content must be valid multipart/form-data with an `image` array located within the `images[]` key

This endpoint returns the following JSON structure on success with a 201 Created status

```
{
  "files": [
    {
      "delete_token": "JFvFhqJA98",
      "file": "lkWZDRVugm.jpg"
    },
    {
      "delete_token": "kAYy9nk2WK",
      "file": "8qFS0QooAn.jpg"
    },
    {
      "delete_token": "0xRpM3sf0Y",
      "file": "1hJaYfGE01.jpg"
    }
  ],
  "msg": "ok"
}
```

- **GET /image/download?url=...** Download an image from a remote server, returning the same JSON payload as the POST endpoint
- **GET /image/original/{file}** for getting a full-resolution image. `file` here is the `file` key from the `/image` endpoint's JSON

v: unstable -

- GET /image/details/original/{file} for getting the details of a full-resolution image. The returned JSON is structured like so:

```
{  
    "width": 800,  
    "height": 537,  
    "content_type": "image/webp",  
    "created_at": [  
        2020,  
        345,  
        67376,  
        394363487  
    ]  
}
```

- GET /image/process.{ext}?src={file}&... get a file with transformations applied. existing transformations include
 - **identity=true**: apply no changes
 - **blur={float}**: apply a gaussian blur to the file
 - **thumbnail={int}**: produce a thumbnail of the image fitting inside an {int} by {int} square using raw pixel sampling
 - **resize={int}**: produce a thumbnail of the image fitting inside an {int} by {int} square using a Lanczos2 filter. This is slower than sampling but looks a bit better in some cases
 - **crop={int-w}x{int-h}**: produce a cropped version of the image with an {int-w} by {int-h} aspect ratio. The resulting crop will be centered on the image. Either the width or height of the image will remain full-size, depending on the image's aspect ratio and the requested aspect ratio. For example, a 1600x900 image cropped with a 1x1 aspect ratio will become 900x900. A 1600x1100 image cropped with a 16x9 aspect ratio will become 1600x900.

Supported **ext** file extensions include **png**, **jpg**, and **webp**

An example of usage could be

v: unstable -

```
GET /image/process.jpg?src=asdf.png&thumbnail=256&blur=3.0
```

which would create a 256x256px JPEG thumbnail and blur it

- `GET /image/details/process.{ext}?src={file}&...` for getting the details of a processed image. The returned JSON is the same format as listed for the full-resolution details endpoint.
- `DELETE /image/delete/{delete_token}/{file}` or `GET /image/delete/{delete_token}/{file}` to delete a file, where `delete_token` and `file` are from the `/image` endpoint's JSON

Missing

- Configuring the secure-api-key is not included yet. The envisioned basic use case is consumption on localhost by other services without exposing the service to the internet.

Nextcloud

Table of Contents

[Basic usage](#)

[Common problems](#)

[Using an alternative webserver as reverse-proxy \(e.g. httpd\)](#)

[Installing Apps and PHP extensions](#)

[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 `nextcloud28` which is also the latest major version available
v: unstable -

Basic usage

Nextcloud is a PHP-based application which requires an HTTP server ([`services.nextcloud`](#) and optionally supports [`services.nginx`](#)).

For the database, you can set [`services.nextcloud.config.dbtype`](#) to either `sqlite` (the default), `mysql`, or `pgsql`. The simplest is `sqlite`, which will be automatically created and managed by the application. For the last two, you can easily create a local database by setting [`services.nextcloud.database.createLocally`](#) to `true`, Nextcloud will automatically be configured to connect to it through socket.

A very basic configuration may look like this:

```
{ pkgs, ... }:
{
  services.nextcloud = {
    enable = true;
    hostName = "nextcloud.tld";
    database.createLocally = true;
    config = {
      dbtype = "pgsql";
      adminpassFile = "/path/to/admin-pass-file";
    };
  };

  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.settings.trusted_domains`](#).

Auto updates for Nextcloud apps can be enabled using [`services.nextcloud.autoUpdateApps`](#).

v: unstable -

Common problems

- **General notes.** 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!

- **Multiple version upgrades.** Nextcloud doesn't allow to move more than one major-version forward.

E.g., if you're 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.

- **Error: Command "upgrade" is not defined.** This error usually occurs if the initial installation (`nextcloud-occ maintenance:install`) has failed. After that, the application is not installed, but the upgrade is attempted to be executed. Further context can be found in [NixOS/nixpkgs#111175](#).

First of all, it makes sense to find out what went wrong by looking at the logs of the installation via `journalctl -u nextcloud-setup` and try to fix the underlying issue.

v: unstable -

- If this occurs on an *existing* setup, this is most likely because the maintenance mode is active. It can be deactivated by running **nextcloud-occ maintenance:mode --off**. It's advisable though to check the logs first on why the maintenance mode was activated.

- **Warning**

Only perform the following measures on *freshly installed instances!*

A re-run of the installer can be forced by *deleting* `/var/lib/nextcloud/config/config.php`. This is the only time advisable because the fresh install doesn't have any state that can be lost. In case that doesn't help, an entire re-creation can be forced via **rm -rf ~nextcloud/**.

- **Server-side encryption.** Nextcloud supports [server-side encryption \(SSE\)](#). This is not an end-to-end encryption, but can be used to encrypt files that will be persisted to external storage such as S3.

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 php-fpm pool`](#).

An exemplary configuration may look like this:

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

    /* further, required options */
  };
  services.phpfpm.pools.nextcloud.settings = {
    "listen.owner" = config.services.httpd.user;
    "listen.group" = config.services.httpd.group;
  };
}
```

v: unstable -

```
services.httpd = {
  enable = true;
  adminAddr = "webmaster@localhost";
  extraModules = [ "proxy_fcgi" ];
  virtualHosts."localhost" = {
    documentRoot = config.services.nextcloud.package;
    extraConfig = ''
      <Directory "${config.services.nextcloud.package}">
        <FilesMatch "\.php$">
          <If "-f %{REQUEST_FILENAME}">
            SetHandler "proxy:unix:${config.services.phpfpm.pools.nextc"
          </If>
        </FilesMatch>
        <IfModule mod_rewrite.c>
          RewriteEngine On
          RewriteBase /
          RewriteRule ^index\.php$ - [L]
          RewriteCond %{REQUEST_FILENAME} !-f
          RewriteCond %{REQUEST_FILENAME} !-d
          RewriteRule . /index.php [L]
        </IfModule>
        DirectoryIndex index.php
        Require all granted
        Options +FollowSymLinks
      </Directory>
    '';
  };
};
```

Installing Apps and PHP extensions

Nextcloud apps are installed statefully through the web interface. Some apps may require extra PHP extensions to be installed. This can be configured with the [services.nextcloud.phpExtraExtensions](#) setting.

Alternatively, extra apps can also be declared with the [services.nextcloud.extraA](#) v: unstable -

When using this setting, apps can no longer be managed statefully because this can lead to Nextcloud updating apps that are managed by Nix. If you want automatic updates it is recommended that you use web interface to install apps.

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>`):

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

Matomo

Table of Contents

[Database Setup](#)

[Archive Processing](#)

[Backup](#)

[Issues](#)

[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.

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:

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

```
# For MySQL
INSTALL PLUGIN auth_socket SONAME 'auth_socket.so';
CREATE DATABASE matomo;
CREATE USER 'matomo'@'localhost' IDENTIFIED WITH auth_socket;
```

v: unstable -

```
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.

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.

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/.

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.

Using other Web Servers than nginx

v: unstable -

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.

Lemmy

Table of Contents

[Quickstart](#)

[Usage](#)

[Missing](#)

Lemmy is a federated alternative to reddit in rust.

Quickstart

the minimum to start lemmy is

```
services.lemmy = {  
    enable = true;  
    settings = {  
        hostname = "lemmy.union.rocks";  
        database.createLocally = true;  
    };  
    caddy.enable = true;  
}
```

this will start the backend on port 8536 and the frontend on port 1234. It will expose your instance with a caddy reverse proxy to the hostname you've provided. Postgres will be initialized on that same instance automatically.

Usage

v: unstable -

On first connection you will be asked to define an admin user.

Missing

- Exposing with nginx is not implemented yet.
- This has been tested using a local database with a unix socket connection. Using different database settings will likely require modifications

Keycloak

Table of Contents

[Administration](#)

[Database access](#)

[Hostname](#)

[Setting up TLS/SSL](#)

[Themes](#)

[Configuration file settings](#)

[Example configuration](#)

[Keycloak](#) is an open source identity and access management server with support for [OpenID Connect](#), [OAuth 2.0](#) and [SAML 2.0](#).

Administration

An administrative user with the username `admin` is automatically created in the `master` realm. Its initial password can be configured by setting [`services.keycloak.initialAdminPassword`](#) and defaults to `changeme`. The password is not stored safely and should be changed immediately in the admin panel.

v: unstable -

Refer to the [Keycloak Server Administration Guide](#) for information on how to administer your Keycloak instance.

Database access

Keycloak can be used with either PostgreSQL, MariaDB or MySQL. Which one is used can be configured in `services.keycloak.database.type`. The selected database will automatically be enabled and a database and role created unless `services.keycloak.database.host` is changed from its default of `localhost` or `services.keycloak.database.createLocally` is set to `false`.

External database access can also be configured by setting `services.keycloak.database.host`, `services.keycloak.database.name`, `services.keycloak.database.username`, `services.keycloak.database.useSSL` and `services.keycloak.database.caCert` as appropriate. Note that you need to manually create the database and allow the configured database user full access to it.

`services.keycloak.database.passwordFile` must be set to the path to a file containing the password used to log in to the database. If `services.keycloak.database.host` and `services.keycloak.database.createLocally` are kept at their defaults, the database role `keycloak` with that password is provisioned on the local database instance.

Warning

The path should be provided as a string, not a Nix path, since Nix paths are copied into the world readable Nix store.

Hostname

The hostname is used to build the public URL used as base for all frontend requests and must be configured through `services.keycloak.settings.hostname`.

Note

If you're migrating an old Wildfly based Keycloak instance and want to keep compatibility with your current clients, you'll likely want to set [`services.keycloak.settings.http-relative-path`](#) to `/auth`. See the option description for more details.

[`services.keycloak.settings.hostname-strict-backchannel`](#) determines whether Keycloak should force all requests to go through the frontend URL. By default, Keycloak allows backend requests to instead use its local hostname or IP address and may also advertise it to clients through its OpenID Connect Discovery endpoint.

For more information on hostname configuration, see the [Hostname section of the Keycloak Server Installation and Configuration Guide](#).

Setting up TLS/SSL

By default, Keycloak won't accept unsecured HTTP connections originating from outside its local network.

HTTPS support requires a TLS/SSL certificate and a private key, both [PEM formatted](#). Their paths should be set through [`services.keycloak.sslCertificate`](#) and [`services.keycloak.sslCertificateKey`](#).

Warning

The paths should be provided as strings, not a Nix paths, since Nix paths are copied into the world readable Nix store.

Themes

You can package custom themes and make them visible to Keycloak through [`services.keycloak.themes`](#). See the [Themes section of the Keycloak Server Development Guide](#) and the description of the aforementioned NixOS option for more information.

Configuration file settings

v: unstable -

Keycloak server configuration parameters can be set in [`services.keycloak.settings`](#). These correspond directly to options in `conf/keycloak.conf`. Some of the most important parameters are documented as suboptions, the rest can be found in the [All configuration section of the Keycloak Server Installation and Configuration Guide](#).

Options containing secret data should be set to an attribute set containing the attribute `_secret` - a string pointing to a file containing the value the option should be set to. See the description of [`services.keycloak.settings`](#) for an example.

Example configuration

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

```
services.keycloak = {  
    enable = true;  
    settings = {  
        hostname = "keycloak.example.com";  
        hostname-strict-backchannel = true;  
    };  
    initialAdminPassword = "e6Wcm0RrtegMEHl"; # change on first login  
    sslCertificate = "/run/keys/ssl_cert";  
    sslCertificateKey = "/run/keys/ssl_key";  
    database.passwordFile = "/run/keys/db_password";  
};
```

Jitsi Meet

Table of Contents

[Basic usage](#)

[Configuration](#)

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

v: unstable -
c: current -
d: dev -
b: beta -
t: test -

solution.

Basic usage

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

```
{  
    services.jitsi-meet = {  
        enable = true;  
        hostName = "jitsi.example.com";  
    };  
    services.jitsi-videobridge.openFirewall = true;  
    networking.firewall.allowedTCPPorts = [ 80 443 ];  
    security.acme.email = "me@example.com";  
    security.acme.acceptTerms = true;  
}
```

Configuration

Here is the minimal configuration with additional configurations:

```
{  
    services.jitsi-meet = {  
        enable = true;  
        hostName = "jitsi.example.com";  
        config = {  
            enableWelcomePage = false;  
            prejoinPageEnabled = true;  
            defaultLang = "fi";  
        };  
        interfaceConfig = {  
            SHOW_JITSI_WATERMARK = false;  
            SHOW_WATERMARK_FOR_GUESTS = false;  
        };  
    };  
    services.jitsi-videobridge.openFirewall = true;
```

v: unstable -

```
networking.firewall.allowedTCPPorts = [ 80 443 ];
security.acme.email = "me@example.com";
security.acme.acceptTerms = true;
}
```

Honk

Table of Contents

Basic usage

With Honk on NixOS you can quickly configure a complete ActivityPub server with minimal setup and support costs.

Basic usage

A minimal configuration looks like this:

```
{
  services.honk = {
    enable = true;
    host = "0.0.0.0";
    port = 8080;
    username = "username";
    passwordFile = "/etc/honk/password.txt";
    servername = "honk.example.com";
  };
}

networking.firewall.allowedTCPPorts = [ 8080 ];
}
```

Grocy

Table of Contents

v: unstable -

Basic usage

Settings

[Grocy](#) is a web-based self-hosted groceries & household management solution for your home.

Basic usage

A very basic configuration may look like this:

```
{ pkgs, ... }:
{
  services.grocy = {
    enable = true;
    hostName = "grocy.tld";
  };
}
```

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.

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, ... }:
{
  services.grocy.settings = {
    # The default currency in the system for invoices etc.
```

v: unstable -

```
# Please note that exchange rates aren't taken into account, this
# is just the setting for what's shown in the frontend.
currency = "EUR";

# The display language (and locale configuration) for grocy.
culture = "de";

calendar = {
    # Whether or not to show the week-numbers
    # in the calendar.
    showWeekNumber = true;

    # Index of the first day to be shown in the calendar (0=Sunday, 1=Monday,
    # 2=Tuesday and so on).
    firstDayOfWeek = 2;
};

};

}
```

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

```
{ lib, ... }:
{
  environment.etc."grocy/config.php".text = lib.mkAfter ''
    // Arbitrary PHP code in grocy's configuration file
  '';
}
```

GoToSocial

Table of Contents

[Service configuration](#)

[Proxy configuration](#)

v: unstable -

User management

[GoToSocial](#) is an ActivityPub social network server, written in Golang.

Service configuration

The following configuration sets up the PostgreSQL as database backend and binds GoToSocial to `127.0.0.1:8080`, expecting to be run behind a HTTP proxy on `gotosocial.example.com`.

```
services.gotosocial = {
    enable = true;
    setupPostgresqlDB = true;
    settings = {
        application-name = "My GoToSocial";
        host = "gotosocial.example.com";
        protocol = "https";
        bind-address = "127.0.0.1";
        port = 8080;
    };
};
```

Please refer to the [GoToSocial Documentation](#) for additional configuration options.

Proxy configuration

Although it is possible to expose GoToSocial directly, it is common practice to operate it behind an HTTP reverse proxy such as nginx.

```
networking.firewall.allowedTCPPorts = [ 80 443 ];
services.nginx = {
    enable = true;
    clientMaxBodySize = "40M";
    virtualHosts = with config.services.gotosocial.settings; {
        "${host}" = {
```

v: unstable -

```
enableACME = true;
forceSSL = true;
locations = {
  "/" = {
    recommendedProxySettings = true;
    proxyWebsockets = true;
    proxyPass = "http://${bind-address}:${toString port}";
  };
};
};

};

};

};
```

Please refer to [SSL/TLS Certificates with ACME](#) for details on how to provision an SSL/TLS certificate.

User management

After the GoToSocial service is running, the `gotosocial-admin` utility can be used to manage users. In particular an administrative user can be created with

```
$ sudo gotosocial-admin account create --username <nickname> --email <email>
$ sudo gotosocial-admin account confirm --username <nickname>
$ sudo gotosocial-admin account promote --username <nickname>
```

Discourse

Table of Contents

[Basic usage](#)

[Using a regular TLS certificate](#)

[Database access](#)

[Email](#)

[Additional settings](#)

v: unstable -

Plugins

[Discourse](#) is a modern and open source discussion platform.

Basic usage

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

```
services.discourse = {
    enable = true;
    hostname = "discourse.example.com";
    admin = {
        email = "admin@example.com";
        username = "admin";
        fullName = "Administrator";
        passwordFile = "/path/to/password_file";
    };
    secretKeyBaseFile = "/path/to/secret_key_base_file";
};
security.acme.email = "me@example.com";
security.acme.acceptTerms = true;
```

Provided a proper DNS setup, you'll be able to connect to the instance at `discourse.example.com` and log in using the credentials provided in `services.discourse.admin`.

Using a regular TLS certificate

To set up TLS using a regular certificate and key on file, use the

[services.discourse.sslCertificate](#) and [services.discourse.sslCertificateKey](#) options:

```
services.discourse = {
    enable = true;
    hostname = "discourse.example.com";
```

v: unstable -

```
sslCertificate = "/path/to/ssl_certificate";
sslCertificateKey = "/path/to/ssl_certificate_key";
admin = {
    email = "admin@example.com";
    username = "admin";
    fullName = "Administrator";
    passwordFile = "/path/to/password_file";
};
secretKeyBaseFile = "/path/to/secret_key_base_file";
};
```

Database access

Discourse uses PostgreSQL to store most of its data. A database will automatically be enabled and a database and role created unless `services.discourse.database.host` is changed from its default of `null` or `services.discourse.database.createLocally` is set to `false`.

External database access can also be configured by setting

`services.discourse.database.host`, `services.discourse.database.username` and `services.discourse.database.passwordFile` as appropriate. Note that you need to manually create a database called `discourse` (or the name you chose in `services.discourse.database.name`) and allow the configured database user full access to it.

Email

In addition to the basic setup, you'll want to configure an SMTP server Discourse can use to send user registration and password reset emails, among others. You can also optionally let Discourse receive email, which enables people to reply to threads and conversations via email.

A basic setup which assumes you want to use your configured `hostname` as email domain can be done like this:

```
services.discourse = {
    enable = true;
    hostname = "discourse.example.com";
    sslCertificate = "/path/to/ssl_certificate";
```

v: unstable -

```
sslCertificateKey = "/path/to/ssl_certificate_key";
admin = {
    email = "admin@example.com";
    username = "admin";
    fullName = "Administrator";
    passwordFile = "/path/to/password_file";
};
mail.outgoing = {
    serverAddress = "smtp.emailprovider.com";
    port = 587;
    username = "user@emailprovider.com";
    passwordFile = "/path/to/smtp_password_file";
};
mail.incoming.enable = true;
secretKeyBaseFile = "/path/to/secret_key_base_file";
};
```

This assumes you have set up an MX record for the address you've set in [hostname](#) and requires proper SPF, DKIM and DMARC configuration to be done for the domain you're sending from, in order for email to be reliably delivered.

If you want to use a different domain for your outgoing email (for example `example.com` instead of `discourse.example.com`) you should set [`services.discourse.mail.notificationEmailAddress`](#) and [`services.discourse.mail.contactEmailAddress`](#) manually.

Note

Setup of TLS for incoming email is currently only configured automatically when a regular TLS certificate is used, i.e. when [`services.discourse.sslCertificate`](#) and [`services.discourse.sslCertificateKey`](#) are set.

Additional settings

Additional site settings and backend settings, for which no explicit NixOS options are provided, can be set in [`services.discourse.siteSettings`](#) and [`services.discourse.backen`](#) v: unstable -

respectively.

Site settings

“Site settings” are the settings that can be changed through the Discourse UI. Their *default* values can be set using `services.discourse.siteSettings`.

Settings are expressed as a Nix attribute set which matches the structure of the configuration in `config/site_settings.yml`. To find a setting’s path, you only need to care about the first two levels; i.e. its category (e.g. `login`) and name (e.g. `invite_only`).

Settings containing secret data should be set to an attribute set containing the attribute `_secret` - a string pointing to a file containing the value the option should be set to. See the example.

Backend settings

Settings are expressed as a Nix attribute set which matches the structure of the configuration in `config/discourse.conf`. Empty parameters can be defined by setting them to `null`.

Example

The following example sets the title and description of the Discourse instance and enables GitHub login in the site settings, and changes a few request limits in the backend settings:

```
services.discourse = {
  enable = true;
  hostname = "discourse.example.com";
  sslCertificate = "/path/to/ssl_certificate";
  sslCertificateKey = "/path/to/ssl_certificate_key";
  admin = {
    email = "admin@example.com";
    username = "admin";
    fullName = "Administrator";
    passwordFile = "/path/to/password_file";
  };
  mail.outgoing = {
```

v: unstable -

```
serverAddress = "smtp.emailprovider.com";
port = 587;
username = "user@emailprovider.com";
passwordFile = "/path/to/smtp_password_file";
};

mail.incoming.enable = true;
siteSettings = {
  required = {
    title = "My Cats";
    site_description = "Discuss My Cats (and be nice plz)";
  };
  login = {
    enable_github_logins = true;
    github_client_id = "a2f6dfe838cb3206ce20";
    github_client_secret._secret = /run/keys/discourse_github_client_secret;
  };
};
backendSettings = {
  max_reqs_per_ip_per_minute = 300;
  max_reqs_per_ip_per_10_seconds = 60;
  max_asset_reqs_per_ip_per_10_seconds = 250;
  max_reqs_per_ip_mode = "warn+block";
};
secretKeyBaseFile = "/path/to/secret_key_base_file";
};
```

In the resulting site settings file, the `login.github_client_secret` key will be set to the contents of the `/run/keys/discourse_github_client_secret` file.

Plugins

You can install Discourse plugins using the `services.discourse.plugins` option. Pre-packaged plugins are provided in `<your_discourse_package_here>.plugins`. If you want the full suite of plugins provided through `nixpkgs`, you can also set the `services.discourse.package` option to `pkgs.discourseAllPlugins`.

Plugins can be built with the `<your_discourse_package_here>.mkDiscoursePl` v: unstable -

Normally, it should suffice to provide a `name` and `src` attribute. If the plugin has Ruby dependencies, however, they need to be packaged in accordance with the [Developing with Ruby](#) section of the Nixpkgs manual and the appropriate gem options set in `bundlerEnvArgs` (normally `gemdir` is sufficient). A plugin's Ruby dependencies are listed in its `plugin.rb` file as function calls to `gem`. To construct the corresponding `Gemfile` manually, run `bundle init`, then add the `gem` lines to it verbatim.

Much of the packaging can be done automatically by the `nixpkgs/pkgs/servers/web-apps/discourse/update.py` script - just add the plugin to the `plugins` list in the `update_plugins` function and run the script:

```
./update.py update-plugins
```

Some plugins provide [site settings](#). Their defaults can be configured using `services.discourse.siteSettings`, just like regular site settings. To find the names of these settings, look in the `config/settings.yml` file of the plugin repo.

For example, to add the [discourse-spoiler-alert](#) and [discourse-solved](#) plugins, and disable `discourse-spoiler-alert` by default:

```
services.discourse = {
    enable = true;
    hostname = "discourse.example.com";
    sslCertificate = "/path/to/ssl_certificate";
    sslCertificateKey = "/path/to/ssl_certificate_key";
    admin = {
        email = "admin@example.com";
        username = "admin";
        fullName = "Administrator";
        passwordFile = "/path/to/password_file";
    };
    mail.outgoing = {
        serverAddress = "smtp.emailprovider.com";
        port = 587;
        username = "user@emailprovider.com";
        passwordFile = "/path/to/smtp_password_file";
    };
};
```

v: unstable -

```
mail.incoming.enable = true;
plugins = with config.services.discourse.package.plugins; [
  discourse-spoiler-alert
  discourse-solved
];
siteSettings = {
  plugins = {
    spoiler_enabled = false;
  };
};
secretKeyBaseFile = "/path/to/secret_key_base_file";
};
```

c2FmZQ

c2FmZQ is an application that can securely encrypt, store, and share files, including but not limited to pictures and videos.

The service `c2fmzq-server` can be enabled by setting

```
{
  services.c2fmzq-server.enable = true;
}
```

This will spin up an instance of the server which is API-compatible with [Stingle Photos](#) and an experimental Progressive Web App (PWA) to interact with the storage via the browser.

In principle the server can be exposed directly on a public interface and there are command line options to manage HTTPS certificates directly, but the module is designed to be served behind a reverse proxy or only accessed via localhost.

```
{
  services.c2fmzq-server = {
    enable = true;
    bindIP = "127.0.0.1"; # default
    port = 8080; # default
  }
}
```

v: unstable -

```
};

services.nginx = {
  enable = true;
  recommendedProxySettings = true;
  virtualHosts."example.com" = {
    enableACME = true;
    forceSSL = true;
    locations."/" = {
      proxyPass = "http://127.0.0.1:8080";
    };
  };
};

}
```

For more information, see <https://github.com/c2FmZQ/c2FmZQ/>.

Akkoma

Table of Contents

[Service configuration](#)

[User management](#)

[Proxy configuration](#)

[Frontend management](#)

[Federation policies](#)

[Upload filters](#)

[Migration from Pleroma](#)

[Advanced deployment options](#)

[Akkoma](#) is a lightweight ActivityPub microblogging server forked from Pleroma.

Service configuration

The Elixir configuration file required by Akkoma is generated automatically from [`services.akkoma.config`](#). Secrets must be included from external files outside of the Nix store by setting the configuration option to an attribute set containing the attribute `_secret` – a string pointing to the file containing the actual value of the option.

For the mandatory configuration settings these secrets will be generated automatically if the referenced file does not exist during startup, unless disabled through [`services.akkoma.initSecrets`](#).

The following configuration binds Akkoma to the Unix socket `/run/akkoma/socket`, expecting to be run behind a HTTP proxy on `fediverse.example.com`.

```
services.akkoma.enable = true;
services.akkoma.config = {
  ":pleroma" = {
    ":instance" = {
      name = "My Akkoma instance";
      description = "More detailed description";
      email = "admin@example.com";
      registration_open = false;
    };
  };

  "Pleroma.Web.Endpoint" = {
    url.host = "fediverse.example.com";
  };
};

};
```

Please refer to the [configuration cheat sheet](#) for additional configuration options.

User management

v: unstable -

After the Akkoma service is running, the administration utility can be used to [manage users](#). In particular an administrative user can be created with

```
$ pleroma_ctl user new <nickname> <email> --admin --moderator --password <password>
```

Proxy configuration

Although it is possible to expose Akkoma directly, it is common practice to operate it behind an HTTP reverse proxy such as nginx.

```
services.akkoma.nginx = {  
    enableACME = true;  
    forceSSL = true;  
};  
  
services.nginx = {  
    enable = true;  
  
    clientMaxBodySize = "16m";  
    recommendedTlsSettings = true;  
    recommendedOptimisation = true;  
    recommendedGzipSettings = true;  
};
```

Please refer to [SSL/TLS Certificates with ACME](#) for details on how to provision an SSL/TLS certificate.

Media proxy

Without the media proxy function, Akkoma does not store any remote media like pictures or video locally, and clients have to fetch them directly from the source server.

```
# Enable nginx slice module distributed with Tengine  
services.nginx.package = pkgs.tengine;  
  
# Enable media proxy
```

v: unstable -

```
services.akkoma.config." :pleroma ". :media_proxy" = {
    enabled = true;
    proxy_opts.redirect_on_failure = true;
};

# Adjust the persistent cache size as needed:
# Assuming an average object size of 128 KiB, around 1 MiB
# of memory is required for the key zone per GiB of cache.
# Ensure that the cache directory exists and is writable by nginx.
services.nginx.commonHttpConfig = ''
  proxy_cache_path /var/cache/nginx/cache/akkoma-media-cache
    levels= keys_zone=akkoma_media_cache:16m max_size=16g
    inactive=1y use_temp_path=off;
';

services.akkoma.nginx = {
  locations."/proxy" = {
    proxyPass = "http://unix:/run/akkoma/socket";

    extraConfig = ''
      proxy_cache akkoma_media_cache;

      # Cache objects in slices of 1 MiB
      slice 1m;
      proxy_cache_key $host$uri$is_args$args$slice_range;
      proxy_set_header Range $slice_range;

      # Decouple proxy and upstream responses
      proxy_buffering on;
      proxy_cache_lock on;
      proxy_ignore_client_abort on;

      # Default cache times for various responses
      proxy_cache_valid 200 1y;
      proxy_cache_valid 206 301 304 1h;

      # Allow serving of stale items
      proxy_cache_use_stale error timeout invalid_header update;
  };
};
```

v: unstable -
unstable -

```
    '';  
};  
};
```

Prefetch remote media

The following example enables the `MediaProxyWarmingPolicy` MRF policy which automatically fetches all media associated with a post through the media proxy, as soon as the post is received by the instance.

```
services.akkoma.config.":pleroma".":mrf".policies =  
  map (pkgs.formats.elixirConf { }).lib.mkRaw [  
    "Pleroma.Web.ActivityPub.MRF.MediaProxyWarmingPolicy"  
];
```

Media previews

Akkoma can generate previews for media.

```
services.akkoma.config.":pleroma".":media_preview_proxy" = {  
  enabled = true;  
  thumbnail_max_width = 1920;  
  thumbnail_max_height = 1080;  
};
```

Frontend management

Akkoma will be deployed with the `akkoma-fe` and `admin-fe` frontends by default. These can be modified by setting `services.akkoma.frontends`.

The following example overrides the primary frontend's default configuration using a custom derivation.

```
services.akkoma.frontends.primary.package = pkgs.runCommand "akkoma-fe" {  
  config = builtins.toJSON {  
    expertLevel = 1;  
  };  
};
```

v: unstable -

```
collapseMessageWithSubject = false;
stopGifs = false;
replyVisibility = "following";
webPushHideIfCW = true;
hideScopeNotice = true;
renderMisskeyMarkdown = false;
hideSiteFavicon = true;
postContentType = "text/markdown";
showNavShortcuts = false;
};

nativeBuildInputs = with pkgs; [ jq xorg.lndir ];
passAsFile = [ "config" ];
} ''
mkdir $out
lndir ${pkgs.akkoma-frontends.akkoma-fe} $out

rm $out/static/config.json
jq -s add ${pkgs.akkoma-frontends.akkoma-fe}/static/config.json ${config}
>$out/static/config.json
'';
```

Federation policies

Akkoma comes with a number of modules to police federation with other ActivityPub instances. The most valuable for typical users is the [:mrf_simple](#) module which allows limiting federation based on instance hostnames.

This configuration snippet provides an example on how these can be used. Choosing an adequate federation policy is not trivial and entails finding a balance between connectivity to the rest of the fediverse and providing a pleasant experience to the users of an instance.

```
services.akkoma.config.":pleroma" = with (pkgs.formats.elixirConf { }).lib;
":mrf".policies = map mkRaw [
  "Pleroma.Web.ActivityPub.MRF.SimplePolicy"
];
```

v: unstable -

```
:mrf_simple" = {
  # Tag all media as sensitive
  media_nsfw = mkMap {
    "nsfw.weird.kinky" = "Untagged NSFW content";
  };

  # Reject all activities except deletes
  reject = mkMap {
    "kiwifarms.cc" = "Persistent harassment of users, no moderation";
  };

  # Force posts to be visible by followers only
  followers_only = mkMap {
    "beta.birdsite.live" = "Avoid polluting timelines with Twitter posts";
  };
};

};
```

Upload filters

This example strips GPS and location metadata from uploads, deduplicates them and anonymises the file name.

```
services.akkoma.config.":pleroma"."Pleroma.Upload".filters =
map (pkgs.formats.elixirConf { }).lib.mkRaw [
  "Pleroma.Upload.Filter.Exiftool"
  "Pleroma.Upload.Filter.Dedupe"
  "Pleroma.Upload.Filter.AnonymizeFilename"
];
```

Migration from Pleroma

Pleroma instances can be migrated to Akkoma either by copying the database and upload data or by pointing Akkoma to the existing data. The necessary database migrations are run automatically during startup of the service.

v: unstable -

The configuration has to be copy-edited manually.

Depending on the size of the database, the initial migration may take a long time and exceed the startup timeout of the system manager. To work around this issue one may adjust the startup timeout `systemd.services.akkoma.serviceConfig.TimeoutStartSec` or simply run the migrations manually:

```
pleroma_ctl migrate
```

Copying data

Copying the Pleroma data instead of re-using it in place may permit easier reversion to Pleroma, but allows the two data sets to diverge.

First disable Pleroma and then copy its database and upload data:

```
# Create a copy of the database
nix-shell -p postgresql --run 'createdb -T pleroma akkoma'

# Copy upload data
mkdir /var/lib/akkoma
cp -R --reflink=auto /var/lib/pleroma/uploads /var/lib/akkoma/
```

After the data has been copied, enable the Akkoma service and verify that the migration has been successful. If no longer required, the original data may then be deleted:

```
# Delete original database
nix-shell -p postgresql --run 'dropdb pleroma'

# Delete original Pleroma state
rm -r /var/lib/pleroma
```

Re-using data

To re-use the Pleroma data in place, disable Pleroma and enable Akkoma, pointing it to the

v: unstable -

database and upload directory.

```
# Adjust these settings according to the database name and upload directory
services.akkoma.config.":pleroma"."Pleroma.Repo".database = "pleroma";
services.akkoma.config.":pleroma"."instance".upload_dir = "/var/lib/plero...
```

Please keep in mind that after the Akkoma service has been started, any migrations applied by Akkoma have to be rolled back before the database can be used again with Pleroma. This can be achieved through `pleroma_ctl ecto.rollback`. Refer to the [Ecto SQL documentation](#) for details.

Advanced deployment options

Confinement

The Akkoma systemd service may be confined to a chroot with

```
services.systemd.akkoma.confinement.enable = true;
```

Confinement of services is not generally supported in NixOS and therefore disabled by default.

Depending on the Akkoma configuration, the default confinement settings may be insufficient and lead to subtle errors at run time, requiring adjustment:

Use [`services.systemd.akkoma.confinement.packages`](#) to make packages available in the chroot.

`services.systemd.akkoma.serviceConfig.BindPaths` and `services.systemd.akkoma.serviceConfig.BindReadOnlyPaths` permit access to outside paths through bind mounts. Refer to [`BindPaths=` of `systemd.exec\(5\)`](#) for details.

Distributed deployment

Being an Elixir application, Akkoma can be deployed in a distributed fashion.

This requires setting [`services.akkoma.dist.address`](#) and [`services.akkoma.dist.cookie`](#). The specifics depend strongly on the deployment environment. For more information please refer to the [Distributed deployment section](#).

relevant [Erlang documentation](#).

systemd-lock-handler

The `systemd-lock-handler` module provides a service that bridges D-Bus events from `logind` to user-level systemd targets:

- `lock.target` started by `logindctl lock-session`,
- `unlock.target` started by `logindctl unlock-session` and
- `sleep.target` started by `systemctl suspend`.

You can create a user service that starts with any of these targets.

For example, to create a service for `swaylock`:

```
{  
    services.systemd-lock-handler.enable = true;  
  
    systemd.user.services.swaylock = {  
        description = "Screen locker for Wayland";  
        documentation = ["man:swaylock(1)"];  
  
        # If swaylock exits cleanly, unlock the session:  
        onSuccess = ["unlock.target"];  
  
        # When lock.target is stopped, stops this too:  
        partOf = ["lock.target"];  
  
        # Delay lock.target until this service is ready:  
        before = ["lock.target"];  
        wantedBy = ["lock.target"];  
  
        serviceConfig = {  
            # systemd will consider this service started when swaylock forks...  
            Type = "forking";
```

v: unstable -

```
# ... and swaylock will fork only after it has locked the screen.  
ExecStart = "${lib.getExe pkgs.swaylock} -f";  
  
# If swaylock crashes, always restart it immediately:  
Restart = "on-failure";  
RestartSec = 0;  
};  
};  
}
```

See [upstream documentation](#) for more information.

Meilisearch

Table of Contents

[Quickstart](#)

[Usage](#)

[Defaults](#)

[Missing](#)

Meilisearch is a lightweight, fast and powerful search engine. Think elastic search with a much smaller footprint.

Quickstart

the minimum to start meilisearch is

```
services.meilisearch.enable = true;
```

this will start the http server included with meilisearch on port 7700.

v: unstable -

```
test with curl -X GET 'http://localhost:7700/health'
```

Usage

you first need to add documents to an index before you can search for documents.

Add a documents to the movies index

```
curl -X POST 'http://127.0.0.1:7700/indexes/movies/documents' --data  
'[{"id": "123", "title": "Superman"}, {"id": 234, "title": "Batman"}]'
```

Search documents in the movies index

```
curl 'http://127.0.0.1:7700/indexes/movies/search' --data '{ "q": "botman"  
}' (note the typo is intentional and there to demonstrate the typo tolerant capabilities)
```

Defaults

- The default nixos package doesn't come with the [dashboard](#), since the dashboard features makes some assets downloads at compile time.
- Anonymized Analytics sent to meilisearch are disabled by default.
- Default deployment is development mode. It doesn't require a secret master key. All routes are not protected and accessible.

Missing

- the snapshot feature is not yet configurable from the module, it's just a matter of adding the relevant environment variables.

Yggdrasil

Table of Contents

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.

Configuration

Simple ephemeral node

An annotated example of a simple configuration:

```
{  
  services.yggdrasil = {  
    enable = true;  
    persistentKeys = false;  
    # The NixOS module will generate new keys and a new IPv6 address each  
    # time it is started if persistentKeys is not enabled.  
  
    settings = {  
      Peers = [  
        # Yggdrasil will automatically connect and "peer" with other nodes  
        # it discovers via link-local multicast announcements. Unless this is  
        # the case (it probably isn't) a node needs peers within the existing  
        # network that it can tunnel to.  
        "tcp://1.2.3.4:1024"  
        "tcp://1.2.3.5:1024"  
        # Public peers can be found at  
        # https://github.com/yggdrasil-network/public-peers  
      ];  
    };  
  };  
};
```

v: unstable -

}

Persistent node with prefix

A node with a fixed address that announces a prefix:

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

  services.yggdrasil = {
    enable = true;
    persistentKeys = true; # Maintain a fixed public key and IPv6 address
    settings = {
      Peers = [ "tcp://1.2.3.4:1024" "tcp://1.2.3.5:1024" ];
      NodeInfo = {
        # This information is visible to the network.
        name = config.networking.hostName;
        location = "The North Pole";
      };
    };
  };

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

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

services.radvd = {
  # Announce the 300::/8 prefix to eth0.
  enable = true;
};
```

v: unstable -

```
config = ''
  interface eth0
  {
    AdvSendAdvert on;
    prefix ${prefix}:::/64 {
      AdvOnLink on;
      AdvAutonomous on;
    };
    route 200::/8 {};
  };
''';
};
```

Yggdrasil attached Container

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

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

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

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

v: unstable -

```
};

containers.foo = {
    autoStart = true;
    privateNetwork = true;
    hostBridge = "br0";
    # Attach the container to the bridge only.
    config = { config, pkgs, ... }: {
        networking.interfaces.eth0.ipv6 = {
            addresses = [{{
                # Configure a prefix address.
                address = "${yggPrefix64}::2";
                prefixLength = 64;
            }};
            routes = [{{
                # Configure the prefix route.
                address = "200::";
                prefixLength = 7;
                via = "${yggPrefix64}::1";
            }};
        };
    };

    services.httpd.enable = true;
    networking.firewall.allowedTCPPorts = [ 80 ];
};

};

}
```

Prosody

Table of Contents

[Basic usage](#)

[Let's Encrypt Configuration](#)

[Prosody](#) is an open-source, modern XMPP server.

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 recommended 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 = {
  enable = true;
  admins = [ "root@example.org" ];
  ssl.cert = "/var/lib/acme/example.org/fullchain.pem";
  ssl.key = "/var/lib/acme/example.org/key.pem";
  virtualHosts."example.org" = {
    enabled = true;
    domain = "example.org";
    ssl.cert = "/var/lib/acme/example.org/fullchain.pem";
    ssl.key = "/var/lib/acme/example.org/key.pem";
  };
  muc = [ {
    domain = "conference.example.org";
  }];
  uploadHttp = {
    domain = "upload.example.org";
  };
};
```

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 = {
  email = "root@example.org";
  acceptTerms = true;
  certs = {
    "example.org" = {
      webroot = "/var/www/example.org";
      email = "root@example.org";
      extraDomainNames = [ "conference.example.org" "upload.ex...
```

```
};  
};  
};
```

Pleroma

Table of Contents

[Generating the Pleroma config](#)

[Initializing the database](#)

[Enabling the Pleroma service locally](#)

[Creating the admin user](#)

[Configuring Nginx](#)

[Pleroma](#) is a lightweight activity pub server.

Generating the Pleroma config

The `pleroma_ctl` CLI utility will prompt you some questions and it will generate an initial config file. This is an example of usage

```
$ mkdir tmp-pleroma  
$ cd tmp-pleroma  
$ nix-shell -p pleroma-otp  
$ pleroma_ctl instance gen --output config.exs --output-psql setup.sql
```

The `config.exs` file can be further customized following the instructions on the [upstream documentation](#). Many refinements can be applied also after the service is running.

Initializing the database

v: unstable -

First, the Postgresql service must be enabled in the NixOS configuration

```
services.postgresql = {  
    enable = true;  
    package = pkgs.postgresql_13;  
};
```

and activated with the usual

```
$ nixos-rebuild switch
```

Then you can create and seed the database, using the `setup.sql` file that you generated in the previous section, by running

```
$ sudo -u postgres psql -f setup.sql
```

Enabling the Pleroma service locally

In this section we will enable the Pleroma service only locally, so its configurations can be improved incrementally.

This is an example of configuration, where `services.pleroma.configs` option contains the content of the file `config.exs`, generated [in the first section](#), but with the secrets (database password, endpoint secret key, salts, etc.) removed. Removing secrets is important, because otherwise they will be stored publicly in the Nix store.

```
services.pleroma = {  
    enable = true;  
    secretConfigFile = "/var/lib/pleroma/secrets.exs";  
    configs = [  
        ''  
        import Config  
  
        config :pleroma, Pleroma.Web.Endpoint,
```

v: unstable -

```
url: [host: "pleroma.example.net", scheme: "https", port: 443],  
http: [ip: {127, 0, 0, 1}, port: 4000]  
  
config :pleroma, :instance,  
  name: "Test",  
  email: "admin@example.net",  
  notify_email: "admin@example.net",  
  limit: 5000,  
  registrations_open: true  
  
config :pleroma, :media_proxy,  
  enabled: false,  
  redirect_on_failure: true  
  
config :pleroma, Pleroma.Repo,  
  adapter: Ecto.Adapters.Postgres,  
  username: "pleroma",  
  database: "pleroma",  
  hostname: "localhost"  
  
# Configure web push notifications  
config :web_push_encryption, :vapid_details,  
  subject: "mailto:admin@example.net"  
  
# ... TO CONTINUE ...  
'  
];  
};
```

Secrets must be moved into a file pointed by `services.pleroma.secretConfigFile`, in our case `/var/lib/pleroma/secrets.exs`. This file can be created copying the previously generated `config.exs` file and then removing all the settings, except the secrets. This is an example

```
# Pleroma instance passwords  
  
import Config
```

v: unstable -

```
config :pleroma, Pleroma.Web.Endpoint,
  secret_key_base: "<the secret generated by pleroma_ctl>",
  signing_salt: "<the secret generated by pleroma_ctl>

config :pleroma, Pleroma.Repo,
  password: "<the secret generated by pleroma_ctl>

# Configure web push notifications
config :web_push_encryption, :vapid_details,
  public_key: "<the secret generated by pleroma_ctl>",
  private_key: "<the secret generated by pleroma_ctl>

# ... TO CONTINUE ...
```

Note that the lines of the same configuration group are comma separated (i.e. all the lines end with a comma, except the last one), so when the lines with passwords are added or removed, commas must be adjusted accordingly.

The service can be enabled with the usual

```
$ nixos-rebuild switch
```

The service is accessible only from the local `127.0.0.1:4000` port. It can be tested using a port forwarding like this

```
$ ssh -L 4000:localhost:4000 myuser@example.net
```

and then accessing <http://localhost:4000> from a web browser.

Creating the admin user

After Pleroma service is running, all [Pleroma administration utilities](#) can be used. In particular an admin user can be created with

```
$ pleroma_ctl user new <nickname> <email> --admin --moderator --password
```

Configuring Nginx

In this configuration, Pleroma is listening only on the local port 4000. Nginx can be configured as a Reverse Proxy, for forwarding requests from public ports to the Pleroma service. This is an example of configuration, using [Let's Encrypt](#) for the TLS certificates

```
security.acme = {
    email = "root@example.net";
    acceptTerms = true;
};

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

    recommendedTlsSettings = true;
    recommendedOptimisation = true;
    recommendedGzipSettings = true;

    recommendedProxySettings = false;
    # NOTE: if enabled, the NixOS proxy optimizations will override the Pleroma
    # specific settings, and they will enter in conflict.

    virtualHosts = {
        "pleroma.example.net" = {
            http2 = true;
            enableACME = true;
            forceSSL = true;

            locations."/" = {
                proxyPass = "http://127.0.0.1:4000";

                extraConfig = ''
                    etag on;
            };
        };
    };
};
```

v: unstable -

```
    gzip on;

    add_header 'Access-Control-Allow-Origin' '*' always;
    add_header 'Access-Control-Allow-Methods' 'POST, PUT, DELETE, GET';
    add_header 'Access-Control-Allow-Headers' 'Authorization, Content-Type';
    add_header 'Access-Control-Expose-Headers' 'Link, X-RateLimit-Remaining';

    if ($request_method = OPTIONS) {
        return 204;
    }

    add_header X-XSS-Protection "1; mode=block";
    add_header X-Permitted-Cross-Domain-Policies none;
    add_header X-Frame-Options DENY;
    add_header X-Content-Type-Options nosniff;
    add_header Referrer-Policy same-origin;
    add_header X-Download-Options noopener;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_set_header Host $host;

    client_max_body_size 16m;
    # NOTE: increase if users need to upload very big files
};

};

};

};

};
```

Netbird

Table of Contents

[Quickstart](#)

[Multiple connections setup](#)

Quickstart

The absolute minimal configuration for the netbird daemon looks like this:

```
services.netbird.enable = true;
```

This will set up a netbird service listening on the port **51820** associated to the **wt0** interface.

It is strictly equivalent to setting:

```
services.netbird.tunnels.wt0.stateDir = "netbird";
```

The **enable** option is mainly kept for backward compatibility, as defining netbird tunnels through the **tunnels** option is more expressive.

Multiple connections setup

Using the **services.netbird.tunnels** option, it is also possible to define more than one netbird service running at the same time.

The following configuration will start a netbird daemon using the interface **wt1** and the port **51830**. Its configuration file will then be located at **/var/lib/netbird-wt1/config.json**.

```
services.netbird.tunnels = {
    wt1 = {
        port = 51830;
    };
};
```

To interact with it, you will need to specify the correct daemon address:

```
netbird --daemon-addr unix:///var/run/netbird-wt1/sock ...
```

The address will by default be **unix:///var/run/netbird-<name>**.

v: unstable -

It is also possible to overwrite default options passed to the service, for example:

```
services.netbird.tunnels.wt1.environment = {
    NB_DAEMON_ADDR = "unix:///var/run/toto.sock"
};
```

This will set the socket to interact with the netbird service to `/var/run/toto.sock`.

Mosquitto

Table of Contents

[Quickstart](#)

[Configuration](#)

Mosquitto is a MQTT broker often used for IoT or home automation data transport.

Quickstart

A minimal configuration for Mosquitto is

```
services.mosquitto = {
    enable = true;
    listeners = [ {
        acl = [ "pattern readwrite #" ];
        omitPasswordAuth = true;
        settings.allow_anonymous = true;
    } ];
};
```

This will start a broker on port 1883, listening on all interfaces of the machine, allowing read/write access to all topics to any user without password requirements.

User authentication can be configured with the `users` key of listeners. A config that give

v: unstable -

access to a user `monitor` and restricted write access to a user `service` could look like

```
services.mosquitto = {
  enable = true;
  listeners = [ {
    users = {
      monitor = {
        acl = [ "read #" ];
        password = "monitor";
      };
      service = {
        acl = [ "write service/#" ];
        password = "service";
      };
    };
  }];
};
```

TLS authentication is configured by setting TLS-related options of the listener:

```
services.mosquitto = {
  enable = true;
  listeners = [ {
    port = 8883; # port change is not required, but helpful to avoid mista
    # ...
    settings = {
      cafile = "/path/to/mqtt.ca.pem";
      certfile = "/path/to/mqtt.pem";
      keyfile = "/path/to/mqtt.key";
    };
  }];
};
```

Configuration

The Mosquitto configuration has four distinct types of settings: the global settings of the `daemon`, `v: unstable -` listeners, plugins, and bridges. Bridges and listeners are part of the global configuration, |

part of listeners. Users of the broker are configured as parts of listeners rather than globally, allowing configurations in which a given user is only allowed to log in to the broker using specific listeners (eg to configure an admin user with full access to all topics, but restricted to localhost).

Almost all options of Mosquitto are available for configuration at their appropriate levels, some as NixOS options written in camel case, the remainders under `settings` with their exact names in the Mosquitto config file. The exceptions are `acl_file` (which is always set according to the `acl` attributes of a listener and its users) and `per_listener_settings` (which is always set to `true`).

Password authentication

Mosquitto can be run in two modes, with a password file or without. Each listener has its own password file, and different listeners may use different password files. Password file generation can be disabled by setting `omitPasswordAuth = true` for a listener; in this case it is necessary to either set `settings.allow_anonymous = true` to allow all logins, or to configure other authentication methods like TLS client certificates with `settings.use_identity_as_username = true`.

The default is to generate a password file for each listener from the users configured to that listener. Users with no configured password will not be added to the password file and thus will not be able to use the broker.

ACL format

Every listener has a Mosquitto `acl_file` attached to it. This ACL is configured via two attributes of the config:

- the `acl` attribute of the listener configures pattern ACL entries and topic ACL entries for anonymous users. Each entry must be prefixed with `pattern` or `topic` to distinguish between these two cases.
- the `acl` attribute of every user configures in the listener configured the ACL for that given user. Only topic ACLs are supported by Mosquitto in this setting, so no prefix is required or allowed.

The default ACL for a listener is empty, disallowing all accesses from all clients. To configure a completely open ACL, set `acl = ["pattern readwrite #"]` in the listener.

GNS3 Server

v: unstable -

Table of Contents

[Basic Usage](#)

[GNS3](#), a network software emulator.

Basic Usage

A minimal configuration looks like this:

```
{  
    services.gns3-server = {  
        enable = true;  
  
        auth = {  
            enable = true;  
            user = "gns3";  
            passwordFile = "/var/lib/secrets/gns3_password";  
        };  
  
        ssl = {  
            enable = true;  
            certFile = "/var/lib/gns3/ssl/cert.pem";  
            keyFile = "/var/lib/gns3/ssl/key.pem";  
        };  
  
        dynamips.enable = true;  
        ubridge.enable = true;  
        vpcs.enable = true;  
    };  
}
```

Firefox Sync server

[Table of Contents](#)

v: unstable -

Quickstart

More detailed setup

A storage server for Firefox Sync that you can easily host yourself.

Quickstart

The absolute minimal configuration for the sync server looks like this:

```
services.mysql.package = pkgs.mariadb;

services.firefox-syncserver = {
    enable = true;
    secrets = builtins.toFile "sync-secrets" ''
        SYNC_MASTER_SECRET=this-secret-is-actually-leaked-to-/nix/store
    '';
    singleNode = {
        enable = true;
        hostname = "localhost";
        url = "http://localhost:5000";
    };
};
```

This will start a sync server that is only accessible locally. Once the services is running you can navigate to `about:config` in your Firefox profile and set `identity.sync.tokensever.uri` to `http://localhost:5000/1.0sync/1.5`. Your browser will now use your local sync server for data storage.

Warning

This configuration should never be used in production. It is not encrypted and stores its secrets in a world-readable location.

More detailed setup

The `firefox-syncserver` service provides a number of options to make setting up small deployment easier. These are grouped under the `singleNode` element of the option tree and allow simple configuration of the most important parameters.

Single node setup is split into two kinds of options: those that affect the sync server itself, and those that affect its surroundings. Options that affect the sync server are `capacity`, which configures how many accounts may be active on this instance, and `url`, which holds the URL under which the sync server can be accessed. The `url` can be configured automatically when using nginx.

Options that affect the surroundings of the sync server are `enableNginx`, `enableTLS` and `hostname`. If `enableNginx` is set the sync server module will automatically add an nginx virtual host to the system using `hostname` as the domain and set `url` accordingly. If `enableTLS` is set the module will also enable ACME certificates on the new virtual host and force all connections to be made via TLS.

For actual deployment it is also recommended to store the `secrets` file in a secure location.

Dnsmasq

Table of Contents

[Configuration](#)

[References](#)

Dnsmasq is an integrated DNS, DHCP and TFTP server for small networks.

Configuration

An authoritative DHCP and DNS server on a home network

On a home network, you can use Dnsmasq as a DHCP and DNS server. New devices on your network will be configured by Dnsmasq, and instructed to use it as the DNS server by default. This v: unstable -

rely on your own server to perform DNS queries and caching, with DNSSEC enabled.

The following example assumes that

- you have disabled your router's integrated DHCP server, if it has one
- your router's address is set in [networking.defaultGateway.address](#)
- your system's Ethernet interface is `eth0`
- you have configured the address(es) to forward DNS queries in [networking.nameservers](#)

```
{  
    services.dnsmasq = {  
        enable = true;  
        settings = {  
            interface = "eth0";  
            bind-interfaces = true; # Only bind to the specified interface  
            dhcp-authoritative = true; # Should be set when dnsmasq is definite-  
            server = config.networking.nameservers; # Upstream dns servers to whi-  
            ch will forward queries  
            dhcp-host = [  
                # Give the current system a fixed address of 192.168.0.254  
                "dc:a6:32:0b:ea:b9,192.168.0.254,${config.networking.hostName},in-  
            ];  
  
            dhcp-option = [  
                # Address of the gateway, i.e. your router  
                "option:router,${config.networking.defaultGateway.address}"  
            ];  
  
            dhcp-range = [  
                # Range of IPv4 addresses to give out  
                # <range start>, <range end>, <lease time>  
                "192.168.0.10,192.168.0.253,24h"  
                # Enable stateless IPv6 allocation  
                "::f,::ff,constructor:eth0,ra-stateless"  
            ];  
    };  
}
```

v: unstable -

```
];  
  
    dhcp-rapid-commit = true; # Faster DHCP negotiation for IPv6  
    local-service = true; # Accept DNS queries only from hosts whose add  
    log-queries = true; # Log results of all DNS queries  
    bogus-priv = true; # Don't forward requests for the local address ra  
    domain-needed = true; # Don't forward requests without dots or domai  
  
    dnssec = true; # Enable DNSSEC  
    # DNSSEC trust anchor. Source: https://data.iana.org/root-anchors/re  
    trust-anchor = ". ,20326,8,2,E06D44B80B8F1D39A95C0B0D7C65D08458E88040";  
};  
};  
}
```

References

- Upstream website: <https://dnsmasq.org>
- Manpage: <https://dnsmasq.org/docs/dnsmasq-man.html>
- FAQ: <https://dnsmasq.org/docs/FAQ>

Litestream

Table of Contents

[Configuration](#)

[Litestream](#) is a standalone streaming replication tool for SQLite.

Configuration

Litestream service is managed by a dedicated user named `litestream` which needs permission to the database file. Here's an example config which gives required permissions to access [v:unstable -](#)

database:

```
{ pkgs, ... }:

{
  users.users.litestream.extraGroups = [ "grafana" ];

  systemd.services.grafana.serviceConfig.ExecStartPost = "+${pkgs.writeScript}/grafana-start";
  systemd.services.grafana.serviceConfig.TimeoutStart = "10s";
  systemd.services.grafana.serviceConfig.TimeoutStop = "10s";

  while [ ! -f /var/lib/grafana/data/grafana.db ];
  do
    if [ "$timeout" == 0 ]; then
      echo "ERROR: Timeout while waiting for /var/lib/grafana/data/grafana.db"
      exit 1
    fi

    sleep 1

    ((timeout--))
  done

  find /var/lib/grafana -type d -exec chmod -v 775 {} \;
  find /var/lib/grafana -type f -exec chmod -v 660 {} \;
  '';

services.litestream = {
  enable = true;

  environmentFile = "/run/secrets/litestream";

  settings = {
    dbs = [
      {
        path = "/var/lib/grafana/data/grafana.db";
        replicas = [
          {
            url = "s3://mybkt.litestream.io/grafana";
          }];
    ];
  };
};
```

v: unstable -

```
        }
    ];
};

};

}
```

Prometheus exporters

Table of Contents

[Configuration](#)

[Adding a new exporter](#)

[Updating an exporter module](#)

Prometheus exporters provide metrics for the [prometheus monitoring system](#).

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:

```
services.prometheus.exporters.node = {
  enable = true;
  port = 9100;
  enabledCollectors = [
    "logind"
    "systemd"
];
  disabledCollectors = [
    "textfile"
];
  openFirewall = true;
  firewallFilter = "-i br0 -p tcp -m tcp --dport 9100";
```

v: unstable -

```
};
```

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](#).

Prometheus can now be configured to consume the metrics produced by the exporter:

```
services.prometheus = {
  # ...

  scrapeConfigs = [
    {
      job_name = "node";
      static_configs = [{{
        targets = [ "localhost:${toString config.services.prometheus.(
      }];
    }
  ];
}

# ...
}
```

Adding a new exporter

To add a new exporter, it has to be packaged first (see `nixpkgs/pkgs/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.nix`:
 - `enable`

v: unstable -

- 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/prometheus/exporters/postfix.nix
{ config, lib, pkgs, options }:

with lib;

let
  # for convenience we define cfg here
  cfg = config.services.prometheus.exporters.postfix;
in
{
  port = 9154; # The postfix exporter listens on this port by default

  # `extraOpts` is an attribute set which contains additional options
  # (and optional overrides for default options).
  # Note that this attribute is optional.
  extraOpts = {
    telemetryPath = mkOption {
      type = types.str;
      default = "/metrics";
      description = ''
        Path under which to expose metrics.
    };
  };
}
```

v: unstable -

```
'';
};

logfilePath = mkOption {
    type = types.path;
    default = /var/log/postfix_exporter_input.log;
    example = /var/log/mail.log;
    description = ''
        Path where Postfix writes log entries.
        This file will be truncated by this exporter!
';
};

showqPath = mkOption {
    type = types.path;
    default = /var/spool/postfix/public/showq;
    example = /var/lib/postfix/queue/public/showq;
    description = ''
        Path at which Postfix places its showq socket.
';
};

};

# `serviceOpts` is an attribute set which contains configuration
# for the exporter's systemd service. One of
# `serviceOpts.script` and `serviceOpts.serviceConfig.ExecStart`
# has to be specified here. This will be merged with the default
# service configuration.
# Note that by default 'DynamicUser' is 'true'.
serviceOpts = {
    serviceConfig = {
        DynamicUser = false;
        ExecStart = ''
            ${pkgs.prometheus-postfix-exporter}/bin/postfix_exporter \
                --web.listen-address ${cfg.listenAddress}:${toString cfg.port}
                --web.telemetry-path ${cfg.telemetryPath} \
                ${concatStringsSep " \\\n" cfg.extraFlags}
';
    };
};

};  
v: unstable -
```

}

- 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`

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 }:

with lib;

let
  cfg = config.services.prometheus.exporters.nginx;
in
{
  port = 9113;
  extraOpts = {
    # additional module options
    # ...
  };
  serviceOpts = {
    # service configuration
    # ...
  };
  imports = [
    # 'services.prometheus.exporters.nginx.telemetryEndpoint' -> 'services.(mkRenamedOptionModule [ "telemetryEndpoint" ] [ "telemetryPath" ])'

    # removed option 'services.prometheus.exporters.nginx.insecure'
    (mkRemovedOptionModule [ "insecure" ] ''
      This option was replaced by 'prometheus.exporters.nginx.: v:unstable - '
    );
  ];
}
```

```
        '')
    ({ options.warnings = options.warnings; })
];
}
```

parsedmarc

Table of Contents

[Basic usage](#)

[Local mail](#)

[Grafana and GeoIP](#)

[parsedmarc](#) is a service which parses incoming [DMARC](#) reports and stores or sends them to a downstream service for further analysis. In combination with Elasticsearch, Grafana and the included Grafana dashboard, it provides a handy overview of DMARC reports over time.

Basic usage

A very minimal setup which reads incoming reports from an external email address and saves them to a local Elasticsearch instance looks like this:

```
services.parsedmarc = {
  enable = true;
  settings imap = {
    host = "imap.example.com";
    user = "alice@example.com";
    password = "/path/to/imap_password_file";
  };
  provision.geoIp = false; # Not recommended!
};
```

Note that GeolP provisioning is disabled in the example for simplicity, but should be turned on for fully functional reports.

Local mail

Instead of watching an external inbox, a local inbox can be automatically provisioned. The recipient's name is by default set to `dmarc`, but can be configured in `services.parsedmarc.provision.localMail.recipientName`. You need to add an MX record pointing to the host. More concretely: for the example to work, an MX record needs to be set up for `monitoring.example.com` and the complete email address that should be configured in the domain's dmarc policy is `dmarc@monitoring.example.com`.

```
services.parsedmarc = {
  enable = true;
  provision = {
    localMail = {
      enable = true;
      hostname = monitoring.example.com;
    };
    geoIp = false; # Not recommended!
  };
};
```

Grafana and GeolP

The reports can be visualized and summarized with parsedmarc's official Grafana dashboard. For all views to work, and for the data to be complete, GeolP databases are also required. The following example shows a basic deployment where the provisioned Elasticsearch instance is automatically added as a Grafana datasource, and the dashboard is added to Grafana as well.

```
services.parsedmarc = {
  enable = true;
  provision = {
    localMail = {
      enable = true;
    };
  };
};
```

v: unstable -

```
        hostname = url;
    };
    grafana = {
        datasource = true;
        dashboard = true;
    };
};

# Not required, but recommended for full functionality
services.geoipupdate = {
    settings = {
        AccountID = 0000000;
        LicenseKey = "/path/to/license_key_file";
    };
};

services.grafana = {
    enable = true;
    addr = "0.0.0.0";
    domain = url;
    rootUrl = "https://" + url;
    protocol = "socket";
    security = {
        adminUser = "admin";
        adminPasswordFile = "/path/to/admin_password_file";
        secretKeyFile = "/path/to/secret_key_file";
    };
};

services.nginx = {
    enable = true;
    recommendedTlsSettings = true;
    recommendedOptimisation = true;
    recommendedGzipSettings = true;
    recommendedProxySettings = true;
    upstreams.grafana.servers."unix:${config.services.grafana.socket}" = {
        virtualHosts.${url} = {
```

```
root = config.services.grafana.staticRootPath;
enableACME = true;
forceSSL = true;
locations."/".tryFiles = "$uri @grafana";
locations."@grafana".proxyPass = "http://grafana";
};

};

users.users.nginx.extraGroups = [ "grafana" ];
```

OCS Inventory Agent

Table of Contents

[Basic Usage](#)

[OCS Inventory NG](#) or Open Computers and Software inventory is an application designed to help IT administrator to keep track of the hardware and software configurations of computers that are installed on their network.

OCS Inventory collects information about the hardware and software of networked machines through the **OCS Inventory Agent** program.

This NixOS module enables you to install and configure this agent so that it sends information from your computer to the OCS Inventory server.

For more technical information about OCS Inventory Agent, refer to [the Wiki documentation](#).

Basic Usage

A minimal configuration looks like this:

```
{
  services.ocsinventory-agent = {
    enable = true;
    settings = {
```

v: unstable -

```
    server = "https://ocsinventory.localhost:8080/ocsinventory";
    tag = "01234567890123";
};

};

}
```

This configuration will periodically run the ocsinventory-agent SystemD service.

The OCS Inventory Agent will inventory the computer and then sends the results to the specified OCS Inventory Server.

Goss

Table of Contents

Basic Usage

[goss](#) is a YAML based serverspec alternative tool for validating a server's configuration.

Basic Usage

A minimal configuration looks like this:

```
{
  services.goss = {
    enable = true;

    environment = {
      GOSS_FMT = "json";
      GOSS_LOGLEVEL = "TRACE";
    };

    settings = {
      addr."tcp://localhost:8080" = {
        reachable = true;
      };
    };
  };
}
```

v: unstable -

```
    local-address = "127.0.0.1";
};

command."check-goss-version" = {
  exec = "${lib.getExe pkgs.goss} --version";
  exit-status = 0;
};
dns.localhost.resolvable = true;
file."/nix" = {
  filetype = "directory";
  exists = true;
};
group.root.exists = true;
kernel-param."kernel.ostype".value = "Linux";
service.goss = {
  enabled = true;
  running = true;
};
user.root.exists = true;
};

};

}
```

Cert Spotter

Table of Contents

[Service Configuration](#)

[Operation](#)

[Hooks](#)

Cert Spotter is a tool for monitoring [Certificate Transparency](#) logs.

Service Configuration

v: unstable -

A basic config that notifies you of all certificate changes for your domain would look as follows:

```
services.certspotter = {  
    enable = true;  
    # replace example.org with your domain name  
    watchlist = [ ".example.org" ];  
    emailRecipients = [ "webmaster@example.org" ];  
};  
  
# Configure an SMTP client  
programs(msmtp).enable = true;  
# Or you can use any other module that provides sendmail, like  
# services.nullmailer, services.opensmtpd, services.postfix
```

In this case, the leading dot in `".example.org"` means that Cert Spotter should monitor not only `example.org`, but also all of its subdomains.

Operation

By default, NixOS configures Cert Spotter to skip all certificates issued before its first launch,

because checking the entire Certificate Transparency logs requires downloading tens of terabytes of data. If you want to check the *entire* logs for previously issued certificates, you have to set `services.certspotter.startAtEnd` to `false` and remove all previously saved log state in `/var/lib/certspotter/logs`. The downloaded logs aren't saved, so if you add a new domain to the watchlist and want Cert Spotter to go through the logs again, you will have to remove `/var/lib/certspotter/logs` again.

After catching up with the logs, Cert Spotter will start monitoring live logs. As of October 2023, it uses around **20 Mbps** of traffic on average.

Hooks

Cert Spotter supports running custom hooks instead of (or in addition to) sending emails. Hooks are shell scripts that will be passed certain environment variables.

To see hook documentation, see Cert Spotter's man pages:

v: unstable -

```
nix-shell -p certspotter --run 'man 8 certspotter-script'
```

For example, you can remove `emailRecipients` and send email notifications manually using the following hook:

```
services.certspotter.hooks = [
  (pkgs.writeShellScript "certspotter-hook" ''
    function print_email() {
      echo "Subject: [certspotter] $SUMMARY"
      echo "Mime-Version: 1.0"
      echo "Content-Type: text/plain; charset=US-ASCII"
      echo
      cat "$TEXT_FILENAME"
    }
    print_email | ${config.services.certspotter.sendmailPath} -i webmaster
  '')
];
```

WeeChat

Table of Contents

[Basic Usage](#)

[Re-attaching to WeeChat](#)

[WeeChat](#) is a fast and extensible IRC client.

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:

v: unstable -

```
{ ... }:

{

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

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

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:

```
{
  programs.screen.screenrc = ''
    multiuser on
    acladd normal_user
  '';
}
```

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

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

The session name can be changed using [`services.weechat.sessionName`](#).

Taskserver

Table of Contents

[Configuration](#)

[The nixos-taskserver tool](#)

[Declarative/automatic CA management](#)

[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>

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.

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 organisati

v: unstable -

option.

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

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:

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

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.

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.

Sourcehut

Table of Contents

[Basic usage](#)

[Configuration](#)

[Using an alternative webserver as reverse-proxy \(e.g. httpd\)](#)

[Sourcehut](#) is an open-source, self-hostable software development platform. The server setup can be automated using `services.sourcehut`.

Basic usage

Sourcehut is a Python and Go based set of applications. This NixOS module also provides basic configuration integrating Sourcehut into locally running `services.nginx`, `services.redis.servers.sourcehut`, `services.postfix` and `services.postgresql` services.

A very basic configuration may look like this:

```
{ pkgs, ... }:
let
  fqdn =
    let
      join = hostName: domain: hostName + optionalString (domain != null)
      in join config.networking.hostName config.networking.domain v: unstable -
```

```
in {

networking = {
  hostName = "srht";
  domain = "tld";
  firewall.allowedTCPPorts = [ 22 80 443 ];
};

services.sourcehut = {
  enable = true;
  git.enable = true;
  man.enable = true;
  meta.enable = true;
  nginx.enable = true;
  postfix.enable = true;
  postgresql.enable = true;
  redis.enable = true;
  settings = {
    "sr.ht" = {
      environment = "production";
      global-domain = fqdn;
      origin = "https://${fqdn}";
      # Produce keys with srht-keygen from sourcehut.coresrht.
      network-key = "/run/keys/path/to/network-key";
      service-key = "/run/keys/path/to/service-key";
    };
    webhooks.private-key= "/run/keys/path/to/webhook-key";
  };
};

security.acme.certs."${fqdn}".extraDomainNames = [
  "meta.${fqdn}"
  "man.${fqdn}"
  "git.${fqdn}"
];

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

v: unstable -

```
# only recommendedProxySettings are strictly required, but the rest may be
recommendedTlsSettings = true;
recommendedOptimisation = true;
recommendedGzipSettings = true;
recommendedProxySettings = true;

# Settings to setup what certificates are used for which endpoint.
virtualHosts = {
  "${fqdn}".enableACME = true;
  "meta.${fqdn}".useACMEHost = fqdn;
  "man.${fqdn}".useACMEHost = fqdn;
  "git.${fqdn}".useACMEHost = fqdn;
};

};

}
```

The `hostName` option is used internally to configure the nginx reverse-proxy. The `settings` attribute set is used by the configuration generator and the result is placed in `/etc/sr.ht/config.ini`.

Configuration

All configuration parameters are also stored in `/etc/sr.ht/config.ini` which is generated by the module and linked from the store to ensure that all values from `config.ini` can be modified by the module.

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

By default, `nginx` is used as reverse-proxy for `sourcehut`. However, it's possible to use e.g. `httpd` by explicitly disabling `nginx` using [`services.nginx.enable`](#) and fixing the `settings`.

GitLab

Table of Contents

[Prerequisites](#)

v: unstable -

Configuring

Maintenance

GitLab is a feature-rich git hosting service.

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 = {  
    enable = true;  
    recommendedGzipSettings = true;  
    recommendedOptimisation = true;  
    recommendedProxySettings = true;  
    recommendedTlsSettings = true;  
    virtualHosts."git.example.com" = {  
        enableACME = true;  
        forceSSL = true;  
        locations."/".proxyPass = "http://unix:/run/gitlab/gitlab-workhorse.socket";  
    };  
};
```

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:

v: unstable -

```
services.gitlab = {
  enable = true;
  databasePasswordFile = "/var/keys/gitlab/db_password";
  initialRootPasswordFile = "/var/keys/gitlab/root_password";
  https = true;
  host = "git.example.com";
  port = 443;
  user = "git";
  group = "git";
  smtp = {
    enable = true;
    address = "localhost";
    port = 25;
  };
  secrets = {
    dbFile = "/var/keys/gitlab/db";
    secretFile = "/var/keys/gitlab/secret";
    otpFile = "/var/keys/gitlab/otp";
    jwsFile = "/var/keys/gitlab/jws";
  };
  extraConfig = {
    gitlab = {
      email_from = "gitlab-no-reply@example.com";
      email_display_name = "Example GitLab";
      email_reply_to = "gitlab-no-reply@example.com";
      default_projects_features = { builds = false; };
    };
  };
};
```

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 `incoming_mail.enabled` is set to `true` in [extraConfig](#) an additional service called `gitlab-mailroom` is enabled for fetching incoming mail.

Refer to [Appendix A](#) for all available configuration options for the [services.gitlab](#) module.

Maintenance

Backups

Backups can be configured with the options in [services.gitlab.backup](#). Use the [services.gitlab.backup.startAt](#) option to configure regular backups.

To run a manual backup, start the `gitlab-backup` service:

```
$ systemctl start gitlab-backup.service
```

Rake tasks

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.

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

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

Forgejo

Table of Contents

[Migration from Gitea](#)

Forgejo is a soft-fork of gitea, with strong community focus, as well as on self-hosting and federation. [Codeberg](#) is deployed from it.

v: unstable -

See [upstream docs](#).

The method of choice for running forgejo is using [`services.forgejo`](#).

Warning

Running forgejo using `services.gitea.package = pkgs.forgejo` is no longer recommended. If you experience issues with your instance using `services.gitea`, **DO NOT** report them to the `services.gitea` module maintainers. **DO** report them to the `services.forgejo` module maintainers instead.

Migration from Gitea

Note

Migrating is, while not strictly necessary at this point, highly recommended. Both modules and projects are likely to diverge further with each release. Which might lead to an even more involved migration.

Full-Migration

This will migrate the state directory (data), rename and chown the database and delete the gitea user.

Note

This will also change the git remote ssh-url user from `gitea@` to `forgejo@`, when using the host's openssh server (default) instead of the integrated one.

Instructions for PostgreSQL (default). Adapt accordingly for other databases:

```
systemctl stop gitea
mv /var/lib/gitea /var/lib/forgejo
runuser -u postgres -- psql -c '
    ALTER USER gitea RENAME TO forgejo;
    ALTER DATABASE gitea RENAME TO forgejo;
```

v: unstable -

```
nixos-rebuild switch  
systemctl stop forgejo  
chown -R forgejo:forgejo /var/lib/forgejo  
systemctl restart forgejo
```

Alternatively, keeping the gitea user

Alternatively, instead of renaming the database, copying the state folder and changing the user, the forgejo module can be set up to re-use the old storage locations and database, instead of having to copy or rename them. Make sure to disable `services.gitea`, when doing this.

```
services.gitea.enable = false;  
  
services.forgejo = {  
    enable = true;  
    user = "gitea";  
    group = "gitea";  
    stateDir = "/var/lib/gitea";  
    database.name = "gitea";  
    database.user = "gitea";  
};  
  
users.users.gitea = {  
    home = "/var/lib/gitea";  
    useDefaultShell = true;  
    group = "gitea";  
    isSystemUser = true;  
};  
  
users.groups.gitea = {};
```

Apache Kafka

Table of Contents

v: unstable -

Basic Usage

KRaft

Migrating to settings

[Apache Kafka](#) is an open-source distributed event streaming platform

Basic Usage

The Apache Kafka service is configured almost exclusively through its [settings](#) option, with each attribute corresponding to the [upstream configuration manual](#) broker settings.

KRaft

Unlike in Zookeeper mode, Kafka in [KRaft](#) mode requires each log dir to be “formatted” (which means a cluster-specific metadata file must exist in each log dir)

The upstream intention is for users to execute the [storage tool](#) to achieve this, but this module contains a few extra options to automate this:

- [services.apache-kafka.clusterId](#)
- [services.apache-kafka.formatLogDirs](#)
- [services.apache-kafka.formatLogDirsIgnoreFormatted](#)

Migrating to settings

Migrating a cluster to the new `settings`-based changes requires adapting removed options to the corresponding upstream settings.

This means that the upstream [Broker Configs documentation](#) should be followed closely.

Note that dotted options in the upstream docs do *not* correspond to nested Nix attrsets, but instead as quoted top level `settings` attributes, as in `services.apache-kafka.settings.`

v: unstable

~~NOT services.apache-kafka.settings.broker.id.~~

Care should be taken, especially when migrating clusters from the old module, to ensure that the same intended configuration is reproduced faithfully via `settings`.

To assist in the comparison, the final config can be inspected by building the config file itself, ie. with:
`nix-build <nixpkgs/nixos> -A config.services.apache-kafka.configFiles.serverProperties`.

Notable changes to be aware of include:

- Removal of `services.apache-kafka.extraProperties` and `services.apache-kafka.serverProperties`
 - Translate using arbitrary properties using [services.apache-kafka.settings](#)
 - [Upstream docs](#)
 - The intention is for all broker properties to be fully representable via [services.apache-kafka.settings](#).
 - If this is not the case, please do consider raising an issue.
 - Until it can be remedied, you can bail out by using [services.apache-kafka.configFiles.serverProperties](#) to the path of a fully rendered properties file.
- Removal of `services.apache-kafka.hostname` and `services.apache-kafka.port`
 - Translate using: `services.apache-kafka.settings.listeners`
 - [Upstream docs](#)
- Removal of `services.apache-kafka.logDirs`
 - Translate using: `services.apache-kafka.settings."log.dirs"`
 - [Upstream docs](#)
- Removal of `services.apache-kafka.brokerId`
 - Translate using: `services.apache-kafka.settings."broker.id"`

v: unstable -

- [Upstream docs](#)
- Removal of `services.apache-kafka.zookeeper`
 - Translate using: `services.apache-kafka.settings."zookeeper.connect"`
 - [Upstream docs](#)

Anki Sync Server

Table of Contents

[Basic Usage](#)

[Alternatives](#)

[Anki Sync Server](#) is the built-in sync server, present in recent versions of Anki. Advanced users who cannot or do not wish to use AnkiWeb can use this sync server instead of AnkiWeb.

This module is compatible only with Anki versions $\geq 2.1.66$, due to [recent enhancements to the Nix anki package](#).

Basic Usage

By default, the module creates a [systemd](#) unit which runs the sync server with an isolated user using the systemd `DynamicUser` option.

This can be done by enabling the `anki-sync-server` service:

```
{ ... }:

{

  services.anki-sync-server.enable = true;
}
```

It is necessary to set at least one username-password pair under `services.anki-sy` v: unstable -

`server.users`. For example

```
{  
    services.anki-sync-server.users = [  
        {  
            username = "user";  
            passwordFile = /etc/anki-sync-server/user;  
        }  
    ];  
}
```

Here, `passwordFile` is the path to a file containing just the password in plaintext. Make sure to set permissions to make this file unreadable to any user besides root.

By default, the server listen address `services.anki-sync-server.host` is set to localhost, listening on port `services.anki-sync-server.port`, and does not open the firewall. This is suitable for purely local testing, or to be used behind a reverse proxy. If you want to expose the sync server directly to other computers (not recommended in most circumstances, because the sync server doesn't use HTTPS), then set the following options:

```
{  
    services.anki-sync-server.host = "0.0.0.0";  
    services.anki-sync-server.openFirewall = true;  
}
```

Alternatives

The [ankisyncd NixOS module](#) provides similar functionality, but using a third-party implementation, [anki-sync-server-rs](#). According to that project's README, it is “no longer maintained”, and not recommended for Anki 2.1.64+.

Matrix

Table of Contents

v: unstable -

[Synapse Homeserver](#)

[Registering Matrix users](#)

[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.

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, lib, config, ... }:
let
  fqdn = "${config.networking.hostName}.${config.networking.domain}";
  baseUrl = "https://${fqdn}";
  clientConfig."m.homeserver".base_url = baseUrl;
  serverConfig."m.server" = "${fqdn}:443";
  mkWellKnown = data: ''
    default_type application/json;
    add_header Access-Control-Allow-Origin *;
    return 200 '${builtins.toJSON data}';
  '';
in {
```

v: unstable -

```
networking.hostName = "myhostname";
networking.domain = "example.org";
networking.firewall.allowedTCPPorts = [ 80 443 ];

services.postgresql.enable = true;
services.postgresql.initialScript = pkgs.writeText "synapse-init.sql" ''
  CREATE ROLE "matrix-synapse" WITH LOGIN PASSWORD 'synapse';
  CREATE DATABASE "matrix-synapse" WITH OWNER "matrix-synapse"
    TEMPLATE template0
    LC_COLLATE = "C"
    LC_CTYPE = "C";
';

services.nginx = {
  enable = true;
  recommendedTlsSettings = true;
  recommendedOptimisation = true;
  recommendedGzipSettings = true;
  recommendedProxySettings = true;
  virtualHosts = {
    # If the A and AAAA DNS records on example.org do not point on the :8448
    # 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.
    # 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
    # /.well-known redirection target.
    "${config.networking.domain}" = {
      enableACME = true;
      forceSSL = true;
      # This section is not needed if the server_name of matrix-synapse
      # the domain (i.e. example.org from @foo@example.org) and the federation
      # port is 8448.
      # Further reference can be found in the docs about delegation under
      # https://element-hq.github.io/synapse/latest/delegate.html
      locations."= /.well-known/matrix/server".extraConfig = mkWellKnownServerExtraConfig;
      # This is usually needed for homeserver discovery (from https://matrix.org)
      # Further reference can be found in the upstream docs at https://matrix.org/docs/spec/...
    };
  };
};
```

```
# https://spec.matrix.org/latest/client-server-api/#getwell-known-locations."= ./well-known/matrix/client".extraConfig = mkWellKnownLocations;
};

"${fqdn}" = {
    enableACME = true;
    forceSSL = true;
    # It's also possible to do a redirect here or something else, this
    # needed for Matrix. It's recommended though to *not* put* element
    # here, see also the section about Element.
    locations."/".extraConfig = ''
        return 404;
    '';
};

# Forward all Matrix API calls to the synapse Matrix homeserver. A
# *must* be used here.
locations._matrix.proxyPass = "http://[::1]:8008";
# Forward requests for e.g. SSO and password-resets.
locations._synapse/client.proxyPass = "http://[::1]:8008";
};

};

services.matrix-synapse = {
    enable = true;
    settings.server_name = config.networking.domain;
    # The public base URL value must match the `base_url` value set in `config`.
    # The default value here is based on `server_name`, so if your `server_name` is
    # from the value of `fqdn` above, you will likely run into some mismatch
    # in client applications.
    settings.public_baseurl = baseUrl;
    settings.listeners = [
        { port = 8008;
            bind_addresses = [ "::1" ];
            type = "http";
            tls = false;
            x_forwarded = true;
            resources = [ {
                names = [ "client" "federation" ];
                compress = true;
            };
        };
    ];
};
```

v: unstable -

```
    } ];
  }
];
};

}
```

Registering Matrix users

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

```
$ nix-shell -p matrix-synapse
$ register_new_matrix_user -k your-registration-shared-secret http://local
New user localpart: your-username
Password:
Confirm password:
Make admin [no]:
Success!
```

In the example, this would create a user with the Matrix Identifier `@your-username:example.org`.

Warning

When using `services.matrix-synapse.settings.registration_shared_secret`, the secret will end up in the world-readable store. Instead it's recommended to deploy the secret in an additional file like this:

- Create a file with the following contents:

```
registration_shared_secret: your-very-secret-secret
```

- Deploy the file with a secret-manager such as `deployment.keys` from `nixops(1)`

v: unstable -

to e.g. `/run/secrets/matrix-shared-secret` and ensure that it's readable by `matrix-synapse`.

- Include the file like this in your configuration:

```
{  
    services.matrix-synapse.extraConfigFiles = [  
        "/run/secrets/matrix-shared-secret"  
    ];  
}
```

Note

It's also possible to user alternative authentication mechanism such as [LDAP \(via matrix-synapse-ldap3\)](#) or [OpenID](#).

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.

```
{  
    services.nginx.virtualHosts."element.${fqdn}" = {  
        enableACME = true;  
        forceSSL = true;  
        serverAliases = [
```

v: unstable -

```
"element.${config.networking.domain}"  
];  
  
root = pkgs.element-web.override {  
    conf = {  
        default_server_config = clientConfig; # see `clientConfig` from the  
    };  
};  
};  
};  
}
```

Note

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.

Mjolnir (Matrix Moderation Tool)

Table of Contents

[Mjolnir Setup](#)

[Synapse Antispam Module](#)

This chapter will show you how to set up your own, self-hosted [Mjolnir](#) instance.

As an all-in-one moderation tool, it can protect your server from malicious invites, spam messages, and whatever else you don't want. In addition to server-level protection, Mjolnir is great for communities wanting to protect their rooms without having to use their personal accounts for moderation.

The bot by default includes support for bans, redactions, anti-spam, server ACLs, room directory changes, room alias transfers, account deactivation, room shutdown, and more.

v: unstable -

See the [README](#) page and the [Moderator's guide](#) for additional instructions on how to setup and use Mjolnir.

For [additional settings](#) see [the default configuration](#).

Mjolnir Setup

First create a new Room which will be used as a management room for Mjolnir. In this room, Mjolnir will log possible errors and debugging information. You'll need to set this Room-ID in [services.mjolnir.managementRoom](#).

Next, create a new user for Mjolnir on your homeserver, if not present already.

The Mjolnir Matrix user expects to be free of any rate limiting. See [Synapse #6286](#) for an example on how to achieve this.

If you want Mjolnir to be able to deactivate users, move room aliases, shutdown rooms, etc. you'll need to make the Mjolnir user a Matrix server admin.

Now invite the Mjolnir user to the management room.

It is recommended to use [Pantalaimon](#), so your management room can be encrypted. This also applies if you are looking to moderate an encrypted room.

To enable the Pantalaimon E2E Proxy for mjolnir, enable [services.mjolnir.pantalaimon](#). This will autoconfigure a new Pantalaimon instance, which will connect to the homeserver set in [services.mjolnir.homeserverUrl](#) and Mjolnir itself will be configured to connect to the new Pantalaimon instance.

```
{  
    services.mjolnir = {  
        enable = true;  
        homeserverUrl = "https://matrix.domain.tld";  
        pantalaimon = {  
            enable = true;  
            username = "mjolnir";  
            passwordFile = "/run/secrets/mjolnir-password";
```

v: unstable -

```
};

protectedRooms = [
    "https://matrix.to/#!/xxx:domain.tld"
];
managementRoom = "!yyy:domain.tld";
};

}
```

Element Matrix Services (EMS)

If you are using a managed “[Element Matrix Services \(EMS\)](#)” server, you will need to consent to the terms and conditions. Upon startup, an error log entry with a URL to the consent page will be generated.

Synapse Antispam Module

A Synapse module is also available to apply the same rulesets the bot uses across an entire homeserver.

To use the Antispam Module, add `matrix-synapse-plugins.matrix-synapse-mjolnir-antispam` to the Synapse plugin list and enable the `mjolnir.Module` module.

```
{
  services.matrix-synapse = {
    plugins = with pkgs; [
      matrix-synapse-plugins.matrix-synapse-mjolnir-antispam
    ];
    extraConfig = '';
    modules:
      - module: mjolnir.Module
        config:
          # Prevent servers/users in the ban lists from inviting users to
          # server to rooms. Default true.
          block_invites: true
          # Flag messages sent by servers/users in the ban lists as spam.
          # this means that spammy messages will appear as ei v:unstable - :]

```

```
# false.
block_messages: false
# Remove users from the user directory search by filtering matrix
# display names by the entries in the user ban list. Default is true.
block_usernames: false
# The room IDs of the ban lists to honour. Unlike other parts
# this list cannot be room aliases or permalinks. This server
# has to already be joined to the room - Mjolnir will not automatically
# ban these rooms.
ban_lists:
  - "!roomid@example.org"
';
};

}
```

Maubot

Table of Contents

[Configuration](#)

[Maubot](#) is a plugin-based bot framework for Matrix.

Configuration

1. Set `services.maubot.enable` to `true`. The service will use SQLite by default.
2. If you want to use PostgreSQL instead of SQLite, do this:

```
services.maubot.settings.database = "postgresql://maubot@localhost/ma...
```

If the PostgreSQL connection requires a password, you will have to add it later on step 8.

3. If you plan to expose your Maubot interface to the web, do something like this:

v: unstable -

```
services.nginx.virtualHosts."matrix.example.org".locations = {
    "/_matrix/maubot/" = {
        proxyPass = "http://127.0.0.1:${toString config.services.maubot.s
        proxyWebsockets = true;
    };
};

services.maubot.settings.server.public_url = "matrix.example.org";
# do the following only if you want to use something other than /_mat
services.maubot.settings.server.ui_base_path = "/another/base/path";
```

4. Optionally, set `services.maubot.pythonPackages` to a list of `python3` packages to make available for Maubot plugins.
5. Optionally, set `services.maubot.plugins` to a list of Maubot plugins (full list available at <https://plugins.maubot.xyz/>):

```
services.maubot.plugins = with config.services.maubot.package.plugins
    reactbot
    # This will only change the default config! After you create a
    # plugin instance, the default config will be copied into that
    # instance's config in Maubot's database, and further base config
    # changes won't affect the running plugin.
    (rss.override {
        base_config = {
            update_interval = 60;
            max_backoff = 7200;
            spam_sleep = 2;
            command_prefix = "rss";
            admins = [ "@chayleaf:pavluk.org" ];
        };
    })
];
# ...or...
services.maubot.plugins = config.services.maubot.package.plugins.all
# ...or...
services.maubot.plugins = config.services.maubot.package.p v:unstable -'
```

```
# ...or...
services.maubot.plugins = with config.services.maubot.package.plugins
  (weather.override {
    # you can pass base_config as a string
    base_config = ''
      default_location: New York
      default_units: M
      default_language:
      show_link: true
      show_image: false
    '';
  })
];
```

6. Start Maubot at least once before doing the following steps (it's necessary to generate the initial config).

7. If your PostgreSQL connection requires a password, add `database`:

```
postgresql://user:password@localhost/maubot to /var/lib/maubot
/config.yaml. This overrides the Nix-provided config. Even then, don't remove the database
line from Nix config so the module knows you use PostgreSQL!
```

8. To create a user account for logging into Maubot web UI and configuring it, generate a password using the shell command `mkpasswd -R 12 -m bcrypt`, and edit `/var/lib/maubot/config.yaml` with the following:

```
admins:
  admin_username: $2b$12$g.oIStUeUCvI58ebYoVMt0/vb9QZJo81PsmV0omHiN
```

Where `admin_username` is your username, and `$2b...` is the bcrypted password.

9. Optional: if you want to be able to register new users with the Maubot CLI (`mbc`), and your homeserver is private, add your homeserver's registration key to `/var/lib/maubot/config.yaml`:

```
homeservers:
```

v: unstable -

```
matrix.example.org:  
  url: https://matrix.example.org  
  secret: your-very-secret-key
```

10. Restart Maubot after editing `/var/lib/maubot/config.yaml`, and Maubot will be available at https://matrix.example.org/_matrix/maubot. If you want to use the `mbc` CLI, it's available using the `maubot` package (`nix-shell -p maubot`).

Mailman

Table of Contents

[Basic usage with Postfix](#)

[Using with other MTAs](#)

[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.

Basic usage with Postfix

For a basic configuration with Postfix as the MTA, the following settings are suggested:

```
{ config, ... }: {  
  services.postfix = {  
    enable = true;  
    relayDomains = ["hash:/var/lib/mailman/data/postfix_domains"];  
    sslCert = config.security.acme.certs."lists.example.org".directory +  
    sslKey = config.security.acme.certs."lists.example.org".directory + "  
    config = {  
      transport_maps = ["hash:/var/lib/mailman/data/postfix_lmtp"];  
      local_recipient_maps = ["hash:/var/lib/mailman/data/postfix_recipient"];  
    };  
  };  
};
```

```
};

services.mailman = {
    enable = true;
    serve.enable = true;
    hyperkitty.enable = true;
    webHosts = ["lists.example.org"];
    siteOwner = "mailman@example.org";
};

services.nginx.virtualHosts."lists.example.org".enableACME = true;
networking.firewall.allowedTCPPorts = [ 25 80 443 ];
}
```

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.

Using with other MTAs

Mailman also supports other MTA, though with a little bit more configuration. For example, to use Mailman with Exim, you can use the following settings:

```
{ config, ... }: {
    services = {
        mailman = {
            enable = true;
            siteOwner = "mailman@example.org";
        };
    };
}
```

v: unstable -

```
enablePostfix = false;
settings.mta = {
    incoming = "mailman.mta.exim4.LMTP";
    outgoing = "mailman.mta.deliver.deliver";
    lmtp_host = "localhost";
    lmtp_port = "8024";
    smtp_host = "localhost";
    smtp_port = "25";
    configuration = "python:mailman.config.exim4";
};
};

exim = {
    enable = true;
    # You can configure Exim in a separate file to reduce configuration
    config = builtins.readFile ./exim.conf;
};
};

}
```

The exim config needs some special additions to work with Mailman. Currently NixOS can't manage Exim config with such granularity. Please refer to [Mailman documentation](#) for more info on configuring Mailman for working with Exim.

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;
```

v: unstable -

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

Emacs

Table of Contents

[Installing Emacs](#)

[Running Emacs as a Service](#)

[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.

Installing Emacs

Emacs can be installed in the normal way for Nix (see [Package Management](#)). In addition, a NixOS service can be enabled.

The Different Releases of Emacs

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

`emacs`

v: unstable -

The latest stable version of Emacs using the [GTK 2](#) widget toolkit.

`emacs-nox`

Emacs built without any dependency on X11 libraries.

`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](#).

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

```
/*
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
all the dependencies from the nixpkgs repository and build the binary files
without interfering with the host distribution.
```

To build the project, type the following from the current directory:

v: unstable -

```
$ nix-build emacs.nix
```

To run the newly compiled executable:

```
$ ./result/bin/emacs  
*/
```

```
# The first non-comment line in this file indicates that  
# the whole file represents a function.  
{ pkgs ? import <nixpkgs> {} }:
```

let

```
# The let expression below defines a myEmacs binding pointing to the  
# current stable version of Emacs. This binding is here to separate  
# the choice of the Emacs binary from the specification of the  
# required packages.  
myEmacs = pkgs.emacs;  
# This generates an emacsWithPackages function. It takes a single  
# argument: a function from a package set to a list of packages  
# (the packages that will be available in Emacs).  
emacsWithPackages = (pkgs.emacsPackagesFor myEmacs).emacsWithPackages;  
in  
# The rest of the file specifies the list of packages to install. In the  
# example, two packages (magit and zerodark-theme) are taken from  
# MELPA stable.  
emacsWithPackages (epkgs: (with epkgs.melpaStablePackages; [  
    magit          # ; Integrate git <C-x g>  
    zerodark-theme # ; Nicolas' theme  
])  
# Two packages (undo-tree and zoom-fm) are taken from MELPA.  
++ (with epkgs.melpaPackages; [  
    undo-tree      # ; <C-x u> to show the undo tree  
    zoom-fm        # ; increase/decrease font size for all buffers %lt;C-  
])  
# Three packages are taken from GNU ELPA.  
++ (with epkgs.elpaPackages; [  
    auctex         # ; LaTeX mode  
    beacon         # ; highlight my cursor when scrolling
```

v: unstable -

```
nameless      # ; hide current package name everywhere in elisp code
])
# notmuch is taken from a nixpkgs derivation which contains an Emacs mod
++ [
  pkgs.notmuch  # From main packages set
])
```

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
$ ./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 6. Querying Emacs packages

```
nix-env -f "<nixpkgs>" -qaP -A emacs.pkgs.elpaPackages
nix-env -f "<nixpkgs>" -qaP -A emacs.pkgs.melpaPackages
nix-env -f "<nixpkgs>" -qaP -A emacs.pkgs.melpaStablePackages
nix-env -f "<nixpkgs>" -qaP -A emacs.pkgs.orgPackages
```

v: unstable -

If you are on NixOS, you can install this particular Emacs for all users by putting the `emacs.nix` file in `/etc/nixos` and adding it to the list of system packages (see [the section called “Declarative Package Management”](#)). Simply modify your file `configuration.nix` to make it contain:

Example 7. Custom Emacs in `configuration.nix`

```
{  
  environment.systemPackages = [  
    # [...]  
    (import ./emacs.nix { inherit pkgs; })  
  ];  
}
```

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

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

Example 8. Custom Emacs in `~/.config/nixpkgs/config.nix`

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

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

Advanced Emacs Configuration

v: unstable -

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 9. Custom Emacs build

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

After building this file as shown in [Example 5](#), you will get an GTK 3-based Emacs binary pre-loaded with your favorite packages.

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`

Enabling the Service

To install and enable the **systemd** user service for Emacs daemon, add the following to `v: unstable -`

configuration.nix:

```
services.emacs.enable = true;
```

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  
$ 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.

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  
emacsclient --create-frame # opens a new frame (window)  
emacsclient --create-frame --tty # opens a new frame on the current term:
```

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.default`.

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
```

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;
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`.

Configuring Emacs

If you want to only use extension packages from Nixpkgs, you can add (`(setq package-archives nil)`) to your init file.

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).

A Major Mode for Nix Expressions

v: unstable -

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.

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>`.

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

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

```

```
<namespace ns="http://docbook.org/ns/docbook"
    uri="../../nix-profile/share/xml/docbook-5.0/rng/docbookxi.rnc"
-->
</locatingRules>
```

Livebook

Table of Contents

[Basic Usage](#)

[Livebook](#) is a web application for writing interactive and collaborative code notebooks.

Basic Usage

Enabling the `livebook` service creates a user [systemd](#) unit which runs the server.

```
{ ... }:

{
  services.livebook = {
    enableUserService = true;
    environment = {
      LIVEBOOK_PORT = 20123;
      LIVEBOOK_PASSWORD = "mypassword";
    };
    # See note below about security
    environmentFile = "/var/lib/livebook.env";
  };
}
```

Note

v: unstable -

The Livebook server has the ability to run any command as the user it is running under, so securing access to it with a password is highly recommended.

Putting the password in the Nix configuration like above is an easy way to get started but it is not recommended in the real world because the resulting environment variables can be read by unprivileged users. A better approach would be to put the password in some secure user-readable location and set `environmentFile = /home/user/secure/livebook.env`.

The [Livebook documentation](#) lists all the applicable environment variables. It is recommended to at least set `LIVEBOOK_PASSWORD` or `LIVEBOOK_TOKEN_ENABLED=false`.

Extra dependencies

By default, the Livebook service is run with minimum dependencies, but some features require additional packages. For example, the machine learning Kinos require `gcc` and `gnumake`. To add these, use `extraPackages`:

```
services.livebook.extraPackages = with pkgs; [ gcc gnumake ];
```

Blackfire profiler

Source: [modules/services/development/blackfire.nix](#)

Upstream documentation: <https://blackfire.io/docs/introduction>

[Blackfire](#) is a proprietary tool for profiling applications. There are several languages supported by the product but currently only PHP support is packaged in Nixpkgs. The back-end consists of a module that is loaded into the language runtime (called *probe*) and a service (*agent*) that the probe connects to and that sends the profiles to the server.

To use it, you will need to enable the agent and the probe on your server. The exact method will depend on the way you use PHP but here is an example of NixOS configuration for PHP-FPM:

```
let
```

```
php = pkgs.php.withExtensions ({ enabled, all }: enabled ++ [ v: unstable - ]
```

```
blackfire
]);
in {
  # Enable the probe extension for PHP-FPM.
  services.phpfpm = {
    phpPackage = php;
  };

  # Enable and configure the agent.
  services.blackfire-agent = {
    enable = true;
    settings = {
      # You will need to get credentials at https://blackfire.io/my/settings
      # You can also use other options described in https://blackfire.io/docs/api
      server-id = "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX";
      server-token = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
    };
  };

  # Make the agent run on start-up.
  # (WantedBy= from the upstream unit not respected: https://github.com/NixOS/nixos/pull/12345)
  # Alternately, you can start it manually with `systemctl start blackfire-agent`.
  systemd.services.blackfire-agent.wantedBy = [ "phpfpm-foo.service" ];
}
```

On your developer machine, you will also want to install [the client](#) (see `blackfire` package) or the browser extension to actually trigger the profiling.

Athens

Table of Contents

[Configuring](#)

[Basic usage for a caching proxy configuration](#)

Source: [modules/services/development/athens.nix](#)

Upstream documentation: <https://docs.gomods.io/>

[Athens](#) is a Go module datastore and proxy

The main goal of Athens is providing a Go proxy (`$GOPROXY`) in regions without access to <https://proxy.golang.org> or to improve the speed of Go module downloads for CI/CD systems.

Configuring

A complete list of options for the Athens module may be found [here](#).

Basic usage for a caching proxy configuration

A very basic configuration for Athens that acts as a caching and forwarding HTTP proxy is:

```
{  
    services.athens = {  
        enable = true;  
    };  
}
```

If you want to prevent Athens from writing to disk, you can instead configure it to cache modules only in memory:

```
{  
    services.athens = {  
        enable = true;  
        storageType = "memory";  
    };  
}
```

To use the local proxy in Go builds, you can set the proxy as environment variable:

```
{ v: unstable -
```

```
environment.variables = {
  GOPROXY = "http://localhost:3000"
};
```

It is currently not possible to use the local proxy for builds done by the Nix daemon. This might be enabled by experimental features, specifically [configurable-impure-env](#), in upcoming Nix versions.

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 ];
xdg.portal.config.common.default = "gtk";
```

Then, you will need to add a repository, for example, [Flathub](#), either using the following commands:

```
$ 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
$ flatpak install flathub org.freedesktop.Bustle
$ flatpak run org.freedesktop.Bustle
```

Again, GNOME Software offers graphical interface for these tasks.

TigerBeetle

Table of Contents

[Configuring](#)

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

Upstream documentation: <https://docs.tigerbeetle.com/>

TigerBeetle is a distributed financial accounting database designed for mission critical safety and performance.

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

```
services.tigerbeetle.enable = true;
```

When first started, the TigerBeetle service will create its data file at `/var/lib/tigerbeetle` unless the file already exists, in which case it will just use the existing file. If you make changes to the configuration of TigerBeetle after its data file was already created (for example increasing the replica count), you may need to remove the existing file to avoid conflicts.

Configuring

By default, TigerBeetle will only listen on a local interface. To configure it to listen on a different interface (and to configure it to connect to other replicas, if you're creating more than one)

v: unstable -

to set the `addresses` option. Note that the TigerBeetle module won't open any firewall ports automatically, so if you configure it to listen on an external interface, you'll need to ensure that connections can reach it:

```
services.tigerbeetle = {  
    enable = true;  
    addresses = [ "0.0.0.0:3001" ];  
};  
  
networking.firewall.allowedTCPPorts = [ 3001 ];
```

A complete list of options for TigerBeetle can be found [here](#).

PostgreSQL

Table of Contents

[Configuring](#)

[Initializing](#)

[Upgrading](#)

[Options](#)

[Plugins](#)

[JIT \(Just-In-Time compilation\)](#)

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

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

PostgreSQL is an advanced, free relational database.

Configuring

v: unstable -

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

```
services.postgresql.enable = true;  
services.postgresql.package = pkgs.postgresql_15;
```

Note that you are required to specify the desired version of PostgreSQL (e.g. `pkgs.postgresql_15`). 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/$pgsqlSchema`. You can override this using `services.postgresql.dataDir`, e.g.

```
services.postgresql.dataDir = "/data/postgresql";
```

Initializing

As of NixOS 23.11, `services.postgresql.ensureUsers.*.ensurePermissions` has been deprecated, after a change to default permissions in PostgreSQL 15 invalidated most of its previous use cases:

- In `psql < 15`, `ALL PRIVILEGES` used to include `CREATE TABLE`, where in `psql >= 15` that would be a separate permission
- `psql >= 15` instead gives only the database owner create permissions
- Even on `psql < 15` (or databases migrated to `>= 15`), it is recommended to manually assign permissions along these lines
 - <https://www.postgresql.org/docs/release/15.0/>
 - <https://www.postgresql.org/docs/15/ddl-schemas.html#DDL-SCHEMAS-PRIV>

Assigning ownership

Usually, the database owner should be a database user of the same name. This can be done via `unstable -`

```
services.postgresql.ensureUsers.*.ensureDBOwnership = true;
```

If the database user name equals the connecting system user name, postgres by default will accept a passwordless connection via unix domain socket. This makes it possible to run many postgres-backed services without creating any database secrets at all

Assigning extra permissions

For many cases, it will be enough to have the database user be the owner. Until `services.postgresql.ensureUsers.*.ensurePermissions` has been re-thought, if more users need access to the database, please use one of the following approaches:

WARNING: `services.postgresql.initialScript` is not recommended for `ensurePermissions` replacement, as that is *only run on first start of PostgreSQL*.

NOTE: all of these methods may be obsoleted, when `ensure*` is reworked, but it is expected that they will stay viable for running database migrations.

NOTE: please make sure that any added migrations are idempotent (re-runnable).

as superuser

Advantage: compatible with postgres < 15, because it's run as the database superuser `postgres`.

in database `postStart`

Disadvantage: need to take care of ordering yourself. In this example, `mkAfter` ensures that permissions are assigned after any databases from `ensureDatabases` and `extraUser1` from `ensureUsers` are already created.

```
systemd.services.postgresql.postStart = lib.mkAfter ''  
    "$PSQL service1 -c 'GRANT SELECT ON ALL TABLES IN SCHEMA public TO \"${extraUser1}\")'  
    "$PSQL service1 -c 'GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO \"${extraUser1}\")'  
    # ....  
'';
```

in intermediate oneshot service

```
systemd.services."migrate-service1-db1" = {
  serviceConfig.Type = "oneshot";
  requiredBy = "service1.service";
  before = "service1.service";
  after = "postgresql.service";
  serviceConfig.User = "postgres";
  environment.PSQL = "psql --port=${toString services.postgresql.port}";
  path = [ postgresql ];
  script = ''
    $PSQL service1 -c 'GRANT SELECT ON ALL TABLES IN SCHEMA public TO
      $PSQL service1 -c 'GRANT SELECT ON ALL SEQUENCES IN SCHEMA public
      # ....
    ';
};
```

as service user

Advantage: re-uses systemd's dependency ordering;

Disadvantage: relies on service user having grant permission. To be combined with ensureDBOwnership.

in service preStart

```
environment.PSQL = "psql --port=${toString services.postgresql.port}"
path = [ postgresql ];
systemd.services."service1".preStart = ''
  $PSQL -c 'GRANT SELECT ON ALL TABLES IN SCHEMA public TO "extraUser";
  $PSQL -c 'GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO "extraUser";
  # ....
'';
```

in intermediate oneshot service

v: unstable -

```
systemd.services."migrate-service1-db1" = {
  serviceConfig.Type = "oneshot";
  requiredBy = "service1.service";
  before = "service1.service";
  after = "postgresql.service";
  serviceConfig.User = "service1";
  environment.PSQL = "psql --port=${toString services.postgresql.port}";
  path = [ postgresql ];
  script = ''
    $PSQL -c 'GRANT SELECT ON ALL TABLES IN SCHEMA public TO "extraUser1"';
    $PSQL -c 'GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO "extraUser1"';
    # ....
  '';
};
```

Upgrading

Note

The steps below demonstrate how to upgrade from an older version to `pkgs.postgresql_13`. These instructions are also applicable to other versions.

Major PostgreSQL upgrades require a downtime and a few imperative steps to be called. This is the case because each major version has some internal changes in the databases' state during major releases. Because of that, NixOS places the state into `/var/lib/postgresql/<version>`, where each `version` can be obtained like this:

```
$ nix-instantiate --eval -A postgresql_13.psqlSchema
"13"
```

For an upgrade, a script like this can be used to simplify the process:

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

v: unstable -

```
environment.systemPackages = [
  (let
    # XXX specify the postgresql package you'd like to upgrade to.
    # Do not forget to list the extensions you need.
    newPostgres = pkgs.postgresql_13.withPackages (pp: [
      # pp.plv8
    ]);
  in pkgs.writeScriptBin "upgrade-pg-cluster" ''
    set -eux
    # XXX it's perhaps advisable to stop all services that depend on postgresql
    systemctl stop postgresql

    export NEWDATA="/var/lib/postgresql/${newPostgres.psqlSchema}"

    export NEWBIN="${newPostgres}/bin"

    export OLDDATA="${config.services.postgresql.dataDir}"
    export OLDBIN="${config.services.postgresql.package}/bin"

    install -d -m 0700 -o postgres -g postgres "$NEWDATA"
    cd "$NEWDATA"
    sudo -u postgres $NEWBIN/initdb -D "$NEWDATA"

    sudo -u postgres $NEWBIN/pg_upgrade \
      --old-datadir "$OLDDATA" --new-datadir "$NEWDATA" \
      --old-bindir $OLDBIN --new-bindir $NEWBIN \
      "$@"
  '')
];
}
```

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 -`)

v: unstable -

3. Run `upgrade-pg-cluster`. It will stop old postgresql, initialize a new one and migrate the old one to the 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 via `services.postgresql.package`. Rebuild NixOS. This should start new postgres using upgraded data directory and all services you stopped during the upgrade.
5. After the upgrade it's advisable to analyze the new cluster.

- For PostgreSQL ≥ 14, use the `vacuumdb` command printed by the upgrades script.
- For PostgreSQL < 14, run (as `su -l postgres` in the `services.postgresql.dataDir`, in this example `/var/lib/postgresql/13`):

```
$ ./analyze_new_cluster.sh
```

Warning

The next step removes the old state-directory!

```
$ ./delete_old_cluster.sh
```

Options

A complete list of options for the PostgreSQL module may be found [here](#).

Plugins

Plugins collection for each PostgreSQL version can be accessed with `.pkgs`. For example, for `pkgs.postgresql_15` package, its plugin collection is accessed by `pkgs.postgresql_15.pkgs`:

```
$ nix repl '<nixpkgs>'
```

```
Loading '<nixpkgs>'...
```

v: unstable -

Added 10574 variables.

```
nix-repl> postgresql_15.pkgs.<TAB><TAB>
postgresql_15.pkgs.cstore_fdw      postgresql_15.pkgs.pg_repack
postgresql_15.pkgs.pg_auto_failover postgresql_15.pkgs.pg_safeupdate
postgresql_15.pkgs.pg_bigm          postgresql_15.pkgs.pg_similarity
postgresql_15.pkgs.pg_cron          postgresql_15.pkgs.pg_topn
postgresql_15.pkgs.pg_hll           postgresql_15.pkgs.pgjwt
postgresql_15.pkgs.pg_partman      postgresql_15.pkgs.pgroonga
...
...
```

To add plugins via NixOS configuration, set `services.postgresql.extraPlugins`:

```
services.postgresql.package = pkgs.postgresql_12;
services.postgresql.extraPlugins = ps: with ps; [
  pg_repack
  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: {
  postgresql_custom = self.postgresql_12.withPackages (ps: [
    ps.pg_repack
    ps.postgis
  ]);
}
```

Here's a recipe on how to override a particular plugin through an overlay:

```
self: super: {
  postgresql_15 = super.postgresql_15.override { this = self.postgresql_15;
  pkgs = super.postgresql_15.pkgs // {
    pg_repack = super.postgresql_15.pkgs.pg_repack.overrideA
    ...
  };
}
```

```
    name = "pg_repack-v20181024";
    src = self.fetchzip {
      url = "https://github.com/reorg/pg_repack/archive/923fa2f3c709a!";
      sha256 = "17k6hq9xaax87yz79j773qyigm4fwk8z4zh5cyp6z0sxnwfqxxw5";
    };
  });
};

}
```

JIT (Just-In-Time compilation)

JIT-support in the PostgreSQL package is disabled by default because of the ~300MiB closure-size increase from the LLVM dependency. It can be optionally enabled in PostgreSQL with the following config option:

```
{
  services.postgresql.enableJIT = true;
}
```

This makes sure that the jit-setting is set to **on** and a PostgreSQL package with JIT enabled is used. Further tweaking of the JIT compiler, e.g. setting a different query cost threshold via jit_above_cost can be done manually via services.postgresql.settings.

The attribute-names of JIT-enabled PostgreSQL packages are suffixed with _jit, i.e. for each `pkgs.postgresql` (and `pkgs.postgresql_<major>`) in `nixpkgs` there's also a `pkgs.postgresql_jit` (and `pkgs.postgresql_<major>_jit`). Alternatively, a JIT-enabled variant can be derived from a given `postgresql` package via `postgresql.withJIT`. This is also useful if it's not clear which attribute from `nixpkgs` was originally used (e.g. when working with `config.services.postgresql.package` or if the package was modified via an overlay) since all modifications are propagated to `withJIT`. I.e.

```
with import <nixpkgs> {
  overlays = [
    (self: super: {
```

v: unstable -

```
    postgresql = super.postgresql.overrideAttrs (_: { pname = "foobar"; })
  );
};

postgresql.withJIT.pname
```

evaluates to "foobar".

FoundationDB

Table of Contents

[Configuring and basic setup](#)

[Scaling processes and backup agents](#)

[Clustering](#)

[Client connectivity](#)

[Client authorization and TLS](#)

[Backups and Disaster Recovery](#)

[Known limitations](#)

[Options](#)

[Full documentation](#)

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

Upstream documentation: <https://apple.github.io/foundationdb/>

Maintainer: Austin Seipp

Available version(s): 7.1.x

v: unstable -

FoundationDB (or “FDB”) is an open source, distributed, transactional key-value store.

Configuring and basic setup

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

```
services.foundationdb.enable = true;  
services.foundationdb.package = pkgs.foundationdb71; # FoundationDB 7.1.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  
Using cluster file `/etc/foundationdb/fdb.cluster'.
```

The database is available.

```
Welcome to the fdbcli. For help, type `help'.  
fdb> status
```

```
Using cluster file `/etc/foundationdb/fdb.cluster'.
```

Configuration:

Redundancy mode	- single
Storage engine	- memory
Coordinators	- 1

Cluster:

FoundationDB processes	- 1
Machines	- 1

v: unstable -

Memory availability	- 5.4 GB per process on machine with least available memory
Fault Tolerance	- 0 machines
Server time	- 04/20/18 15:21:14

...

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).

```
a@link> cat fdb-status.py
#!/usr/bin/env nix-shell
#! nix-shell -i python -p python pythonPackages.foundationdb71

import fdb
import json

def main():
    fdb.api_version(520)
    db = fdb.open()

    @fdb.transactional
    def get_status(tr):
        return str(tr['\xff\xff/status/json'])

    obj = json.loads(get_status(db))
    print('FoundationDB available: %s' % obj['client']['database_status'])

if __name__ == "__main__":
    main()
a@link> chmod +x fdb-status.py
a@link> ./fdb-status.py
FoundationDB available: True
a@link>
```

v: unstable -

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.

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.

v: unstable -

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 `v:unstable - 1`

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  
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.

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.

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.

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 **fdbackup** 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 nc v: unstable -

the private FoundationDB namespace (and it will not add it dynamically at runtime).

You can now perform a backup:

```
$ sudo -u foundationdb fdbbackup start -t default -d file:///opt/fdb-backup
$ sudo -u foundationdb fdbbackup status -t default
```

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.

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.

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/>.

BorgBackup

Table of Contents

[Configuring](#)

v: unstable -

[Basic usage for a local backup](#)

[Create a borg backup server](#)

[Backup to the borg repository server](#)

[Backup to a hosting service](#)

[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.

Configuring

A complete list of options for the Borgbase module may be found [here](#).

Basic usage for a local backup

A very basic configuration for backing up to a locally accessible directory is:

```
{  
  opt.services.borgbackup.jobs = {  
    { rootBackup = {  
      paths = "/";
      exclude = [ "/nix" "/path/to/local/repo" ];  
      repo = "/path/to/local/repo";  
      doInit = true;  
    };  
  };  
};
```

v: unstable -

```
        encryption = {
            mode = "repokey";
            passphrase = "secret";
        };
        compression = "auto,lzma";
        startAt = "weekly";
    };
}
};

}
```

Warning

If you do not want the passphrase to be stored in the world-readable Nix store, use passCommand. You find an example below.

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.

```
# sudo ssh-keygen -N '' -t ed25519 -f /run/keys/id_ed25519_my_borg_repo
# cat /run/keys/id_ed25519_my_borg_repo
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAID78zm0yA+5uPG40t0hfAy+sLDPU1L4AiIoRYI
```

Add the following snippet to your NixOS configuration:

```
{
  services.borgbackup.repos = {
    my_borg_repo = {
      authorizedKeys = [
        "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAID78zm0yA+5uPG40t0hfAy+sLDPU1L4AiIoRYI"
      ];
      path = "/var/lib/my_borg_repo" ;
    }
  }
};
```

v: unstable -

```
    };
};

}
```

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

```
{
  services.borgbackup.jobs = {
    backupToLocalServer = {
      paths = [ "/etc/nixos" ];
      doInit = true;
      repo = "borg@nixos:." ;
      encryption = {
        mode = "repokey-blake2";
        passCommand = "cat /run/keys/borgbackup_passphrase";
      };
      environment = { BORG_RSH = "ssh -i /run/keys/id_ed25519_my_borg_repo" };
      compression = "auto,lzma";
      startAt = "hourly";
    };
  };
};
```

The following few commands (run as root) let you test your backup.

```
> nixos-rebuild switch
...restarting the following units: polkit.service
> systemctl restart borgbackup-job-backupToLocalServer
> sleep 10
> systemctl restart borgbackup-job-backupToLocalServer
> export BORG_PASSPHRASE=topSecret
> borg list --rsh='ssh -i /run/keys/id_ed25519_my_borg_repo' b... v: unstable -
```

```
nixos-backupToLocalServer-2020-03-30T21:46:17 Mon, 2020-03-30 21:46:19 [84]
nixos-backupToLocalServer-2020-03-30T21:46:30 Mon, 2020-03-30 21:46:32 [e]
```

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.

```
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.

```
sudo borg init --encryption=repokey-blake2 \
--rsh "ssh -i /run/keys/id_ed25519_borgbase" \
zzz2aaaaa@zzz2aaaaa.repo.borgbase.com:repo
```

- Add it to your NixOS configuration, e.g.

```
{
  services.borgbackup.jobs = {
    my_Remote_Backup = {
      paths = [ "/" ];
      exclude = [ "/nix" "'**/.cache'" ];
      repo = "zzz2aaaaa@zzz2aaaaa.repo.borgbase.com:repo";
      encryption = {
        mode = "repokey-blake2";
        passCommand = "cat /run/keys/borgbackup_passphrase";
      };
      environment = { BORG_RSH = "ssh -i /run/keys/id_ed25519_borgbase" };
      compression = "auto,lzma";
      startAt = "daily";
    };
  };
}
```

v: unstable -

```
    };
};

}}
```

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>.

Castopod

Table of Contents

[Quickstart](#)

Castopod is an open-source hosting platform made for podcasters who want to engage and interact with their audience.

Quickstart

Use the following configuration to start a public instance of Castopod on `castopod.example.com` domain:

```
networking.firewall.allowedTCPPorts = [ 80 443 ];
services.castopod = {
  enable = true;
  database.createLocally = true;
  nginx.virtualHost = {
    serverName = "castopod.example.com";
    enableACME = true;
    forceSSL = true;
```

v: unstable -

```
};  
};
```

Go to <https://castopod.example.com/cp-install> to create superadmin account after applying the above configuration.

SSL/TLS Certificates with ACME

Table of Contents

[Prerequisites](#)

[Using ACME certificates in Nginx](#)

[Using ACME certificates in Apache/httpd](#)

[Manual configuration of HTTP-01 validation](#)

[Configuring ACME for DNS validation](#)

[Using DNS validation with web server virtual hosts](#)

[Using ACME with services demanding root owned certificates](#)

[Regenerating certificates](#)

[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.

Prerequisites

v: unstable -

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.defaults.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.defaults.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.

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 this:

```
security.acme.acceptTerms = true;
security.acme.defaults.email = "admin+acme@example.com";
services.nginx = {
    enable = true;
    virtualHosts = {
        "foo.example.com" = {
            forceSSL = true;
            enableACME = true;
            # All serverAliases will be added as extra domain names on the cert:
            serverAliases = [ "bar.example.com" ];
            locations."/" = {
                root = "/var/www";
            }
        }
    }
}
```

v: unstable -

```
};

};

# We can also add a different vhost and reuse the same certificate
# but we have to append extraDomainNames manually beforehand:
# security.acme.certs."foo.example.com".extraDomainNames = [ "baz.exar
"baz.example.com" = {
    forceSSL = true;
    useACMEHost = "foo.example.com";
    locations."/" = {
        root = "/var/www";
    };
};
};

};

};
```

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.

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;
security.acme.defaults.email = "admin+acme@example.com";

# /var/lib/acme/.challenges must be writable by the ACME user
# and readable by the Nginx user. The easiest way to achieve
# this is to add the Nginx user to the ACME group.
users.users.nginx.extraGroups = [ "acme" ];

services.nginx = {
```

v: unstable -

```
enable = true;
virtualHosts = {
  "acmechallenge.example.com" = {
    # Catchall vhost, will redirect users to HTTPS for all vhosts
    serverAliases = [ "*.example.com" ];
    locations."/well-known/acme-challenge" = {
      root = "/var/lib/acme/.challenges";
    };
    locations."/" = {
      return = "301 https://$host$request_uri";
    };
  };
};

# Alternative config for Apache
users.users.wwwrun.extraGroups = [ "acme" ];
services.httpd = {
  enable = true;
  virtualHosts = {
    "acmechallenge.example.com" = {
      # Catchall vhost, will redirect users to HTTPS for all vhosts
      serverAliases = [ "*.example.com" ];
      # /var/lib/acme/.challenges must be writable by the ACME user and re
      # By default, this is the case.
      documentRoot = "/var/lib/acme/.challenges";
      extraConfig = ''
        RewriteEngine On
        RewriteCond %{HTTPS} off
        RewriteCond %{REQUEST_URI} !^/\well-known/acme-challenge [NC]
        RewriteRule (.*) https://{$HTTP_HOST}%{REQUEST_URI} [R=301]
      '';
    };
  };
};
```

Now you need to configure ACME to generate a certificate.

v: unstable -

```
security.acme.certs."foo.example.com" = {
  webroot = "/var/lib/acme/.challenges";
  email = "foo@example.com";
  # Ensure that the web server you use can read the generated certs
  # Take a look at the group option for the web server you choose.
  group = "nginx";
  # Since we have a wildcard vhost to handle port 80,
  # we can generate certs for anything!
  # Just make sure your DNS resolves them.
  extraDomainNames = [ "mail.example.com" ];
};
```

The private key `key.pem` and certificate `fullchain.pem` will be put into `/var/lib/acme/foo.example.com`.

Refer to [Appendix A](#) for all available configuration options for the `security.acme` module.

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 are 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 = {
  enable = true;
  extraConfig = ''
    include "/var/lib/secrets/dnskeys.conf";
  '';
  zones = [
    rec {
      name = "example.com";
      file = "/var/db/bind/${name}";
      master = true;
      extraConfig = "allow-update { key rfc2136key.example.com . } .";
    }
  ];
};
```

v: unstable -

```
];
};

# Now we can configure ACME
security.acme.acceptTerms = true;
security.acme.defaults.email = "admin+acme@example.com";
security.acme.certs."example.com" = {
  domain = "*.{example}.com";
  dnsProvider = "rfc2136";
  environmentFile = "/var/lib/secrets/certs.secret";
  # We don't need to wait for propagation since this is a local DNS server
  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 a systemd service:

```
systemd.services.dns-rfc2136-conf = {
  requiredBy = ["acme-example.com.service" "bind.service"];
  before = ["acme-example.com.service" "bind.service"];
  unitConfig = {
    ConditionPathExists = "!/var/lib/secrets/dnskeys.conf";
  };
  serviceConfig = {
    Type = "oneshot";
    UMask = 0077;
  };
  path = [ pkgs.bind ];
  script = ''
    mkdir -p /var/lib/secrets
    chmod 755 /var/lib/secrets
    tsig-keygen rfc2136key.example.com > /var/lib/secrets/dnskeys.conf
    chown named:root /var/lib/secrets/dnskeys.conf
    chmod 400 /var/lib/secrets/dnskeys.conf

    # extract secret value from the dnskeys.conf
    while read x y; do if [ "$x" = "secret" ]; then secret="$y"; fi
  ''
```

```
cat > /var/lib/secrets/certs.secret << EOF
RFC2136_NAMESERVER='127.0.0.1:53'
RFC2136_TSIG_ALGORITHM='hmac-sha256.'
RFC2136_TSIG_KEY='rfc2136key.example.com'
RFC2136_TSIG_SECRET='$secret'
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.

Using DNS validation with web server virtual hosts

It is possible to use DNS-01 validation with all certificates, including those automatically configured via the Nginx/Apache [enableACME](#) option. This configuration pattern is fully supported and part of the module's test suite for Nginx + Apache.

You must follow the guide above on configuring DNS-01 validation first, however instead of setting the options for one certificate (e.g. [`security.acme.certs.<name>.dnsProvider`](#)) you will set them as defaults (e.g. [`security.acme.defaults.dnsProvider`](#)).

```
# Configure ACME appropriately
security.acme.acceptTerms = true;
security.acme.defaults.email = "admin+acme@example.com";
security.acme.defaults = {
    dnsProvider = "rfc2136";
    environmentFile = "/var/lib/secrets/certs.secret";
    # We don't need to wait for propagation since this is a local DNS server
    dnsPropagationCheck = false;
};

# For each virtual host you would like to use DNS-01 validation
```

```
# set acmeRoot = null
services.nginx = {
  enable = true;
  virtualHosts = {
    "foo.example.com" = {
      enableACME = true;
      acmeRoot = null;
    };
  };
};
```

And that's it! Next time your configuration is rebuilt, or when you add a new virtualHost, it will be DNS-01 validated.

Using ACME with services demanding root owned certificates

Some services refuse to start if the configured certificate files are not owned by root. PostgreSQL and OpenSMTPD are examples of these. There is no way to change the user the ACME module uses (it will always be `acme`), however you can use systemd's `LoadCredential` feature to resolve this elegantly. Below is an example configuration for OpenSMTPD, but this pattern can be applied to any service.

```
# Configure ACME however you like (DNS or HTTP validation), adding
# the following configuration for the relevant certificate.
# Note: You cannot use `systemctl reload` here as that would mean
# the LoadCredential configuration below would be skipped and
# the service would continue to use old certificates.
security.acme.certs."mail.example.com".postRun =
  systemctl restart opensmtpd
';

# Now you must augment OpenSMTPD's systemd service to load
# the certificate files.
systemd.services.opensmtpd.requires = ["acme-finished-mail.example.com.target"]
systemd.services.opensmtpd.serviceConfig.LoadCredential = let
  certDir = config.security.acme.certs."mail.example.com".directory;
in [
  v: unstable -
```

```
"cert.pem:${certDir}/cert.pem"
"key.pem:${certDir}/key.pem"
];

# Finally, configure OpenSMTPD to use these certs.
services.opensmtpd = let
  credsDir = "/run/credentials/opensmtpd.service";
in {
  enable = true;
  setSendmail = false;
  serverConfiguration = ''
    pki mail.example.com cert "${credsDir}/cert.pem"
    pki mail.example.com key "${credsDir}/key.pem"
    listen on localhost tls pki mail.example.com
    action act1 relay host smtp://127.0.0.1:10027
    match for local action act1
  '';
};

};
```

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.

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
systemctl cat acme-example.com.service | grep -Po 'accounts/[^\r\n]+'
```

```
export accountdir="$(!!)"  
# Move this folder to some place else  
mv /var/lib/acme/.lego/$accountdir{,.bak}  
# Recreate the folder using systemd-tmpfiles  
systemd-tmpfiles --create  
# Get a new account and reissue certificates  
# Note: Do this for all certs that share the same account email address  
systemctl start acme-example.com.service
```

Oh my ZSH

Table of Contents

[Basic usage](#)

[Custom additions](#)

[Custom environments](#)

[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.

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`.

```
{  
  programs.zsh.ohMyZsh = {  
    enable = true;  
    plugins = [ "git" "python" "man" ];  
    theme = "agnoster";  
  };
```

v: unstable -

{

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`.

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:

```
{  
  programs.zsh.ohMyZsh.custom = "~/path/to/custom/scripts";  
}
```

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:

```
{ pkgs, ... }:  
{  
  programs.zsh.ohMyZsh.customPkgs = [  
    pkgs.nix-zsh-completions  
    # and even more...  
  ];  
}
```

Internally a single store path will be created using `buildEnv`. Please refer to the docs of [nix-zsh-completions v: unstable](#).

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.

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:

```
{ stdenv, fetchFromGitHub }:

stdenv.mkDerivation rec {
  name = "exemplary-zsh-customization-${version}";
  version = "1.0.0";
  src = fetchFromGitHub {
    # path to the upstream repository
  };

  dontBuild = true;
  installPhase = ''
    mkdir -p $out/share/zsh/site-functions
    cp {themes,plugins} $out/share/zsh
```

v: unstable -

```
cp completions $out/share/zsh/site-functions
';
}
```

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;
```

Digital Bitbox

Table of Contents

[Package](#)

[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;
```

v: unstable -

and bundles the `digitalbitbox` package (see [the section called “Package”](#)), which contains the `dbb-app` and `dbb-cli` binaries, along with the hardware module (see [the section called “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.

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:

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

Hardware

The `digitalbitbox` hardware package enables the udev rules for Digital Bitbox devices and may be installed as follows:

```
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 = {  
    enable = true;  
    package = pkgs.digitalbitbox.override {  
        udevRule51 = "something else";  
    };  
};
```

v: unstable -

Input Methods

Table of Contents

[IBus](#)

[Fcith5](#)

[Nabi](#)

[Uim](#)

[Hime](#)

[Kime](#)

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.
- Fcith5: The next generation of fcith, addons (including engines, dictionaries, skins) can be added using `i18n.inputMethod.fcith5.addons`.
- Nabi: A Korean input method based on XIM.
- Uim: The universal input method, is a library with a XIM bridge.
- Hime: An extremely easy-to-use input method framework.
- Kime: Korean IME

IBus

v: unstable -

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 = {  
    enabled = "ibus";  
    ibus.engines = with pkgs.ibus-engines; [ anthy hangul mozc ];  
};
```

`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 dep v: unstable -

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.

Fcitx5

Fcitx5 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 Fcitx:

```
i18n.inputMethod = {  
    enabled = "fcitx5";  
    fcitx5.addons = with pkgs; [ fcitx5-mozc fcitx5-hangul fcitx5-m17n ];  
};
```

`i18n.inputMethod.fcitx5.addons` is optional and can be used to add extra Fcitx5 addons.

Available extra Fcitx5 addons are:

- Anthy (`fcitx5-anthy`): Anthy is a system for Japanese input method. It converts Hiragana text to Kana Kanji mixed text.
- Chewing (`fcitx5-chewing`): Chewing is an intelligent Zhuyin input method. It is one of the most popular input methods among Traditional Chinese Unix users.
- Hangul (`fcitx5-hangul`): Korean input method.
- Unikey (`fcitx5-unikey`): Vietnamese input method.
- m17n (`fcitx5-m17n`): m17n is an input method that uses input methods and corresponding icons in the m17n database.
- mozc (`fcitx5-mozc`): A Japanese input method from Google.
- table-others (`fcitx5-table-other`): Various table-based input methods.

- `chinese-addons (fcitx5-chinese-addons)`: Various chinese input methods.
- `rime (fcitx5-rime)`: RIME support for fcitx5.

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 = {  
    enabled = "nabi";  
};
```

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:

```
i18n.inputMethod = {  
    enabled = "uim";  
};
```

Note: The `i18n.inputMethod.uim.toolbar` option can be used to choose uim toolbar.

Hime

Hime is an extremely easy-to-use input method framework. It is lightweight, stable, powerful and supports many commonly used input methods, including Cangjie, Zhuyin, Dayi, Rank, Shrimp, Greek, Korean Pinyin, Latin Alphabet, etc...

The following snippet can be used to configure Hime:

v: unstable -

```
i18n.inputMethod = {  
    enabled = "hime";  
};
```

Kime

Kime is Korean IME. it's built with Rust language and let you get simple, safe, fast Korean typing

The following snippet can be used to configure Kime:

```
i18n.inputMethod = {  
    enabled = "kime";  
};
```

Profiles

Table of Contents

[All Hardware](#)

[Base](#)

[Clone Config](#)

[Demo](#)

[Docker Container](#)

[Graphical](#)

[Hardened](#)

[Headless](#)

[Installation Device](#)

[Perless](#)

v: unstable -

Minimal

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:

```
imports = [
  <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.

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.

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 k v: unstable -

modules).

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-installer.nix`.

Demo

This profile just enables a `demo` user, with password `demo`, uid `1000`, `wheel` group and [autologin in the SDDM display manager](#).

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](#).

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.

Hardened

A profile with most (vanilla) hardening options enabled by default, potentially at the cost of stability.

v: unstable -

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.

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.

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.

v: unstable -

Perlless

Warning

If you enable this profile, you will NOT be able to switch to a new configuration and thus you will not be able to rebuild your system with nixos-rebuild!

Render your system completely perlless (i.e. without the perl interpreter). This includes a mechanism so that your build fails if it contains a Nix store path that references the string “perl”.

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](#).

QEMU Guest

This profile contains common configuration for virtual machines running under QEMU (using virtio).

It makes virtio modules available on the initrd and sets the system time from the hardware clock to work around a bug in qemu-kvm.

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 = {  
    apiserver.enable = true;  
    controllerManager.enable = true;  
    scheduler.enable = true;  
    addonManager.enable = true;
```

v: unstable -

```
proxy.enable = true;
flannel.enable = true;
};
```

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:

```
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:

```
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.

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](#)

v: unstable -

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.

Administration

This chapter describes various aspects of managing a running NixOS system, such as how to use the `systemd` service manager.

Table of Contents

v: unstable -

[Service Management](#)

[Rebooting and Shutting Down](#)

[User Sessions](#)

[Control Groups](#)

[Logging](#)

[Necessary system state](#)

[Cleaning the Nix Store](#)

[Container Management](#)

[Troubleshooting](#)

Service Management

Table of Contents

[Interacting with a running systemd](#)

[systemd in NixOS](#)

[Template units](#)

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.

Interacting with a running systemd

The command `systemctl` is the main way to interact with `systemd`. The following paragraphs demonstrate ways to interact with any OS running `systemd` as init system. NixOS is of no exception. The [next section](#) explains NixOS specific things worth knowing.

Without any arguments, `systemctl` the status of active units:

```
$ systemctl
● .mount           loaded active mounted   /
● swapfile.swap   loaded active active   /swapfile
● sshd.service    loaded active running  SSH Daemon
● graphical.target loaded active active   Graphical Interface
...
...
```

You can ask for detailed status information about a unit, for instance, the PostgreSQL database service:

```
$ systemctl status postgresql.service
postgresql.service - PostgreSQL Server
   Loaded: loaded (/nix/store/pn3q73mvh75gsrl8w7fd1fk3fq5qm5mw-unit
   Active: active (running) since Mon, 2013-01-07 15:55:57 CET; 9h
     Main PID: 2390 (postgres)
        CGroup: name=systemd:/system/postgresql.service
                  ├─2390 postgres
                  ├─2418 postgres: writer process
                  ├─2419 postgres: wal writer process
                  ├─2420 postgres: autovacuum launcher process
                  ├─2421 postgres: stats collector process
                  └─2498 postgres: zabbix zabbix [local] idle
```

```
Jan 07 15:55:55 hagbard postgres[2394]: [1-1] LOG:  database system was si
Jan 07 15:55:57 hagbard postgres[2390]: [1-1] LOG:  database system is ready
Jan 07 15:55:57 hagbard postgres[2420]: [1-1] LOG:  autovacuum launcher st
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 belongi

v: unstable -

service, as well as the most recent log messages from the service.

Units can be stopped, started or restarted:

```
# systemctl stop postgresql.service  
# systemctl start postgresql.service  
# 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).

systemd in NixOS

Packages in Nixpkgs sometimes provide systemd units with them, usually in e.g `#pkg-out#/lib/systemd/`. Putting such a package in `environment.systemPackages` doesn't make the service available to users or the system.

In order to enable a systemd system service with provided upstream package, use (e.g):

```
systemd.packages = [ pkgs.packagekit ];
```

Usually NixOS modules written by the community do the above, plus take care of other details. If a module was written for a service you are interested in, you'd probably need only to use `services.#name#.enable = true;`. These services are defined in Nixpkgs' [nixos/modules/directory](#). In case the service is simple enough, the above method should work, and start the service on boot.

User systemd services on the other hand, should be treated differently. Given a package that has a systemd unit file at `#pkg-out#/lib/systemd/user/`, using [`systemd.packages`](#) will make you able to start the service via `systemctl --user start`, but it won't start automatically on login. However, You can imperatively enable it by adding the package's attribute to [`systemd.packages`](#) and then do this (e.g):

```
$ mkdir -p ~/.config/systemd/user/default.target.wants  
$ ln -s /run/current-system/sw/lib/systemd/user/syncthing.serv...
```

```
$ systemctl --user daemon-reload  
$ systemctl --user enable syncthing.service
```

If you are interested in a timer file, use `timers.target.wants` instead of `default.target.wants` in the 1st and 2nd command.

Using `systemctl --user enable syncthing.service` instead of the above, will work, but it'll use the absolute path of `syncthing.service` for the symlink, and this path is in `/nix/store /.../lib/systemd/user/`. Hence [garbage collection](#) will remove that file and you will wind up with a broken symlink in your systemd configuration, which in turn will not make the service / timer start on login.

Template units

systemd supports templated units where a base unit can be started multiple times with a different parameter. The syntax to accomplish this is `service-name@instance-name.service`. Units get the instance name passed to them (see `systemd.unit(5)`). NixOS has support for these kinds of units and for template-specific overrides. A service needs to be defined twice, once for the base unit and once for the instance. All instances must include `overrideStrategy = "asDropin"` for the change detection to work. This example illustrates this:

```
{  
    systemd.services = {  
        "base-unit@".serviceConfig = {  
            ExecStart = "...";  
            User = "...";  
        };  
        "base-unit@instance-a" = {  
            overrideStrategy = "asDropin"; # needed for templates to work  
            wantedBy = [ "multi-user.target" ]; # causes NixOS to manage the instance  
        };  
        "base-unit@instance-b" = {  
            overrideStrategy = "asDropin"; # needed for templates to work  
            wantedBy = [ "multi-user.target" ]; # causes NixOS to manage the instance  
            serviceConfig.User = "root"; # also override something from this config  
        };  
    };  
}
```

```
};  
}
```

Rebooting and Shutting Down

The system can be shut down (and automatically powered off) by doing:

```
# shutdown
```

This is equivalent to running `systemctl poweroff`.

To reboot the system, run

```
# 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.

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 `logindctl` allows querying and manipulating user sessions. For instance, to list all user sessions:

```
$ logindctl  
SESSION          UID  USER           SEAT  
v: unstable -
```

c1	500 eelco	seat0
c3	0 root	seat0
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:

```
$ loginctl session-status c3
c3 - root (0)
    Since: Tue, 2013-01-08 01:17:56 CET; 4min 42s ago
    Leader: 2536 (login)
        Seat: seat0; vc3
        TTY: /dev/tty3
    Service: login; type tty; class user
        State: online
    CGroup: name=systemd:/user/root/c3
            └─ 2536 /nix/store/10mn4xip9n7y9bxqwnsx7xwx2v2g34xn-sha
            ├─ 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
```

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:

```
$ systemctl-cgls
└─user
  └─eelco
    └─c1
      └─ 2567 -:0
        └─ 2682 kdeinit4: kdeinit4 Running...
          └─ ...
            └─ 10851 sh -c less -R
└─system
  └─httpd.service
    └─2444 httpd -f /nix/store/3pyacby5cpr55a03qwbnnndizpciq161-httpd.con...
      └─...
  └─dhcpcd.service
    └─2376 dhcpcd --config /nix/store/f8dif8dsi2yaa70n03xir8r653776ka6-dhc...
      └─...
```

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";
```

v: unstable -

The command `systemd-cgtop` shows a continuously updated list of all cgroups with their CPU and memory usage.

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:

```
$ journalctl -u postgresql.service
-- Logs begin at Mon, 2013-01-07 13:28:01 CET, end at Tue, 2013-01-08 01:0
...
Jan 07 15:44:14 hagbard postgres[2681]: [2-1] LOG:  database system is shi
-- Reboot --
Jan 07 15:45:10 hagbard postgres[2532]: [1-1] LOG:  database system was si
Jan 07 15:45:13 hagbard postgres[2500]: [1-1] LOG:  database system is rea
```

Or to get all messages since the last reboot that have at least a "critical" severity level:

```
$ journalctl -b -p crit
Dec 17 21:08:06 mandark sudo[3673]: pam_unix(sudo:auth): auth could not i
Dec 29 01:30:22 mandark kernel[6131]: [1053513.909444] CPU6: Core temperat
```

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`.

Necessary system state

v: unstable -

Table of Contents

[NixOS](#)

[systemd](#)

[ZFS](#)

Normally – on systems with a persistent `rootfs` – system services can persist state to the filesystem without administrator intervention.

However, it is possible and not-uncommon to create [impermanent systems](#), whose `rootfs` is either a `tmpfs` or reset during boot. While NixOS itself supports this kind of configuration, special care needs to be taken.

NixOS

/nix

NixOS needs the entirety of `/nix` to be persistent, as it includes:

- `/nix/store`, which contains all the system's executables, libraries, and supporting data;
- `/nix/var/nix`, which contains:
 - the Nix daemon's database;
 - roots whose transitive closure is preserved when garbage-collecting the Nix store;
 - system-wide and per-user profiles.

/boot

`/boot` should also be persistent, as it contains:

- the kernel and initrd which the bootloader loads,

v: unstable -

- the bootloader's configuration, including the kernel's command-line which determines the store path to use as system environment.

Users and groups

- `/var/lib/nixos` should persist: it holds state needed to generate stable uids and gids for declaratively-managed users and groups, etc.
- `users.mutableUsers` should be false, or the following files under `/etc` should all persist:
 - `passwd(5)` and `group(5)`,
 - `shadow(5)` and `gshadow(5)`,
 - `subuid(5)` and `subgid(5)`.

systemd

machine-id(5)

`systemd` uses per-machine identifier – [machine-id\(5\)](#) – which must be unique and persistent; otherwise, the system journal may fail to list earlier boots, etc.

`systemd` generates a random `machine-id(5)` during boot if it does not already exist, and persists it in `/etc/machine-id`. As such, it suffices to make that file persistent.

Alternatively, it is possible to generate a random `machine-id(5)`; while the specification allows for any hex-encoded 128b value, `systemd` itself uses [UUIDv4](#), i.e. random UUIDs, and it is thus preferable to do so as well, in case some software assumes `machine-id(5)` to be a UUIDv4. Those can be generated with `uuidgen -r | tr -d -` (`tr` being used to remove the dashes).

Such a `machine-id(5)` can be set by writing it to `/etc/machine-id` or through the kernel's command-line, though NixOS' `systemd` maintainers [discourage](#) the latter approach.

/var/lib/systemd

Moreover, `systemd` expects its state directory – `/var/lib/systemd` – to persist, for: v: unstable -

- [systemd-random-seed\(8\)](#), which loads a 256b “seed” into the kernel’s RNG at boot time, and saves a fresh one during shutdown;
- [systemd.timer\(5\)](#) with **Persistent=yes**, which are then run after boot if the timer would have triggered during the time the system was shut down;
- [systemd-coredump\(8\)](#) to store core dumps there by default; (see [coredump.conf\(5\)](#))
- [systemd-timesyncd\(8\)](#);
- [systemd-backlight\(8\)](#) and [systemd-rfkill\(8\)](#) persist hardware-related state;
- possibly other things, this list is not meant to be exhaustive.

In any case, making `/var/lib/systemd` persistent is recommended.

/var/log/journal/{machine-id}

Lastly, [systemd-journald\(8\)](#) writes the system’s journal in binary form to `/var/log/journal/{machine-id}`; if (locally) persisting the entire log is desired, it is recommended to make all of `/var/log/journal` persistent.

If not, one can set **Storage=volatile** in [journald.conf\(5\)](#) (`services.journald.storage = "volatile";`), which disables journal persistence and causes it to be written to `/run/log/journal`.

ZFS

When using ZFS, `/etc/zfs/zpool.cache` should be persistent (or a symlink to a persistent location) as it is the default value for the [cachefile](#) [property](#).

This cachefile is used on system startup to discover ZFS pools, so ZFS pools holding the `rootfs` and/or early-boot datasets such as `/nix` can be set to `cachefile=none`.

In principle, if there are no other pools attached to the system, `zpool.cache` does not need to be persisted; it is however *strongly recommended* to persist it, in case additional pools are added later on, temporarily or permanently:

v: unstable -

While mishandling the cachefile does not lead to data loss by itself, it may cause zpools not to be imported during boot, and services may then write to a location where a dataset was expected to be mounted.

Cleaning the Nix Store

Table of Contents

[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:

```
# 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;
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.

```
$ nix-env -p /nix/var/nix/profiles/per-user/eelco/profile --delete-generated-profile
```

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.

NixOS Boot Entries

If your `/boot` partition runs out of space, after clearing old profiles you must rebuild your system with `nixos-rebuild boot` or `nixos-rebuild switch` to update the `/boot` partition and clear space.

Container Management

Table of Contents

[Imperative Container Management](#)

[Declarative Container Specification](#)

[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

v: unstable -

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.

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/nixos-containers/foo` and a small configuration file in `/etc/nixos-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 '
  services.openssh.enable = true;
  users.users.root.openssh.authorizedKeys.keys = ["ssh-dss AAAAB3N..."];
'
```

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 v: unstable -
```

```
--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:

```
# 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
foo login: alice
Password: ***
```

With `nixos-container run`, you can execute arbitrary commands in the container:

```
# 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/Lin
```

There are several ways to change the configuration of the container. First, on the host, you can edit `/var/lib/nixos-containers/foo/etc/nixos/configuration.nix`, and run

v: unstable -

```
# nixos-container update foo
```

This will build and activate the new configuration. You can also specify a new configuration on the command line:

```
# nixos-container update foo --config '
  services.httpd.enable = true;
  services.httpd.adminAddr = "foo@example.org";
  networking.firewall.allowedTCPPorts = [ 80 ];
'

# 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
```

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

v: unstable -

```
{ services.postgresql.enable = true;
  services.postgresql.package = pkgs.postgresql_14;
};
};
```

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 = {
  privateNetwork = true;
  hostAddress = "192.168.100.10";
  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/nixos-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`.

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:

v: unstable -

```
# nixos-container show-ip foo  
10.233.4.2  
  
$ 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;  
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.

Troubleshooting

Table of Contents

[Boot Problems](#)

v: unstable -

[Maintenance Mode](#)

[Rolling Back Configuration Changes](#)

[Nix Store Corruption](#)

[Network Problems](#)

This chapter describes solutions to common problems you might encounter when you manage your NixOS system.

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`

v: unstable -

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\(1\)](#).

In addition, these arguments are recognised by the live image only:

live.nixos.passwd=password

Set the password for the **nixos** live user. This can be used for SSH access if there are issues using the terminal.

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.)

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.

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:

```
# /run/current-system/bin/switch-to-configuration boot
```

Second, you can switch to the previous configuration in a running system:

```
# nixos-rebuild switch --rollback
```

This is equivalent to running:

```
# /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
...
lrwxrwxrwx 1 root root 78 Aug 12 13:54 /nix/var/nix/profiles/system-268-l:
```

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.

v: unstable -

If the corruption is in a path in the closure of the NixOS system configuration, you can fix it by doing

```
# 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:

```
# 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.

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:

```
# nixos-rebuild switch --option binary-caches http://my-cache.example.org,
```

Development

This chapter describes how you can modify and extend NixOS.

v: unstable -

Table of Contents

[Getting the Sources](#)

[Writing NixOS Modules](#)

[Building Specific Parts of NixOS](#)

[Experimental feature: Bootspec](#)

[What happens during a system switch?](#)

[Writing NixOS Documentation](#)

[NixOS Tests](#)

[Developing the NixOS Test Driver](#)

[Testing the Installer](#)

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
$ 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 [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 ju v: unstable -

(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  
17.09pre104379.6e0b727 (Hummingbird)  
  
$ git checkout -b local 6e0b727
```

Or, to base your local branch on the latest version available in a NixOS channel:

```
$ git remote update origin  
$ 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  
$ 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.

log out and log in again and it should have been recreated with a link to the root channels.

Writing NixOS Modules

Table of Contents

[Option Declarations](#)

[Options Types](#)

[Option Definitions](#)

[Warnings and Assertions](#)

[Meta Attributes](#)

[Importing Modules](#)

[Replace Modules](#)

[Freeform modules](#)

[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`.

v: unstable -

In [Configuration Syntax](#), we saw the following structure of NixOS modules:

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

{ option definitions
}
```

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: Structure of NixOS Modules](#).

Example 11. Structure of NixOS Modules

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

{

imports =
  [ paths of other modules
  ];

options = {
  option declarations
};

config = {
  option definitions
};
}
```

The meaning of each part is as follows.

- The first line makes the current Nix expression a function. The variable `pkgs` contains Nixpkgs (by default, it takes the `nixpkgs` entry of `NIX_PATH`, see the [Nix manual](#) for further details), while `config` contains the full system configuration. This line can be omitted if there is no reference to `pkgs` and `config` inside the module.

v: unstable -

- This **imports** list enumerates the paths to other NixOS modules that should be included in the evaluation of the system configuration. A default set of modules is defined in the file `modules/module-list.nix`. These don't need to be added in the import list.
- The attribute **options** is a nested set of *option declarations* (described below).
- The attribute **config** is a nested set of *option definitions* (also described below).

[Example: 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`).

Care must be taken when writing systemd services using `Exec*` directives. By default systemd performs substitution on `%<char>` specifiers in these directives, expands environment variables from `$FOO` and `${FOO}`, splits arguments on whitespace, and splits commands on `;`. All of these must be escaped to avoid unexpected substitution or splitting when interpolating into an `Exec*` directive, e.g. when using an `extraArgs` option to pass additional arguments to the service. The functions `utils.escapeSystemdExecArg` and `utils.escapeSystemdExecArgs` are provided for this, see [Example: Escaping in Exec directives](#) for an example. When using these functions system environment substitution should *not* be disabled explicitly.

Example 12. NixOS Module for the “locate” Service

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

with lib;

let
  cfg = config.services.locate;
in {
  options.services.locate = {
    enable = mkOption {
```

v: unstable -

```
type = types.bool;
default = false;
description = ''
  If enabled, NixOS will periodically update the database of
  files used by the locate command.
';
};

interval = mkOption {
  type = types.str;
  default = "02:15";
  example = "hourly";
  description = ''
    Update the locate database at this interval. Updates by
    default at 2:15 AM every day.

  The format is described in
  systemd.time(7).
';
};

# Other options omitted for documentation
};

config = {
  systemd.services.update-locatedb =
  { description = "Update Locate Database";
    path  = [ pkgs.su ];
    script =
    ''
      mkdir -m 0755 -p $(dirname ${toString cfg.output})
      exec updatedb \
        --localuser=${cfg.localuser} \
        ${optionalString (!cfg.includeStore) "--prunepaths='/nix/sto"
        --output=${toString cfg.output} ${concatStringsSep " " cfg.e
    '';
  };
};
```

v: unstable -

```
    systemd.timers.update-locatedb = mkIf cfg.enable
      { description = "Update timer for locate database";
        partOf      = [ "update-locatedb.service" ];
        wantedBy    = [ "timers.target" ];
        timerConfig.OnCalendar = cfg.interval;
      };
    };
}
```

Example 13. Escaping in Exec directives

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

with lib;

let
  cfg = config.services.echo;
  echoAll = pkgs.writeScript "echo-all" ''
    #! ${pkgs.runtimeShell}
    for s in "$@"; do
      printf '%s\n' "$s"
    done
  '';
  args = [ "a%Nything" "lang=\${LANG}" ";" "/bin/sh -c date" ];
in {
  systemd.services.echo =
    { description = "Echo to the journal";
      wantedBy = [ "multi-user.target" ];
      serviceConfig.Type = "oneshot";
      serviceConfig.ExecStart = ''
        ${echoAll} ${utils.escapeSystemdExecArgs args}
      '';
    };
}
```

v: unstable -

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:

```
options = {  
    name = mkOption {  
        type = type specification;  
        default = default value;  
        example = example value;  
        description = lib.mdDoc "Description for use in the NixOS manual."  
    };  
};
```

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 [the section called "Options Types"](#)). This argument is mandatory for nixpkgs modules. Setting this is highly recommended for the sake of documentation and type checking. In case it is not set, a fallback type with unspecified behavior is used.

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.

defaultText

A textual representation of the default value to be rendered verbatim in the manual. Useful if the default value is a complex expression or depends on other values or packages. Use `v: unstable -`

`lib.literalExpression` for a Nix expression, `lib.literalMD` for a plain English description in [Nixpkgs-flavored Markdown](#) format.

example

An example value that will be shown in the NixOS manual. You can use `lib.literalExpression` and `lib.literalMD` in the same way as in `defaultText`.

description

A textual description of the option, in [Nixpkgs-flavored Markdown](#) format, that will be included in the NixOS manual. During the migration process from DocBook it is necessary to mark descriptions written in CommonMark with `lib.mdDoc`. The description may still be written in DocBook (without any marker), but this is discouraged and will be deprecated in the future.

Utility functions for common option patterns

mkEnableOption

Creates an Option attribute set for a boolean value option i.e an option to be toggled on or off.

This function takes a single string argument, the name of the thing to be toggled.

The option's description is "Whether to enable <name>".

For example:

Example 14. `mkEnableOption` usage

```
lib.mkEnableOption (lib.mdDoc "magic")
# is like
lib.mkOption {
  type = lib.types.bool;
  default = false;
  example = true;
  description = lib.mdDoc "Whether to enable magic.";
}
```

mkPackageOption

Usage:

```
mkPackageOption pkgs "name" { default = [ "path" "in" "pkgs" ]; example = }
```

Creates an Option attribute set for an option that specifies the package a module should use for some purpose.

Note: You shouldn't necessarily make package options for all of your modules. You can always overwrite a specific package throughout nixpkgs by using [nixpkgs overlays](#).

The package is specified in the third argument under `default` as a list of strings representing its attribute path in nixpkgs (or another package set). Because of this, you need to pass nixpkgs itself (or a subset) as the first argument.

The second argument may be either a string or a list of strings. It provides the display name of the package in the description of the generated option (using only the last element if the passed value is a list) and serves as the fallback value for the `default` argument.

To include extra information in the description, pass `extraDescription` to append arbitrary text to the generated description. You can also pass an `example` value, either a literal string or an attribute path.

The default argument can be omitted if the provided name is an attribute of pkgs (if name is a string) or a valid attribute path in pkgs (if name is a list).

If you wish to explicitly provide no default, pass `null` as `default`.

Examples:

Example 15. Simple mkPackageOption usage

```
lib.mkPackageOption pkgs "hello" { }  
# is like
```

v: unstable -

```
lib.mkOption {  
    type = lib.types.package;  
    default = pkgs.hello;  
    defaultText = lib.literalExpression "pkgs.hello";  
    description = lib.mdDoc "The hello package to use.";  
}
```

Example 16. `mkPackageOption` with explicit default and example

```
lib.mkPackageOption pkgs "GHC" {  
    default = [ "ghc" ];  
    example = "pkgs.haskell.packages.ghc92.ghc.withPackages (h�gs: [ h�gs.p  
}  
# is like  
lib.mkOption {  
    type = lib.types.package;  
    default = pkgs.ghc;  
    defaultText = lib.literalExpression "pkgs.ghc";  
    example = lib.literalExpression "pkgs.haskell.packages.ghc92.ghc.withPac  
    description = lib.mdDoc "The GHC package to use.";  
}
```

Example 17. `mkPackageOption` with additional description text

```
mkPackageOption pkgs [ "python39Packages" "pytorch" ] {  
    extraDescription = "This is an example and doesn't actually do anything"  
}  
# is like  
lib.mkOption {  
    type = lib.types.package;  
    default = pkgs.python39Packages.pytorch;  
    defaultText = lib.literalExpression "pkgs.python39Packages.p" v: unstable -
```

```
description = "The pytorch package to use. This is an example and doesn
{}
```

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 approaches we could take with this module structure:

- Configuring the display managers independently by adding an enable option to every display manager module backend. (NixOS)
- Configuring 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 only be one enabled at a time, but the type system cannot enforce this restriction as there is no relation between each backend's **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 manager backends in the central module will require changing 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: Extensible type placeholder in the service module](#)), and to extend it in each backend module ([Example: Extending services.xserver.displayManager.enable in the gdm mo](#) v: unstable -

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 enforces that there can only be a single display manager enabled.

Example 18. Extensible type placeholder in the service module

```
services.xserver.displayManager.enable = mkOption {  
    description = "Display manager to use";  
    type = with types; nullOr (enum [ ]);  
};
```

Example 19. Extending `services.xserver.displayManager.enable` in the `gdm` module

```
services.xserver.displayManager.enable = mkOption {  
    type = with types; nullOr (enum [ "gdm" ]);  
};
```

Example 20. Extending `services.xserver.displayManager.enable` in the `sddm` module

```
services.xserver.displayManager.enable = mkOption {  
    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" v: unstable -`

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.

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.bool`

A boolean, its values can be `true` or `false`. All definitions must have the same value, after priorities. An error is thrown in case of a conflict.

`types.boolByOr`

A boolean, its values can be `true` or `false`. The result is `true` if *any* of multiple definitions is `true`. In other words, definitions are merged with the logical *OR* operator.

`types.path`

A filesystem path is anything that starts with a slash when coerced to a string. Even if derivations can be considered as paths, the more specific `types.package` should be preferred.

`types.pathInStore`

A path that is contained in the Nix store. This can be a top-level store path like `pkgs.hello` or a descendant like " `${pkgs.hello}/bin/hello`".

`types.package`

A top-level store path. This can be an attribute set pointing to a store path, like a derivation or a flake input.

`types.enum l`

One element of the list `l`, e.g. `types.enum ["left" "right"]`. Multiple definitions cannot be merged.

`types.anything`

A type that accepts any value and recursively merges attribute sets together. This type is recommended when the option type is unknown.

v: unstable -

Example 21. `types.anything`

Two definitions of this type like

```
{  
  str = lib.mkDefault "foo";  
  pkg.hello = pkgs.hello;  
  fun.fun = x: x + 1;  
}
```

```
{  
  str = lib.mkIf true "bar";  
  pkg.gcc = pkgs.gcc;  
  fun.fun = lib.mkForce (x: x + 2);  
}
```

will get merged to

```
{  
  str = "bar";  
  pkg.gcc = pkgs.gcc;  
  pkg.hello = pkgs.hello;  
  fun.fun = x: x + 2;  
}
```

`types.raw`

A type which doesn't do any checking, merging or nested evaluation. It accepts a single arbitrary value that is not recursed into, making it useful for values coming from outside the module system, such as package sets or arbitrary data. Options of this type are still evaluated according to priorities and conditionals, so `mkForce`, `mkIf` and co. still work on the option value itself, but not for any value nested within it. This type should only be used when checking, merging and nested evaluation are not desirable.

v: unstable -

types.optionType

The type of an option's type. Its merging operation ensures that nested options have the correct file location annotated, and that if possible, multiple option definitions are correctly merged together. The main use case is as the type of the `_module.freeformType` option.

types.attrs

A free-form attribute set.

Warning

This type will be deprecated in the future because it doesn't recurse into attribute sets, silently drops earlier attribute definitions, and doesn't discharge `lib.mkDefault`, `lib.mkIf` and co. For allowing arbitrary attribute sets, prefer `types.attrsOf` `types.anything` instead which doesn't have these problems.

types.pkgs

A type for the top level Nixpkgs package set.

Numeric 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.between *lowest* *highest*

An integer between *lowest* and *highest* (both inclusive).

v: unstable -

types.ints.positive

A positive integer (that is > 0).

types.port

A port number. This type is an alias to `types.ints.u16`.

types.float

A floating point number.

Warning

Converting a floating point number to a string with `toString` or `toJSON` may result in [precision loss](#).

types.number

Either a signed integer or a floating point number. No implicit conversion is done between the two types, and multiple equal definitions will only be merged if they have the same type.

types.numbers.between *lowest* *highest*

An integer or floating point number between *lowest* and *highest* (both inclusive).

types.numbers.nonnegative

A nonnegative integer or floating point number (that is ≥ 0).

types.numbers.positive

A positive integer or floating point number (that is > 0).

String types

types.str

A string. Multiple definitions cannot be merged.

types.separatedString *sep*

A string. Multiple definitions are concatenated with *sep*, e.g. `types.separatedString " | "`.

types.lines

A string. Multiple definitions are concatenated with a new line "`\n`".

v: unstable -

types.commas

A string. Multiple definitions are concatenated with a comma ", ".

types.envVar

A string. Multiple definitions are concatenated with a colon ":".

types.strMatching

A string matching a specific regular expression. Multiple definitions cannot be merged. The regular expression is processed using `builtins.match`.

Submodule types

Submodules are detailed in [Submodule](#).

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; }`.

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.

v: unstable -

- **shorthandOnlyDefinesConfig** Whether definitions of this type should default to the `config` section of a module (see [Example: 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 = { ... }; }.`

`types.deferredModule`

Whereas `submodule` represents an option tree, `deferredModule` represents a module value, such as a module file or a configuration.

It can be set multiple times.

Module authors can use its value in `imports`, in `submoduleWith`'s modules` or in `evalModules`'modules` parameter, among other places.

Note that `imports` must be evaluated before the module fixpoint. Because of this, deferred modules can only be imported into “other” fixpoints, such as submodules.

One use case for this type is the type of a “default” module that allow the user to affect all submodules in an `attrsOf` submodule` at once. This is more convenient and discoverable than expecting the module user to type-merge with the `attrsOf` submodule` option.

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`

v: unstable -

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.unique { message = m } t`

Ensures that type t cannot be merged. Prints the message m , after the line `The option <option path> is defined multiple times.` and before a list of definition locations.

`types.either t1 t2`

Type $t1$ or type $t2$, e.g. with `types; either int str`. Multiple definitions cannot be merged.

v: unstable -

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*

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.

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: Directly defined submodule](#)) or as reference ([Example: Submodule defined as a reference](#)).

Note that even if your submodule's options all have a default value, you will still need to provide a default value (e.g. an empty attribute set) if you want to allow users to leave it undefined.

Example 22. Directly defined submodule

```
options.mod = mkOption {  
    description = "submodule example";  
    type = with types; submodule {  
        options = {  
            foo = mkOption {  
                type = int;  
            };  
            bar = mkOption {  
                type = str;  
            };  
        };  
    };  
};
```

v: unstable -

```
};  
};
```

Example 23. Submodule defined as a reference

```
let  
  modOptions = {  
    options = {  
      foo = mkOption {  
        type = int;  
      };  
      bar = mkOption {  
        type = int;  
      };  
    };  
  };  
in  
  options.mod = mkOption {  
    description = "submodule example";  
    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: Declaration of a list of submodules](#)), `submodule` allows multiple definitions of the submodule option set ([Example: Definition of a list of submodules](#)).

Example 24. Declaration of a list of submodules

```
options.mod = mkOption {  
  description = "submodule example";  
  type = with types; listOf (submodule {  
    options = {  
      foo = mkOption {
```

v: unstable -

```
    type = int;
};

bar = mkOption {
    type = str;
};

};

});

};

};
```

Example 25. Definition of a list of submodules

```
config.mod = [
{ foo = 1; bar = "one"; }
{ foo = 2; bar = "two"; }
];
```

When composed with `attrsOf` ([Example: Declaration of attribute sets of submodules](#)), `submodule` allows multiple named definitions of the submodule option set ([Example: Definition of attribute sets of submodules](#)).

Example 26. Declaration of attribute sets of submodules

```
options.mod = mkOption {
  description = "submodule example";
  type = with types; attrsOf (submodule {
    options = {
      foo = mkOption {
        type = int;
      };
      bar = mkOption {
        type = str;
      };
    };
  });
};
```

v: unstable -

```
});  
};
```

Example 27. Definition of attribute sets of submodules

```
config.mod.one = { foo = 1; bar = "one"; };  
config.mod.two = { foo = 2; bar = "two"; };
```

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: Adding a type check](#)), or to fully override the `check` function ([Example: Overriding a type check](#)).

Example 28. Adding a type check

```
byte = mkOption {  
    description = "An integer between 0 and 255.";  
    type = types.addCheck types.int (x: x >= 0 && x <= 255);  
};
```

Example 29. Overriding a type check

```
nixThings = mkOption {  
    description = "words that start with 'nix'";  
    type = types.str // {  
        v: unstable -
```

```
    check = (x: lib.hasPrefix "nix" x)
  };
};
```

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.

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.

description

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

v: unstable -

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:

v: unstable -

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.

Option Definitions

Option definitions are generally straight-forward bindings of values to option names, like

```
config = {  
    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 {  
    environment.systemPackages = [ ... ];  
    ...  
} 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 {  
    services.httpd.enable = false;  
} else {  
    services.httpd.enable = true;  
};
```

The solution is to write:

```
config = mkIf config.services.httpd.enable {  
    environment.systemPackages = [ ... ];  
    ...  
};
```

The special function `mkIf` causes the evaluation of the conditional to be “pushed down” into the individual definitions, as if you had written:

```
config = {  
    environment.systemPackages = if config.services.httpd.enable then [ ...  
    ...  
];
```

Setting Priorities

A module can override the definitions of an option in other modules by setting an *override priority*. All option definitions that do not have the lowest priority value are discarded. By default, option definitions have priority 100 and option defaults have priority 1500. You can specify an explicit priority by using `mkOverride`, e.g.

```
services.openssh.enable = mkOverride 10 false;
```

v: unstable -

This definition causes all other definitions with priorities above 10 to be discarded. The function `mkForce` is equal to `mkOverride 50`, and `mkDefault` is equal to `mkOverride 1000`.

Ordering Definitions

It is also possible to influence the order in which the definitions for an option are merged by setting an *order priority* with `mkOrder`. The default order priority is 1000. The functions `mkBefore` and `mkAfter` are equal to `mkOrder 500` and `mkOrder 1500`, respectively. As an example,

```
hardware.firmware = mkBefore [ myFirmware ];
```

This definition ensures that `myFirmware` comes before other unordered definitions in the final list value of `hardware.firmware`.

Note that this is different from [override priorities](#): setting an order does not affect whether the definition is included or not.

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
[ # Unconditional stuff.
{ environment.systemPackages = [ ... ];
}
# Conditional stuff.
(mkIf config.services.bla.enable {
    environment.systemPackages = [ ... ];
})
];
```

Warnings and Assertions

v: unstable -

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.

Warnings

This is an example of using `warnings`.

```
{ config, lib, ... }:
{
  config = lib.mkIf config.services.foo.enable {
    warnings =
      if config.services.foo.bar
        then [ "'You have enabled the bar feature of the foo service.
              This is known to cause some specific problems in certain s:
              ''' ]
      else [];
  }
}
```

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, ... }:
{
  config = lib.mkIf config.services.syslogd.enable {
    assertions =
      [ { assertion = !config.services.rsyslogd.enable;
          message = "rsyslogd conflicts with syslogd";
        }
  ]
}
```

v: unstable -

```
];
}

}
```

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`, `doc`, and `buildDocsInSandbox`.

Each of the meta-attributes must be defined at most once per module file.

```
{ config, lib, pkgs, ... }:
{
  options = {
    ...
  };

  config = {
    ...
  };

  meta = {
    maintainers = with lib.maintainers; [ ericsagnes ];
    doc = ./default.md;
    buildDocsInSandbox = true;
  };
}
```

- `maintainers` contains a list of the module maintainers.
- `doc` points to a valid [Nixpkgs-flavored CommonMark](#) file containing the module documentation. Its contents is automatically added to [Configuration](#). Changes to a module documentation have to be checked to not break building the NixOS manual:

v: unstable -

```
$ nix-build nixos/release.nix -A manual.x86_64-linux
```

- **buildDocsInSandbox** indicates whether the option documentation for the module can be built in a derivation sandbox. This option is currently only honored for modules shipped by `nixpkgs`. User modules and modules taken from `NIXOS_EXTRA_MODULE_PATH` are always built outside of the sandbox, as has been the case in previous releases.

Building NixOS option documentation in a sandbox allows caching of the built documentation, which greatly decreases the amount of time needed to evaluate a system configuration that has NixOS documentation enabled. The sandbox also restricts which attributes may be referenced by documentation attributes (such as option descriptions) to the `options` and `lib` module arguments and the `pkgs.formats` attribute of the `pkgs` argument, `config` and the rest of `pkgs` are disallowed and will cause doc build failures when used. This restriction is necessary because we cannot reproduce the full `nixpkgs` instantiation with configuration and overlays from a system configuration inside the sandbox. The `options` argument only includes options of modules that are also built inside the sandbox, referencing an option of a module that isn't built in the sandbox is also forbidden.

The default is `true` and should usually not be changed; set it to `false` only if the module requires access to `pkgs` in its documentation (e.g. because it loads information from a linked package to build an option type) or if its documentation depends on other modules that also aren't sandboxed (e.g. by using types defined in the other module).

Importing Modules

Sometimes NixOS modules need to be used in configuration but exist outside of `Nixpkgs`. These modules can be imported:

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

{
  imports =
    [ # Use a locally-available module definition in
      # ./example-module/default.nix
```

v: unstable -

```
./example-module
];
services.exampleModule.enable = true;
}
```

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
[
  ./example-module1
  ./example-module2
]
```

```
# ./extra-module/default.nix
{ imports = import ./module-list.nix; }
```

```
# NIXOS_EXTRA_MODULE_PATH=/absolute/path/to/extra-module
{ config, lib, pkgs, ... }:

{
  # No `imports` needed

  services.exampleModule1.enable = true;
}
```

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

v: unstable -

`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),
- or an attribute set containing a specific `key` attribute.

The latter allows some modules to be disabled, despite them being distributed via attributes instead of file paths. The `key` should be globally unique, so it is recommended to include a file path in it, or rely on a framework to do it for you.

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, ... }:

{
  disabledModules = [ "services/databases/postgresql.nix" ];

  imports =
    [ # Use postgresql service from nixos-unstable channel.
      # sudo nix-channel --add https://nixos.org/channels/nixos-unstable \
      <nixos-unstable/nixos/modules/services/databases/postgresql.nix>
    ];
  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.

```
{ config, lib, pkgs, ... }: v: unstable -
```

```
with lib;

let
  cfg = config.programs.man;
in

{
  disabledModules = [ "services/programs/man.nix" ];

  options = {
    programs.man.enable = mkOption {
      type = types.bool;
      default = true;
      description = "Whether to enable manual pages.";
    };
  };

  config = mkIf cfg.enabled {
    warnings = [ "disabled manpages for production deployments." ];
  };
}
```

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 30. Freeform submodule

v: unstable -

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: Declaring a type-checked `settings` attribute](#) for a more complete example.

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

  options.settings = lib.mkOption {
    type = lib.types.submodule {

      freeformType = with lib.types; attrsOf str;

      # We want this attribute to be checked for the correct type
      options.port = lib.mkOption {
        type = lib.types.port;
        # Declaring the option also allows defining a default value
        default = 8080;
      };

    };
  };
}
```

And the following shows what such a module then allows

```
{
  # Not a declared option, but the freeform type allows this
  settings.logLevel = "debug";

  # Not allowed because the freeform type only allows strings
  # settings.enable = true;

  # Allowed because there is a port option declared
  settings.port = 80;

  # Not allowed because the port option doesn't allow strings
  v: unstable -
```

```
# settings.port = "443";  
}
```

Note

Freeform attributes cannot depend on other attributes of the same set without infinite recursion:

```
{  
  # This throws infinite recursion encountered  
  settings.logLevel = lib.mkIf (config.settings.port == 80) "debug";  
}
```

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.

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` op

lines to allow arbitrary text after the autogenerated part of the file.

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.javaProperties { comment? "Generated with Nix" }`

A function taking an attribute set with values

`comment`

A string to put at the start of the file in a comment. It can have multiple lines.

It returns the type: `attrsOf str` and a function `generate` to build a Java `.properties` file, taking care of the correct escaping, etc.

`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, listToValue? null, ... }`

A function taking an attribute set with values

`listsAsDuplicateKeys`

A boolean for controlling whether list values can be used to represent duplicate INI keys

`listToValue`

A function for turning a list of values into a single value.

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](#).

v: unstable -

`pkgs.formats.elixirConf { elixir ? pkgs.elixir }`

A function taking an attribute set with values

`elixir`

The Elixir package which will be used to format the generated output

It returns a set with Elixir-Config-specific attributes `type`, `lib`, and `generate` as specified [below](#).

The `lib` attribute contains functions to be used in settings, for generating special Elixir values:

`mkRaw elixirCode`

Outputs the given string as raw Elixir code

`mkGetEnv { envVariable, fallback ? null }`

Makes the configuration fetch an environment variable at runtime

`mkAtom atom`

Outputs the given string as an Elixir atom, instead of the default Elixir binary string. Note: lowercase atoms still needs to be prefixed with :

`mkTuple array`

Outputs the given array as an Elixir tuple, instead of the default Elixir list

`mkMap attrset`

Outputs the given attribute set as an Elixir map, instead of the default Elixir keyword list

These functions all return an attribute set with these values:

`type`

A module system type representing a value of the format

`lib`

Utility functions for convenience, or special interactions with the format. This attribute is optional.

It may contain inside a `types` attribute containing types specific to this format.

`generate filename jsonValue`

A function that can render a value of the format to a file. Returns a file path.

Note

v: unstable -

This function puts the value contents in the Nix store. So this should be avoided for secrets.

Example 31. 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, ... }:
let
  cfg = config.services.foo;
  # Define the settings format used for this program
  settingsFormat = pkgs.formats.json {};
in {

  options.services.foo = {
    enable = lib.mkEnableOption "foo service";

    settings = lib.mkOption {
      # Setting this type allows for correct merging behavior
      type = settingsFormat.type;
      default = {};
      description = ''
        Configuration for foo, see
        <link xlink:href="https://example.com/docs/foo"/>
        for supported settings.
    '';
  };
};

config = lib.mkIf cfg.enable {
  # We can assign some default settings here to make the service work by
  # enabling it. We use `mkDefault` for values that can be changed without
  # problems
  services.foo.settings = {
```

v: unstable -

```
# Fails at runtime without any value set
log_level = lib.mkDefault "WARN";

# We assume systemd's `StateDirectory` is used, so we require this
# therefore no mkDefault
data_path = "/var/lib/foo";

# Since we use this to create a user we need to know the default value
# eval time
user = lib.mkDefault "foo";
};

environment.etc."foo.json".source =
  # The formats generator function takes a filename and the Nix value
  # representing the format value and produces a filepath with that value
  # rendered in the format
  settingsFormat.generate "foo-config.json" cfg.settings;

# 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} = { isSystemUser = true; };

# ...
};

}
```

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 [the section called “Freeform modules”](#).

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.

Example 32. Declaring a type-checked `settings` attribute

```
settings = lib.mkOption {
  type = lib.types.submodule {
    freeformType = settingsFormat.type;

    # Declare an option for the port such that the type is checked and th:
    # is shown in the manual.
    options.port = lib.mkOption {
      type = lib.types.port;
      default = 8080;
      description = ''
        Which port this service should listen on.
      '';
    };
  };
  default = {};
  description = ''
    Configuration for Foo, see
    <link xlink:href="https://example.com/docs/foo"/>
    for supported values.
  '';
};
```

Building Specific Parts of NixOS

With the command `nix-build`, you can build specific parts of your NixOS configuration. This is done as follows:

```
$ cd /path/to/nixpkgs/nixos
$ nix-build -A config.option
```

v: unstable -

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  
$ nix-build -A config.system.build.kernel -o kernel  
$ qemu-system-x86_64 -kernel ./kernel/bzImage -initrd ./initrd/initrd
```

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'
```

v: unstable -

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 /run/systemd/system/tmp-httppd.service  
# systemctl daemon-reload  
# systemctl start tmp-httppd.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-httppd.service` here.

Experimental feature: Bootspec

Table of Contents

[Schema](#)

[Extensions mechanism](#)

[External bootloaders](#)

Bootspec is a experimental feature, introduced in the [RFC-0125 proposal](#), the reference implementation can be found [there](#) in order to standardize bootloader support and advanced boot workflows such as SecureBoot and potentially more.

You can enable the creation of bootspec documents through `boot.bootspec.enable = true`, which will prompt a warning until [RFC-0125](#) is officially merged.

Schema

The bootspec schema is versioned and validated against [a CUE schema file](#) which should considered as the source of truth for your applications.

You will find the current version [here](#).

v: unstable -

Extensions mechanism

Bootspec cannot account for all usecases.

For this purpose, Bootspec offers a generic extension facility [`boot.bootspec.extensions`](#) which can be used to inject any data needed for your usecases.

An example for SecureBoot is to get the Nix store path to `/etc/os-release` in order to bake it into a unified kernel image:

```
{ config, lib, ... }: {  
    boot.bootspec.extensions = {  
        "org.secureboot.osRelease" = config.environment.etc."os-release".source;  
    };  
}
```

To reduce incompatibility and prevent names from clashing between applications, it is **highly recommended** to use a unique namespace for your extensions.

External bootloaders

It is possible to enable your own bootloader through [`boot.loader.external.installHook`](#) which can wrap an existing bootloader.

Currently, there is no good story to compose existing bootloaders to enrich their features, e.g. SecureBoot, etc. It will be necessary to reimplement or reuse existing parts.

What happens during a system switch?

Table of Contents

[Unit handling](#)

[Activation script](#)

[Non Switchable Systems](#)

[/etc via overlay filesystem](#)

Running `nixos-rebuild switch` is one of the more common tasks under NixOS. This chapter explains some of the internals of this command to make it simpler for new module developers to configure their units correctly and to make it easier to understand what is happening and why for curious administrators.

`nixos-rebuild`, like many deployment solutions, calls `switch-to-configuration` which resides in a NixOS system at `$out/bin/switch-to-configuration`. The script is called with the action that is to be performed like `switch`, `test`, `boot`. There is also the `dry-activate` action which does not really perform the actions but rather prints what it would do if you called it with `test`. This feature can be used to check what service states would be changed if the configuration was switched to.

If the action is `switch` or `boot`, the bootloader is updated first so the configuration will be the next one to boot. Unless `NIXOS_NO_SYNC` is set to 1, `/nix/store` is synced to disk.

If the action is `switch` or `test`, the currently running system is inspected and the actions to switch to the new system are calculated. This process takes two data sources into account: `/etc/fstab` and the current systemd status. Mounts and swaps are read from `/etc/fstab` and the corresponding actions are generated. If the options of a mount are modified, for example, the proper `.mount` unit is reloaded (or restarted if anything else changed and it's neither the root mount or the nix store). The current systemd state is inspected, the difference between the current system and the desired configuration is calculated and actions are generated to get to this state. There are a lot of nuances that can be controlled by the units which are explained here.

After calculating what should be done, the actions are carried out. The order of actions is always the same:

v: unstable -

- Stop units (`systemctl stop`)
- Run activation script (`$out/activate`)
- See if the activation script requested more units to restart
- Restart systemd if needed (`systemd daemon-reexec`)
- Forget about the failed state of units (`systemctl reset-failed`)
- Reload systemd (`systemctl daemon-reload`)
- Reload systemd user instances (`systemctl --user daemon-reload`)
- Reactivate sysinit (`systemctl restart sysinit-reactivation.target`)
- Reload units (`systemctl reload`)
- Restart units (`systemctl restart`)
- Start units (`systemctl start`)
- Inspect what changed during these actions and print units that failed and that were newly started

By default, some units are filtered from the outputs to make it less spammy. This can be disabled for development or testing by setting the environment variable `STC_DISPLAY_ALL_UNITS=1`

Most of these actions are either self-explaining but some of them have to do with our units or the activation script. For this reason, these topics are explained in the next sections.

Unit handling

To figure out what units need to be started/stopped/restarted/reloaded, the script first checks the current state of the system, similar to what `systemctl list-units` shows. For each of the units, the script goes through the following checks:

- Is the unit file still in the new system? If not, `stop` the service unless it sets `X-StopOnRemoval` in the `[Unit]` section to `false`.
- Is it a `.target` unit? If so, `start` it unless it sets `RefuseManualStart` in the `[Unit]` v: unstable -

`true` or `X-OnlyManualStart` in the `[Unit]` section to `true`. Also `stop` the unit again unless it sets `X-StopOnReconfiguration` to `false`.

- Are the contents of the unit files different? They are compared by parsing them and comparing their contents. If they are different but only `X-Reload-Triggers` in the `[Unit]` section is changed, `reload` the unit. The NixOS module system allows setting these triggers with the option `systemd.services.<name>.reloadTriggers`. There are some additional keys in the `[Unit]` section that are ignored as well. If the unit files differ in any way, the following actions are performed:
 - `.path` and `.slice` units are ignored. There is no need to restart them since changes in their values are applied by systemd when systemd is reloaded.
 - `.mount` units are `reloaded` if only their `Options` changed. If anything else changed (like `What`), they are `restarted` unless they are the mount unit for `/` or `/nix` in which case they are reloaded to prevent the system from crashing. Note that this is the case for `.mount` units and not for mounts from `/etc/fstab`. These are explained in [*What happens during a system switch?*](#).
 - `.socket` units are currently ignored. This is to be fixed at a later point.
 - The rest of the units (mostly `.service` units) are then `reloaded` if `X-ReloadIfChanged` in the `[Service]` section is set to `true` (exposed via `systemd.services.<name>.reloadIfChanged`). A little exception is done for units that were deactivated in the meantime, for example because they require a unit that got stopped before. These are `started` instead of reloaded.
 - If the reload flag is not set, some more flags decide if the unit is skipped. These flags are `X-RestartIfChanged` in the `[Service]` section (exposed via `systemd.services.<name>.restartIfChanged`), `RefuseManualStop` in the `[Unit]` section, and `X-OnlyManualStart` in the `[Unit]` section.
 - Further behavior depends on the unit having `X-StopIfChanged` in the `[Service]` section set to `true` (exposed via `systemd.services.<name>.stopIfChanged`). This is set to `true` by default and must be explicitly turned off if not wanted. If the flag is enabled, the unit is `stopped` and then `started`. If not, the unit is `restarted`. The goal of the flag is to make sure that the new unit never runs in the old environment which is still in place before the activation script is run. This behavior is different when the service is socket-activated, as outlined in the following steps.
 - The last thing that is taken into account is whether the unit is a service and socket-a v: unstable -

X-StopIfChanged is **not** set, the service is **restart**ed with the others. If it is set, both the service and the socket are **stop**ped and the socket is **start**ed, leaving socket activation to start the service when it's needed.

Sysinit reactivation

[**sysinit.target**](#) is a systemd target that encodes system initialization (i.e. early startup). A few units that need to run very early in the bootup process are ordered to finish before this target is reached. Probably the most notable one of these is [**systemd-tmpfiles-setup.service**](#). We will refer to these units as "sysinit units".

"Normal" systemd units, by default, are ordered AFTER [**sysinit.target**](#). In other words, these "normal" units expect all services ordered before [**sysinit.target**](#) to have finished without explicitly declaring this dependency relationship for each dependency. See the [systemd bootup](#) for more details on the bootup process.

When restarting both a unit ordered before [**sysinit.target**](#) as well as one after, this presents a problem because they would be started at the same time as they do not explicitly declare their dependency relations.

To solve this, NixOS has an artificial [**sysinit-reactivation.target**](#) which allows you to ensure that services ordered before [**sysinit.target**](#) are restarted correctly. This applies both to the ordering between these sysinit services as well as ensuring that sysinit units are restarted before "normal" units.

To make an existing sysinit service restart correctly during system switch, you have to declare:

```
systemd.services.my-sysinit = {
    requiredBy = [ "sysinit-reactivation.target" ];
    before = [ "sysinit-reactivation.target" ];
    restartTriggers = [ config.environment/etc."my-sysinit.d".source ];
};
```

You need to configure appropriate **restartTriggers** specific to your service.

Activation script

The activation script is a bash script called to activate the new configuration which resides in a NixOS system in `$out/activate`. Since its contents depend on your system configuration, the contents may differ. This chapter explains how the script works in general and some common NixOS snippets. Please be aware that the script is executed on every boot and system switch, so tasks that can be performed in other places should be performed there (for example letting a directory of a service be created by systemd using mechanisms like `StateDirectory`, `CacheDirectory`, ... or if that's not possible using `preStart` of the service).

Activation scripts are defined as snippets using `system.activationScripts`. They can either be a simple multiline string or an attribute set that can depend on other snippets. The builder for the activation script will take these dependencies into account and order the snippets accordingly. As a simple example:

```
system.activationScripts.my-activation-script = {
  deps = [ "etc" ];
  # supportsDryActivation = true;
  text = ''
    echo "Hallo i bims"
  '';
};
```

This example creates an activation script snippet that is run after the `etc` snippet. The special variable `supportsDryActivation` can be set so the snippet is also run when `nixos-rebuild dry-activate` is run. To differentiate between real and dry activation, the `$NIXOS_ACTION` environment variable can be read which is set to `dry-activate` when a dry activation is done.

An activation script can write to special files instructing `switch-to-configuration` to restart/reload units. The script will take these requests into account and will incorporate the unit configuration as described above. This means that the activation script will “fake” a modified unit file and `switch-to-configuration` will act accordingly. By doing so, configuration like `systemd.services.<name>.restartIfChanged` is respected. Since the activation script is run **after** services are already stopped, `systemd.services.<name>.stopIfChanged` cannot be taken into account anymore and the unit is always restarted instead of being stopped and started afterward

The files that can be written to are `/run/nixos/activation-restart-list` and `/run/nixos/activation-reload-list` with their respective counterparts for dry activation being `/run/nixos/dry-activation-restart-list` and `/run/nixos/dry-activation-reload-list`. Those files can contain newline-separated lists of unit names where duplicates are being ignored. These files are not created automatically and activation scripts must take the possibility into account that they have to create them first.

NixOS snippets

There are some snippets NixOS enables by default because disabling them would most likely break your system. This section lists a few of them and what they do:

- `binsh` creates `/bin/sh` which points to the runtime shell
- `etc` sets up the contents of `/etc`, this includes systemd units and excludes `/etc/passwd`, `/etc/group`, and `/etc/shadow` (which are managed by the `users` snippet)
- `hostname` sets the system's hostname in the kernel (not in `/etc`)
- `modprobe` sets the path to the `modprobe` binary for module auto-loading
- `nix` prepares the nix store and adds a default initial channel
- `specialfs` is responsible for mounting filesystems like `/proc` and `sys`
- `users` creates and removes users and groups by managing `/etc/passwd`, `/etc/group` and `/etc/shadow`. This also creates home directories
- `usrbinenv` creates `/usr/bin/env`
- `var` creates some directories in `/var` that are not service-specific
- `wrappers` creates setuid wrappers like `sudo`

Non Switchable Systems

In certain systems, most notably image based appliances, updates are handled outside the system. This means that you do not need to rebuild your configuration on the system itself anymore.

v: unstable -

If you want to build such a system, you can use the `image-based-appliance` profile:

```
{ modulesPath, ... }: {
  imports = [ "${modulesPath}/profiles/image-based-appliance.nix" ]
}
```

The most notable deviation of this profile from a standard NixOS configuration is that after building it, you cannot switch to the configuration anymore. The profile sets `config.system.switch.enable = false;`, which excludes `switch-to-configuration`, the central script called by `nixos-rebuild`, from your system. Removing this script makes the image lighter and slightly more secure.

/etc via overlay filesystem

Note

This is experimental and requires a kernel version ≥ 6.6 because it uses new overlay features and relies on the new mount API.

Instead of using a custom perl script to activate `/etc`, you activate it via an overlay filesystem:

```
system.etc.overlay.enable = true;
```

Using an overlay has two benefits:

1. it removes a dependency on perl
2. it makes activation faster (up to a few seconds)

By default, the `/etc` overlay is mounted writable (i.e. there is a writable upper layer). However, you can also mount `/etc` immutably (i.e. read-only) by setting:

```
system.etc.overlay.mutable = false;
```

The overlay is atomically replaced during system switch. However, files that have been modified will v: unstable -

NOT be overwritten. This is the biggest change compared to the perl-based system.

If you manually make changes to `/etc` on your system and then switch to a new configuration where `system.etc.overlay.mutable = false;`, you will not be able to see the previously made changes in `/etc` anymore. However the changes are not completely gone, they are still in the upperdir of the previous overlay in `./.rw-etc/upper`.

Writing NixOS Documentation

Table of Contents

[Building the Manual](#)

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.

Building the Manual

The 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  
$ nix-shell  
nix-shell$ devmode
```

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 v: unstable -

will be accessible through the `result` symlink at `./result/share/doc/nixos/index.html`.

NixOS Tests

Table of Contents

[Writing Tests](#)

[Running Tests](#)

[Running Tests interactively](#)

[Linking NixOS tests to packages](#)

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.

Writing Tests

A NixOS test is a module that has the following structure:

```
{  
  
  # One or more machines:  
  nodes =  
    { machine =  
        { config, pkgs, ... }: { ... };  
        machine2 =  
          { config, pkgs, ... }: { ... };  
          ...  
    };  
  
  testScript =  
    ''  
      Python code...
```

v: unstable -

```
    ..;  
}
```

We refer to the whole test above as a test module, whereas the values in `nodes.<name>` are NixOS modules themselves.

The option `testScript` is a piece 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 option `nodes`.

An example of a single-node test is `login.nix`. It 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. An interesting multi-node test is `nfs/simple.nix`. It uses two client nodes to test correct locking across server crashes.

Calling a test

Tests are invoked differently depending on whether the test is part of NixOS or lives in a different project.

Testing within NixOS

Tests that are part of NixOS are added to `nixos/tests/all-tests.nix`.

```
hostname = runTest ./hostname.nix;
```

Overrides can be added by defining an anonymous module in `all-tests.nix`.

```
hostname = runTest {  
    imports = [ ./hostname.nix ];  
    defaults.networking.firewall.enable = false;  
};
```

You can run a test with attribute name `hostname` in `nixos/tests/all-tests.nix` by invoking:

v: unstable -

```
cd /my/git/clone/of/nixpkgs  
nix-build -A nixosTests.hostname
```

Testing outside the NixOS project

Outside the `nixpkgs` repository, you can instantiate the test by first importing the NixOS library,

```
let nixos-lib = import (nixpkgs + "/nixos/lib") { };  
in  
  
nixos-lib.runTest {  
    imports = [ ./test.nix ];  
    hostPkgs = pkgs; # the Nixpkgs package set used outside the VMs  
    defaults.services.foo.package = mypkg;  
}
```

`runTest` returns a derivation that runs the test.

Configuring the nodes

There are a few special NixOS 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 `v: unstable -`

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 specified a node `nodes.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()
machine.wait_for_unit("default.target")
if not "Linux" in machine.succeed("uname"):
    raise Exception("Wrong OS")
```

The first line is technically 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()
```

If the hostname of a node contains characters that can't be used in a Python variable name, those characters will be replaced with underscores in the variable name, so `nodes.machine-a` will be exposed to Python as `machine_a`.

Machine objects

The following methods are available on machine objects:

`block()`

Simulate unplugging the Ethernet cable that connects the machine to the other machines. This happens by shutting down `eth1` (the multicast interface used to talk to the other VMs). `eth0` is kept online to still enable the test driver to communicate with the machine.

`console_interact()`

Allows you to directly interact with QEMU's `stdin`, by forwarding terminal input to the QEMU process. This is for use with the interactive test driver, not for production tests, which run unattended. Output from QEMU is only read line-wise. `Ctrl-c` kills QEMU and `Ctrl-d` closes console and returns to the test runner.

v: unstable -

copy_from_host(source, target)

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. Note that the “host” refers to the environment in which the test driver runs, which is typically the Nix build sandbox.

The second argument is the location of the file on the machine that will be written to.

The file is copied via the `shared_dir` directory which is shared among all the VMs (using a temporary directory). The access rights bits will mimic the ones from the host file and user:group will be root:root.

copy_from_host_via_shell(source, target)

Copy a file from the host into the guest by piping it over the shell into the destination file. Works without host-guest shared folder. Prefer `copy_from_host` for whenever possible.

copy_from_vm(source, target_dir)

Copy a file from the VM (specified by an in-VM source path) to a path relative to `$out`. The file is copied via the `shared_dir` shared among all the VMs (using a temporary directory).

crash()

Simulate a sudden power failure, by telling the VM to exit immediately.

dump_tty_contents(tty)

Debugging: Dump the contents of the TTY<n>

execute(command, check_return, check_output, timeout)

Execute a shell command, returning a list (`status, stdout`).

Commands are run with `set -euo pipefail` set:

- If several commands are separated by ; and one fails, the command as a whole will fail.
- For pipelines, the last non-zero exit status will be returned (if there is one; otherwise zero will be returned).
- Dereferencing unset variables fails the command.

v: unstable -

- It will wait for stdout to be closed.

If the command detaches, it must close stdout, as `execute` will wait for this to consume all output reliably. This can be achieved by redirecting stdout to stderr `>&2`, to `/dev/console`, `/dev/null` or a file. Examples of detaching commands are `sleep 365d &`, where the shell forks a new process that can write to stdout and `xclip -i`, where the `xclip` command itself forks without closing stdout.

Takes an optional parameter `check_return` that defaults to `True`. Setting this parameter to `False` will not check for the return code and return `-1` instead. This can be used for commands that shut down the VM and would therefore break the pipe that would be used for retrieving the return code.

A timeout for the command can be specified (in seconds) using the optional `timeout` parameter, e.g., `execute(cmd, timeout=10)` or `execute(cmd, timeout=None)`. The default is 900 seconds.

`fail()`

Like `succeed`, but raising an exception if the command returns a zero status.

`forward_port(host_port, guest_port)`

Forward a TCP port on the host to a TCP port on the guest. Useful during interactive testing.

`get_screen_text()`

Return a textual representation of what is currently visible on the machine's screen using optical character recognition.

Note

This requires `enableOCR` to be set to `true`.

`get_screen_text_variants()`

Return a list of different interpretations of what is currently visible on the machine's screen using optical character recognition. The number and order of the interpretations is not specified and is subject to change, but if no exception is raised at least one will be returned.

Note

This requires `enableOCR` to be set to `true`.

`reboot()`

Press Ctrl+Alt+Delete in the guest.

Prepares the machine to be reconnected which is useful if the machine was started with

`allow_reboot = True`

`screenshot(filename)`

Take a picture of the display of the virtual machine, in PNG format. The screenshot will be available in the derivation output.

`send_chars(chars, delay)`

Simulate typing a sequence of characters on the virtual keyboard, e.g., `send_chars("foobar")` will type the string `foobar` followed by the Enter key.

`send_console(chars)`

Send keys to the kernel console. This allows interaction with the systemd emergency mode, for example. Takes a string that is sent, e.g., `send_console("\n\nsystemctl default\n")`.

`send_key(key, delay, log)`

Simulate pressing keys on the virtual keyboard, e.g., `send_key("ctrl-alt-delete")`.

Please also refer to the QEMU documentation for more information on the input syntax:

https://en.wikibooks.org/wiki/QEMU/Monitor#sendkey_keys

`send_monitor_command(command)`

Send a command to the QEMU monitor. This allows attaching virtual USB disks to a running machine, among other things.

`shell_interact(address)`

Allows you to directly interact with the guest shell. This should only be used during test development, not in production tests. Killing the interactive session with `Ctrl-d` or `Ctrl-c` also ends the guest session.

shutdown()

Shut down the machine, waiting for the VM to exit.

start(allow_reboot)

Start the virtual machine. This method is asynchronous – it does not wait for the machine to finish booting.

succeed()

Execute a shell command, raising an exception if the exit status is not zero, otherwise returning the standard output. Similar to `execute`, except that the timeout is `None` by default. See `execute` for details on command execution.

switch_root()

Transition from stage 1 to stage 2. This requires the machine to be configured with `testing.initrdBackdoor = true` and `boot.initrd.systemd.enable = true`.

systemctl(q, user)

Runs `systemctl` commands with optional support for `systemctl --user`

```
# run `systemctl list-jobs --no-pager`  
machine.systemctl("list-jobs --no-pager")  
  
# spawn a shell for `any-user` and run  
# `systemctl --user list-jobs --no-pager`  
machine.systemctl("list-jobs --no-pager", "any-user")
```

unblock()

Undo the effect of `block`.

wait_for_closed_port(port, addr, timeout)

Wait until nobody is listening on the given TCP port and IP address (default `localhost`).

wait_for_console_text(regex, timeout)

Wait until the supplied regular expressions match a line of the serial console output. This method is useful when OCR is not possible or inaccurate.

wait_for_file(filename, timeout)

v: unstable -

Waits until the file exists in the machine's file system.

`wait_for_open_port(port, addr, timeout)`

Wait until a process is listening on the given TCP port and IP address (default `localhost`).

`wait_for_open_unix_socket(addr, is_datagram, timeout)`

Wait until a process is listening on the given UNIX-domain socket (default to a UNIX-domain stream socket).

`wait_for_qmp_event(event_filter, timeout)`

Wait for a QMP event which you can filter with the `event_filter` function. The function takes as an input a dictionary of the event and if it returns True, we return that event, if it does not, we wait for the next event and retry.

It will skip all events received in the meantime, if you want to keep them, you have to do the bookkeeping yourself and store them somewhere.

By default, it will wait up to 10 minutes, `timeout` is in seconds.

`wait_for_text(regex, timeout)`

Wait until the supplied regular expressions matches the textual contents of the screen by using optical character recognition (see `get_screen_text` and `get_screen_text_variants`).

Note

This requires `enableOCR` to be set to `true`.

`wait_for_unit(unit, user, timeout)`

Wait for a systemd unit to get into "active" state. Throws exceptions on "failed" and "inactive" states as well as after timing out.

`wait_for_window(regexp, timeout)`

Wait until an X11 window has appeared whose name matches the given regular expression, e.g., `wait_for_window("Terminal")`.

`wait_for_x(timeout)`

Wait until it is possible to connect to the X server.

v: unstable -

`wait_until_fails(command, timeout)`

Like `wait_until_succeeds`, but repeating the command until it fails.

`wait_until_succeeds(command, timeout)`

Repeat a shell command with 1-second intervals until it succeeds. Has a default timeout of 900 seconds which can be modified, e.g. `wait_until_succeeds(cmd, timeout=10)`. See `execute` for details on command execution. Throws an exception on timeout.

`wait_until_tty_matches(tty, regexp, timeout)`

Wait until the visible output on the chosen TTY matches regular expression. Throws an exception on timeout.

To test user units declared by `systemd.user.services` the optional `user` argument can be used:

```
machine.start()
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):

```
{
  skipLint = true;
  nodes.machine =
    { config, pkgs, ... }:
    { configuration...
    };

  testScript =
  ''
    Python code...
  '';
}
```

This will produce a Nix warning at evaluation time. To fully disable the linter, wrap the test script in a `let` block:

v: unstable -

comment directives to disable the Black linter directly (again, don't commit this within the Nixpkgs repository):

```
testScript =  
''  
  # fmt: off  
  Python code...  
  # fmt: on  
'';
```

Similarly, the type checking of test scripts can be disabled in the following way:

```
{  
  skipTypeCheck = true;  
  nodes.machine =  
    { config, pkgs, ... }:  
      { configuration...  
    };  
}
```

Failing tests early

To fail tests early when certain invariants are no longer met (instead of waiting for the build to time out), the decorator `polling_condition` is provided. For example, if we are testing a program `foo` that should not quit after being started, we might write the following:

```
@polling_condition  
def foo_running():  
    machine.succeed("pgrep -x foo")  
  
    machine.succeed("foo --start")  
    machine.wait_until_succeeds("pgrep -x foo")  
  
with foo_running:
```

v: unstable -

```
... # Put `foo` through its paces
```

`polling_condition` takes the following (optional) arguments:

`seconds_interval`

specifies how often the condition should be polled:

```
@polling_condition(seconds_interval=10)
def foo_running():
    machine.succeed("pgrep -x foo")
```

`description`

is used in the log when the condition is checked. If this is not provided, the description is pulled from the docstring of the function. These two are therefore equivalent:

```
@polling_condition
def foo_running():
    "check that foo is running"
    machine.succeed("pgrep -x foo")
```

```
@polling_condition(description="check that foo is running")
def foo_running():
    machine.succeed("pgrep -x foo")
```

Adding Python packages to the test script

When additional Python libraries are required in the test script, they can be added using the parameter `extraPythonPackages`. For example, you could add `numpy` like this:

```
{
  extraPythonPackages = p: [ p.numpy ];
  nodes = { };
```

v: unstable -

```
# Type checking on extra packages doesn't work yet
skipTypeCheck = true;

testScript = ''
  import numpy as np
  assert str(np.zeros(4) == "array([0., 0., 0., 0.])")
';
}
```

In that case, `numpy` is chosen from the generic `python3Packages`.

Test Options Reference

The following options can be used when writing tests.

enableOCR

Whether to enable Optical Character Recognition functionality for testing graphical programs. See [Machine objects](#).

Type: boolean

Default: `false`

Declared by:

[nixos/lib/testing/driver.nix](#)

defaults

NixOS configuration that is applied to all [nodes](#).

Type: module

Default: `{ }`

Declared by:

[nixos/lib/testing/nodes.nix](#)

driver

v: unstable -

Package containing a script that runs the test.

Type: package

Default: set by the test framework

Declared by:

[nixos/lib/testing/driver.nix](#)

extraBaseModules

NixOS configuration that, like [defaults](#), is applied to all [nodes](#) and can not be undone with [specialisation.<name>.inheritParentConfig](#).

Type: module

Default: { }

Declared by:

[nixos/lib/testing/nodes.nix](#)

extraDriverArgs

Extra arguments to pass to the test driver.

They become part of [driver](#) via [wrapProgram](#).

Type: list of string

Default: []

Declared by:

[nixos/lib/testing/driver.nix](#)

extraPythonPackages

Python packages to add to the test driver.

The argument is a Python package set, similar to [pkgs.pythonPackages](#).

Type: function that evaluates to a(n) list of package

v: unstable -

Default: <function>

Example:

```
p: [ p.numpy ]
```

Declared by:

[nixos/lib/testing/driver.nix](#)

globalTimeout

A global timeout for the complete test, expressed in seconds. Beyond that timeout, every resource will be killed and released and the test will fail.

By default, we use a 1 hour timeout.

Type: signed integer

Default: 3600

Example: 600

Declared by:

[nixos/lib/testing/driver.nix](#)

hostPkgs

Nixpkgs attrset used outside the nodes.

Type: raw value

Example:

```
import nixpkgs { inherit system config overlays; }
```

Declared by:

v: unstable -

[nixos/lib/testing/driver.nix](#)

interactive

Tests [can be run interactively](#) using the program in the test derivation's `.driverInteractive` attribute.

When they are, the configuration will include anything set in this submodule.

You can set any top-level test option here.

Example test module:

```
{ config, lib, ... }: {  
  
  nodes.rabbitmq = {  
    services.rabbitmq.enable = true;  
  };  
  
  # When running interactively ...  
  interactive.nodes.rabbitmq = {  
    # ... enable the web ui.  
    services.rabbitmq.managementPlugin.enable = true;  
  };  
}
```

For details, see the section about [running tests interactively](#).

Type: submodule

Declared by:

[nixos/lib/testing/interactive.nix](#)

meta

The `meta` attributes that will be set on the returned derivations.

Not all `meta` attributes are supported, but more can be added as desired.

Type: submodule

v: unstable -

Default: { }

Declared by:

<nixos/lib/testing/meta.nix>

meta.broken

Sets the [meta.broken](#) attribute on the [test](#) derivation.

Type: boolean

Default: false

Declared by:

<nixos/lib/testing/meta.nix>

meta.maintainers

The [list of maintainers](#) for this test.

Type: list of raw value

Default: []

Declared by:

<nixos/lib/testing/meta.nix>

meta.timeout

The [test](#)'s [meta.timeout](#) in seconds.

Type: null or signed integer

Default: 3600

Declared by:

<nixos/lib/testing/meta.nix>

name

The name of the test.

v: unstable -

This is used in the derivation names of the [**driver**](#) and [**test**](#) runner.

Type: string

Declared by:

[nixos/lib/testing/name.nix](#)

[node.pkgs](#)

The Nixpkgs to use for the nodes.

Setting this will make the `nixpkgs.*` options read-only, to avoid mistakenly testing with a Nixpkgs configuration that diverges from regular use.

Type: null or Nixpkgs package set

Default: `null`, so construct `pkgs` according to the `nixpkgs.*` options as usual.

Declared by:

[nixos/lib/testing/nodes.nix](#)

[node.pkgsReadOnly](#)

Whether to make the `nixpkgs.*` options read-only. This is only relevant when [node.pkgs](#) is set.

Set this to `false` when any of the [nodes](#) needs to configure any of the `nixpkgs.*` options. This will slow down evaluation of your test a bit.

Type: boolean

Default: `node.pkgs != null`

Declared by:

[nixos/lib/testing/nodes.nix](#)

[node.specialArgs](#)

An attribute set of arbitrary values that will be made available as module arguments during the resolution of module `imports`.

v: unstable -

Note that it is not possible to override these from within the NixOS configurations. If your argument is not relevant to `imports`, consider setting `defaults._module.args.<name>` instead.

Type: lazy attribute set of raw value

Default: { }

Declared by:

[nixos/lib/testing/nodes.nix](#)

nodes

An attribute set of NixOS configuration modules.

The configurations are augmented by the [defaults](#) option.

They are assigned network addresses according to the [nixos/lib/testing/network.nix](#) module.

A few special options are available, that aren't in a plain NixOS configuration. See [Configuring the nodes](#)

Type: lazy attribute set of module

Declared by:

[nixos/lib/testing/nodes.nix](#)

passthru

Attributes to add to the returned derivations, which are not necessarily part of the build.

This is a bit like doing `drv // { myAttr = true; }` (which would be lost by `overrideAttrs`). It does not change the actual derivation, but adds the attribute nonetheless, so that consumers of what would be `drv` have more information.

Type: lazy attribute set of raw value

Declared by:

v: unstable -

[nixos/lib/testing/run.nix](#)

[qemu.package](#)

Which qemu package to use for the virtualisation of [nodes](#).

Type: package

Default: "hostPkgs.qemu_test"

Declared by:

[nixos/lib/testing/driver.nix](#)

[skipLint](#)

Do not run the linters. This may speed up your iteration cycle, but it is not something you should commit.

Type: boolean

Default: **false**

Declared by:

[nixos/lib/testing/driver.nix](#)

[skipTypeCheck](#)

Disable type checking. This must not be enabled for new NixOS tests.

This may speed up your iteration cycle, unless you're working on the [testScript](#).

Type: boolean

Default: **false**

Declared by:

[nixos/lib/testing/driver.nix](#)

[test](#)

Derivation that runs the test as its "build" process.

v: unstable -

This implies that NixOS tests run isolated from the network, making them more dependable.

Type: package

Declared by:

[nixos/lib/testing/run.nix](#)

testScript

A series of python declarations and statements that you write to perform the test.

Type: string or function that evaluates to a(n) string

Declared by:

[nixos/lib/testing/testScript.nix](#)

Running Tests

You can run tests using `nix-build`. For example, to run the test [login.nix](#), you do:

```
$ cd /my/git/clone/of/nixpkgs  
$ nix-build -A nixosTests.login
```

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 log of the test:

```
$ nix-store --read-log result
```

Running Tests interactively

The test itself can be run interactively. This is particularly useful when developing or debugging a test:

```
$ nix-build . -A nixosTests.login.driverInteractive
```

v: unstable -

```
$ ./result/bin/nixos-test-driver  
[...]  
>>>
```

You can then take any Python statement, e.g.

```
>>> start_all()  
>>> test_script()  
>>> machine.succeed("touch /tmp/foo")  
>>> 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).

Shell access in interactive mode

The function `<yourmachine>.shell_interact()` grants access to a shell running inside a virtual machine. To use it, replace `<yourmachine>` with the name of a virtual machine defined in the test, for example: `machine.shell_interact()`. Keep in mind that this shell may not display everything correctly as it is running within an interactive Python REPL, and logging output from the virtual machine may overwrite input and output from the guest shell:

```
>>> machine.shell_interact()  
machine: Terminal is ready (there is no initial prompt):  
$ hostname  
machine
```

As an alternative, you can proxy the guest shell to a local TCP server by first starting a TCP server in a terminal using the command:

```
$ socat 'READLINE,PROMPT=$ ' tcp-listen:4444,reuseaddr`
```

In the terminal where the test driver is running, connect to this server by using:

v: unstable -

```
>>> machine.shell_interact("tcp:127.0.0.1:4444")
```

Once the connection is established, you can enter commands in the socat terminal where socat is running.

Port forwarding to NixOS test VMs

If your test has only a single VM, you may use e.g.

```
$ QEMU_NET_OPTS="hostfwd=tcp:127.0.0.1:2222-:22" ./result/bin/nixos-test-d
```

to port-forward a port in the VM (here 22) to the host machine (here port 2222).

This naturally does not work when multiple machines are involved, since a single port on the host cannot forward to multiple VMs.

If the test defines multiple machines, you may opt to *temporarily* set `virtualisation.forwardPorts` in the test definition for debugging.

Such port forwardings connect via the VM's virtual network interface. Thus they cannot connect to ports that are only bound to the VM's loopback interface (127.0.0.1), and the VM's NixOS firewall must be configured to allow these connections.

Reuse VM state

You can re-use the VM states coming from a previous run by setting the `--keep-vm-state` flag.

```
$ ./result/bin/nixos-test-driver --keep-vm-state
```

The machine state is stored in the `$TMPDIR/vm-state-machinename` directory.

Interactive-only test configuration

The `.driverInteractive` attribute combines the regular test configuration with definitions from the [interactive submodule](#). This gives you a more usable, graphical, but slightly different v: unstable -

You can add your own interactive-only test configuration by adding extra configuration to the [interactive submodule](#).

To interactively run only the regular configuration, build the `<test>.driver` attribute instead, and call it with the flag `result/bin/nixos-test-driver --interactive`.

Linking NixOS tests to packages

You can link NixOS module tests to the packages that they exercised, so that the tests can be run automatically during code review when the package gets changed. This is [described in the nixpkgs manual](#).

Developing the NixOS Test Driver

Table of Contents

[Testing changes to the test framework](#)

The NixOS test framework is a project of its own.

It consists of roughly the following components:

- `nixos/lib/test-driver`: The Python framework that sets up the test and runs the [testScript](#)
- `nixos/lib/testing`: The Nix code responsible for the wiring, written using the (NixOS) Module System.

These components are exposed publicly through:

- `nixos/lib/default.nix`: The public interface that exposes the `nixos/lib/testing` entrypoint.
- `flake.nix`: Exposes the `lib.nixos`, including the public test interface.

Beyond the test driver itself, its integration into NixOS and Nixpkgs is important.

- `pkgs/top-level/all-packages.nix`: Defines the `nixosTests` attribute, used b v: unstable -

tests attributes and OfBorg.

The project

[Channel Status](#)
[Packages search](#)
[Options search](#)
[Reproducible Builds Status](#)
[Security](#)

Get in Touch

[Forum](#)
[Matrix Chat](#)
[Commercial support](#)

Contribute

[Contributing Guide](#)
[Donate](#)

Stay up to date

[Blog](#)
[Newsletter](#)

Copyright © 2024 NixOS contributors
CC-BY-SA-4.0

Connect with us:

accidentally compare iterations of the PR instead of changes to the PR base.

As we currently have some flaky tests, newly failing tests are expected, but should be reviewed to make sure that

v: unstable -

- The number of failures did not increase significantly.
- All failures that do occur can reasonably be assumed to fail for a different reason than the changes.

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
# nixos-generate-config --root /mnt
$ nix-build '<nixpkgs/nixos>' -A config.system.build.nixos-install
# ./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
# ./result/bin/nixos-enter
```

Contributing to this manual

Table of Contents

[Contributing to the `configuration.nix` options documentation](#)

[Contributing to `nixos-*` tools' manpages](#)

The sources of the NixOS manual are in the [nixos/doc/manual](#) subdirectory of the [Nixpkgs](#) repository. This manual uses the [Nixpkgs manual syntax](#).

You can quickly check your edits with the following:

```
$ cd /path/to/nixpkgs
```

v: unstable -

```
$ $EDITOR doc/nixos/manual/... # edit the manual  
$ nix-build nixos/release.nix -A manual.x86_64-linux
```

If the build succeeds, the manual will be in `./result/share/doc/nixos/index.html`.

There's also [a convenient development daemon](#).

The above instructions don't deal with the appendix of available `configuration.nix` options, and the manual pages related to NixOS. These are built, and written in a different location and in a different format, as explained in the next sections.

Contributing to the `configuration.nix` options documentation

The documentation for all the different `configuration.nix` options is automatically generated by reading the `descriptions` of all the NixOS options defined at `nixos/modules/`. If you want to improve such `description`, find it in the `nixos/modules/` directory, and edit it and open a pull request.

To see how your changes render on the web, run again:

```
$ nix-build nixos/release.nix -A manual.x86_64-linux
```

And you'll see the changes to the appendix in the path `result/share/doc/nixos/options.html`.

You can also build only the `configuration.nix(5)` manual page, via:

```
$ cd /path/to/nixpkgs  
$ nix-build nixos/release.nix -A nixos-configuration-reference-manpage.x86_64-linux
```

And observe the result via:

```
$ man --local-file result/share/man/man5/configuration.nix.5  
v: unstable -
```

If you're on a different architecture that's supported by NixOS (check file `nixos/release.nix` on Nixpkgs' repository) then replace `x86_64-linux` with the architecture. `nix-build` will complain otherwise, but should also tell you which architecture you have + the supported ones.

Contributing to nixos-* tools' manpages

The manual pages for the tools available in the installation image can be found in Nixpkgs by running (e.g for `nixos-rebuild`):

```
$ git ls | grep nixos-rebuild.8
```

Man pages are written in [mdoc\(7\) format](#) and should be portable between mandoc and groff for rendering (except for minor differences, notably different spacing rules.)

For a preview, run `man --local-file path/to/file.8`.

Being written in `mdoc`, these manpages use semantic markup. This following subsections provides a guideline on where to apply which semantic elements.

Command lines and arguments

In any manpage, commands, flags and arguments to the *current* executable should be marked according to their semantics. Commands, flags and arguments passed to *other* executables should not be marked like this and should instead be considered as code examples and marked with `Ql`.

- Use `F1` to mark flag arguments, `A1` for their arguments.
- Repeating arguments should be marked by adding an ellipsis (spelled with periods, `...`).
- Use `Cm` to mark literal string arguments, e.g. the `boot` command argument passed to `nixos-rebuild`.
- Optional flags or arguments should be marked with `O1`. This includes optional repeating arguments.
- Required flags or arguments should not be marked.

v: unstable -

- Mutually exclusive groups of arguments should be enclosed in curly brackets, preferably created with **Bro/Brc** blocks.

When an argument is used in an example it should be marked up with **Ar** again to differentiate it from a constant. For example, a command with a `--host name` option that calls ssh to retrieve the host's local time would signify this thusly:

```
This will run
.Ic ssh Ar name Ic time
to retrieve the remote time.
```

Paths, NixOS options, environment variables

Constant paths should be marked with **Pa**, NixOS options with **Va**, and environment variables with **Ev**.

Generated paths, e.g. `result/bin/run-hostname-vm` (where `hostname` is a variable or arguments) should be marked as **Ql** inline literals with their variable components marked appropriately.

- When `hostname` refers to an argument, it becomes `.Ql result/bin/run- Ns Ar hostname Ns -vm`
- When `hostname` refers to a variable, it becomes `.Ql result/bin/run- Ns Va hostname Ns -vm`

Code examples and other commands

In free text names and complete invocations of other commands (e.g. `ssh` or `tar -xvf src.tar`) should be marked with **Ic**, fragments of command lines should be marked with **Ql**.

Larger code blocks or those that cannot be shown inline should use indented literal display block markup for their contents, i.e.

```
.Bd -literal -offset indent
...
.Ed
```

v: unstable -

Contents of code blocks may be marked up further, e.g. if they refer to arguments that will be substituted into them:

```
.Bd -literal -offset indent
{
    config.networking.hostname = "\c
.Ar hostname Ns \c
";
}
.Ed
```

v: unstable -