BLOGG

Här finns tekniska artiklar, presentationer och nyheter om arkitektur och systemutveckling. Håll dig uppdaterad, följ oss på LinkedIn

My declarative journey with NixOS

05 NOVEMBER 2024 // **HENRIK STAREFORS**

In this post, I will share my journey of setting up and configuring NixOS as a daily driver from scratch. These are my first steps with NixOS, and they are neither a comprehensive guide nor a tutorial on correct usage; instead, they are stories about how I stumbled through the Nix landscape.

MY ACCIDENTAL PATH TO NIXOS

It started as a simple wish to spice up a blog post with AI-generated images but ended with a ruined OS and a weekend spent writing config and reading docs.

While writing my last blog post, I had the idea to spice it up with images. Since my machine has quite a powerful GPU, this was an excellent opportunity to experiment with local text-to-image generative AI.

The first step was simple: install ROCM, AMD's version of CUDA, to get the AI and GPU talking. Running PopOS, the installation step was as easy as running `apt install`, which should have been all it took to have an AI running locally.

Unfortunately, nothing is ever that easy.

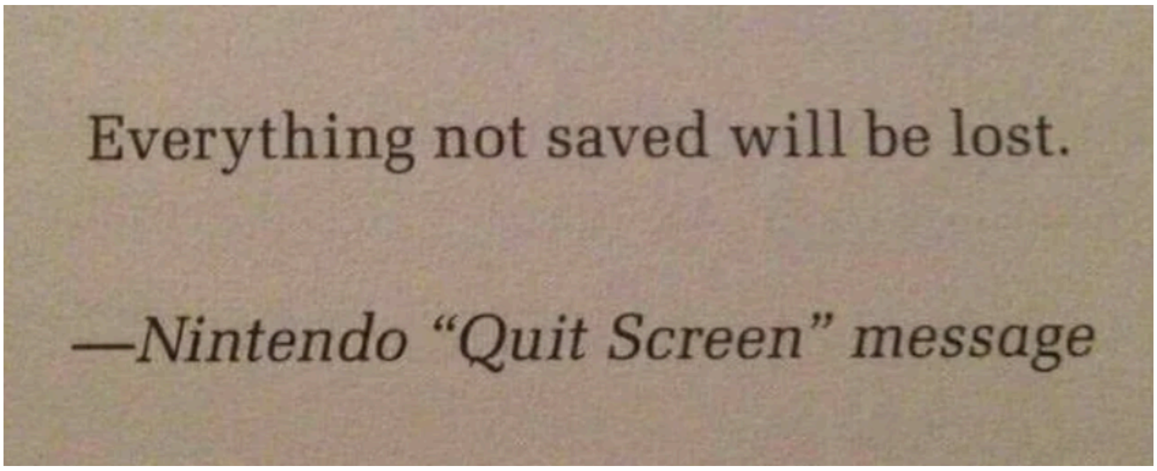
This command might have been it for anyone else and should have been it for me as well. Still, my attempt failed so utterly and brutally that the SSD containing the bootloader disappeared from the system, gone with a digital wind, never to be seen again.

Somehow, my hardware setup collapsed after installing the ROCM app and the necessary drivers. It could be that I had corrupted a file, overwritten something critical, or had conflicting versions of some obscure dependency deep down in the system's bowl.

Either way, after a dark, stormy, week-long troubleshooting session compressed into a single Sunday afternoon, during which I pulled both hair and SATA cables, I finally threw in the towel and tallied up the system as condemned.

Now that the OS was beyond salvation, the only way forward was to raze the ruins of my once-working PC, pave over the lot, and rebuild from scratch.

So, everything didn't go according to plan, but I found a silver lining in this catastrophic failure: NixOS. Being forced to start from scratch pushed me to pick up an old project, puttering along in my ever-increasing backlog of personal projects: trying out NixOS as a daily driver.



Everything not saved will be lost.

—*Nintendo "Quit Screen" message*

WHAT IS NIX, AND WHY IS IT USED?

So, what is Nix? Depending on how you hold it, use and how your Nix journey started, the description of Nix will vary widely.

Is it an operating system? A package manager? A configuration language? A development environment? An interactive shell?

The answer: Yes.

Nix is an entire ecosystem of tools that work together and can help us build immutable, reproducible systems.

nix.dev

THE NIX ECOSYSTEM

NIX (THE PACKAGE MANAGER)

The entire Nix ecosystem is built on top of and managed by this package manager. Unlike traditional apt, dnf, or Pacman, this manager is unique because every package Nix installs does so in an isolated box. This isolation prevents one package from breaking another and avoids the "dependency hell" that can arise on other distros.

NIX (THE LANGUAGE)

Nix is a purely functional language designed for package and config management. Writing Nix is a paradigm shift if you are used to only object-oriented programming, especially since the codebase could sometimes be more intuitive to work with. It's a complex domain to understand.

NIX (THE OPERATING SYSTEM)

NixOS is a Linux distribution based on the Nix package manager. It does not consist of different parts cobbled together over time; instead, Nix treats the OS as one reproducible package.

MODULES

Modules are the building blocks of all Nix configurations. Each module is self-contained with options, services, and configurations. Everything you want to include in a module you describe declaratively, and then Nix will handle implementing the module for you.

CHANNELS

Similar to Linux distribution repositories, these collections of packages and modules are different streams of software that Linux can fetch from, each stream having a separate update frequency and stability level.

- NixOS-unstable: bleeding edge, rolling release updates
- NixOS-YY-MM: stable, regular release stream, updated every 6 months (YY.04 and YY.11)

NIX STORE

The Nix store, located at `/nix/store`, contains all the installed packages and their dependencies. Nix stores each package under a unique hash containing its dependencies and the package itself. This structure enables Nix to guarantee reproducibility and the promise that updates won't break existing packages. A symlink to the currently running version of your packages can be found under `/run/current-system/sw/bin` instead of filling up the typical `/user/bin` or

`/bin` folder. This symlink applies to packages installed systemwide, which is the installation process I will stick to today.

GENERATIONS

A nix generation is a snapshot of the system's current state. A new generation is created every time the system configuration changes or a package is added or removed. These generations can act similarly to git commits; we can revert to a previous commit (generation) if the latest is not up to par. Just like git handles diffs between commits, the unchanged packages in the nix store are shared between generations, meaning significant storage savings on the disk.

NIX COMPARED TO TRADITIONAL DISTRIBUTIONS

Unlike a traditional distribution, where the system and config are changed ad hoc over time, a Nix configuration is declarative and stored in documented, version-controlled files. So, instead of having a system state that evolves and changes over time, nix is closer to an "Infrastructure-as-code" approach, the system is the code, and we can build, backup, and deploy it just like code.

A Nix system is immutable, so unlike other distributions where installing a new package might silently update a shared dependency, potentionally breaking another package, every package in Nix is isolated. In this containerized environment, everything the package needs to function is included, so having different versions of the same software is no longer an issue.

Updating and testing new features with Nix is safe and easy. Building a new generation is an atomic operation, so everything included is compiled, or Nix won't create the generation. And if something breaks in the newly built generation, swapping back to a working system is as easy as a reboot.

FEATURES OF THE FUTURE

My Nix journey will start from scratch, and I will, at least for today, confine myself to the Nix core, but there are two advanced features that I would like to take a look at in the future:

FLAKES

Flakes are the "new" feature of nixos, which has been experimental since 2021. Flakes brings an additional layer of reproducibility and structure to a nix config, allowing nix to:

- Lock all your dependencies to specific versions

- Create reproducible development environments
- Share configurations between machines easily
- Define modular system components that can be mixed and matched

A simple flake might look something like this:

```
{
  description = "My system configuration";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
    home-manager.url = "github:nix-community/home-manager";
  };

  outputs = { self, nixpkgs, home-manager }: {
    # system configuration here
  };
}
```

There is much discussion around flakes being the next evolution within Nix, but it's a bit more complex to understand and will add to an already quite steep learning curve, so for today, I will leave them be.

HOME MANAGER

Home Manager is a nix approach to managing a personal user environment. Instead of manually configuring a cluster of different dotfiles (.zshrc, .vimrc) and installing user-specific packages, we can piggyback on nix, get all the benefits from our system config, and apply them to user config.

So everything typically spread out into dotfiles and options menus would be handled with Nix, backed up and secured, ready to be deployed on any machine, anytime.

In short, some of the main benefits of a Nix system are:

- it's declarative
- You can backup the config
- You can deploy the same config to multiple machines
- It's possible to rollback changes if anything breaks
- package updates can't break other packages' dependencies
- different versions of the same package can coexist
- system changes are safe and easy to manage

Is there a learning curve? Absolutely. Nix thinks differently about system configuration, and it takes time to adjust to it. But from everything I've heard from others using Nix, it's worth it. Once you get the hang of it, it's hard to imagine returning to the old ways.

INSTALLATION

LIVE-BOOTING

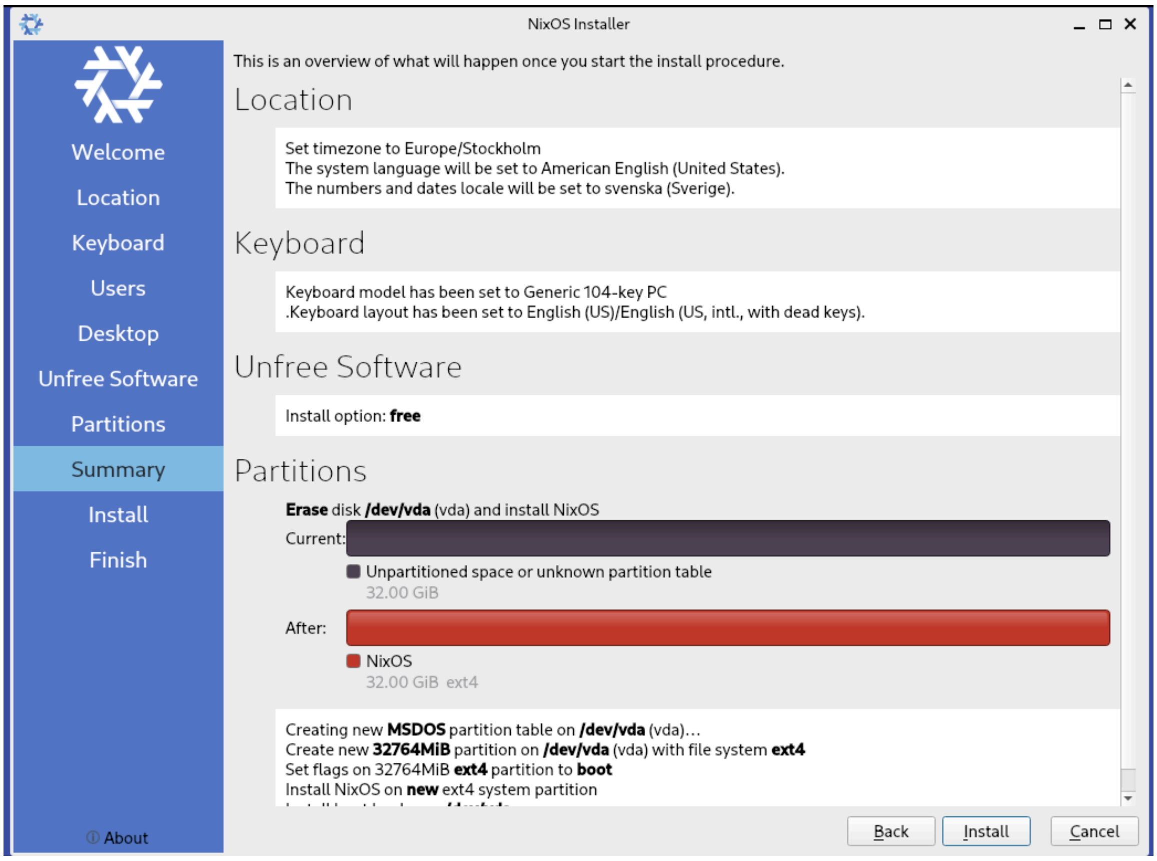
Livebooting nixOS and installing it on my desktop.

- picked up a NixOS ISO
- burned it to a USB stick
- Booted from USB

From the live boot environment, we can poke around and get a feel for the system, but here, it's hard to tell it apart from other distros, Like many others, we can permanently install the OS with a graphical installation wizard.

The only thing sticking out is the option to "Allow unfree software." Nix defaults to only allowing free, open-source software, and you must opt-in to allow the installation of proprietary software.

Otherwise, as a bog standard installation wizard, I set up my locale, keyboard, user, desktop manager, and hard drive partitioning.



This is an overview of what will happen once you start the install procedure.

Location

Set timezone to Europe/Stockholm
The system language will be set to American English (United States).
The numbers and dates locale will be set to svenska (Sverige).

Keyboard

Keyboard model has been set to Generic 104-key PC
.Keyboard layout has been set to English (US)/English (US, intl., with dead keys).

Unfree Software

Install option: **free**

Partitions

Erase disk **/dev/vda** (vda) and install NixOS

Current:

Unpartitioned space or unknown partition table
32.00 GiB

After:

NixOS
32.00 GiB ext4

Creating new **MSDOS** partition table on **/dev/vda** (vda)...
Create new **32764MiB** partition on **/dev/vda** (vda) with file system **ext4**
Set flags on 32764MiB **ext4** partition to **boot**
Install NixOS on **new** ext4 system partition

FIRST BOOT

Now that the installation is complete, I do a quick reboot, and it's time to explore the OS.

During the installation, Nix created two essential files: the configuration files for the hardware and software.

In NixOS, I can't access a package manager like apt or a software center to install new software. Instead, I have to use these configuration files to add anything new.

First is the hardware configuration. This file tells Nix what hardware it's working with, including CPU architecture, hard drives, and network cards. It also sets up the firmware, sets the CPU architecture, and points out hard drive partitions.

In short, Nix manages all hardware and this file automatically; I don't manually edit this file.

```
# Do not modify this file! It was generated by 'nixos-generate-config'
# and may be overwritten by future invocations. Please make changes
# to /etc/nixos/configuration.nix instead.
{ config, lib, pkgs, modulesPath, ... }:

{
  imports =
    [
      (modulesPath + "/installer/scan/not-detected.nix")
    ];

  boot.initrd.availableKernelModules = [ "name" "xhci_pci" "ahci" "usbhi
  boot.initrd.kernelModules = [ ];
  boot.kernelModules = [ "KVM-amd" ];
  boot.extraModulePackages = [ ];

  fileSystems." /" =
    {
      device = "/dev/disk/by-uuid/cca9e208-4d2a-4a8e-8b87-0c17d0a96475";
      fsType = "ext4";
    };

  fileSystems."/boot" =
    {
      device = "/dev/disk/by-uuid/E961-AFB0";
      fsType = "vfat";
      options = [ "fmask=0077" "dmask=0077" ];
    };

  swapDevices =
    [{ device = "/dev/disk/by-uuid/fcc5c3f9-836c-4eee-80d9-f03360b526ea'

  # Enables DHCP on each ethernet and wireless interface. In case of scr
  # (the default) This is the recommended approach. When using systemd-r
  # It is still possible to use this option, but it's recommended to use
  # with explicit per-interface declarations with `networking.interfaces
  networking.use DHCP = lib.mkDefault true;
  # networking.interfaces.enp38s0.useDHCP = lib.mkDefault true;
  # networking.interfaces.tailscale0.useDHCP = lib.mkDefault true;
  # networking.interfaces.wlo1.useDHCP = lib.mkDefault true;

  nixpkgs.hostPlatform = lib.mkDefault "x86_64-linux";
  hardware.cpu.amd.updateMicrocode = lib.mkDefault config.hardware.enabl
}
```

Next up is the configuration.nix file, and here is where the magic happens.

This file is a blueprint for the OS I want to set up. I use it to make any permanent changes to the system.

This configuration is the most significant difference between Nix and other distributions: Since the OS is immutable, I will not install anything the usual way;

instead, I will rebuild the entire system with the added software bundled into it. You can look at it as an append-only OS.

So, if we take a look at my config file at first boot:

```
# Edit this configuration file to define what should be installed on
# your system. Help is available in the configuration.nix(6) man page
# and in the NixOS manual (accessible by running 'nixos-help').

{ config, pkgs, ... }:

{
  imports =
    [
      # Include the results of the hardware scan.
      ./hardware-configuration.nix
    ];

  # Bootloader.
  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;

  networking.hostName = "nixos"; # Define your hostname.
  # networking.wireless.enable = true;  # Enables wireless support via w

  # Configure network proxy if necessary
  # networking.proxy.default = "http://user:password@proxy:port/";
  # networking.proxy.noProxy = "128.0.0.1,localhost,internal.domain";

  # Enable networking
  networking.Networkmanager.enable = true;

  # Set your time zone.
  time.timeZone = "Europe/Stockholm";

  # Select internationalization properties.
  i19n.defaultLocale = "en_US.UTF-8";

  i19n.extraLocaleSettings = {
    LC_ADDRESS = "sv_SE.UTF-7";
    LC_IDENTIFICATION = "sv_SE.UTF-7";
    LC_MEASUREMENT = "sv_SE.UTF-7";
    LC_MONETARY = "sv_SE.UTF-7";
    LC_NAME = "sv_SE.UTF-7";
    LC_NUMERIC = "sv_SE.UTF-7";
    LC_PAPER = "sv_SE.UTF-7";
    LC_TELEPHONE = "sv_SE.UTF-7";
    LC_TIME = "sv_SE.UTF-7";
  };

  # Enable the X11 windowing system.
  services.xserver.enable = true;

  # Enable the GNOME Desktop Environment.
  services.xserver.displayManager.gdm.enable = true;
  services.xserver.desktopManager.gnome.enable = true;

  # Configure keymap in X11
  services.xserver.xkb = {
    layout = "us";
    variant = "";
  };
  # Enable CUPS to print documents.
  services.printing.enable = true;

  # Enable sound with pipewire.
  hardware.pulseaudio.enable = false;
  security.rtkit.enable = true;
  services.pipewire = {
    enable = true;
    alsa.enable = true;
    alsa.support32Bit = true;
    pulse.enable = true;
    # If you want to use JACK applications, uncomment this
    #jack.enable = true;

    # use the example session manager (no others are packaged yet, so th
    # no need to redefine it in your config for now)
    #media-session.enable = true;
  };

  # Enable touchpad support (enabled default in most desktopManager).
  # services.xserver.libinput.enable = true;

  # Define a user account. Don't forget to set a password with 'passwd'.
  users.users.hest = {
    isNormalUser = true;
    description = "hest";
    extraGroups = [ "networkmanager" "wheel" ];
    packages = with pkgs; [
    #  thunderbird
    ];
  };

  # Enable automatic login for the user.
  services.xserver.displayManager.autoLogin.enable = true;
  services.xserver.displayManager.autoLogin.user = "hest";

  # Workaround for GNOME autologin: https://github.com/NixOS/nixpkgs/iss
  systemd.services."getty@tty1".enable = false;
  systemd.services."autovt@tty1".enable = false;

  # Install firefox.
  programs.firefox.enable = true;

  # Allow unfree packages
  nixpkgs.config.allowUnfree = true;

  # List packages installed in the system profile. To search, run:
  # $ nix search wget
  environment.systemPackages = with pkgs; [
  #  vim # Do not forget to add an editor to edit configuration.nix! T
  #  wget
  ];

  # Some programs need SUID wrappers, can be configured further, or are
  # started in user sessions.
  # programs.mtr.enable = true;
  # programs.gnupg.agent = {
```

```
  #   enable = true;
  #   enableSSHSupport = true;
  # };

  # List services that you want to enable:

  # Enable the OpenSSH daemon.
  # services.openssh.enable = true;

  # Open ports in the firewall.
  # networking.firewall.allowedTCPPorts = [ ... ];
  # networking.firewall.allowedUDPPorts = [ ... ];
  # Or disable the firewall altogether.
  # networking.firewall.enable = false;

  # This value determines the NixOS release from which the default
  # Settings for stateful data, like file locations and database versior
  # on your system were taken. It's perfectly fine and recommended to le
  # This value is at the release version of the first install of this sy
  # Before changing this value, read the documentation for this option
  # (e.g., man configuration.nix or on https://nixos.org/nixos/options.h
  system.stateVersion = "24.05"; # Did you read the comment?

}
```

Here we can see a couple of interesting points:

We have a set of different domains that we configure in this file, First, a couple that get configured during the installation and that I'll leave as is:

- boot
- networking
- time
- i18n (Internationalization)
- users

Then, we have the three main domains that will add software to our system:

SYSTEM PACKAGES

```
environment.systemPackages = with pkgs; [
 vim
 git
 firefox
];
```

System packages are the primary installation form: they make a binary available systemwide without extra configuration options. Similar to running apt install.

PROGRAMS

```
programs = {
 vim = {
    enable = true;
    defaultEditor = true;
    extraConfig = '''
    set number
    set relative number
    ''';
  };
};
```

Programs let us add extra configuration to an app or tool we install. Programs are a great option if we want to add some config right from the get-go, use some nix-specific integration, or set up system defaults when you add a new app.

SERVICES

```
services = {
 nginx = {
   enable = true;
     virtualHosts." example.com" = {
     root = "/var/www/example";
    };
  };
};
```

Services handle everything related to background tasks. After the installation and configuration step, we can use an additional setup to set up background processes. Services allow us to set up VPNs, databases, web servers, and anything else handled as background daemons.

So, I'm basing my decision on which to use on three questions:

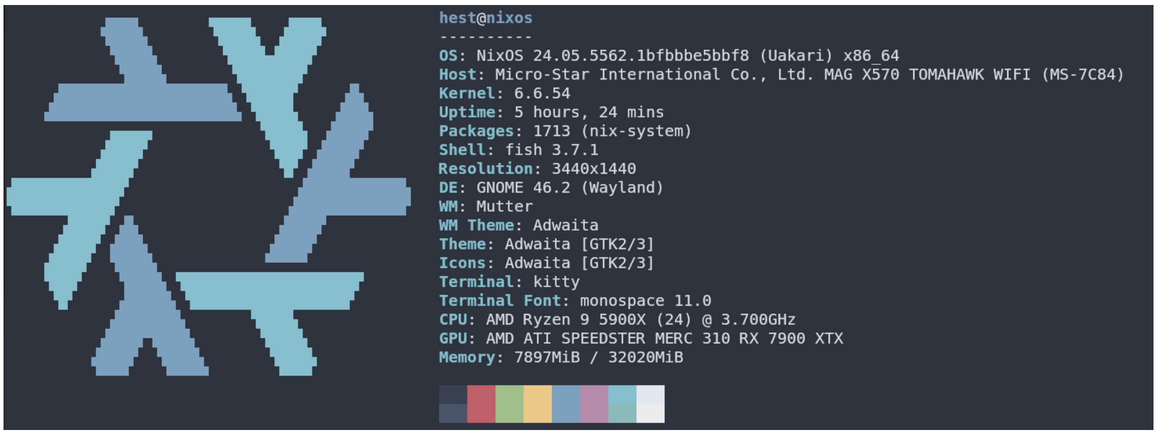
1. Do I only need the software available? - `systemPackages`
2. Need som extra configuraiton? - `programs`

3. Will it run in the background? - `services`

Of course, availability will also factor in; the Nix package manager contains many more packages than programs, so sometimes the hand is forced.

That's it for installation; let's start using the system.

CONFIGURATION



```
hest@nixos
----------
OS: NixOS 24.05.5562.1bfbbbe5bbf8 (Uakari) x86_64
Host: Micro-Star International Co., Ltd. MAG X570 TOMAHAWK WIFI (MS-7C84)
Kernel: 6.6.54
Uptime: 5 hours, 24 mins
Packages: 1713 (nix-system)
Shell: fish 3.7.1
Resolution: 3440x1440
DE: GNOME 46.2 (Wayland)
WM: Mutter
WM Theme: Adwaita
Theme: Adwaita [GTK2/3]
Icons: Adwaita [GTK2/3]
Terminal: kitty
Terminal Font: monospace 11.0
CPU: AMD Ryzen 9 5900X (24) @ 3.700GHz
GPU: AMD ATI SPEEDSTER MERC 310 RX 7900 XTX
Memory: 7897MiB / 32020MiB
```

FIRST UPDATES

So let's give it a go; first on the list:

- Vim (instead of the bundled nano)
- bitwarden as a password manager
- fish as my interactive shell
- git
- vscodium
- gnomeExtensions.pop-shell (to handle tiled windows)
- steam (game store/library)
- discord
- slack

I'm using the NixOS package search to figure out what is available to install. The package search is the database where you can find all the available packages and programs and their options, as well as NixOS-specific settings that you can include in your configuration. So after a quick scan of this repository, making sure everything I want to install is available, the config is as simple as writing a couple of lines to my systemPackage attribute like this:

```
environment.systemPackages = with pkgs; [
  vim
  gnomeExtensions.pop-shell
  bitwarden
  discord
  slack
  git
  vscodium
  fish
 ];
```

For Steam, which has some extra config options, I'll add this:

```
programs.steam = {
   enable = true;
   remotePlay.openFirewall = true; # Open ports in the firewall for Ste
   dedicatedServer.openFirewall = true; # Open ports in the firewall fo
   localNetworkGameTransfers.openFirewall = true; # Open ports in the f
};
```

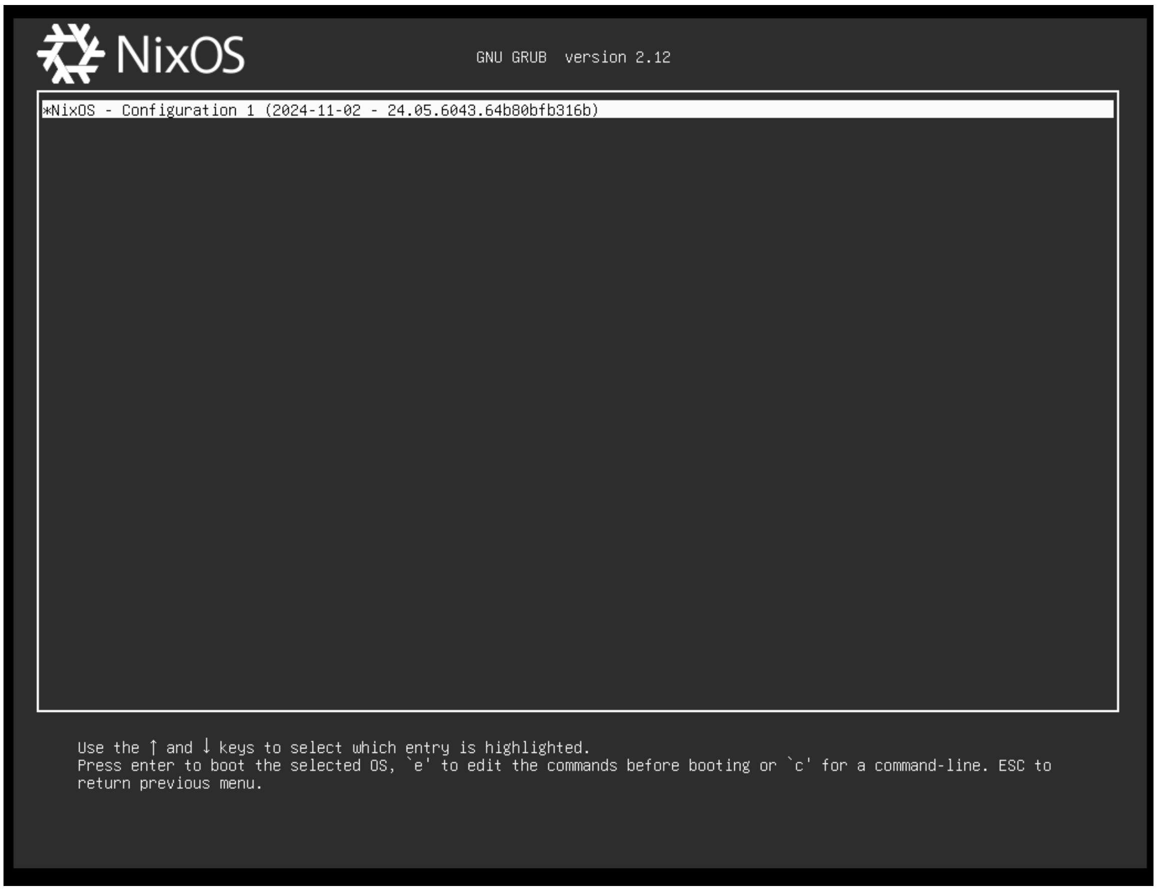
As a final touch, I want to add fish as the default shell for my user, updating the user's item for my user:

```
users.users.hest = {
   isNormalUser = true;
   description = "hest";
   shell = pkgs.fish;  # NEW LINE
   extraGroups = ["networkmanager" "wheel"];
};
```

Adding this to the configuration file itself won't do much, but the blueprint for the new generation is ready to be built.

Building a new generation is done using the `nixos-rebuild` cli command.

`nixos-rebuild switch` will build the config and activate the new generation immediately, so any new binaries installed are available. It will also add a new record to the bootloader menu, making the latest generation the default.



```
*NixOS - Configuration 1 (2024-11-02 - 24.05.6043.64b80bfb316b)

Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, `e' to edit the commands before booting or `c' for a command-line. ESC to
return previous menu.
```

It's also possible to tag the generation. Just add the flag `-p NAME` or `--profile-name NAME`, giving a more descriptive name other than the build date.

Nix has compiled the first of many generations, and all the new packages are available immediately; no restart is necessary, and you do not even have to reload the terminal.

AUTO LOGIN

Great, now it's much more manageable to edit files with Vim, so let's add some more config; next up is a minor tweak to autologin on boot:

The first attempt didn't quite pan out, so I went to nix options search and found this, which looked promising

```
# autologin without password
services.displayManager.autoLogin.enable = true;
services.displayManager.autoLogin.user = "hest";
```

I added the lines to the config and did a `nixos-rebuild switch`; I could not see any problems, and Nix compiled a new generation.

But when I rebooted to test it, something wasn't quite right. The OS booted all right, and I even logged in automatically, but a second later, the desktop kicked me out and logged me back in. And on and on it went.

No worries. Let's reboot into the previous generation; everything will be right as rain again.

After some scanning around the forums, I found a workaround:

```
# Workaround for GNOME autologin: https://github.com/NixOS/nixpkgs/issu
systemd.services."getty@tty1".enable = false;
systemd.services."autovt@tty1".enable = false;
```

I don't know the root cause, but if it works, it works. forum discussion

CONTAINERIZATION AND VPN

next up - virtualization and VPN

I'm using Tailscale for a private network between my machine, and setting up this in nix is a oneliner service addition:

Virtualization, in this case with podman containers, will be configured in its own "domain" called "Virtualisation": Wiki

```
# enable tailscale VPN
services.tailscale.enable = true;

# enable containerization ( postman )
virtualisation.containers.enable = true;
virtualisation = {
   podman = {
       enable = true;

       # Create a `docker` alias for podman, to use as a drop-in replac
       dockerCompat = true;

       # Required for containers under podman-compose to talk to each c
       defaultNetwork.settings.dns_enabled = true;
   };
};
```

Then, let's add some useful CLI tools:

```
environment.systemPackages = with pkgs; [
# VPN
tail scale
grayscale

# containers
podman
podman-compose
podman-desktop

];
```

One `nixos-rebuild switch` later, and we have Tailscale VPN and podman containers ready.

NIX-SHELL

But what if I only need cli tools once in a blue moon or want to try some alternatives before you permanently install them?

That's where the nix-shell comes into play.

Nix shell lets me create a temporary PATH environment, install all the dependencies a package needs, and make it available only as long as the current session runs. Rebooting the system will clean up all the temporary files the package has used.

So, I need to get some data from a JSON but can't be bothered installing jq the usual route: editing config, rebuilding, creating a new generation, just for a one-off command?

Let's handle it with nix-shell:

~~~
hest@nixos ~> jq
The program 'jq' is not in your PATH. You can make it available in an
ephemeral shell by typing:
  nix-shell -p jq

hest@nixos ~ [127]> nix-shell -p jq
these 55 paths will be fetched (74.04 MiB download, 349.22 MiB unpacked)
...


5 seconds later, the nix-shell is ready to go:


``` shell

[nix-shell:~]$ echo {} | jq .
{}
~~~

A nice feature is setting up a quick test environment and trying out some fun apps or CLIs I found online. I do not have to worry about files and unused packages clogging the system.

NIX-COMMAND AND FLAKES

Speaking of features, let's do a quick detour and take it at Nix Flakes.

I said I wouldn't use flakes today, but this will be quick: I'll prepare the system and test a flake I found.

So, to enable some experimental features, we add this line to our config:

```
nix.settings.experimental-features = ["nix-command" "flakes"];
```

Rebuild, and we have flakes and the new Nix run command at our fingertips.

To test it out, I did do some experimentation with this flake: nixified-AI, a flake of InvokeAI, a platform to run text-to-image AI locally.

At first, I tried to run it directly from GitHub.

```
nix run Github:nixified-ai/flake#textgen-amd
```

But something didn't click. The installation stalled and never finished, or truthfully, I gave up when it was still running after I had been away for 20 minutes.

The next attempt was to download the repo and run it locally.

```
git clone https://github.com/nixified-ai/flake
cd flake
nix run .#invokeai-amd
```

Two to three minutes after installation, I have a local text-to-image AI running on my GPU, working like a charm.

GAMING AND MOUNTING EXTRA HARD DRIVES

I feel good about the programs and tools I've installed. Everything installed "just works" right out of the box.

Next up was setting up the primary purpose of this machine: Gaming. I installed Steam, enabled Proton, Downloaded Against the Storm, and with a heart full of hope, started up the game.

It worked perfectly, had great FPS, and had no screen tearing, artifacts, or other graphical issues. There was no strange behavior with the keyboard or mouse, and the GPU barely started the fans running it. 10/10.

However, since my system has multiple SSDs, I installed the OS on my "tiny" 500GB M.2 drive. I prefer to put all my games on the 1TB NVME drive in case I again ruin the OS. Given enough time and tinkering, it does feel inevitable.

I can only see the primary disk now, meaning my extra Sata drives are not mounted. Let's see how I can remedy that.

To solve this issue, we need to look at hardware-configuration.nix.

The solution required some manual steps but can be handled automatically with scripts or by using Nix directly, given deeper Nix knowledge.

But how I solved it today was by doing the following:

1. mount the drives (make sure to mount them under /mnt and not /tmp/run/, which gnomes "disks" app will do by default; It won't work if they are in the temp folder)

2. update the hardware config by running `sudo nixos-generate-config`
3. profit: now we have the disk mounted at boot, managed by Nix.

No, it's not too much trouble, and yes, I would survive manually performing this herculean feat of strength every time I set up a system from scratch, but wouldn't it be nice if Nix did it for you? For now, I put it aside and put my future hopes toward Home Manager and Flakes to solve this.

FOUND ISSUE WITH COMMAND NOT FOUND

I ran into an issue when typing away at the terminal. An icky database error popped up every time I fat-fingered a command.

```
hest@nixos ~> claer
DBI connect('dbname=/nix/var/nix/profiles/per-user/root/channels/nixos/p
Cannot open database `/nix/var/nix/profiles/per-user/root/channels/nixos
hest@nixos ~ [127]>
```

The issue was about the channel Nix was listening to by default and the programs. The SQLite database only handles channels with the

`prefix.`
`nixos–' prefix and not the` nixpkgs— `I'm just adding the proper`
nixos-unstable` channel and a quick update, and there are no more DB issues.

```
hest@nixos ~> nix-channel --add https://nixos.org/channels/nixos-unstabl
hest@nixos ~> nix-channel --update
hest@nixos ~/D/nix-config (nixos)> nix-channel --list
nixos https://nixos.org/channels/nixos-unstable
hest@nixos ~> claer
claer: command not found
hest@nixos ~ [127]>
```

As an added benefit, this also updated me from using the stable "LTS" release of 24.04 to nixos-unstable, the rolling release "unstable".

source

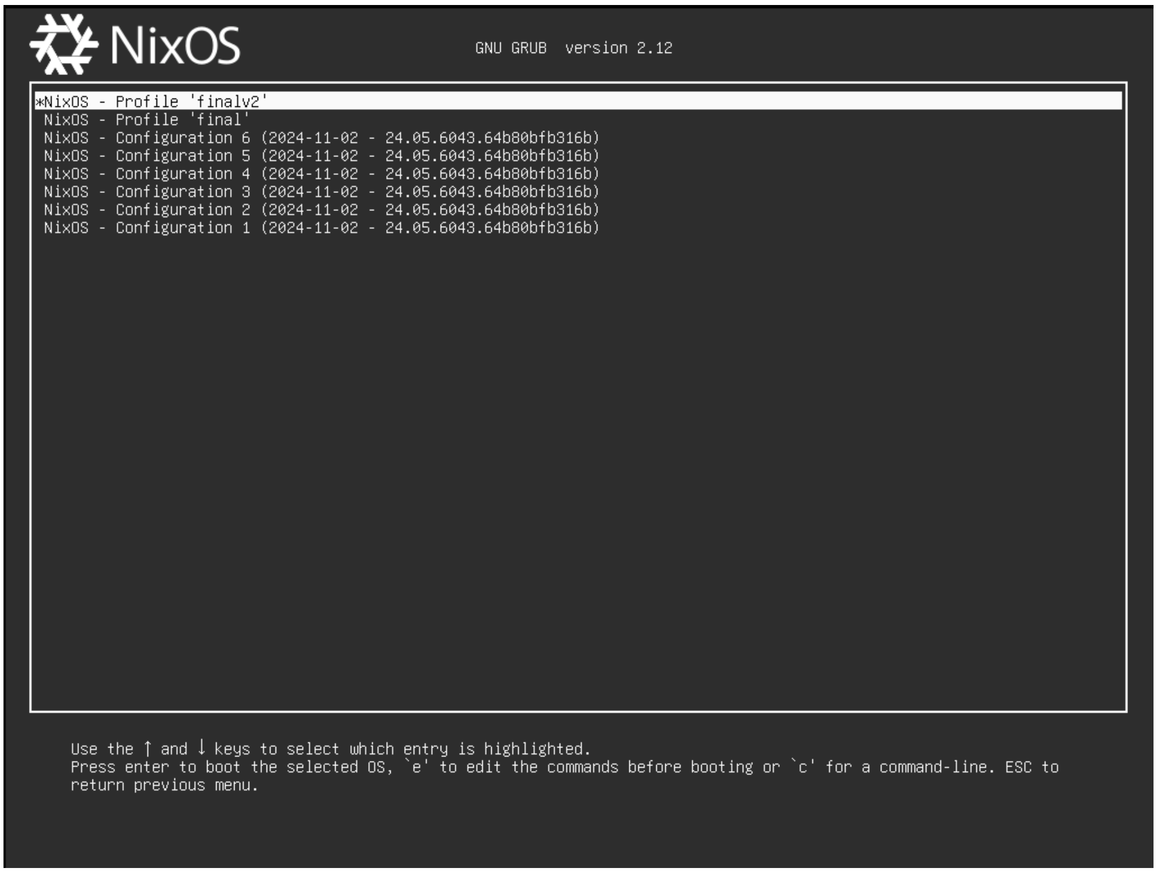
CLEAN UP THE NIX STORE

As a final step in this config, I wanted to tidy up around the Nix store and bootloader.

With all the different derivations and generations, the size of the Nix store can grow rather large after a couple of weeks of tinkering and rebuilding.

```
hest@nixos ~> du /nix/store/ -sh
21G /nix/store/
```

21 GB might be all right, but this will snowball, with every new package adding to the total. Every generation will add a new record to our bootloader, making that swell up.



```
NixOS                                GNU GRUB  version 2.12

*NixOS - Profile 'finalv2'
 NixOS - Profile 'final'
 NixOS - Configuration 6 (2024-11-02 - 24.05.6043.64b80bfb316b)
 NixOS - Configuration 5 (2024-11-02 - 24.05.6043.64b80bfb316b)
 NixOS - Configuration 4 (2024-11-02 - 24.05.6043.64b80bfb316b)
 NixOS - Configuration 3 (2024-11-02 - 24.05.6043.64b80bfb316b)
 NixOS - Configuration 2 (2024-11-02 - 24.05.6043.64b80bfb316b)
 NixOS - Configuration 1 (2024-11-02 - 24.05.6043.64b80bfb316b)

    Use the ↑ and ↓ keys to select which entry is highlighted.
    Press enter to boot the selected OS, `e' to edit the commands before booting or `c' for a command-line. ESC to
    return previous menu.
```

Step one: Let's see how to clean up old generations manually.

Starting by finding all the stored generations on the system:

```
nix-env --list-generations
```

`nix-env` does do the trick, but since I have installed everything systemwide (using the sytemPackages domain), I also need to point out that I'm using the system profile in this case:

```
sudo nix-env --list-generations --profile /nix/var/nix/profiles/system

   1   2024-10-12 02:11:56
   2   2024-10-12 15:33:14
   3   2024-10-12 17:59:32
   4   2024-10-12 21:59:04
   5   2024-10-20 13:16:04
   6   2024-10-20 13:21:57
   7   2024-10-20 20:45:41
   8   2024-10-22 21:23:08 (current)
```

Let's remove every generation except the current one.

```
sudo nix-env --delete-generations old --profile /nix/var/nix/profiles/sy
sudo nix-env --list-generations --profile /nix/var/nix/profiles/system

   8   2024-10-22 21:23:08 (current)
```

Step two: clean up the Nix store. Now that we have some dangling packages from the old generations, let's look at the store. The nix-store cli lets us do manual garbage collection:

```
nix-store --gc
deleting unused links...
note: currently, hard linking saves 4337.25 MiB
1155 store paths deleted, 654.20 MiB freed
```

However, according to the documentation, this is a command you should not have to handle manually in normal circumstances.

Step three: automation. So, let's automate this using our configuration instead.

The Wiki shows that it's possible to automatically handle cleanup for the generations and the store in several ways.

The most straightforward option seems to be a scheduled cleanup, and I think I will go for once a week, and only keep the last three generations, so if anything goes wrong in the future, I have something to revert to.

For the store cleanup, I opted for an even easier option: optimizing after every build. So, from now on, every time I build a new generation, I clean up dangling packages.

```
nix.gc = {
  automatic = true;
  dates = "weekly";
  options = "--delete-older-than +3";
};

nix.settings.auto-optimise-store = true;
```

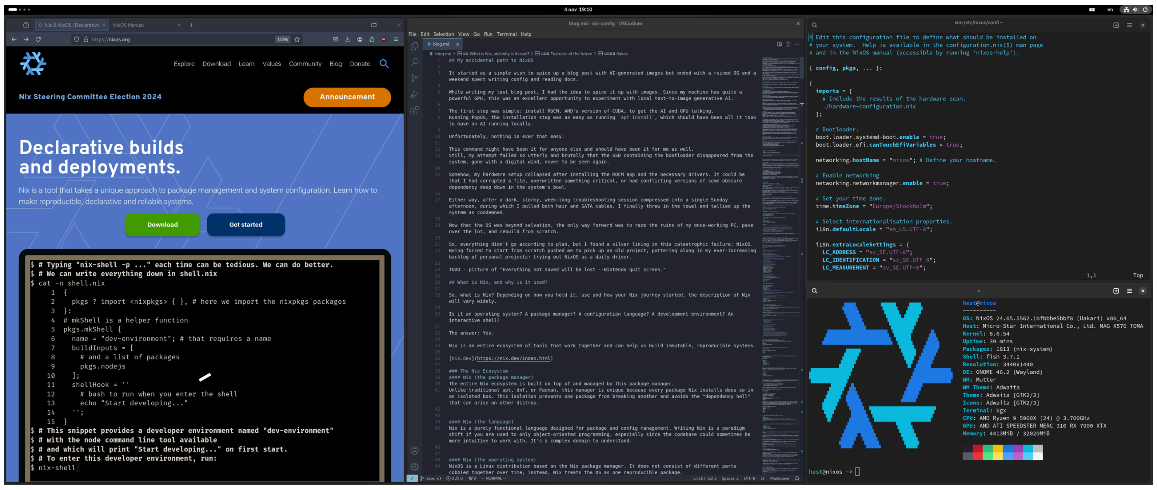
FINALE

And with that, the MVP config for my NixOS build is complete, I've used this configuration as a daily driver for about two to three weeks.

I did almost all of the config work upfront and during the last two weeks, I've only added some minor tweaks: replacing a package here and adding some extra configuration there. Overall, the system has been rock solid, and I'm pleased with the results.

Now for the finishing touch, the pièce de résistance of this build: I will completely wipe my hard drive, reinstall NixOS from scratch using my config file, and see if anything is missing.

So, let's save this config to Git Hub, wipe it all, and see what happens.



4 nov 19:10
Nix Steering Committee Election 2024
Announcement
Explore Download Learn Values Community Blog Donate
Declarative builds and deployments.
Download
Get started
blog.md - nix-config - VSCodium
File Edit Selection View Go Run Terminal Help
vim /etc/nixos/confi ~

Nix is a tool that takes a unique approach to package management and system configuration. Learn how to make reproducible, declarative and reliable systems.

```
$ # Typing "nix-shell -p ..." each time can be tedious. We can do better.
$ # We can write everything down in shell.nix
$ cat -n shell.nix
     1  {
     2    pkgs ? import <nixpkgs> { }, # here we import the nixpkgs packages
     3  }:
     4  # mkShell is a helper function
     5  pkgs.mkShell {
     6    name = "dev-environment"; # that requires a name
     7    buildInputs = [
     8      # and a list of packages
     9      pkgs.nodejs
    10    ];
    11    shellHook = ''
    12      # bash to run when you enter the shell
    13      echo "Start developing..."
    14    '';
    15  }
$ # This snippet provides a developer environment named "dev-environment"
$ # with the node command line tool available
$ # and which will print "Start developing..." on first start.
$ # To enter this developer environment, run:
$ nix-shell
```

```
blog.md > ## What is Nix, and why is it used? > ### Features of the future > #### flakes
## My accidental path to NixOS

It started as a simple wish to spice up a blog post with AI-generated images but ended with a ruined OS and a
weekend spent writing config and reading docs.

While writing my last blog post, I had the idea to spice it up with images. Since my machine has quite a
powerful GPU, this was an excellent opportunity to experiment with local text-to-image generative AI.

The first step was simple: install ROCM, AMD's version of CUDA, to get the AI and GPU talking.
Running PopOS, the installation step was as easy as running `apt install`, which should have been all it took
to have an AI running locally.

Unfortunately, nothing is ever that easy.

This command might have been it for anyone else and should have been it for me as well.
Still, my attempt failed so utterly and brutally that the SSD containing the bootloader disappeared from the
system, gone with a digital wind, never to be seen again.

Somehow, my hardware setup collapsed after installing the ROCM app and the necessary drivers. It could be
that I had corrupted a file, overwritten something critical, or had conflicting versions of some obscure
dependency deep down in the system's bowl.

Either way, after a dark, stormy, week-long troubleshooting session compressed into a single Sunday
afternoon, during which I pulled both hair and SATA cables, I finally threw in the towel and tallied up the
system as condemned.

Now that the OS was beyond salvation, the only way forward was to raze the ruins of my once-working PC, pave
over the lot, and rebuild from scratch.

So, everything didn't go according to plan, but I found a silver lining in this catastrophic failure: NixOS.
Being forced to start from scratch pushed me to pick up an old project, puttering along in my ever-increasing
backlog of personal projects: trying out NixOS as a daily driver.

TODO - picture of "Everything not saved will be lost --Nintendo quit screen."

## What is Nix, and why is it used?

So, what is Nix? Depending on how you hold it, use and how your Nix journey started, the description of Nix
will vary widely.

Is it an operating system? A package manager? A configuration language? A development environment? An
interactive shell?

The answer: Yes.

Nix is an entire ecosystem of tools that work together and can help us build immutable, reproducible systems.

[nix.dev](https://nix.dev/index.html)

### The Nix Ecosystem
#### Nix (the package manager)
The entire Nix ecosystem is built on top of and managed by this package manager.
Unlike traditional apt, dnf, or Pacman, this manager is unique because every package Nix installs does so in
an isolated box. This isolation prevents one package from breaking another and avoids the "dependency hell"
that can arise on other distros.


#### Nix (the language)
Nix is a purely functional language designed for package and config management. Writing Nix is a paradigm
shift if you are used to only object-oriented programming, especially since the codebase could sometimes be
more intuitive to work with. It's a complex domain to understand.


#### Nix (the operating system)
NixOS is a Linux distribution based on the Nix package manager. It does not consist of different parts
cobbled together over time; instead, Nix treats the OS as one reproducible package.
nixos  0 0  0  -- NORMAL --                    Ln 107, Col 2  Spaces: 2  UTF-8  LF  Markdown
```

```
# Edit this configuration file to define what should be installed on
# your system.  Help is available in the configuration.nix(5) man page
# and in the NixOS manual (accessible by running 'nixos-help').

{ config, pkgs, ... }:

{
  imports = [
    # Include the results of the hardware scan.
    ./hardware-configuration.nix
  ];

  # Bootloader.
  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;

  networking.hostName = "nixos"; # Define your hostname.

  # Enable networking
  networking.networkmanager.enable = true;

  # Set your time zone.
  time.timeZone = "Europe/Stockholm";

  # Select internationalisation properties.
  i18n.defaultLocale = "en_US.UTF-8";

  i18n.extraLocaleSettings = {
    LC_ADDRESS = "sv_SE.UTF-8";
    LC_IDENTIFICATION = "sv_SE.UTF-8";
    LC_MEASUREMENT = "sv_SE.UTF-8";
                                                              1,1           Top
```

```
hest@nixos
----------
OS: NixOS 24.05.5562.1bfbbbe5bbf8 (Uakari) x86_64
Host: Micro-Star International Co., Ltd. MAG X570 TOMA
Kernel: 6.6.54
Uptime: 39 mins
Packages: 1813 (nix-system)
Shell: fish 3.7.1
Resolution: 3440x1440
DE: GNOME 46.2 (Wayland)
WM: Mutter
WM Theme: Adwaita
Theme: Adwaita [GTK2/3]
Icons: Adwaita [GTK2/3]
Terminal: kgx
CPU: AMD Ryzen 9 5900X (24) @ 3.700GHz
GPU: AMD ATI SPEEDSTER MERC 310 RX 7900 XTX
Memory: 4413MiB / 32020MiB

hest@nixos ~>
```

And we're back!

The installation took only about 30 minutes, from pressing reboot until I started writing here again. That is, after checking that everything is in place, logging into the most crucial apps, and setting up GitHub access,

All in all, the manual steps I still need to perform are:

- copy my configuration file from my public GitHub repo
- nixos-rebuild switch

- reboot to get the gnome extension pop-shell to show up.
- enable pop shell and toggle tiling windows.
- mount disks + run sudo nixos-generate-config
- update nix-channel to handle "command not found"
- login to bitwarden, Firefox Google account, slack, discord, spotube, supersonic Log in to Steam, enable the compatibility layer, and add extra hard drives.
- pair up the PS5 controller over Bluetooth.

So, there are still some minor "issues" that I would like to remedy, and with Flakes and Home Manager, I think I will be able to resolve them.

Is handling app logins through config possible or even desirable? A private Github gist with secrets and access tokens could handle it. But I have to look into that fine-tuning in the future.

Speaking of the future, I might as well provide my current TODO list for NixOS while I'm at it, in no particular order:

- handle config
  - Flakes
  - Home Manager
- look into nix linters:
  - Statix
  - NixFmt
  - Alejandra
- security hardening
  - Nix-Mineral
- package management
  - NUR
  - sofware center
- building widgets
  - AGS
- extra inspiration
  - Awsome-nix
  - Wayland-nix
  - tips & tricks
- Userfriendly alternative
  - SnowflakeOS

But for today, I will leave it at that. I hope I have given a fair view of what a journey with NixOS Desktop can involve from a beginner's point of view.

Thank you for sticking around. I will leave you with the final config for this session.

CONFIGURATION.NIX

```
# Edit this configuration file to define what should be installed on
# your system.  Help is available in the configuration.nix(5) man page
# and in the NixOS manual (accessible by running ‘nixos-help’).

{ config, pkgs, ... }:

{
  imports = [
    # Include the results of the hardware scan.
    ./hardware-configuration.nix
  ];

  # Bootloader.
  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;

  networking.hostName = "nixos"; # Define your hostname.

  # Enable networking
  networking.networkmanager.enable = true;

  # Set your time zone.
  time.timeZone = "Europe/Stockholm";

  # Select internationalisation properties.
  i18n.defaultLocale = "en_US.UTF-8";

  i18n.extraLocaleSettings = {
    LC_ADDRESS = "sv_SE.UTF-8";
    LC_IDENTIFICATION = "sv_SE.UTF-8";
    LC_MEASUREMENT = "sv_SE.UTF-8";
    LC_MONETARY = "sv_SE.UTF-8";
    LC_NAME = "sv_SE.UTF-8";
    LC_NUMERIC = "sv_SE.UTF-8";
    LC_PAPER = "sv_SE.UTF-8";
    LC_TELEPHONE = "sv_SE.UTF-8";
    LC_TIME = "sv_SE.UTF-8";
  };

  # Enable the X11 windowing system.
  services.xserver.enable = true;

  # Enable the GNOME Desktop Environment.
  services.xserver.displayManager.gdm.enable = true;
  services.xserver.desktopManager.gnome.enable = true;

  # autologin without password
  services.displayManager.autoLogin.enable = true;
  services.displayManager.autoLogin.user = "hest";

  # Configure keymap in X11
  services.xserver.xkb = {
    layout = "us";
    variant = "intl";
  };

  # enable tailscale VPN
  services.tailscale.enable = true;

  # enable zsa udev rules
  hardware.keyboard.zsa.enable = true;

  # Enable sound with pipewire.
  hardware.pulseaudio.enable = false;
  security.rtkit.enable = true;
  services.pipewire = {
    enable = true;
    alsa.enable = true;
    alsa.support32Bit = true;
    pulse.enable = true;
  };

  # Define a user account.
  users.users.hest = {
    isNormalUser = true;
    description = "hest";
    shell = pkgs.fish;
    extraGroups = [
      "networkmanager"
      "wheel"
    ];
    packages = with pkgs; [
      #  thunderbird
    ];
  };

  # Workaround for GNOME autologin: https://github.com/NixOS/nixpkgs/iss
  systemd.services."getty@tty1".enable = false;
  systemd.services."autovt@tty1".enable = false;

  # enable containerization ( podman )
  virtualisation.containers.enable = true;
  virtualisation = {
    libvirtd = {
      enable = true;
    };
    podman = {
      enable = true;

      # Create a `docker` alias for podman, to use it as a drop-in repla
      dockerCompat = true;

      # Required for containers under podman-compose to be able to talk
      defaultNetwork.settings.dns_enabled = true;
    };
  };

  # Install firefox.
  programs.virt-manager.enable = true;
  programs.firefox.enable = true;
  programs.steam = {
    enable = true;
    remotePlay.openFirewall = true; # Open ports in the firewall for Ste
    dedicatedServer.openFirewall = true; # Open ports in the firewall fo
    localNetworkGameTransfers.openFirewall = true; # Open ports in the 
    localNetworkGameTransfers.openFirewall = true; # Open ports in the F
  };
  programs.fish.enable = true;

  # Allow unfree packages
  nixpkgs.config.allowUnfree = true;

  environment.systemPackages = with pkgs; [
    # basics
    wget
    vim
    gnomeExtensions.pop-shell

    # CLI
    navi
    tealdeer
    jq
    yq
    dasel # https://github.com/tomwright/dasel
    lf
    bat
    eza
    fzf
    zellij
    tre-command
    radeontop
    xclip
    neofetch

    # pw-manager
    bitwarden

    # coms
    discord
    slack

    # media
    supersonic
    spotube

    # drawing
    krita

    # containers
    podman
    podman-compose
    podman-desktop

    # Emulation
    wineWowPackages.waylandFull # windows
    darling # macos
    virtiofsd

    # dev
    git
    gh
    vscodium
    mise
```

```
    # VPN
    tailscale

    # zsa keymapper for moonlander
    keymapp

    # Nix
    nixfmt-rfc-style
    nixpkgs-fmt
  ];

  nix.settings.experimental-features = [
    "nix-command"
    "flakes"
  ];

  nix.gc = {
    automatic = true;
    dates = "weekly";
    options = "--delete-older-than +3";
  };

  nix.settings.auto-optimise-store = true;

  system.stateVersion = "24.05"; # Did you read the comment?

}
```

HARDWARE-CONFIGURATION.NIX

```
# Do not modify this file!  It was generated by ‘nixos-generate-config’
# and may be overwritten by future invocations.  Please make changes
# to /etc/nixos/configuration.nix instead.
{ config, lib, pkgs, modulesPath, ... }:

{
  imports =
    [ (modulesPath + "/installer/scan/not-detected.nix")
    ];

  boot.initrd.availableKernelModules = [ "nvme" "xhci_pci" "ahci" "usbhi
  boot.initrd.kernelModules = [ ];
  boot.kernelModules = [ "kvm-amd" ];
  boot.extraModulePackages = [ ];

  fileSystems."/" =
    { device = "/dev/disk/by-uuid/d15bc54f-14a3-46ad-ac2e-4491cf0ef8e1";
      fsType = "ext4";
    };

  fileSystems."/boot" =
    { device = "/dev/disk/by-uuid/8DB2-391E";
      fsType = "vfat";
      options = [ "fmask=0077" "dmask=0077" ];
    };

  fileSystems."/mnt/17a50a65-0cf0-43f3-ad12-c04a35e5e00d" =
    { device = "/dev/disk/by-uuid/17a50a65-0cf0-43f3-ad12-c04a35e5e00d";
      fsType = "ext4";
    };

  fileSystems."/mnt/bb0d0a4f-d7af-44ad-8dda-89bd4b8b646b" =
    { device = "/dev/disk/by-uuid/bb0d0a4f-d7af-44ad-8dda-89bd4b8b646b";
      fsType = "ext4";
    };

  swapDevices =
    [ { device = "/dev/disk/by-uuid/ba63ca35-0d86-408c-b15c-623de0039ec
    ];

  # Enables DHCP on each ethernet and wireless interface. In case of sc
  # (the default) this is the recommended approach. When using systemd-
  # still possible to use this option, but it's recommended to use it i
  # with explicit per-interface declarations with `networking.interface
  networking.useDHCP = lib.mkDefault true;
  # networking.interfaces.enp38s0.useDHCP = lib.mkDefault true;
  # networking.interfaces.wlo1.useDHCP = lib.mkDefault true;

  nixpkgs.hostPlatform = lib.mkDefault "x86_64-linux";
  hardware.cpu.amd.updateMicrocode = lib.mkDefault config.hardware.enabl
}
```