Linux Server Handbook

Flavio Copes
flaviocopes.com

November 2026

Contents

What a Linux server is 4
Finding your way around Ubuntu 7
Packages and updates 13
Creating the VPS 18
The first hour: locking it down 23
A domain for the server 29
Services with systemd 32
Timers instead of cron 45
Reading the logs 48
Nginx 52
Caddy, and when to pick it 64
Disks and filesystems 69
Backups you can restore 79
Security, day two 87
Monitoring and alerts 92
A troubleshooting method 95
A home server 102
Tailscale 107
How I run Linux servers 114

Preface

This is the Linux Server Handbook, written by Flavio Copes. You can find more programming tutorials and free handbooks at https://flaviocopes.com.

In this book we take one Ubuntu server from nothing to production, and then we keep it running.

We start with an empty Droplet on DigitalOcean. We lock it down in the first hour, run a small Node.js API as a systemd service, put Nginx and HTTPS in front of it, move its data to a separate volume, back it up with restic and restore it on purpose. Then we break it in a few different ways to learn how to find out what went wrong.

The same small app follows us through the whole book. It's a notes API, about 60 lines of JavaScript with no dependencies, so the focus stays on the server and not on the code.

Every command and every output in this book comes from a real run. In November 2026 I created a fresh Ubuntu 24.04 Droplet and went through the book from the first SSH connection to the last restore drill. Then I destroyed it.

Some things went wrong along the way, and I kept them in. A leftover process blocked the port of the new service, an Nginx directive didn't exist in the installed version, a Docker container ignored the firewall. That's what a real server does, and seeing how it fails is the most useful part.

The server's real IP addresses are replaced by documentation addresses. The Droplet is `203.0.113.42` (and `203-0-113-42.sslip.io`), and my home connection is `203.0.113.7`.

These are the versions I used:

Software	Version
Ubuntu	24.04.5 LTS (Noble Numbat)
Linux kernel	6.8.0-151-generic
systemd	255
OpenSSH	9.6p1
UFW	0.36.2
Node.js	24.22.3 (NodeSource)
Nginx	1.24.0
Certbot	2.9.0
Caddy	2.11.9
LVM	2.03.16
restic	0.16.4
Docker	29.1.3
Docker Compose	2.40.3

You need a computer with a terminal and SSH, a DigitalOcean account (any VPS

provider works, the commands after the first chapter are the same on any Ubuntu 24.04 server), and a couple of hours. The Droplet I used costs $6 a month, which is less than a cent an hour while you follow along.

What a Linux server is

A server is a computer that runs programs for other computers, all day, without anyone sitting in front of it.

That's it. There's no special hardware that makes a machine a server.

Your laptop becomes a server the moment it runs a web server and someone else connects to it. What changes is the job: a server is expected to be up when you're asleep, to start its programs again after a reboot, and to stay reachable while strangers on the internet poke at it.

The server we build in this book

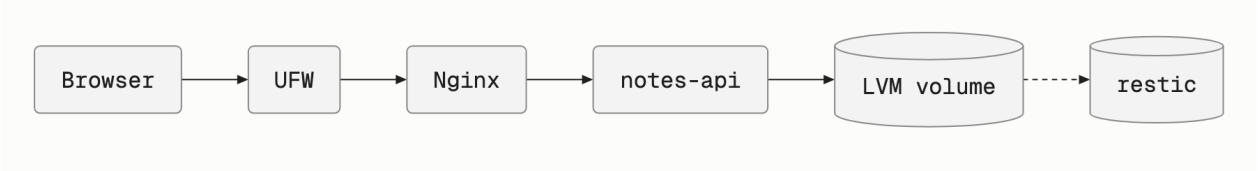
Everything you'll read comes from one real machine I created for this book: an Ubuntu 24.04 LTS virtual server on DigitalOcean, called `handbook-linux-test`, with 1 CPU and 1 GB of RAM, in the Frankfurt region. It costs $6 a month.

On it we run a small Node.js application, the **notes API**. It saves short text notes in a JSON file and answers on two URLs, `/health` and `/notes`.

It's small on purpose. The app is not the point.

The point is everything around it: the user it runs as, the service that keeps it alive, the reverse proxy in front of it, the disk its data lives on, the backups, the logs, the alerts.

Here's the path a request takes once the whole book is done:



The browser finds the server through DNS. UFW lets in only ports 22, 80 and 443. Nginx terminates HTTPS and passes the request to the notes API on `127.0.0.1:3000`, a systemd service that keeps its data on a separate LVM volume, backed up every night with restic.

Each box is a chapter. We'll build them in that order, from the left.

VPS, home server or managed platform

Before renting anything, ask yourself whether you need a server you maintain yourself at all.

There are three common homes for an application:

Option	Typical cost	Network	Who patches the OS	Good fit
Cloud VPS (DigitalOcean, Hetzner)	$4 to $20 a month	Static public IP, data center power and network	You	Web apps, APIs, databases, self-hosted tools, background jobs
Home server (mini PC, old laptop)	Hardware once, then electricity	Residential connection, often behind NAT	You	Media, personal backups, private tools, build boxes
Managed platform (edge functions, PaaS)	Free tier, then usage based	Global, managed for you	The provider	Static sites, small APIs, webhooks

A server you run yourself is great for anything stateful or long-running: a database, a newsletter server, a job that runs for an hour, a self-hosted analytics tool. It's overkill for a static marketing site, which a managed platform serves for free and better.

I run both kinds. Static sites go on managed platforms. Things that need a disk, a long-running process or a database I control go on a VPS.

The last chapter covers the home server option. Most of this book is about the VPS.

What a VPS is, and what's your job

A **VPS**, a Virtual Private Server, is a virtual machine running on a provider's hardware. DigitalOcean calls them **Droplets**.

From the inside it looks like a normal Linux machine: its own users, processes, disks, network interfaces and a public IP address. From the outside, the provider can create, resize, snapshot or destroy it in seconds through a control panel or an API.

The split of responsibilities is what matters most.

DigitalOcean runs the data center, the physical machines and the virtualization layer. Everything inside the operating system is your job:

- user accounts and SSH access
- operating system and application updates
- the firewall on the server
- the web server and the application services
- the application data and its secrets
- backups, restore tests, monitoring and what you do when something breaks

DigitalOcean will happily show the Droplet as ”on” while your app has crashed, the disk is full or the TLS certificate expired. Nobody watches those things for you.

Most of this book is about making that work small and predictable.

Linux, the kernel and the distribution

Linux, strictly speaking, is the **kernel**: the program at the center of the operating system that manages processes, memory, disks, the network and the hardware. Linus Torvalds started it in 1991.

A kernel alone can't do much for you. It has no shell, no editor, no way to install software.

A **distribution** wraps the kernel with system tools, libraries, a package manager and a collection of maintained software. Debian, Fedora and Ubuntu are distributions.

They all run the Linux kernel, and they differ in everything around it.

On the server, two commands show the two layers. The distribution:

```
cat /etc/os-release | head -4
```

```
PRETTY_NAME="Ubuntu 24.04.5 LTS"
NAME="Ubuntu"
VERSION_ID="24.04"
VERSION="24.04.5 LTS (Noble Numbat)"
```

And the kernel it's running:

```
uname -r
```

```
6.8.0-149-generic
```

Why does the difference matter? Because instructions are distribution-specific.

Ubuntu installs software with `apt`, Fedora with `dnf`. When a tutorial's install command says `command not found`, the first thing I check is `/etc/os-release`.

Usually I'm reading instructions for a different distribution.

Kernel space and user space

The kernel is the only code with full control of the hardware. Every other program runs in **user space**, with limited privileges, and asks the kernel to do things for it through **system calls**: open this file, send these bytes, start this process.

Even `root` runs in user space. Root is a powerful account that the kernel trusts broadly, but a root shell still goes through system calls like every other program.

This split gives you a useful rule when something breaks.

One application crashing again and again is a user space problem: look at that program, its configuration and its logs. The whole machine freezing or rebooting by itself points at the kernel, a driver or the hardware, and that's much rarer.

On our server the notes API crashed a few times while I wrote this book. The machine never did.

Ubuntu LTS

Ubuntu ships a new release every six months.

Every two years, the April release is a **Long Term Support** release, LTS. 24.04 is one, released in April 2024, codenamed Noble Numbat.

`lsb_release` prints the same facts in a shorter form:

```
lsb_release -a
```

```
Distributor ID: Ubuntu
Description:    Ubuntu 24.04.5 LTS
Release:    24.04
Codename:   noble
```

For a server I always pick an LTS release.

It gets security updates for years, I upgrade it to the next major version only every couple of years, and tutorials written for it stay valid longer. The `.5` in `24.04.5` is a point release: the same 24.04 with all updates rolled in, so a new install has less to download.

Keep the codename in mind, `noble`. It shows up in package repository names, and in the next chapters you'll see it in the APT output.

Finding your way around Ubuntu

When I connect to a server for the first time, before changing anything, I want to know what I'm talking to. Which machine, which release, how much memory, how much disk.

In this chapter and the next one I'm logged in as `root` on the brand new Droplet, the way you are after the very first SSH connection. The chapter "Creating the VPS" shows how I created it and how we stop using root.

If you want to type along now, jump there, create the server, and come back.

Identify the system

`hostnamectl` gives a good summary in one go:

```
hostnamectl
```

```
 Static hostname: handbook-linux-test
       Icon name: computer-vm
         Chassis: vm
```

```
      Machine ID: 7afa1aad300342ee8878d9fb24051219
         Boot ID: 1710cae489654efb8bb97805e92e8cb6
  Virtualization: kvm
Operating System: Ubuntu 24.04.5 LTS
          Kernel: Linux 6.8.0-149-generic
    Architecture: x86-64
 Hardware Vendor: DigitalOcean
  Hardware Model: Droplet
Firmware Version: 20171212
   Firmware Date: Tue 2017-12-12
    Firmware Age: 8y 11month 2w 2d
```

It's a virtual machine (`kvm`), on x86-64, running Ubuntu 24.04.5 with kernel 6.8.0-149. The firmware date is the virtual BIOS DigitalOcean provides, not the age of any real hardware.

The architecture decides which programs you can run.

A binary built for `x86_64` doesn't run on an ARM server, and the other way around. When I download a release from GitHub, this is the line that tells me which file to pick.

Then the resources:

```
nproc; free -h
```

```
1
               total        used        free      shared  buff/cache   available
Mem:           961Mi       322Mi       227Mi       4.0Mi       560Mi       639Mi
Swap:             0B          0B          0B
```

One CPU and about 1 GB of RAM.

Look at the `available` column, not `free`. Linux uses spare memory as a disk cache (`buff/cache`) and gives it back as soon as a program needs it, so `available` is the honest number: 639 MB.

Notice there's no swap. DigitalOcean images ship without it.

When a program asks for more memory than the server has, there's nowhere to spill over, and the kernel kills something. We'll make that happen on purpose in the troubleshooting chapter.

And the disk:

```
df -h /
```

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/vda1        24G  2.1G   22G   9% /
```

A 24 GB disk with 2.1 GB used by a fresh Ubuntu. The `-h` flag prints human-readable sizes.

Before any command that deletes, reboots or changes many things at once, I print the host and the user. It takes two seconds and it's the cheapest protection against running the right command on the wrong server:

```
printf 'host=%s user=%s\n' "$(hostname)" "$(whoami)"
```

The filesystem tree

Linux has no drive letters. There's one tree of directories, starting at `/`, and every file on the machine lives somewhere in it.

A second disk doesn't become `D:`: it gets **mounted** on a directory inside the tree. We'll do exactly that with a data volume later in the book.

Here's the top of the tree on our server:

```
ls /
```

```
bin
bin.usr-is-merged
boot
dev
etc
home
lib
lib.usr-is-merged
lib64
lost+found
media
mnt
opt
proc
root
run
sbin
sbin.usr-is-merged
snap
srv
sys
tmp
usr
var
```

The `*.usr-is-merged` directories look odd.

On modern Ubuntu, `/bin`, `/sbin` and `/lib` are links into `/usr`, and those three directories are markers left behind by that change. You can ignore them.

The directories that matter

A few directories cover almost everything you'll touch on a server.

`/etc` holds system-wide configuration. The SSH server config, the Nginx sites, the firewall rules, the systemd units we'll write: all under `/etc`.

`/var` holds data that changes while the system runs.

Logs in `/var/log`, package caches, and application state in `/var/lib`. Our notes API will keep its data in `/var/lib/notes-api`.

`/usr` holds the installed programs and their shared files. `/usr/bin` alone has hundreds of commands.

`/home` has one folder per regular user. `/root` is root's home, outside `/home`, and regular users can't even list it.

`/tmp` is for temporary files, and its contents don't survive a reboot.

`/srv` and `/opt` are the conventional places for services and add-on software that don't come from the package manager. We'll put the notes API code in `/opt/notes-api` and the backup repository in `/srv/restic`.

`/dev`, `/proc`, `/sys` and `/run` aren't real files on disk.

The kernel generates them to expose devices, processes and runtime state as files. `/dev/null`, the black hole you redirect output into, lives in `/dev`.

The rule I keep in my head: configuration in `/etc`, changing data in `/var`, programs in `/usr`. And I never edit files under `/usr` to change how a program behaves.

Those files belong to the package manager, and the next update overwrites them without asking. The supported knobs are in `/etc`.

Configuration files, and who owns them

Before editing a file in `/etc`, I check whether it's the real source of truth or something generated by another tool.

Ubuntu on a cloud server has a good example. The list of package repositories begins like this:

```
cat /etc/apt/sources.list.d/ubuntu.sources | head -6
```

```
## Note, this file is written by cloud-init on first boot of an instance
## modifications made here will not survive a re-bundle.
##
## If you wish to make changes you can:
## a.) add 'apt_preserve_sources_list: true' to /etc/cloud/cloud.cfg
##     or do the same in user-data
```

cloud-init is the tool that configures a cloud server on its first boot: it sets the hostname, installs your SSH key, writes files like this one. A file with a header like that tells you where the real setting lives.

Many services also read a directory of small fragments next to their main file. SSH

is one of them:

```
ls -la /etc/ssh/sshd_config.d/
```

```
total 16
drwxr-xr-x 2 root root 4096 Nov 28 16:13 .
drwxr-xr-x 4 root root 4096 Nov 28 16:13 ..
-rw------- 1 root root   26 Nov 28 16:13 50-cloud-init.conf
-rw-r--r-- 1 root root   26 Oct 17 12:57 60-cloudimg-settings.conf
```

Two fragments: one written by cloud-init when the server booted, one shipped in the Ubuntu cloud image. When we harden SSH, we'll add our own fragment here instead of editing the main `sshd_config`.

A small file of your own is easy to review and easy to remove when it breaks something.

When I'm not sure where a file comes from, `dpkg -S` tells me which package installed it.

The safe sequence for any configuration change is always the same: keep a copy of the current file, change one thing, run the program's own config check, reload only if the check passes, then test the real behavior. We'll do it with SSH, Nginx and Caddy.

Reading permissions

Look again at that `ls -la` output. The two fragments have different permissions:

```
-rw------- 1 root root   26 Nov 28 16:13 50-cloud-init.conf
-rw-r--r-- 1 root root   26 Oct 17 12:57 60-cloudimg-settings.conf
```

The first column is the permission string. The first character is the type: `-` for a file, `d` for a directory, `l` for a link. The other nine are three groups of three, for the **owner**, the **group** and **everyone else**, and each group says whether that class can read (`r`), write (`w`) and execute (`x`).

So `-rw-------` means the owner, root, can read and write it, and nobody else can do anything. `-rw-r--r--` means root can read and write, and everyone else can read.

The next two columns after the link count are the owner and the group, here both `root`.

You'll often see permissions written as numbers. Each digit is the sum of read (4), write (2) and execute (1) for one class: `rw-------` is `600`, `rw-r--r--` is `644`, and `rwxr-x---` is `750`. On a directory, `x` means you can enter it, so a directory is usually `755` or `750` where a file would be `644` or `640`.

`chmod` changes the permissions and `chown` changes the owner:

```
sudo chmod 640 /etc/notes-api/env
sudo chown root:notes /etc/notes-api/env
```

These two lines are the shape of what we'll do with the notes API's secrets file later: owned by root, readable by the service's group, and invisible to everyone else.

Users, groups and sudo

Every process on Linux runs as some user, and every file has an owner and a group. That's how the system decides who can read or change what.

Right now I'm root:

```
id
```

```
uid=0(root) gid=0(root) groups=0(root)
```

User ID 0 is root, the account that can do anything. Working as root all day is how a typo becomes an outage, so in a moment we'll create a regular user and use `sudo` only for the commands that need it.

User accounts are listed in `/etc/passwd`, and `getent` reads them for you:

```
getent passwd | wc -l; getent group sudo
```

```
32
sudo:x:27:
```

32 accounts on a fresh server. Most of them aren't people: they're **system users** that services run as, like `www-data` for the web server or `systemd-resolve` for DNS.

Separate accounts mean a bug in one service can't read another service's files. We'll create one of these for the notes API.

The `sudo` group is empty. On Ubuntu, members of `sudo` can run commands as root by typing their own password. Adding our user to it is how they become an administrator.

`sudo` runs one command with elevated privileges and logs it.

One trap catches everyone once. This doesn't work:

```
sudo echo enabled > /etc/notes.conf
```

The redirection `>` is done by your own shell, as your user, before `sudo` even starts. To write a root-owned file, pipe into `sudo tee`:

```
echo enabled | sudo tee /etc/notes.conf
```

You'll see `sudo tee` many times in this book for that reason.

Packages and updates

Software on Ubuntu comes from **packages**, and **APT** is the tool that installs and updates them.

Ubuntu's repositories hold package files plus signed metadata: names, versions, dependencies, checksums. APT reads that metadata, works out what needs to be installed or upgraded, downloads the files, checks their signatures and hands them to `dpkg`, the lower-level tool that unpacks them.

The first apt update, and the lock

The first thing I did on the new server was refresh the package index:

```
apt update
```

And it failed:

```
Reading package lists...
E: Could not get lock /var/lib/apt/lists/lock. It is held by process 1616 (apt-get)
E: Unable to lock directory /var/lib/apt/lists/
```

APT allows only one process at a time to change the package database. Something else was already running `apt-get`, one minute after boot.

What was it?

```
ps -o pid,etime,cmd -C apt-get,unattended-upgr,apt.systemd.daily
```

```
    PID     ELAPSED CMD
    936       01:12 /usr/bin/python3 /usr/share/unattended-upgrades/unattended-upgrade-shutdown --wait-
    ↪ for-signal
   1187       01:06 apt-get -qq --fix-missing install -y droplet-agent droplet-agent-keyring
```

DigitalOcean installs its `droplet-agent` on first boot, and that holds the lock. The system wasn't even done starting:

```
systemctl is-system-running
```

```
starting
```

The fix is to wait, not to delete the lock file (a classic bad tip on forums, which can leave the package database half-written). cloud-init can tell you when first-boot setup is finished:

```
cloud-init status --wait
```

A minute later `systemctl is-system-running` said `running`, and `apt update` worked:

```
Reading state information...
35 packages can be upgraded. Run 'apt list --upgradable' to see them.
```

You'll hit this lock again later, when Ubuntu's automatic updates run in the background. Same answer: wait for the other process to finish.

Update the index, then read what changes

`apt update` doesn't upgrade anything. It only downloads fresh metadata, so APT knows which versions are available. Upgrading is a separate step, and I like to look at the list first:

```
apt list --upgradable
```

```
Listing...
krb5-locales/noble-updates,noble-security 1.20.1-6ubuntu2.12 all [upgradable from: 1.20.1-6ubuntu2.10]
libdw1t64/noble-updates,noble-security 0.190-1.1ubuntu0.2 amd64 [upgradable from: 0.190-1.1ubuntu0.1]
libelf1t64/noble-updates,noble-security 0.190-1.1ubuntu0.2 amd64 [upgradable from: 0.190-1.1ubuntu0.1]
libexpat1/noble-updates,noble-security 2.6.1-2ubuntu0.7 amd64 [upgradable from: 2.6.1-2ubuntu0.6]
libgssapi-krb5-2/noble-updates,noble-security 1.20.1-6ubuntu2.12 amd64 [upgradable from: 1.20.1-6
    ↪ ubuntu2.10]
...
```

There's the codename again: `noble-updates` and `noble-security` are the repositories for regular and security updates of 24.04. A fresh image is always a bit behind, so 35 updates on day one is normal.

You can ask APT which version of a package it would install and from where, before installing it:

```
apt-cache policy nginx | head -4
```

```
nginx:
  Installed: (none)
  Candidate: 1.24.0-2ubuntu7.20
  Version table:
```

Ubuntu keeps the Nginx version of the release (1.24) for the whole life of 24.04 and backports security fixes into it. That's why the version ends in `ubuntu7.20`: it's the 20th update of that package.

In November 2026 that exact version fixed two HTTP/2 bugs in Nginx, as I explained in Turn on automatic security updates on your VPS. The upstream version number stays old, but the package is patched.

Upgrade, and the kernel that was kept back

Now the upgrade itself:

```
apt upgrade
```

```
31 upgraded, 0 newly installed, 0 to remove and 4 not upgraded.
```

31 out of 35. Why were four left out? A simulation run shows it:

```
apt-get -s upgrade
```

```
The following packages have been kept back:
  linux-headers-generic linux-headers-virtual linux-image-virtual
  linux-virtual
0 upgraded, 0 newly installed, 0 to remove and 4 not upgraded.
```

Those are the kernel meta-packages. A new kernel means installing new packages (the new `linux-image-6.8.0-151-generic`), and plain `apt upgrade` never installs new packages, so it keeps them back.

`full-upgrade` is allowed to add and remove packages:

```
apt full-upgrade
```

```
The following NEW packages will be installed:
  linux-headers-6.8.0-151 linux-headers-6.8.0-151-generic
  linux-image-6.8.0-151-generic linux-modules-6.8.0-151-generic
...
The following packages will be upgraded:
  linux-headers-generic linux-headers-virtual linux-image-virtual
```

Kept-back packages aren't always kernels.

Ubuntu also rolls out some updates gradually (phased updates), and those show up as kept back for a few days. I don't force them one by one.

When a reboot is needed

A new kernel is installed but not running.

The running kernel changes only at boot. Ubuntu leaves a flag file when a reboot is needed, together with the list of packages that asked for it:

```
cat /var/run/reboot-required /var/run/reboot-required.pkgs
```

```
*** System restart required ***
linux-image-6.8.0-151-generic
linux-base
```

And `uname` confirms we're still on the old one:

```
uname -r
```

```
6.8.0-149-generic
```

On a server with nothing on it yet, I reboot right away:

```
systemctl reboot
```

On a server with users, I plan the reboot: a quiet hour, a working SSH session, a way to check the services come back. After about 30 seconds I reconnected:

```
uname -r; uptime
```

```
6.8.0-151-generic
 16:18:14 up 0 min,  1 user,  load average: 0.90, 0.27, 0.09
```

The new kernel is running. `last reboot` keeps the history of boots, with the kernel each one used:

```
last reboot | head -3
```

```
reboot   system boot  6.8.0-151-generi Sat Nov 28 16:17   still running
reboot   system boot  6.8.0-149-generi Sat Nov 28 16:13 - 16:17  (00:04)
```

The first boot lasted four minutes. That's how long it took to update a fresh server.

Installing and removing packages

Installing a package is one command. I added `htop`, a nicer process viewer, and `tree`:

```
apt install htop tree
```

APT prints a summary of what it will install, including dependencies, before doing anything. Read it.

If the list is much longer than you expected, stop.

Removing is the same shape:

```
apt remove tree
```

`apt remove` keeps the package's configuration files in `/etc`, in case you install it again. `apt purge` removes them too.

To read about a package before installing it:

```
apt-cache show htop | grep -E '^(Package|Version|Description-en)'
```

```
Package: htop
Version: 3.3.0-4build1
Description-en: interactive processes viewer
```

A fresh 24.04 Droplet has a few hundred packages installed. `dpkg -l` lists all of them, one per line:

```
dpkg -l | wc -l
```

```
676
```

A small detail for scripts: `apt` is designed for people at a terminal, and its output can change between versions. In scripts and automation, use `apt-get`, which has a stable interface.

Services that use old libraries

When I installed those packages, the last line APT printed came from a different tool:

```
No VM guests are running outdated hypervisor (qemu) binaries on this host.
```

That's **needrestart**, which Ubuntu runs after every package transaction. An upgraded library is on disk, but programs that were already running keep using the old copy in memory until they restart. needrestart finds those programs and, on a server, restarts their services for you. The line about VM guests is its last check, and it only matters on machines that host virtual machines.

So after an upgrade, a service may have restarted under you. Usually that's what you want, since the security fix only applies once the process restarts. It's also one more reason to check your services after an update: `systemctl --failed` lists anything that didn't come back.

Automatic security updates

Ubuntu 24.04 comes with **unattended-upgrades** already turned on. That's why you rarely need to run `apt upgrade` for security fixes, and why you'll sometimes find APT busy when you connect.

Two lines in a small config file enable it:

```
cat /etc/apt/apt.conf.d/20auto-upgrades
```

```
APT::Periodic::Update-Package-Lists "1";
APT::Periodic::Unattended-Upgrade "1";
```

Refresh the package lists every day, install the updates every day. By default it installs only packages from the security repository, and it doesn't reboot the server on its own.

Kernel updates installed this way still wait for a reboot, and they leave the same `/var/run/reboot-required` flag.

The service that coordinates it is running:

```
systemctl status unattended-upgrades --no-pager | head -5
```

```
unattended-upgrades.service - Unattended Upgrades Shutdown
    Loaded: loaded (/usr/lib/systemd/system/unattended-upgrades.service; enabled; preset: enabled)
    Active: active (running) since Sat 2026-11-28 16:13:35 UTC; 59s ago
      Docs: man:unattended-upgrade(8)
  Main PID: 936 (unattended-upgr)
```

That's PID 936, the same process we saw waiting next to the lock.

I keep it on for every server.

Whether to also let it reboot automatically, and which packages to include besides security fixes, depends on the server: I tested all those options on a Droplet in Turn on automatic security updates on your VPS. And Ubuntu isn't always fast with kernel fixes, which I wrote about in Ubuntu's slow kernel patches, and the model behind them.

One more distinction: `apt upgrade` updates packages **inside** 24.04. Moving to the next LTS release is a different operation, `do-release-upgrade`, and I treat it like a migration, with a backup and a plan.

Creating the VPS

Now let's go back to the beginning: renting the server and logging in for the first time without leaving the door open.

Pick the size and the region

I picked the smallest regular Droplet: 1 shared CPU, 1 GB of RAM, 25 GB of disk, $6 a month. That's enough for a small Node.js API, Nginx and a backup job.

Memory is usually the first thing a small server runs out of, so if your app keeps a big process in memory, start with 2 GB.

My advice is to start small, turn on monitoring, and resize only when you can point at the resource that ran out. Resizing a Droplet takes a few clicks.

The region is the hard choice to change later. Moving to another region means a new server, copying the data and moving traffic.

Pick the one closest to your users, or to the database the app talks to. I picked Frankfurt, `fra1`.

Know how billing works before you start. A powered-off Droplet still costs money, because it reserves the CPU, RAM and disk.

Only destroying it stops the charges. Volumes, snapshots and backups are billed separately and can keep charging after the Droplet is gone.

Create an SSH key

We log in with an SSH key, not a password.

A key pair has two halves: the **private key** stays on your computer and proves who you are, the **public key** goes on every server that should let you in. You can give the public half to anyone.

I create a separate key per project, so if one leaks I know exactly what it opens:

```
ssh-keygen -t ed25519 -f ~/.ssh/handbook_linux -C "handbook-linux-test"
```

Ed25519 is the modern default: short keys, fast, supported everywhere.

The `-C` comment ends up at the end of the public key, and it's how you'll recognize the key in a list a year from now. Pick a passphrase when it asks: it protects the private key if someone copies the file.

You get two files, `~/.ssh/handbook_linux` and `~/.ssh/handbook_linux.pub`.

The `.pub` one is the only one you ever paste anywhere. Never `cat` the private key into a chat, a server or a Git repository.

Check the key fingerprint

Before adding the public key to DigitalOcean, look at its fingerprint:

```
ssh-keygen -lf ~/.ssh/handbook_linux.pub
```

```
256 SHA256:XfvNML5wguVZCLchYJtkRcs9t7cUnpO9AygFNra258g handbook-linux-test (ED25519)
```

It prints the key type, a short `SHA256:` hash and the comment. After you add the key, the control panel shows a fingerprint for it too, and the two must match. A fingerprint is short enough to compare by eye, which a 70-character key line isn't.

Keep a copy of the private key somewhere safe, like a password manager. Losing it doesn't give anyone access to your server, but it locks you out. For a server that matters, I add a second admin key from another machine, so one lost laptop isn't a lockout.

Create the Droplet

From the DigitalOcean control panel the path is: **Create**, **Droplets**, choose the region, the **Ubuntu 24.04 (LTS) x64** image, the **Basic** plan with the size you

picked, then under **Authentication method** choose **SSH Key** and add the public key you just created. Give it a hostname that says what the machine does.

I skip the Marketplace images with Node or Docker preinstalled: starting from plain Ubuntu means I see every service that runs on the box.

I did the same from the terminal with `doctl`, DigitalOcean's command line tool. First the public key:

```
doctl compute ssh-key create handbook-linux-test --public-key "$(cat ~/.ssh/handbook_linux.pub)"
```

That returns the key's ID, which goes in the create command:

```
doctl compute droplet create handbook-linux-test \
  --region fra1 \
  --image ubuntu-24-04-x64 \
  --size s-1vcpu-1gb \
  --ssh-keys 61228784 \
  --wait
```

```
ID           Name                    Public IPv4      Status
624334546    handbook-linux-test     203.0.113.42     active
```

`--wait` blocks until the Droplet is up, and then prints the public IP. About a minute.

Before logging in, I go back to the control panel and check the region, size, image and key are what I meant. Mistakes are cheapest now, when the server has nothing on it.

Picked the wrong key or the wrong region? Destroy it and create the right one.

The first connection

```
ssh -i ~/.ssh/handbook_linux root@203.0.113.42
```

Two identities get checked in the first connection.

Your key proves who you are to the server. The server's **host key** proves which machine answered you.

Since this server isn't in your `~/.ssh/known_hosts` yet, OpenSSH shows the host key's fingerprint and asks if you trust it. Most people type `yes` without reading.

If you want to be sure, open the Droplet's **Console** in the control panel (a path that doesn't go through the network you're about to trust) and print the fingerprint there:

```
ssh-keygen -lf /etc/ssh/ssh_host_ed25519_key.pub
```

```
256 SHA256:Qoxs9ER2bc7lZ2spfQdOIkqlcuBDsK9cHPEkHKRx/04 root@handbook-linux-test (ED25519)
```

That's the same `SHA256:` value OpenSSH shows on your laptop in the prompt below.

When they match, accept. OpenSSH saves the key:

```
Warning: Permanently added '203.0.113.42' (ED25519) to the list of known hosts.
```

From now on, if the server's key ever changes, OpenSSH refuses to connect and prints `REMOTE HOST IDENTIFICATION HAS CHANGED!`. After you rebuild a server that's expected.

Out of the blue, it's a reason to stop and check.

What cloud-init did for us

How did my key end up on the server? cloud-init put it in root's `~/.ssh/authorized_keys` on first boot. It also turned off password logins.

Remember the two SSH fragments from earlier:

```
cat /etc/ssh/sshd_config.d/50-cloud-init.conf /etc/ssh/sshd_config.d/60-cloudimg-settings.conf
```

```
PasswordAuthentication no
PasswordAuthentication no
```

Good. But what does the SSH server actually use? `sshd -T` prints the effective configuration, after all files are read:

```
sshd -T | grep -E '^(permitrootlogin|passwordauthentication|kbdinteractiveauthentication|
    ↪ pubkeyauthentication|port) '
```

```
port 22
permitrootlogin yes
pubkeyauthentication yes
passwordauthentication no
kbdinteractiveauthentication no
```

Passwords are off, but `root` can still log in with a key. We'll close that in the next chapter, once we have another way in.

Create your own user

First a regular user for me:

```
adduser --disabled-password --gecos '' flavio
```

```
info: Adding user `flavio' ...
info: Selecting UID/GID from range 1000 to 59999 ...
info: Adding new group `flavio' (1000) ...
info: Adding new user `flavio' (1000) with group `flavio (1000)' ...
info: Creating home directory `/home/flavio' ...
info: Copying files from `/etc/skel' ...
```

```
info: Adding new user `flavio' to supplemental / extra groups `users' ...
info: Adding user `flavio' to group `users' ...
```

Without the flags, `adduser` asks for a password and a few details like your full name. The user still needs a password, though, because `sudo` asks for it.

That password is never used for SSH (password logins are off), only to confirm `sudo`:

```
passwd flavio
```

Then the user joins the `sudo` group:

```
usermod -aG sudo flavio && id flavio
```

```
uid=1000(flavio) gid=1000(flavio) groups=1000(flavio),27(sudo),100(users)
```

The `-a` is important. It **appends** the group. Without it, `usermod -G` replaces the user's whole list of extra groups.

Now the key. The quickest way to give `flavio` the same key root has is to copy root's `.ssh` directory with the right owner:

```
rsync --archive --chown=flavio:flavio ~/.ssh /home/flavio && ls -la /home/flavio/.ssh
```

```
total 12
drwx------ 2 flavio flavio 4096 Nov 28 16:13 .
drwxr-x--- 3 flavio flavio 4096 Nov 28 16:18 ..
-rw------- 1 flavio flavio  101 Nov 28 16:13 authorized_keys
```

Look at the permissions. The directory is `drwx------` and the file `-rw-------`: only `flavio` can read them. SSH refuses keys in files that other users can write, so if you create these by hand, keep those permissions.

Test before you depend on it

Keep the root session open. In a **second** terminal, log in as the new user:

```
ssh -i ~/.ssh/handbook_linux flavio@203.0.113.42
whoami; id
```

```
flavio
uid=1000(flavio) gid=1000(flavio) groups=1000(flavio),27(sudo),100(users)
```

And check `sudo` works:

```
sudo whoami
```

```
root
```

Only now, with a second working way in, is it safe to start closing doors. That's the next chapter.

If you lose every way in

Everything in this chapter assumes SSH keeps working. Sometimes it doesn't: you delete the wrong key, or a firewall rule blocks you.

The Droplet's **Recovery Console**, in the control panel, gives you a terminal on the server without going through SSH or the network. It logs in with a password, though, and our root user has none (cloud-init set up key access only). The control panel can reset the root password and show it to you once, so you can log in at the console, fix the problem, and get back to keys.

I'd rather find out how that works on a test server than at 2 a.m. on a real one. Destroying a Droplet like this one and creating it again also takes a minute, which is a good reason to keep the setup of a server written down, or in a script.

The first hour: locking it down

A server with a public IP gets found by scanners within minutes. That's not a sign you've been targeted.

It's background noise of the internet, and on this Droplet I saw scanners asking Nginx for `/v2/_catalog` and `/trace.axd` less than an hour after it booted.

So the first hour on a new server is for the boring basics: no root login over SSH, only keys, a firewall that drops everything we don't use, and a list of what's listening.

Harden SSH with a drop-in file

Ubuntu's SSH server reads `/etc/ssh/sshd_config` and then every `.conf` file in `/etc/ssh/sshd_config.d/`. I add my settings in a file of my own instead of editing the main one:

```
printf 'PermitRootLogin no\nPasswordAuthentication no\nKbdInteractiveAuthentication no\nMaxAuthTries 3\
    ↪ n' | sudo tee /etc/ssh/sshd_config.d/10-hardening.conf
```

```
PermitRootLogin no
PasswordAuthentication no
KbdInteractiveAuthentication no
MaxAuthTries 3
```

`PermitRootLogin no` blocks root over SSH, even with a key. The two `Authentication no` lines block every kind of password prompt (cloud-init already did one of them, but I want it in my file too, so it doesn't depend on cloud-init).

`MaxAuthTries 3` closes the connection after three failed attempts. None of these touch key logins, which is what we use.

Why `10-` in the name?

The files load in alphabetical order, and for most SSH options **the first value read wins**. Starting the name with a low number puts my file before `50-cloud-init.conf`, so my values aren't overridden by something that loads earlier.

Check, then reload

`sshd -t` checks the syntax. It prints nothing when the file is valid, so I add an `echo`:

```
sudo sshd -t && echo config ok
```

```
config ok
```

Only then reload:

```
sudo systemctl reload ssh
```

A reload re-reads the config and keeps existing connections open. That's why you keep your sessions open while you do this: if the new config locks you out, you can still fix it from a session that's already in.

Then check the values the server really uses:

```
sudo sshd -T | grep -E '^(permitrootlogin|passwordauthentication|maxauthtries) '
```

```
maxauthtries 3
permitrootlogin no
passwordauthentication no
```

Test the door that should be closed

A hardening step you didn't test from the outside might not be doing anything. So from my Mac, I tried to log in as root:

```
ssh root@203.0.113.42
```

```
Received disconnect from 203.0.113.42 port 22:2: Too many authentication failures
Disconnected from 203.0.113.42 port 22
```

That wasn't the error I expected, and the reason is something you'll want to know.

My Mac runs an SSH agent with four keys loaded.

When you connect, the client offers the agent's keys one after the other. Root is now refused whatever the key, so every attempt failed, and after three failures `MaxAuthTries 3` closed the connection.

That's the right result for root. But think about what it means for your own user: if the right key happens to be the fourth one your agent offers, you get the same error, and you're locked out of a perfectly working account.

The fix is on the client side. `IdentitiesOnly=yes` tells SSH to offer only the key you name:

```
ssh -o IdentitiesOnly=yes -i ~/.ssh/handbook_linux root@203.0.113.42


root@203.0.113.42: Permission denied (publickey).
```

That's the error I wanted to see: the right key, refused because it's root. And `flavio` still gets in.

To avoid typing the flags every time, I put them in `~/.ssh/config` on my Mac:

```
Host handbook
  HostName 203.0.113.42
  User flavio
  IdentityFile ~/.ssh/handbook_linux
  IdentitiesOnly yes
```

Now `ssh handbook` does the right thing. I wrote more about client configuration, agents and keys in SSH for developers.

If the new config ever locks you out of your own user, don't touch the session you kept open. Move your file out of `sshd_config.d`, run `sudo sshd -t`, reload, and try again.

If every session is gone, the Droplet's **Recovery Console** in the control panel is the way back in.

What about changing the port, or fail2ban?

Two tips come up in every SSH hardening guide: move SSH to a port other than 22, and install fail2ban to ban IPs that fail too many logins.

Moving the port cuts down the noise in the logs, because most scanners only try 22. It doesn't make the server safer against anyone who actually looks, since a port scan finds the new one in seconds, and it adds one more thing to remember (and to open in two firewalls). With passwords turned off, a bot that guesses passwords can't get in no matter how many times it tries. So I keep 22.

fail2ban is useful when some service still accepts passwords. With key-only SSH, it mostly saves log space. I didn't install it on this server. If you need SSH to be invisible to the internet, the cleaner answer is to close port 22 entirely and reach the server over a private network, which we'll see with Tailscale.

What's listening?

Before turning on a firewall, I want to know what's listening on the network. `ss` lists the sockets, and with `sudo` it also shows which process owns each one:

```
sudo ss -tulpn
```

```
Netid State  Recv-Q Send-Q Local Address:Port Peer Address:PortProcess
udp   UNCONN 0      0         127.0.0.54:53        0.0.0.0:*    users:(("systemd-resolve",pid=462,fd
    ↪ =16))
udp   UNCONN 0      0      127.0.0.53%lo:53        0.0.0.0:*    users:(("systemd-resolve",pid=462,fd
    ↪ =14))
tcp   LISTEN 0      4096   127.0.0.53%lo:53        0.0.0.0:*    users:(("systemd-resolve",pid=462,fd
    ↪ =15))
tcp   LISTEN 0      4096         0.0.0.0:22        0.0.0.0:*    users:(("sshd",pid=890,fd=3),("systemd
    ↪ ",pid=1,fd=91))
tcp   LISTEN 0      4096      127.0.0.54:53        0.0.0.0:*    users:(("systemd-resolve",pid=462,fd
    ↪ =17))
tcp   LISTEN 0      4096            [::]:22           [::]:*    users:(("sshd",pid=890,fd=4),("systemd
    ↪ ",pid=1,fd=92))
```

The flags mean TCP, UDP, listening sockets, with the process, and numeric ports.

The column to read is `Local Address`.

`0.0.0.0:22` and `[::]:22` mean SSH listens on every IPv4 and IPv6 address the server has, so it's reachable from the internet. Everything on `127.0.0.53` and `127.0.0.54` is the local DNS resolver, `systemd-resolved`, and it's bound to loopback: only processes on this machine can reach it.

That's the whole exposure of a fresh Ubuntu Droplet: port 22. Later we'll bind the notes API to `127.0.0.1:3000` for the same reason, so it's invisible from outside even without a firewall.

Running services

The services that run are a different list from the ones that listen:

```
systemctl list-units --type=service --state=running --no-pager --no-legend | awk '{print $1}'
```

```
cron.service
dbus.service
droplet-agent.service
getty@tty1.service
ModemManager.service
multipathd.service
polkit.service
rsyslog.service
serial-getty@ttyS0.service
ssh.service
systemd-journald.service
systemd-logind.service
systemd-networkd.service
systemd-resolved.service
systemd-timesyncd.service
```

```
systemd-udevd.service
udisks2.service
unattended-upgrades.service
user@1000.service
```

A couple of them make little sense on a cloud server. `ModemManager` manages mobile broadband modems, `udisks2` manages removable disks for desktop environments.

Neither listens on the network, as the `ss` output shows, so they don't add exposure. I left them alone on this server.

When a service does listen and the server doesn't need it, I stop it and keep it from starting at the next boot in one command, `sudo systemctl disable --now <name>`, and if nothing needs the package, `sudo apt purge` it. Then I run `ss` again and reboot once, to catch a socket or a timer that starts it again.

Turn on the firewall

Ubuntu comes with **UFW**, the Uncomplicated Firewall, a friendly front end for the kernel's packet filter. It's installed but off:

```
sudo ufw status
```

```
Status: inactive
```

The order of the next steps is what keeps you from locking yourself out: set the defaults, allow SSH, and only then enable.

Packages can register named rules, called **application profiles**. On a fresh server there's one:

```
sudo ufw app list
```

```
Available applications:
  OpenSSH
```

Deny everything incoming, allow everything outgoing:

```
sudo ufw default deny incoming && sudo ufw default allow outgoing
```

```
Default incoming policy changed to 'deny'
(be sure to update your rules accordingly)
Default outgoing policy changed to 'allow'
(be sure to update your rules accordingly)
```

Allow SSH:

```
sudo ufw allow OpenSSH
```

```
Rules updated
Rules updated (v6)
```

And enable it:

```
sudo ufw enable
```

```
Firewall is active and enabled on system startup
```

It asks for confirmation, because it may cut existing SSH connections. Answer `y`.

The verbose status shows the whole picture:

```
sudo ufw status verbose
```

```
Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip

To                         Action      From
--                         ------      ----
22/tcp (OpenSSH)           ALLOW IN    Anywhere
22/tcp (OpenSSH (v6))      ALLOW IN    Anywhere (v6)
```

Only port 22 gets in, on IPv4 and IPv6. When we install Nginx, we'll open 80 and 443.

A rule in the list proves nothing until a new connection works. So in another terminal I logged in again, from scratch:

```
ssh flavio@203.0.113.42
```

```
still in
```

(That's the output of the `echo` I ran on the server through the new connection.) If that second login hangs, don't close the first session: in it, `sudo ufw status numbered` lists the rules with numbers, and `sudo ufw delete <number>` removes a wrong one.

The firewall before the firewall

DigitalOcean also has **Cloud Firewalls**, configured in the control panel. They filter traffic before it reaches the Droplet, so a connection has to pass both: the Cloud Firewall first, then UFW on the server.

I like having both. The Cloud Firewall still protects the server if I break UFW from the inside, and UFW still protects it if someone changes the Cloud Firewall.

But I change one layer at a time. When a login fails right after changing two firewalls, you have two suspects instead of one.

One warning that becomes important later in the book: Docker writes its own firewall rules, and a container port published on all interfaces is reachable from the internet even when UFW has no rule for it. I tested it on this server, with the result you'd fear.

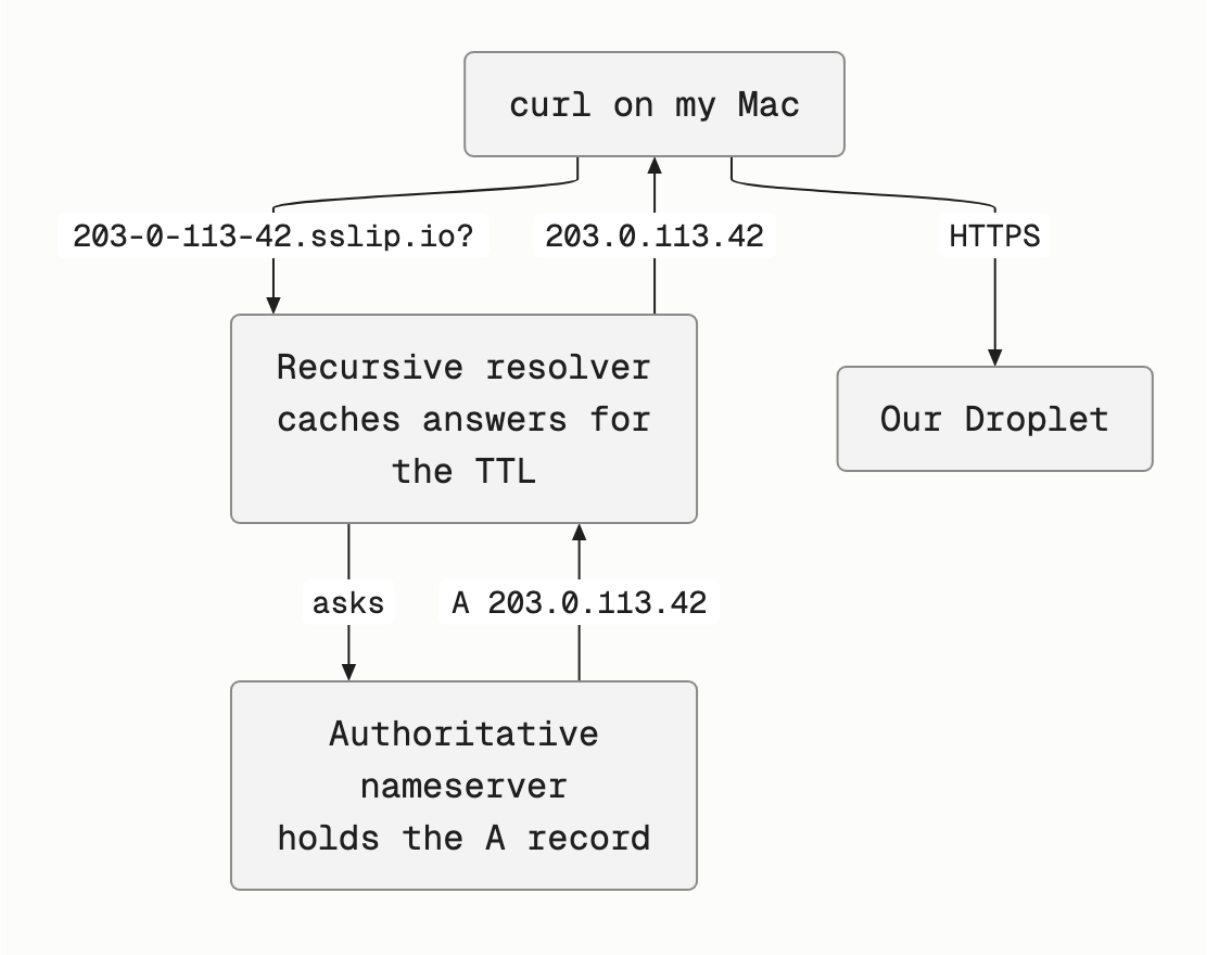
We'll see it in the home server chapter.

There's a different approach where you close port 22 to the internet completely and reach SSH over a private network like Tailscale. I set that up in How to secure a VPS with Tailscale and Cloudflare Tunnel, and we'll come back to Tailscale at the end of the book.

A domain for the server

People don't type IP addresses, and HTTPS certificates are issued for names, not for IPs. So before we put a website on the server, it needs a name.

DNS is the system that turns a name into an address. The record that does it for IPv4 is the **A record**: "this name means this IPv4 address".



With a real domain

With your own domain, you go to the DNS provider that hosts it (the registrar, Cloudflare, DigitalOcean's DNS) and create an A record. For example `notes` in the `flaviocopes.com` zone, pointing to the Droplet's IP, gives you `notes.flaviocopes.com`.

Two kinds of DNS servers are involved. The **authoritative nameservers** hold your zone and always answer with the current record.

Recursive resolvers (your ISP's, `1.1.1.1`, the one on the server) cache answers for the record's **TTL**, the time to live in seconds. When you change a record, the authoritative answer changes right away, and caches catch up when their copy expires.

So when a new record doesn't seem to work, ask the authority directly first. `dig NS` gives you the nameservers of the domain, and `dig @<nameserver> A <name>` asks one of them.

If the authority is right and a resolver is wrong, it's serving a cached answer: wait for the TTL to pass instead of editing the record again.

I'd hold off on the **AAAA record** (the IPv6 version) until the whole IPv6 path works: the Droplet has public IPv6 on, Nginx listens on IPv6, UFW allows it. A broken AAAA record makes a healthy site look flaky, because some clients try IPv6 first.

The name I used: sslip.io

For this book I didn't want to touch my real domains, so I used **sslip.io**, a free DNS service that answers with whatever IP you write in the name. `203-0-113-42.sslip.io` resolves to `203.0.113.42`.

Let's Encrypt issues real certificates for these names, so everything in the following chapters works exactly like with your own domain.

`dig` is the tool I use for DNS questions. With `+short` it prints only the answer:

```
dig +short 203-0-113-42.sslip.io
```

```
203.0.113.42
```

Without `+short`, and with the answer section only, you also see the TTL:

```
dig 203-0-113-42.sslip.io A +noall +answer +stats
```

```
203-0-113-42.sslip.io.  299 IN  A   203.0.113.42
;; Query time: 0 msec
;; SERVER: 127.0.0.53#53(127.0.0.53) (UDP)
;; WHEN: Sat Nov 28 16:24:05 UTC 2026
```

The TTL is 299 seconds, so a resolver keeps this answer for about five minutes. The `SERVER` line says who answered: `127.0.0.53`, the local resolver we saw listening in the previous chapter.

The query time was 0 ms because it was already in the cache.

The resolver on the server

On Ubuntu, programs on the server don't talk to DNS servers directly. They ask `systemd-resolved` on `127.0.0.53`, which forwards the question and caches the answer.

`resolvectl` shows where it forwards to:

```
resolvectl status
```

```
Global
         Protocols: -LLMNR -mDNS -DNSOverTLS DNSSEC=no/unsupported
  resolv.conf mode: stub
Current DNS Server: 10.114.15.254
       DNS Servers: 10.114.15.254
```

`10.114.15.254` is DigitalOcean's resolver on the private network. `resolvectl query` resolves a name and tells you how it got the answer:

```
resolvectl query 203-0-113-42.sslip.io
```

```
203-0-113-42.sslip.io: 203.0.113.42                       -- link: eth1

-- Information acquired via protocol DNS in 29.9ms.
-- Data is authenticated: no; Data was acquired via local or encrypted transport: no
-- Data from: cache network
```

One subtle thing: `dig` asks DNS and nothing else, while programs resolve names through the system's resolver library, which also reads `/etc/hosts`. When `dig` and an application disagree, `getent` shows what the application sees:

```
getent hosts 203-0-113-42.sslip.io
```

```
203.0.113.42   203-0-113-42.sslip.io
```

Check from outside

What matters is what the rest of the internet sees, so I ran the same lookup from my Mac:

```
dig +short 203-0-113-42.sslip.io
```

```
203.0.113.42
```

Same answer. DNS only proves the name points to the right place, though. Whether something answers on that name is the next step: in the Nginx chapter we'll open ports 80 and 443 and request `http://203-0-113-42.sslip.io` from outside, which tests DNS, the firewall and the web server in one go.

Services with systemd

A **service** is a program that runs in the background, starts when the server boots, and keeps running without anyone logged in.

On Ubuntu, the program in charge of services is systemd. It's the first process the kernel starts (PID 1), and it starts, stops, restarts and watches everything else.

In this chapter we take a small Node.js app, run it by hand once, and then hand it over to systemd. Along the way I broke it a few times, and I kept those failures in the chapter, because they're the ones you'll hit too.

The notes API

The app we're going to run for the rest of the book is a tiny notes API. It stores notes in a JSON file and has three endpoints: `GET /health`, `GET /notes` and `POST /notes`.

It has no dependencies, so there's no `npm install` to worry about. Here's `server.js`:

```
import { createServer } from 'node:http'
import { readFile, writeFile, rename } from 'node:fs/promises'
import path from 'node:path'

const host = process.env.HOST || '127.0.0.1'
const port = Number(process.env.PORT || 3000)
const dataDir = process.env.DATA_DIR || './data'
const file = path.join(dataDir, 'notes.json')

async function loadNotes() {
  try {
    return JSON.parse(await readFile(file, 'utf8'))
  } catch (error) {
    if (error.code === 'ENOENT') return []
    throw error
  }
}

async function saveNotes(notes) {
  await writeFile(`${file}.tmp`, JSON.stringify(notes, null, 2))
  await rename(`${file}.tmp`, file)
}

const server = createServer(async (req, res) => {
  const send = (status, body) => {
    res.writeHead(status, { 'content-type': 'application/json' })
    res.end(JSON.stringify(body))
  }

  if (req.url === '/health') return send(200, { ok: true })
```

```
  if (req.url === '/notes' && req.method === 'GET') {
    return send(200, await loadNotes())
  }

  if (req.url === '/notes' && req.method === 'POST') {
    let body = ''
    for await (const chunk of req) body += chunk
    const { text } = JSON.parse(body || '{}')
    if (!text) return send(400, { error: 'text is required' })
    const notes = await loadNotes()
    const note = { id: notes.length + 1, text, createdAt: new Date().toISOString() }
    notes.push(note)
    await saveNotes(notes)
    console.log(`saved note ${note.id}`)
    return send(201, note)
  }

  send(404, { error: 'not found' })
})

server.listen(port, host, () => {
  console.log(`notes-api listening on http://${host}:${port}`)
})

process.on('SIGTERM', () => {
  console.log('SIGTERM received, closing')
  server.close(() => process.exit(0))
})
```

A few details matter later in the book.

The app reads its host, port and data folder from environment variables, and it listens on `127.0.0.1` by default, so only the server itself can reach it. Nginx will sit in front of it in the next part.

It writes to a temporary file and then renames it, so a crash in the middle of a write never leaves half a JSON file behind.

And it handles `SIGTERM`, the signal systemd sends when it stops a service, by closing the server and exiting cleanly.

This is `package.json`:

```
{
  "name": "notes-api",
  "version": "1.0.0",
  "type": "module",
  "engines": { "node": ">=22" },
  "scripts": { "start": "node server.js" }
}
```

Installing Node from NodeSource

Ubuntu 24.04 ships an older Node in its own repositories. I want a current LTS, so I install it from NodeSource, which publishes Node packages for Debian and Ubuntu.

Their instructions pipe a script straight into a root shell. On a server I prefer to download the script and read it first:

```
curl -fsSL https://deb.nodesource.com/setup_24.x -o nodesource_setup.sh
head -3 nodesource_setup.sh
```

```
#!/bin/bash

# Logger Function
```

Then I read it. It's 121 lines, and the interesting part is short. It downloads NodeSource's signing key into `/usr/share/keyrings` and adds an APT source signed by that key:

```
grep -nE "gpg|keyring" nodesource_setup.sh
```

```
55:    if ! mkdir -p /usr/share/keyrings; then
56:      handle_error "$?" "Makes sure the path /usr/share/keyrings exist or run ' mkdir -p /usr/share/
    ↪ keyrings' with sudo"
59:    rm -f /usr/share/keyrings/nodesource.gpg || true
63:    # Run 'curl' and 'gpg' to download and import the NodeSource signing key
64:    if ! curl -fsSL https://deb.nodesource.com/gpgkey/nodesource-repo.gpg.key | gpg --dearmor -o /
    ↪ usr/share/keyrings/nodesource.gpg; then
...
89:Signed-By: /usr/share/keyrings/nodesource.gpg
```

Nothing else, so I run it:

```
sudo bash nodesource_setup.sh
```

The result is one new file in `/etc/apt/sources.list.d`. That's the whole change, and it's easy to remove later:

```
cat /etc/apt/sources.list.d/nodesource.sources
```

```
Types: deb
URIs: https://deb.nodesource.com/node_24.x
Suites: nodistro
Components: main
Architectures: amd64
Signed-By: /usr/share/keyrings/nodesource.gpg
```

Now Node installs like any other package, and `apt upgrade` keeps it updated:

```
sudo apt-get install -y nodejs
node --version; npm --version
```

```
v24.22.3
11.19.1
```

Copying the app and giving it a user

From my Mac, I copy the two files to my home folder on the server with rsync:

```
rsync -av notes-api/ flavio@203.0.113.42:notes-api/
```

```
Transfer starting: 3 files
created directory notes-api
./
package.json
server.js

sent 2097 bytes  received 70 bytes  28475 bytes/sec
total size is 1857  speedup is 0.86
```

The app should not run as `flavio`, and certainly not as root. It gets its own **system user**, with no password, no login shell, and a home folder pointing at its data:

```
sudo useradd --system --home-dir /var/lib/notes-api --shell /usr/sbin/nologin notes
getent passwd notes
```

```
notes:x:999:988::/var/lib/notes-api:/usr/sbin/nologin
```

System users get IDs below 1000. Remember `999`, it shows up again in the logs.

The code goes in `/opt/notes-api`, owned by root. The `notes` user can read it but can't change it, so a bug in the app can't rewrite the app:

```
sudo mkdir -p /opt/notes-api && sudo cp notes-api/* /opt/notes-api/
ls -l /opt/notes-api
```

```
total 8
-rw-r--r-- 1 root root  145 Nov 28 16:21 package.json
-rw-r--r-- 1 root root 1712 Nov 28 16:21 server.js
```

The data goes in `/var/lib/notes-api`, the standard place for a service's state. This folder is owned by `notes`, and nobody else can read it:

```
sudo install -d -o notes -g notes -m 750 /var/lib/notes-api
ls -ld /var/lib/notes-api
```

```
drwxr-x--- 2 notes notes 4096 Nov 28 16:21 /var/lib/notes-api
```

A first run by hand

Before systemd gets involved, I start the app once as the `notes` user, in the background, and call it:

```
cd /opt/notes-api
sudo -u notes env DATA_DIR=/var/lib/notes-api node server.js > /tmp/manual.log 2>&1 &
curl -s http://127.0.0.1:3000/health


{"ok":true}
```

Then I check where it's listening. This line is the one I care about most:

```
sudo ss -lntp | grep 3000


LISTEN 0      511        127.0.0.1:3000      0.0.0.0:*    users:(("MainThread",pid=4212,fd=21))
```

`127.0.0.1:3000` means only the server itself can connect. If it said `0.0.0.0:3000`, the app would answer the whole internet the day the firewall slipped.

Notice the process name, `MainThread`. I didn't pay attention to it at the time, and it cost me a few minutes.

To stop the manual run, I typed what I always type:

```
sudo pkill -u notes node
```

It printed nothing, and I moved on. That was a mistake, as we'll see in a moment.

A minimal unit file

systemd reads services from **unit files**. Packages install theirs in `/usr/lib/systemd/system`. Ours goes in `/etc/systemd/system`, which is where local units and local changes live, and which wins over the package copies.

This is the smallest unit that does the job, `/etc/systemd/system/notes-api.service`:

```
[Unit]
Description=Notes API
After=network.target

[Service]
User=notes
Group=notes
WorkingDirectory=/opt/notes-api
Environment=HOST=127.0.0.1 PORT=3000 DATA_DIR=/var/lib/notes-api
ExecStart=/usr/bin/node server.js
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

`[Unit]` describes the unit and its ordering. `[Service]` says how to run the process: as which user, from which folder, with which environment, and what to do when it

dies. `[Install]` says what `enable` hooks it into. `multi-user.target` is the normal "server is up" state, so enabling the unit means "start this at boot".

Every time we add or change a unit file, systemd has to reread its configuration:

```
sudo systemctl daemon-reload
```

Then `enable --now` enables it for the next boot and starts it right away:

```
sudo systemctl enable --now notes-api
```

```
Created symlink /etc/systemd/system/multi-user.target.wants/notes-api.service -> /etc/systemd/system/
    ↪ notes-api.service.
```

Enabling a unit is just that symlink. Let's look at the result:

```
systemctl status notes-api --no-pager
```

```
 notes-api.service - Notes API
     Loaded: loaded (/etc/systemd/system/notes-api.service; enabled; preset: enabled)
     Active: activating (auto-restart) (Result: exit-code) since Sat 2026-11-28 16:21:34 UTC; 56ms ago
    Process: 4518 ExecStart=/usr/bin/node server.js (code=exited, status=1/FAILURE)
   Main PID: 4518 (code=exited, status=1/FAILURE)
        CPU: 191ms
```

It failed with exit code 1, and systemd is about to try again (`auto-restart`).

Why the first start failed

The strange part was that the API answered anyway:

```
curl -s -X POST http://127.0.0.1:3000/notes -d '{"text":"buy milk"}'
curl -s http://127.0.0.1:3000/notes
```

```
{"id":1,"text":"buy milk","createdAt":"2026-11-28T16:21:35.718Z"}
[{"id":1,"text":"buy milk","createdAt":"2026-11-28T16:21:35.718Z"}]
```

So something was serving port 3000, and it wasn't the service. A second check confirmed it:

```
systemctl is-enabled notes-api; systemctl is-active notes-api
```

```
enabled
failed
```

The natural next step is the service's log. As `flavio`, I got nothing:

```
journalctl -u notes-api --no-pager
```

```
-- No entries --
Hint: You are currently not seeing messages from other users and the system.
      Users in groups 'adm', 'systemd-journal' can see all messages.
      Pass -q to turn off this notice.
```

The hint is right. A normal user only sees their own messages in the journal. With `sudo` the whole story appears:

```
sudo journalctl -u notes-api --no-pager
```

```
...
Nov 28 16:21:35 handbook-linux-test node[4632]:   code: 'EADDRINUSE',
Nov 28 16:21:35 handbook-linux-test node[4632]:   errno: -98,
Nov 28 16:21:35 handbook-linux-test node[4632]:   syscall: 'listen',
Nov 28 16:21:35 handbook-linux-test node[4632]:   address: '127.0.0.1',
Nov 28 16:21:35 handbook-linux-test node[4632]:   port: 3000
Nov 28 16:21:35 handbook-linux-test node[4632]: }
Nov 28 16:21:35 handbook-linux-test node[4632]: Node.js v24.22.3
Nov 28 16:21:35 handbook-linux-test systemd[1]: notes-api.service: Main process exited, code=exited,
    ↪ status=1/FAILURE
Nov 28 16:21:35 handbook-linux-test systemd[1]: notes-api.service: Failed with result 'exit-code'.
Nov 28 16:21:36 handbook-linux-test systemd[1]: notes-api.service: Scheduled restart job, restart
    ↪ counter is at 5.
Nov 28 16:21:36 handbook-linux-test systemd[1]: notes-api.service: Start request repeated too quickly.
Nov 28 16:21:36 handbook-linux-test systemd[1]: notes-api.service: Failed with result 'exit-code'.
Nov 28 16:21:36 handbook-linux-test systemd[1]: Failed to start notes-api.service - Notes API.
```

`EADDRINUSE` means the address is already in use. systemd restarted the service five times in a row, and then gave up with ”Start request repeated too quickly”. That's the start rate limit: by default, five failed starts in ten seconds and systemd stops trying.

So who has port 3000? `ss` tells us:

```
sudo ss -lntp | grep 3000
```

```
LISTEN 0      511        127.0.0.1:3000      0.0.0.0:*    users:(("MainThread",pid=4212,fd=21))
```

PID 4212 is my manual run, still alive. My `pkill -u notes node` didn't match it, and `ps -C node` doesn't find it either. Here's why:

```
ps -p 4212 -o user,pid,comm,args
cat /proc/4212/comm
```

```
USER          PID COMMAND         COMMAND
notes        4212 MainThread      node server.js
MainThread
```

Node 24 names its main thread `MainThread`, and that's the process name `pkill` and `ps -C` compare against. The full command line still says `node server.js`, so `pgrep -f 'node server.js'` would have found it. Or the PID from `ss`, which is what I used:

```
sudo kill 4212
sudo ss -lntp | grep 3000 || echo 'port 3000 free'
```

```
port 3000 free
```

A unit that hit the start limit stays in the `failed` state, and systemd won't start it again until we clear it:

```
sudo systemctl reset-failed notes-api
sudo systemctl start notes-api
systemctl status notes-api --no-pager
```

```
 notes-api.service - Notes API
     Loaded: loaded (/etc/systemd/system/notes-api.service; enabled; preset: enabled)
     Active: active (running) since Sat 2026-11-28 16:22:03 UTC; 30ms ago
   Main PID: 5225 (node)
      Tasks: 1 (limit: 1107)
```

And the note I saved through the stray process is still there, because both processes used the same data folder:

```
curl -s http://127.0.0.1:3000/notes
```

```
[{"id":1,"text":"buy milk","createdAt":"2026-11-28T16:21:35.718Z"}]
```

Two lessons from this. Stop manual test runs by PID, and check that the port is free before blaming the unit. And when a service fails, `sudo journalctl -u` is the first command, not the last.

To stop typing `sudo` for logs, I added `flavio` to the `adm` group, which can read the whole journal and most files in `/var/log`:

```
sudo usermod -aG adm flavio && id flavio
```

```
uid=1000(flavio) gid=1000(flavio) groups=1000(flavio),4(adm),27(sudo),100(users)
```

The new group applies from the next login.

Restart policy

`Restart=on-failure` tells systemd to bring the service back when it crashes. Let's crash it on purpose with `kill -9`, which the process can't catch:

```
systemctl show notes-api -p MainPID -p NRestarts
```

```
MainPID=5225
NRestarts=0
```

```
sudo kill -9 $(systemctl show notes-api -p MainPID --value)
systemctl show notes-api -p MainPID -p NRestarts
```

```
MainPID=5551
NRestarts=1
```

New PID, one restart. The journal shows what happened:

```
journalctl -u notes-api --no-pager -n 5
```

```
Nov 28 16:22:24 handbook-linux-test systemd[1]: notes-api.service: Main process exited, code=killed,
    ↪ status=9/KILL
Nov 28 16:22:24 handbook-linux-test systemd[1]: notes-api.service: Failed with result 'signal'.
Nov 28 16:22:25 handbook-linux-test systemd[1]: notes-api.service: Scheduled restart job, restart
    ↪ counter is at 1.
Nov 28 16:22:25 handbook-linux-test systemd[1]: Started notes-api.service - Notes API.
Nov 28 16:22:25 handbook-linux-test node[5551]: notes-api listening on http://127.0.0.1:3000
```

Later, while testing signals, I sent a plain `SIGTERM` instead:

```
sudo kill -TERM $(systemctl show notes-api -p MainPID --value)
journalctl -u notes-api --no-pager -o cat -n 4
```

```
Started notes-api.service - Notes API.
notes-api listening on http://127.0.0.1:3000
SIGTERM received, closing
notes-api.service: Deactivated successfully.
```

The app did exactly what I wrote: closed the server and exited with code 0. And the service stayed down:

```
systemctl is-active notes-api; systemctl show notes-api -p NRestarts -p Result
```

```
inactive
Result=success
NRestarts=0
```

For systemd that's not a failure. A process that exits with 0 finished its job, so `on-failure` has nothing to restart.

Is that what we want? It depends on who sends the signal.

When I run `systemctl stop`, I want it to stay stopped, and systemd never restarts a unit you stopped yourself, whatever the policy. But a `SIGTERM` from somewhere else, like the stray `kill` I just sent, leaves the API down until someone notices.

`Restart=always` restarts after clean exits too, which is the safer choice for a long-running API that should never exit on its own.

I kept `on-failure` in the notes API to show the difference, and the health check in the monitoring chapter is what would catch it.

This is what happened to the notes API in each case I tried, and what `Restart=always` would change:

How the process ended	`Restart=on-failure`	`Restart=always`
`systemctl stop`	stays stopped	stays stopped
Crash or `kill -9`	restarted	restarted
OOM kill (chapter 16)	restarted	restarted
Clean exit with code 0, like after `kill -TERM`	stays stopped	restarted

Each restart adds 1 to `NRestarts`. Too many starts in a short interval and systemd gives up with `start-limit-hit`, the "Start request repeated too quickly" we saw earlier, and the unit needs `systemctl reset-failed`.

Two more settings go with the policy. `RestartSec=` is the pause before restarting, 100 ms by default, which turns a broken service into a tight loop. I use 2 seconds. And the start limit we hit earlier is set with `StartLimitIntervalSec=` and `StartLimitBurst=` in the `[Unit]` section.

My advice is to watch `NRestarts`. A service that restarts fifty times a day is broken, even if `is-active` always says `active`.

Configuration and secrets

A service doesn't see your shell. The `export` lines in your `.bashrc` don't exist when systemd starts the process, so every value has to come from the unit.

`Environment=` is fine for ordinary values like the port. For anything secret I use an `EnvironmentFile=`, a separate file of `KEY=VALUE` lines that the unit loads. The notes API doesn't use a token yet, but I set one up to show where it goes.

The folder is readable only by root and the `notes` group:

```
sudo install -d -m 750 -o root -g notes /etc/notes-api
printf 'API_TOKEN=%s\n' $(openssl rand -hex 16) | sudo tee /etc/notes-api/env >/dev/null
sudo chmod 640 /etc/notes-api/env
sudo chown root:notes /etc/notes-api/env
ls -l /etc/notes-api/env
```

```
ls: cannot access '/etc/notes-api/env': Permission denied
```

That error is the permissions working. `flavio` isn't in the `notes` group, so it can't even list the folder. Only root and the service can read the token.

Values from `Environment=` are visible to anyone who asks systemd. Values from the file aren't:

```
systemctl show notes-api -p Environment
```

```
Environment=HOST=127.0.0.1 PORT=3000 DATA_DIR=/var/lib/notes-api
```

The token still ends up in the process environment, though, where child processes inherit it and debugging tools can dump it. For secrets that matter, systemd has `LoadCredential=`, which hands the service a private file instead of a variable. I go through the options in managing secrets and API keys on your own infrastructure.

Reducing privileges

Running as `notes` already limits the damage a bug can do. systemd can go much further, and it can measure how exposed a service is. This is the first unit:

```
systemd-analyze security notes-api --no-pager
```

```
...
 UMask=                                                  Files created by service are world-
    ↪ readable by default                    0.1

-> Overall exposure level for notes-api.service: 9.2 UNSAFE :-{
```

9.2 out of 10. The process can see the whole filesystem, every home folder, the kernel tunables, and it could gain privileges through a setuid binary.

Here's the version I run now. It adds the environment file, the restart pause, a stop timeout, and a block of sandboxing options:

```
[Unit]
Description=Notes API
After=network.target

[Service]
User=notes
Group=notes
WorkingDirectory=/opt/notes-api
Environment=HOST=127.0.0.1 PORT=3000 DATA_DIR=/var/lib/notes-api
EnvironmentFile=/etc/notes-api/env
ExecStart=/usr/bin/node server.js
Restart=on-failure
RestartSec=2
TimeoutStopSec=10

StateDirectory=notes-api
StateDirectoryMode=0750
UMask=0027
NoNewPrivileges=true
ProtectSystem=strict
ProtectHome=true
PrivateTmp=true
PrivateDevices=true
ProtectKernelTunables=true
ProtectKernelModules=true
ProtectControlGroups=true
```

```
RestrictAddressFamilies=AF_INET AF_INET6 AF_UNIX
RestrictNamespaces=true
LockPersonality=true
CapabilityBoundingSet=
SystemCallArchitectures=native

MemoryMax=256M
TasksMax=64

[Install]
WantedBy=multi-user.target
```

The ones I'd add to every service:

- `ProtectSystem=strict` makes the whole filesystem read-only for the service.
- `StateDirectory=notes-api` creates `/var/lib/notes-api`, gives it to the service user, and makes it the one writable exception.
- `ProtectHome=true` hides `/home` and `/root` completely.
- `PrivateTmp=true` gives the service its own `/tmp`.
- `NoNewPrivileges=true` and an empty `CapabilityBoundingSet=` mean the process can never become more powerful than it starts.
- `TimeoutStopSec=10` is how long systemd waits after `SIGTERM` before it sends `SIGKILL`.

Deploying the change is the same three steps every time. Copy the unit, reload systemd, restart the service, and check:

```
sudo cp notes-api.service /etc/systemd/system/notes-api.service
sudo systemctl daemon-reload
sudo systemctl restart notes-api
systemctl is-active notes-api
curl -s http://127.0.0.1:3000/notes
```

```
active
[{"id":1,"text":"buy milk","createdAt":"2026-11-28T16:21:35.718Z"}]
```

The score after the change:

```
systemd-analyze security notes-api --no-pager
```

```
...
-> Overall exposure level for notes-api.service: 3.6 OK :-)
```

From 9.2 UNSAFE to 3.6 OK, and the app didn't notice.

You can try the same sandbox on a one-off command with `systemd-run`, which is a quick way to see what an option does before putting it in a unit. Here a process running as `notes` with `ProtectHome=true` tries to create a file in my home folder:

```
sudo systemd-run --pty --quiet -p User=notes -p ProtectSystem=strict -p ProtectHome=true touch /home/
    ↪ flavio/x
```

```
/usr/bin/touch: cannot touch '/home/flavio/x': Permission denied
```

And the plain user permissions still apply on top. The `notes` user can't read the password hashes:

```
sudo -u notes cat /etc/shadow
```

```
cat: /etc/shadow: Permission denied
```

My advice: add the options a few at a time, restart, and test the app after each batch. If something breaks, the journal usually names the path or the system call that was blocked.

Limiting memory and tasks

On a 1 GB server, one runaway process can take everything with it. `MemoryMax=` and `TasksMax=` put a ceiling on the service:

```
systemctl show notes-api -p MemoryMax -p TasksMax -p MemoryCurrent
```

```
MemoryCurrent=17227776
MemoryMax=268435456
TasksMax=64
```

The API uses about 17 MB, against a limit of 256 MB. If it ever goes over, the kernel kills the service's processes instead of something else on the server, and the restart policy brings it back. In the troubleshooting chapter I lower the limit on purpose to see exactly what that looks like in the logs.

`TasksMax=` caps processes and threads, which stops a fork bomb or a thread leak inside one service. `CPUQuota=` does the same for CPU time when you need it.

Startup order and readiness

`After=network.target` only orders the start: our service starts after the network setup has been started. It doesn't wait for the network to work, and it doesn't make our service depend on it.

There's a second, subtler point. With the default `Type=simple`, systemd considers the service started as soon as it has launched the process. Not when the app is ready.

I hit this when I moved the data to a new disk. I started the service and called it in the same command:

```
sudo systemctl start notes-api && curl -s http://127.0.0.1:3000/notes
```

curl exited with code 7, ”couldn't connect”. A second later everything was fine:

```
systemctl is-active notes-api; journalctl -u notes-api --no-pager -n 4 -o cat
curl -s http://127.0.0.1:3000/notes
```

```
active
notes-api.service: Deactivated successfully.
Stopped notes-api.service - Notes API.
Started notes-api.service - Notes API.
notes-api listening on http://127.0.0.1:3000
[{"id":1,"text":"buy milk","createdAt":"2026-11-28T16:21:35.718Z"}]
```

`systemctl start` returned while Node was still loading. For a deploy script, that means checking the health endpoint in a loop for a few seconds before calling the deploy good. Services that tell systemd when they're ready use `Type=notify`, but Node doesn't do that out of the box, so the health check is the simplest fix.

Timers instead of cron

Most servers need something to run on a schedule: a backup, a cleanup, a report. The classic tool is cron, and it still works. On a systemd server I use **timers** instead.

A timer is a unit that starts another unit at a given time. The job itself is a normal service, so it gets everything from the previous chapter: its own user, sandboxing, resource limits, and logs in the journal under its own name.

This server doesn't have a single cron job of its own:

```
crontab -l
ls /etc/cron.d
```

```
no crontab for flavio
e2scrub_all
sysstat
```

Ubuntu itself moved most of its scheduled work to timers, as we'll see in a minute. If you want the cron side of things, I wrote about it in how to use cron.

A service and a timer

Our first scheduled job copies the notes file to `/var/backups/notes-api` once a day. It's not a real backup yet (that's the restic chapter), but it's a good first timer.

The job is a service of `Type=oneshot`, which means it runs a command to completion and then it's done. `/etc/systemd/system/notes-snapshot.service`:

```
[Unit]
Description=Copy the notes data to /var/backups/notes-api

[Service]
Type=oneshot
ExecStart=/usr/bin/bash -c 'install -d -m 700 /var/backups/notes-api && cp /var/lib/notes-api/notes.
    ↪ json /var/backups/notes-api/notes-$(date +%%F-%%H%%M).json'
```

Notice the double `%%` in the date format. systemd uses `%` for its own placeholders, so a literal percent sign has to be written twice.

The timer has the same name with a `.timer` extension, and that's how it knows which service to start. `/etc/systemd/system/notes-snapshot.timer`:

```
[Unit]
Description=Copy the notes data every day

[Timer]
OnCalendar=*-*-* 03:15:00
RandomizedDelaySec=15m
Persistent=true

[Install]
WantedBy=timers.target
```

`OnCalendar=` is the schedule, in systemd's calendar syntax: any year, any month, any day, at 03:15. When I'm not sure an expression means what I think, `systemd-analyze` tells me the next times it fires:

```
systemd-analyze calendar '*-*-* 03:15:00' --iterations=2
```

```
Normalized form: *-*-* 03:15:00
    Next elapse: Sun 2026-11-29 03:15:00 UTC
       From now: 10h left
   Iteration #2: Mon 2026-11-30 03:15:00 UTC
       From now: 1 day 10h left
```

The server runs on UTC, so 03:15 is 03:15 UTC.

Enabling the timer

We enable and start the timer, and the timer starts the service when it's time:

```
sudo cp notes-snapshot.service notes-snapshot.timer /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable --now notes-snapshot.timer
```

```
Created symlink /etc/systemd/system/timers.target.wants/notes-snapshot.timer -> /etc/systemd/system/
    ↪ notes-snapshot.timer.
```

```
systemctl list-timers notes-snapshot.timer --no-pager
```

```
NEXT                         LEFT LAST PASSED UNIT                ACTIVATES
Sun 2026-11-29 03:25:42 UTC  11h -         - notes-snapshot.timer notes-snapshot.service

1 timers listed.
Pass --all to see loaded but inactive timers, too.
```

The next run is 03:25:42, not 03:15. That's `RandomizedDelaySec=15m` at work: systemd picks a random delay up to 15 minutes. On one server it doesn't matter much. On fifty servers backing up to the same storage, it stops them all from hitting it at the same second.

I never wait for the first scheduled run to find out whether a job works. I start the service by hand:

```
sudo systemctl start notes-snapshot.service
sudo ls -l /var/backups/notes-api/
```

```
total 4
-rw-r--r-- 1 root root 92 Nov 28 16:22 notes-2026-11-28-1622.json
```

The file is readable by everyone, but it sits in a folder with mode 700 that only root can enter, so that's fine. The journal has the run:

```
journalctl -u notes-snapshot --no-pager -n 3
```

```
Nov 28 16:22:56 handbook-linux-test systemd[1]: Starting notes-snapshot.service - Copy the notes data
    ↪ to /var/backups/notes-api...
Nov 28 16:22:56 handbook-linux-test systemd[1]: notes-snapshot.service: Deactivated successfully.
Nov 28 16:22:56 handbook-linux-test systemd[1]: Finished notes-snapshot.service - Copy the notes data
    ↪ to /var/backups/notes-api.
```

If the copy had failed, the service would be in the `failed` state and the error would be right there. With cron, the output of a failing job goes to a local mail nobody reads.

Missed runs

What happens if the server is off at 03:15? With cron, the job doesn't run that day.

`Persistent=true` changes that. systemd stores when the timer last fired, and if a scheduled time passed while the server was down, it runs the job once as soon as the server is back. For backups that's what I want.

The timers Ubuntu already runs

`--all` shows every timer on the system, and it's a good tour of what Ubuntu does on its own:

```
systemctl list-timers --all --no-pager
```

```
NEXT                         LEFT LAST                              PASSED UNIT
    ↪       ACTIVATES
Sat 2026-11-28 16:30:00 UTC     7min Sat 2026-11-28 16:20:15 UTC 2min 42s ago sysstat-collect.timer
    ↪       sysstat-collect.service
Sat 2026-11-28 16:32:28 UTC     9min -                                     - systemd-tmpfiles-clean.
    ↪ timer   systemd-tmpfiles-clean.service
```

```
Sat 2026-11-28 17:13:18 UTC    50min Sat 2026-11-28 16:14:51 UTC           - droplet-agent-update.
    ↪ timer     droplet-agent-update.service
Sat 2026-11-28 17:25:44 UTC  1h 2min Sat 2026-11-28 16:13:33 UTC           - fwupd-refresh.timer
    ↪       fwupd-refresh.service
Sun 2026-11-29 00:00:00 UTC       7h Sat 2026-11-28 16:13:33 UTC           - dpkg-db-backup.timer
    ↪       dpkg-db-backup.service
Sun 2026-11-29 00:00:00 UTC       7h Sat 2026-11-28 16:13:33 UTC           - logrotate.timer
    ↪       logrotate.service
Sun 2026-11-29 00:02:32 UTC       7h Sat 2026-11-28 16:13:33 UTC           - motd-news.timer
    ↪       motd-news.service
...
```

Log rotation, the package database backup, temporary file cleanup: all timers. `sysstat-collect` records CPU and memory every ten minutes, and we'll read that history in the monitoring chapter.

One-off jobs with systemd-run

Sometimes I want to run something once, a little later, without writing two unit files. `systemd-run` creates a **transient** timer and service on the fly:

```
sudo systemd-run --on-active=20s --unit=notes-check curl -s http://127.0.0.1:3000/health
```

```
Running timer as unit: notes-check.timer
Will run service as unit: notes-check.service
```

Twenty seconds later the check ran, and its output is in the journal like any other service:

```
journalctl -u notes-check.service --no-pager
```

```
Nov 28 16:23:24 handbook-linux-test systemd[1]: Started notes-check.service - /usr/bin/curl -s http
    ↪ ://127.0.0.1:3000/health.
Nov 28 16:23:24 handbook-linux-test curl[6612]: {"ok":true}
Nov 28 16:23:24 handbook-linux-test systemd[1]: notes-check.service: Deactivated successfully.
```

Then the unit is gone. Transient units only exist until they finish:

```
systemctl status notes-check.service --no-pager
```

```
Unit notes-check.service could not be found.
```

I use this for things like "restart this service in ten minutes, after the migration" or for running a long command so it survives my SSH session closing.

Reading the logs

When something goes wrong on a server, the answer is almost always in a log. The hard part is finding the right lines among thousands of others.

On Ubuntu there are two places to look: the systemd **journal**, and the text files in `/var/log`.

The journal

The journal collects messages from the kernel, from systemd, and from everything services print on standard output and standard error. That's why the notes API needs no logging setup at all: its `console.log` lines go straight into the journal under `notes-api.service`.

On Ubuntu 24.04 the journal is stored on disk, so it survives reboots. You can check how much space it takes:

```
journalctl --disk-usage
```

```
Archived and active journals take up 18.1M in the file system.
```

journald caps its own size, by default at a percentage of the disk. If you want a hard limit, `SystemMaxUse=` in `/etc/systemd/journald.conf` sets it.

As we saw in the systemd chapter, a normal user sees only their own messages. Add your admin user to the `adm` group (or use `sudo`) before you need the logs in a hurry.

Filtering by unit and time

Running `journalctl` with no options prints everything since the journal started, which is useless. I always start from the service and the time of the problem.

`-u` picks one unit. `--since` takes a timestamp or a phrase like `10 min ago`. `-o short-iso` prints full timestamps, which I prefer when I compare logs from different machines:

```
journalctl -u notes-api --since '10 min ago' --no-pager -o short-iso
```

```
...
2026-11-28T16:22:31+00:00 handbook-linux-test systemd[1]: notes-api.service: Deactivated successfully.
2026-11-28T16:22:31+00:00 handbook-linux-test systemd[1]: Stopped notes-api.service - Notes API.
2026-11-28T16:22:31+00:00 handbook-linux-test systemd[1]: Started notes-api.service - Notes API.
2026-11-28T16:22:31+00:00 handbook-linux-test node[5762]: notes-api listening on http://127.0.0.1:3000
```

Notice that the unit's log mixes two voices: what systemd says about the service (`systemd[1]`), and what the app prints (`node[5762]`). Both matter. The app tells you what went wrong, systemd tells you what it did about it.

A few more filters I use all the time:

- `-n 20` shows the last 20 lines.
- `-f` follows new lines as they arrive, like `tail -f`. I keep it open in one terminal while I reproduce a problem in another.
- `-o cat` prints only the message, without timestamps, which is handy for grepping.
- `-t sudo` filters by the program name instead of the unit.

The last one is a quick audit of who ran what as root:

```
journalctl -t sudo --no-pager -n 2
```

```
Nov 28 16:22:58 handbook-linux-test sudo[6561]: pam_unix(sudo:session): session opened for user root(
    ↪ uid=0) by flavio(uid=1000)
Nov 28 16:22:58 handbook-linux-test sudo[6561]: pam_unix(sudo:session): session closed for user root
```

Filtering by priority

Every journal entry has a priority, from `emerg` (0) to `debug` (7).

`-p err` shows errors and worse, and `-b` limits it to the current boot. It's my first scan of a server I don't know:

```
journalctl -p err -b --no-pager
```

```
...
Nov 28 16:19:12 handbook-linux-test sshd[2144]: error: maximum authentication attempts exceeded for
    ↪ root from 203.0.113.7 port 39067 ssh2 [preauth]
Nov 28 16:21:36 handbook-linux-test systemd[1]: Failed to start notes-api.service - Notes API.
```

Both lines are stories from this book. The first is my own Mac running into `MaxAuthTries` while I tested the SSH hardening. The second is the `EADDRINUSE` failure from the systemd chapter.

Priority is a hint, not a verdict, though. Apps often print real failures as plain output, which the journal stores at priority 6 (`info`). When `-p err` shows nothing for a service, I go back to the full unit log.

The fields behind each line

Each line you see is a structured record with many more fields. `-o json-pretty` shows them all:

```
journalctl -u notes-api -o json-pretty -n 1 --no-pager | grep -E '"(MESSAGE|_PID|_UID|PRIORITY|
    ↪ _SYSTEMD_UNIT)"'
```

```
    "MESSAGE" : "notes-api listening on http://127.0.0.1:3000",
    "_SYSTEMD_UNIT" : "notes-api.service",
    "PRIORITY" : "6",
    "_PID" : "5762",
    "_UID" : "999"
```

`_UID` 999 is our `notes` user. The fields that start with an underscore are added by journald itself, so a process can't fake them. Any field works as a filter too, so `journalctl _UID=999` shows everything the `notes` user logged, whatever the unit.

Previous boots

After a crash or an unexpected reboot, the interesting logs are in the previous boot. The journal keeps track of boots:

```
journalctl --list-boots --no-pager
```

```
IDX BOOT ID                          FIRST ENTRY                 LAST ENTRY
 -1 1710cae489654efb8bb97805e92e8cb6 Sat 2026-11-28 16:13:19 UTC Sat 2026-11-28 16:17:24 UTC
  0 0bbfda88b87b44eab59ba89c52b19093 Sat 2026-11-28 16:17:32 UTC Sat 2026-11-28 16:18:15 UTC
```

`-b -1` reads the one before the current boot. Here are its last lines, from the reboot after the kernel update:

```
journalctl -b -1 --no-pager | tail -2
```

```
Nov 28 16:17:24 handbook-linux-test systemd-journald[4073]: Received SIGTERM from PID 1 (systemd-
    ↪ shutdow).
Nov 28 16:17:24 handbook-linux-test systemd-journald[4073]: Journal stopped
```

A clean shutdown ends like that. If the last lines of a boot are ordinary messages that stop in the middle of nothing, the server lost power or the kernel froze.

The files in /var/log

Not everything is in the journal only. rsyslog still writes the classic text files, and some programs (Nginx, APT, cloud-init) write their own:

```
ls -l /var/log/
```

```
total 1184
lrwxrwxrwx  1 root      root               39 Oct 17 12:55 README -> ../../usr/share/doc/systemd/
    ↪ README.logs
-rw-r--r--  1 root      root             1628 Nov 28 16:20 alternatives.log
-rw-r-----  1 root      adm                 0 Nov 28 16:13 apport.log
drwxr-xr-x  2 root      root             4096 Nov 28 16:20 apt
-rw-r-----  1 syslog    adm            179893 Nov 28 16:23 auth.log
-rw-rw----  1 root      utmp                0 Nov 28 16:13 btmp
-rw-r-----  1 root      adm             12437 Nov 28 16:17 cloud-init-output.log
-rw-r-----  1 syslog    adm            221651 Nov 28 16:17 cloud-init.log
drwxr-xr-x  2 root      root             4096 Sep  3 21:11 dist-upgrade
-rw-r-----  1 root      adm             46172 Nov 28 16:17 dmesg
-rw-r-----  1 root      adm             48250 Nov 28 16:13 dmesg.0
-rw-r--r--  1 root      root            75215 Nov 28 16:20 dpkg.log
drwxr-sr-x+ 3 root      systemd-journal  4096 Nov 28 16:13 journal
-rw-r-----  1 syslog    adm            128180 Nov 28 16:23 kern.log
...
-rw-r-----  1 syslog    adm            423247 Nov 28 16:23 syslog
...
```

Most of them belong to the `adm` group, which is the other reason to have your admin user in it. The ones I open most often:

- `auth.log` has SSH logins and every `sudo`. On this server it already had 1,381 lines after half an hour, mostly my own `sudo` commands.
- `syslog` is the general system log, the text twin of the journal.
- `dpkg.log` and the `apt` folder record every package installed or upgraded, with times. Great for "what changed last night?".

- `cloud-init-output.log` shows what happened at the very first boot of a cloud server.
- `nginx/` gets its access and error logs once we install Nginx. We'll read those in the Nginx chapter.

These files don't grow forever. The `logrotate.timer` we saw in the previous chapter runs every night, renames the current file (`syslog.1`), compresses the older ones, and deletes the oldest. The rules live in `/etc/logrotate.d`, one file per program.

My routine when something breaks: the unit log first (`journalctl -u` with `--since`), then the errors of the current boot (`-p err -b`), then the specific file in `/var/log` for that program. In that order, I rarely need to read more than a hundred lines.

Nginx

Right now the notes API listens on `127.0.0.1:3000`. Only processes on the server itself can reach it, and that's exactly what I want.

The internet talks to a **reverse proxy** instead. It listens on ports 80 and 443, handles HTTPS, writes the access logs, and forwards each request to the API on localhost. The API never deals with certificates or with clients directly.

The first reverse proxy we'll use is Nginx. It's in the Ubuntu archive, it's been around forever, and most of the configuration you'll find online is written for it.

Install Nginx and open the firewall

Nginx comes straight from Ubuntu:

```
sudo apt-get install -y nginx
nginx -v
```

```
nginx version: nginx/1.24.0 (Ubuntu)
```

Ubuntu 24.04 ships 1.24, and that version number will matter later in this chapter.

The package starts the service right away. Let's look at the processes:

```
ps -o user,pid,cmd -C nginx
```

```
USER         PID CMD
root        7707 nginx: master process /usr/sbin/nginx -g daemon on; master_process on;
www-data    7710 nginx: worker process
```

Nginx runs as two kinds of process. The **master** runs as root, reads the configuration and opens the listening ports (binding to port 80 needs privileges). The **workers** run as the unprivileged `www-data` user and handle the actual requests.

There's one worker because `worker_processes` is set to `auto`, which means one per CPU core, and this Droplet has one core.

Nginx is listening on port 80:

```
sudo ss -lntp | grep nginx
```

```
LISTEN 0      511          0.0.0.0:80        0.0.0.0:*    users:(("nginx",pid=7710,fd=5),("nginx",pid
    ↪ =7707,fd=5))
LISTEN 0      511             [::]:80           [::]:*    users:(("nginx",pid=7710,fd=6),("nginx",pid
    ↪ =7707,fd=6))
```

But UFW still blocks everything except SSH. The Nginx package drops a few application profiles into UFW, so we don't have to remember port numbers:

```
sudo ufw app list
```

```
Available applications:
  Nginx Full
  Nginx HTTP
  Nginx HTTPS
  OpenSSH
```

`Nginx Full` opens both 80 and 443. We need 80 for the HTTP to HTTPS redirect and for the Let's Encrypt challenge, and 443 for everything else:

```
sudo ufw allow 'Nginx Full'
sudo ufw status
```

```
Status: active

To                         Action      From
--                         ------      ----
OpenSSH                    ALLOW       Anywhere
Nginx Full                 ALLOW       Anywhere
OpenSSH (v6)               ALLOW       Anywhere (v6)
Nginx Full (v6)            ALLOW       Anywhere (v6)
```

Notice what's missing: port 3000. We never open it.

How the configuration is organized

Everything lives in `/etc/nginx`:

```
ls /etc/nginx/ /etc/nginx/sites-enabled/
```

```
/etc/nginx/:
conf.d
fastcgi.conf
fastcgi_params
```

```
koi-utf
koi-win
mime.types
modules-available
modules-enabled
nginx.conf
proxy_params
scgi_params
sites-available
sites-enabled
snippets
uwsgi_params
win-utf

/etc/nginx/sites-enabled/:
default
```

The entry point is `nginx.conf`, and it pulls in the other files with `include`:

```
grep -nE 'worker_processes|worker_connections|include' /etc/nginx/nginx.conf
```

```
2:worker_processes auto;
5:include /etc/nginx/modules-enabled/*.conf;
8:  worker_connections 768;
26: include /etc/nginx/mime.types;
59: include /etc/nginx/conf.d/*.conf;
60: include /etc/nginx/sites-enabled/*;
```

The configuration is a tree of **contexts**.

Process settings like `worker_processes` sit at the top level, connection settings like `worker_connections` go in `events`, and everything about HTTP goes in `http`. Inside `http` each site is a `server` block, with `location` blocks inside that.

A directive applies to the context it's written in, and to the contexts nested inside it.

On Ubuntu, each site gets its own file in `sites-available`, and you enable it with a symlink in `sites-enabled`. Line 60 above includes only the enabled ones. Removing the symlink disables a site without deleting its configuration.

A server block for the notes API

Here's the first version of the site, in `/etc/nginx/sites-available/notes`:

```
server {
    listen 80;
    listen [::]:80;
    server_name 203-0-113-42.sslip.io;

    access_log /var/log/nginx/notes.access.log;
    error_log /var/log/nginx/notes.error.log;
```

```
    location / {
        proxy_pass http://127.0.0.1:3000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

`server_name` is the domain we pointed at the server in the DNS chapter. `proxy_pass` sends every request to the API.

Then the four `proxy_set_header` lines. Without them, the API would see every request coming from `127.0.0.1` over plain HTTP, because from its point of view Nginx *is* the client. These headers pass along the original host, the real client IP, and whether the request arrived over HTTPS.

Be careful: an application should trust `X-Forwarded-For` only when the request comes from the proxy. If the API were also reachable directly, anyone could send a fake header.

The site gets its own access and error logs, which makes them much easier to read than one shared file.

Now we enable the site, test the configuration, and reload:

```
sudo cp notes.conf /etc/nginx/sites-available/notes
sudo ln -s /etc/nginx/sites-available/notes /etc/nginx/sites-enabled/notes
sudo nginx -t
```

```
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

```
sudo systemctl reload nginx
```

`nginx -t` reads the whole configuration and checks it without touching the running server. I run it before every single reload, and later in this chapter you'll see why.

Test it from outside

From my Mac, the API now answers through Nginx:

```
curl -si http://203-0-113-42.sslip.io/notes
```

```
HTTP/1.1 200 OK
Server: nginx/1.24.0 (Ubuntu)
Date: Sat, 28 Nov 2026 16:24:42 GMT
Content-Type: application/json
Transfer-Encoding: chunked
```

```
Connection: keep-alive

[{"id":1,"text":"buy milk","createdAt":"2026-11-28T16:21:35.718Z"}]
```

And port 3000 is not reachable from the internet, also from my Mac:

```
curl -s -o /dev/null -w '%{http_code}\n' http://203-0-113-42.sslip.io:3000/health --max-time 5


000
```

`000` means curl got no HTTP response at all. It gave up after 5 seconds with exit code 28, a timeout, because UFW drops the packets without answering. That's the result we want.

On the server, each request shows up in the site's access log:

```
tail -3 /var/log/nginx/notes.access.log


203.0.113.7 - - [28/Nov/2026:16:24:42 +0000] "GET /health HTTP/1.1" 200 21 "-" "curl/8.7.1"
203.0.113.7 - - [28/Nov/2026:16:24:42 +0000] "GET /notes HTTP/1.1" 200 78 "-" "curl/8.7.1"
```

Each line has my home IP, the request, the status code, the response size and curl's user agent.

Which server block answers

One Nginx can host many sites. For each request it picks a `server` block in two steps: first the `listen` directives (which address and port the client connected to), then the `server_name` compared with the `Host` header.

When no name matches, Nginx uses the **default server** for that port. And here's what that looked like on our fresh server, requesting the bare IP address from my Mac:

```
curl -si http://203.0.113.42/ | head -4


HTTP/1.1 200 OK
Server: nginx/1.24.0 (Ubuntu)
Date: Sat, 28 Nov 2026 16:24:43 GMT
Content-Type: text/html
```

That's the "Welcome to nginx!" page from the `default` site Ubuntu enables. Harmless here, but every scanner that hits your IP learns you run Nginx 1.24 on Ubuntu, and on a server with several sites the default could end up being one you didn't mean to expose.

If you never mark a default, Nginx uses the first server block it loads for that port. So the order of the files in `sites-enabled` decides what strangers see. I prefer an explicit catch-all that answers nothing, and we'll add it in a moment.

Inside a server block, the request path picks one `location`.

An exact match (`location = /health`) wins immediately. Otherwise Nginx remembers the longest matching prefix, then checks the regex locations (`location ~ \.png$`) in file order, and the first matching regex wins over that prefix. A prefix marked `^~` skips the regex check.

Our site has a single `location /`, which matches every path. I keep the set of locations small enough that I can tell which one wins for any URL without testing. When I can't, I test with curl.

HTTPS with Let's Encrypt

On Nginx, certificates come from **certbot**, with a plugin that edits the Nginx configuration for you:

```
sudo apt-get install -y certbot python3-certbot-nginx
certbot --version
```

```
certbot 2.9.0
```

Certbot proves to Let's Encrypt that we control the domain by answering an HTTP challenge on port 80, so that's one reason we opened it. Then it installs the certificate and, with `--redirect`, adds the HTTP to HTTPS redirect:

```
sudo certbot --nginx -d 203-0-113-42.sslip.io --non-interactive --agree-tos --register-unsafely-without
    ↪ -email --redirect
```

```
...
Deploying certificate
Successfully deployed certificate for 203-0-113-42.sslip.io to /etc/nginx/sites-enabled/notes
Congratulations! You have successfully enabled HTTPS on https://203-0-113-42.sslip.io
...
```

I used `--register-unsafely-without-email` because this is a throwaway test server. On a real one, pass `-m` with your email, so Let's Encrypt can warn you when something about your certificates needs attention.

Here's what certbot did to our file:

```
server {
    server_name 203-0-113-42.sslip.io;

    access_log /var/log/nginx/notes.access.log;
    error_log /var/log/nginx/notes.error.log;

    location / {
        proxy_pass http://127.0.0.1:3000;
        ...
    }

    listen [::]:443 ssl ipv6only=on; # managed by Certbot
    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/203-0-113-42.sslip.io/fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/203-0-113-42.sslip.io/privkey.pem; # managed by Certbot
```

```
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot

}
server {
    if ($host = 203-0-113-42.sslip.io) {
        return 301 https://$host$request_uri;
    } # managed by Certbot


    listen 80;
    listen [::]:80;
    server_name 203-0-113-42.sslip.io;
    return 404; # managed by Certbot


}
```

It moved our server block to port 443 and added a second block on port 80 that redirects. It works, but it's not how I'd write it by hand. We'll clean it up shortly.

From my Mac, the redirect and the certificate:

```
curl -sI http://203-0-113-42.sslip.io/notes | head -3
```

```
HTTP/1.1 301 Moved Permanently
Server: nginx/1.24.0 (Ubuntu)
Date: Sat, 28 Nov 2026 16:31:51 GMT
```

```
echo | openssl s_client -connect 203-0-113-42.sslip.io:443 -servername 203-0-113-42.sslip.io 2>/dev/
    ↪ null | openssl x509 -noout -issuer -subject -dates
```

```
issuer=C=US, O=Let's Encrypt, CN=YE2
subject=CN=203-0-113-42.sslip.io
notBefore=Nov 28 15:26:56 2026 GMT
notAfter=Feb 26 15:26:55 2027 GMT
```

The certificate is valid for 90 days. Renewal is automatic, through a systemd timer the package installed:

```
systemctl list-timers certbot.timer --no-pager
```

```
NEXT                         LEFT LAST PASSED UNIT          ACTIVATES
Sun 2026-11-29 04:16:57 UTC  11h -         - certbot.timer certbot.service
```

I always test the renewal once with a dry run, which talks to the Let's Encrypt staging server and changes nothing. The first time I ran it, it sat there for minutes with no output, and I thought it was stuck.

It wasn't. When certbot runs without a terminal attached (as it does from a timer, and from my SSH script), it waits a random amount of time before renewing, so millions of servers don't hit Let's Encrypt at the same second. The flag `--no-random-sleep-on-renew` skips the wait:

```
sudo certbot renew --dry-run --no-random-sleep-on-renew


...
Congratulations, all simulated renewals succeeded:
  /etc/letsencrypt/live/203-0-113-42.sslip.io/fullchain.pem (success)
...
```

More details about certbot and Nginx are in How to configure Nginx for HTTPS.

The final configuration

Now I rewrite the site by hand, keeping the certificate paths certbot created. This is the file I want to maintain:

```
limit_req_zone $binary_remote_addr zone=notes_api:10m rate=5r/s;

server {
    listen 80;
    listen [::]:80;
    server_name 203-0-113-42.sslip.io;
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl http2;
    listen [::]:443 ssl http2;
    server_name 203-0-113-42.sslip.io;

    ssl_certificate /etc/letsencrypt/live/203-0-113-42.sslip.io/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/203-0-113-42.sslip.io/privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;

    add_header Strict-Transport-Security "max-age=31536000" always;

    access_log /var/log/nginx/notes.access.log;
    error_log /var/log/nginx/notes.error.log;

    location / {
        limit_req zone=notes_api burst=10 nodelay;
        proxy_pass http://127.0.0.1:3000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

The port 80 block does one thing: redirect to HTTPS. Certbot renewals still work with it, because the certificate's renewal file says to use the Nginx plugin, which adds its challenge location temporarily while it renews:

```
sudo grep -E 'authenticator|installer' /etc/letsencrypt/renewal/203-0-113-42.sslip.io.conf
```

```
authenticator = nginx
installer = nginx
```

I ran the dry run again after this rewrite to be sure:

```
sudo certbot renew --dry-run --no-random-sleep-on-renew
```

```
Congratulations, all simulated renewals succeeded:
  /etc/letsencrypt/live/203-0-113-42.sslip.io/fullchain.pem (success)
```

`Strict-Transport-Security` (HSTS) tells browsers to use HTTPS for this domain for the next year, even if someone types `http://`. Once a browser has seen it, you can't easily go back to plain HTTP, so add it when HTTPS works and you plan to keep it. `always` sends the header on error responses too.

The `limit_req` lines are rate limiting, which we'll test in a moment.

Then the catch-all, in its own file `/etc/nginx/sites-available/catch-all`, replacing Ubuntu's `default` site:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    listen 443 ssl default_server;
    listen [::]:443 ssl default_server;
    ssl_reject_handshake on;
    return 444;
}
```

`return 444` is special to Nginx: it closes the connection without sending any response. `ssl_reject_handshake on` does the same at the TLS level for HTTPS, so the server doesn't even need a certificate for unknown names.

Last, `server_tokens off` in `nginx.conf` hides the version number. Ubuntu ships the line commented out:

```
sudo cp /etc/nginx/sites-available/notes /etc/nginx/sites-available/notes.bak
sudo cp notes-final.conf /etc/nginx/sites-available/notes
sudo rm /etc/nginx/sites-enabled/default
sudo cp default.conf /etc/nginx/sites-available/catch-all
sudo ln -s /etc/nginx/sites-available/catch-all /etc/nginx/sites-enabled/catch-all
sudo sed -i 's/# server_tokens off;/server_tokens off;/' /etc/nginx/nginx.conf
```

Notice the first line: I keep a copy of the working file before I change it. That copy is my rollback.

When nginx -t saves you

My first version of that file had `http2 on;` on its own line, which is how the current Nginx documentation enables HTTP/2. Here's what happened:

```
sudo nginx -t
```

```
2026/11/28 16:33:09 [emerg] 9441#9441: unknown directive "http2" in /etc/nginx/sites-enabled/notes:13
nginx: configuration file /etc/nginx/nginx.conf test failed
```

The `http2` directive arrived in Nginx 1.25.1. Ubuntu 24.04 has 1.24, which only knows the older form, `listen 443 ssl http2;`. Copying configuration from the latest docs onto the version your distribution ships is one of the most common ways to break Nginx.

Now, what happens if you reload anyway? I tried it on purpose:

```
sudo systemctl reload nginx; systemctl is-active nginx
```

```
Job for nginx.service failed.
See "systemctl status nginx.service" and "journalctl -xeu nginx.service" for details.
active
```

The reload failed, but Nginx stayed `active`, still serving the previous configuration. The master process refuses to load a broken config and keeps the old workers running. A `restart` would have been a different story: stop, then fail to start, and the site goes down. That's why I use `reload`, and always after `nginx -t`.

The fix was to move `http2` to the `listen` lines:

```
sudo sed -i '/http2 on;/d; s/listen 443 ssl;/listen 443 ssl http2;/; s/listen \[::\]:443 ssl;/listen
    ↪ [::]:443 ssl http2;/' /etc/nginx/sites-available/notes
sudo nginx -t && sudo systemctl reload nginx && echo reloaded
```

```
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
reloaded
```

A reload keeps the master process and replaces the workers. You can see it in the start times:

```
ps -o pid,lstart,cmd -C nginx
```

```
    PID                  STARTED CMD
   7707 Sat Nov 28 16:24:13 2026 nginx: master process /usr/sbin/nginx -g daemon on; master_process on;
   9755 Sat Nov 28 16:33:33 2026 nginx: worker process
```

The master has been running since the install, while the worker is new.

The old workers stop accepting connections and exit when their current requests are done, so nobody sees a dropped connection.

My first check from the Mac, right after that reload, still came back with the old answer (`HTTP/1.1` and the version in the `Server` header). On the server, the new configuration was already loaded:

```
sudo nginx -T 2>/dev/null | grep -nE 'server_tokens|Strict|limit_req|listen 443'
```

```
22: server_tokens off;
193:    listen 443 ssl default_server;
200:limit_req_zone $binary_remote_addr zone=notes_api:10m rate=5r/s;
210:    listen 443 ssl http2;
218:    add_header Strict-Transport-Security "max-age=31536000" always;
224:        limit_req zone=notes_api burst=10 nodelay;
```

`nginx -T` (capital T) prints the complete configuration Nginx would load, with every include expanded. It's the best way to answer "what is Nginx actually using?".

Testing again from the Mac a moment later, everything was there:

```
curl -sI https://203-0-113-42.sslip.io/health | grep -iE '^(HTTP|server|strict)'
```

```
HTTP/2 200
server: nginx
strict-transport-security: max-age=31536000
```

HTTP/2, no version number, and the HSTS header. So my habit now is to check twice after a reload before deciding something is wrong.

The catch-all works too. From my Mac, the bare IP over HTTP gets no reply at all (curl exits with code 52, "empty reply from server"), and over HTTPS the handshake is refused (exit code 35, a TLS error):

```
curl -sI http://203.0.113.42/ ; echo "exit $?"
```

```
exit 52
```

```
curl -skI https://203.0.113.42/ ; echo "exit $?"
```

```
exit 35
```

Rate limiting

These two lines from the final config limit how fast a single IP can call the API:

```
limit_req_zone $binary_remote_addr zone=notes_api:10m rate=5r/s;
```

```
limit_req zone=notes_api burst=10 nodelay;
```

The zone tracks clients by IP in 10 MB of shared memory, and allows 5 requests per second each. `burst=10` lets a client go 10 requests over that rate before Nginx starts

refusing, and `nodelay` serves those extra requests right away instead of queueing them.

My first test sent 20 requests in a row from my Mac:

```
for i in $(seq 1 20); do curl -s -o /dev/null -w '%{http_code} ' https://203-0-113-42.sslip.io/health;
    ↪ done; echo


200 200 200 200 200 200 200 200 200 200 200 200 200 200 200 200 200 200 200 200
```

All 200. Each request waits for the previous one to finish, with a new HTTPS connection every time, so they arrived more slowly than the limit. A real burst arrives in parallel, so I sent 30 requests at once:

```
seq 1 30 | xargs -P 30 -I{} curl -s -o /dev/null -w '%{http_code}\n' https://203-0-113-42.sslip.io/
    ↪ health | sort | uniq -c


  11 200
  19 503
```

One request within the rate, plus the burst of 10, got through. The other 19 got a `503`. The error log says why:

```
sudo tail -1 /var/log/nginx/notes.error.log


2026/11/28 16:34:23 [error] 9755#9755: *349 limiting requests, excess: 10.645 by zone "notes_api",
    ↪ client: 203.0.113.7, server: 203-0-113-42.sslip.io, request: "GET /health HTTP/2.0", host:
    ↪ "203-0-113-42.sslip.io"
```

`503 Service Unavailable` is Nginx's default for rate limited requests. If your clients should back off and retry, `limit_req_status 429;` sends `429 Too Many Requests` instead, which is the code meant for this.

Pick the numbers from your real traffic. A page that loads 30 assets from the same server would trip this limit on the first visit, so I put it on the API location only.

WebSockets and long requests

The notes API doesn't use WebSockets, but it's the first thing that breaks when you put an app with live updates behind Nginx, so here's what it needs. I added this block above `location /` on the test server, and `nginx -t` accepted it. A request to `/ws/` reached the API, which answered 404 because it has no WebSocket endpoint, so this is as far as I could test it:

```
location /ws/ {
    proxy_pass http://127.0.0.1:3000;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_read_timeout 300s;
}
```

A WebSocket starts as an HTTP request with an `Upgrade` header. By default Nginx talks HTTP/1.0 to the backend and doesn't forward `Upgrade`, so the upgrade never happens. `proxy_http_version 1.1` and the two headers fix that.

`proxy_read_timeout` counts the time between two messages on the connection, and the default is 60 seconds. A quiet but healthy socket gets closed after a minute unless you raise it, or the app sends pings more often than that. Server-sent events need the same timeout plus `proxy_buffering off;`, otherwise Nginx holds the events in its buffer.

I wrote more about proxying several Node.js apps in Nginx reverse proxy for multiple Node.js apps in subfolders.

My routine for every change

Every Nginx change on this server followed the same steps, and I'd keep them on any server:

1. Copy the working file (`notes.bak`).
2. Edit.
3. `sudo nginx -t`.
4. `sudo systemctl reload nginx`, never `restart`.
5. Test from outside with curl, twice if the first answer looks old.
6. If something's wrong, copy the `.bak` file back, `nginx -t`, reload.

The error log is in `/var/log/nginx/notes.error.log`, the access log next to it. When a request misbehaves, I look for it there before changing anything.

Caddy, and when to pick it

Nginx needed a separate tool, certbot, plus a timer, to get and renew certificates. **Caddy** is a web server that does HTTPS on its own: you write the domain name, and it gets the certificate, renews it, and redirects HTTP to HTTPS.

I wanted to see that on the same server, so I stopped Nginx for a few minutes and put Caddy in front of the notes API, on a second name: `caddy.203-0-113-42.sslip.io`. sslip.io resolves any name that ends with the IP, so it points at the same Droplet.

Install Caddy from its repository

Caddy isn't in the Ubuntu archive at a recent version, so I used the official repository. The commands add Caddy's signing key and its APT source:

```
sudo apt-get install -y debian-keyring debian-archive-keyring apt-transport-https curl
curl -1sLf 'https://dl.cloudsmith.io/public/caddy/stable/gpg.key' | sudo gpg --dearmor -o /usr/share/
    ↪ keyrings/caddy-stable-archive-keyring.gpg
curl -1sLf 'https://dl.cloudsmith.io/public/caddy/stable/debian.deb.txt' | sudo tee /etc/apt/sources.
    ↪ list.d/caddy-stable.list
```

The second `curl` prints the source it installs:

```
# Source: Caddy
# Site: https://github.com/caddyserver/caddy
# Repository: Caddy / stable
# Description: Fast, multi-platform web server with automatic HTTPS


deb [signed-by=/usr/share/keyrings/caddy-stable-archive-keyring.gpg] https://dl.cloudsmith.io/public/
    ↪ caddy/stable/deb/debian any-version main

deb-src [signed-by=/usr/share/keyrings/caddy-stable-archive-keyring.gpg] https://dl.cloudsmith.io/
    ↪ public/caddy/stable/deb/debian any-version main
```

`signed-by` means APT accepts packages from this source only when they're signed with that key, the same setup we used for Node.js.

Caddy wants ports 80 and 443 too, so Nginx has to stop first:

```
sudo systemctl stop nginx
sudo apt-get update -qq && sudo apt-get install -y caddy
caddy version


v2.11.9 h1:bhZtcRKQOq0fuLvEgCdPYvUxXyGn9kzjG4lh2332DTE=
```

Like Nginx, the package starts the service immediately:

```
systemctl is-active caddy; sudo ss -lntp | grep caddy


active
LISTEN 0      4096       127.0.0.1:2019      0.0.0.0:*    users:(("caddy",pid=11004,fd=3))
LISTEN 0      4096               *:80              *:*    users:(("caddy",pid=11004,fd=6))
```

Port 80 serves a default welcome page. Port 2019, on localhost only, is Caddy's **admin API**. That's how Caddy reloads its configuration without a restart, and it's why it must never be reachable from outside.

A Caddyfile for the notes API

Caddy's configuration lives in `/etc/caddy/Caddyfile`. The default one serves static files from `/usr/share/caddy` on port 80. I replaced it with this:

```
caddy.203-0-113-42.sslip.io {
    encode zstd gzip
    reverse_proxy 127.0.0.1:3000
    log {
        output file /var/log/caddy/notes.log
    }
}
```

Seven lines. The site address is a domain name, and that alone turns on automatic HTTPS. `encode` compresses responses, `reverse_proxy` forwards to the API (Caddy sets `X-Forwarded-For` and `X-Forwarded-Proto` on its own), and `log` writes the access log to a file.

`caddy fmt` formats a Caddyfile the canonical way, and `--diff` shows what it would change:

```
sudo cp Caddyfile /etc/caddy/Caddyfile
caddy fmt --diff /etc/caddy/Caddyfile
```

```
  caddy.203-0-113-42.sslip.io {
    encode zstd gzip
    reverse_proxy 127.0.0.1:3000
    log {
        output file /var/log/caddy/notes.log
    }
  }
```

No `+` or `-` lines, so the file was already formatted. Then `caddy validate`, which is Caddy's `nginx -t`:

```
caddy validate --config /etc/caddy/Caddyfile
```

```
{"level":"info","ts":1795883744.397653,"msg":"adapted config to JSON","adapter":"caddyfile"}
{"level":"error","ts":1795883744.3982644,"msg":"setting up custom log 'log0': opening log writer using
    ↪ &logging.FileWriter{...}: open /var/log/caddy/notes.log: permission denied"}
```

Permission denied? The configuration was fine. `caddy validate` does more than check syntax: it sets up the modules, including opening the log file. I ran it as `flavio`, who can't write in `/var/log/caddy`. The service runs as the `caddy` user, so the honest test runs as that user too:

```
sudo -u caddy caddy validate --config /etc/caddy/Caddyfile
```

```
Valid configuration
```

Automatic HTTPS

Now the reload. With the domain in the Caddyfile, Caddy talks to Let's Encrypt on its own:

```
sudo systemctl reload caddy
```

Fifteen seconds later I grouped the messages in Caddy's journal (Caddy logs in JSON, one object per line):

```
journalctl -u caddy --no-pager -n 40 -o cat | grep -oE '"msg":"[^"]+"' | sort | uniq -c | sort -rn
```

```
      7 "msg":"served key authentication"
      3 "msg":"got renewal info"
      2 "msg":"server running"
      2 "msg":"creating new account because no account for configured email is known to us"
```

```
  1 "msg":"waiting on internal rate limiter"
  1 "msg":"validations succeeded; finalizing order"
  1 "msg":"using ACME account"
  1 "msg":"trying to solve challenge"
  1 "msg":"successfully downloaded available certificate chains"
  1 "msg":"stopped previous server"
  1 "msg":"servers shutting down with eternal grace period"
  1 "msg":"server is listening only on the HTTPS port but has no TLS connection policies; adding
↪ one to enable TLS"
```

Put in order, the story is: Caddy created an ACME account, solved the challenge (the "served key authentication" lines are Let's Encrypt's validation servers asking for the token), and downloaded the certificate. All within those 15 seconds, with no extra command.

The certificates live in Caddy's data directory:

```
sudo ls /var/lib/caddy/.local/share/caddy/certificates/
```

```
acme-v02.api.letsencrypt.org-directory
```

From my Mac:

```
curl -sI https://caddy.203-0-113-42.sslip.io/health | grep -iE '^(HTTP|server|alt-svc)'
```

```
HTTP/2 200
alt-svc: h3=":443"; ma=2592000
```

HTTP/2, and an `alt-svc` header announcing HTTP/3 on the same port, both on by default. No `Server` header with a version came back either.

Plain HTTP redirects:

```
curl -sI http://caddy.203-0-113-42.sslip.io/notes | head -3
```

```
HTTP/1.1 308 Permanent Redirect
Connection: close
Location: https://caddy.203-0-113-42.sslip.io/notes
```

Caddy uses `308` where certbot's Nginx config used `301`. Both are permanent, but `308` guarantees that a `POST` stays a `POST` after the redirect.

The certificate:

```
echo | openssl s_client -connect caddy.203-0-113-42.sslip.io:443 -servername caddy.203-0-113-42.sslip.
↪ io 2>/dev/null | openssl x509 -noout -issuer -dates
```

```
issuer=C=US, O=Let's Encrypt, CN=YE1
notBefore=Nov 28 15:37:18 2026 GMT
notAfter=Feb 26 15:37:17 2027 GMT
```

Same 90 days as certbot's. Caddy renews it on its own, well before it expires, from inside the running process. There's no timer to check, but it also means Caddy must keep running and be able to reach Let's Encrypt.

The access log file was created by the service, owned by `caddy`, readable only by that user:

```
ls -l /var/log/caddy/
```

```
total 48
-rw------- 1 caddy caddy 48771 Nov 28 16:36 notes.log
```

Reloading and running it as a service

`systemctl reload caddy` sends the new configuration through the admin API. If the new config is invalid, Caddy keeps running the old one, the same safety net we saw with Nginx.

The Ubuntu package already sets up the systemd unit, running as the `caddy` user. That user owns `/var/lib/caddy`, where the certificates and the ACME account live. Losing that directory isn't a disaster, Caddy would ask for new certificates, but if you run many sites and lose it often, you can hit Let's Encrypt's rate limits. I include it in backups.

Back to Nginx

The rest of this book uses Nginx, so I turned Caddy off and Nginx back on:

```
sudo systemctl disable --now caddy && sudo systemctl start nginx && systemctl is-active nginx caddy
```

```
Removed "/etc/systemd/system/multi-user.target.wants/caddy.service".
active
inactive
```

`disable --now` stops the service and removes it from boot in one step, so the two servers don't fight over port 80 after the next reboot.

Nginx or Caddy?

Here's what I saw on this server, plus the idle memory I measured in a separate test with the `nginx:alpine` and `caddy:2-alpine` containers:

	Nginx 1.24 (Ubuntu)	Caddy 2.11
Where it comes from	Ubuntu archive, security fixes from Ubuntu	Caddy's own repository, always the latest release
HTTPS	certbot + `certbot.timer`	Built in, renews inside the process

	Nginx 1.24 (Ubuntu)	Caddy 2.11
Config for the notes API	About 30 lines, two server blocks	7 lines
HTTP to HTTPS redirect	You write it (or certbot does), `301`	Automatic, `308`
HTTP/2 / HTTP/3	HTTP/2 with `listen ... http2`	HTTP/2 and HTTP/3 on by default
Rate limiting	Built in (`limit_req`)	Needs a third-party module
Logs	Text, combined format	JSON
Idle memory (container test)	4.56 MiB	15.98 MiB

My advice: for a server with one or a few apps, where you want HTTPS working without thinking about it, Caddy is the faster path, and its Caddyfile is hard to get wrong.

Nginx is what I pick when I need fine control over caching, rate limits or routing, when the server is tiny, or when I want the web server to get security updates through `apt` like everything else. And so much of the internet's configuration is written for Nginx that you'll be reading it anyway.

Both put the app behind a proxy on localhost, and both refuse to load a broken config on reload. Whichever you pick, test before you reload, and keep port 3000 closed.

Disks and filesystems

So far the notes live in `/var/lib/notes-api`, on the Droplet's main 25 GB disk, together with the operating system. If the data grows, it competes with logs and packages for the same space. And if I ever want to move it to a bigger server, I have to copy it out first.

So I gave the data its own disk. On DigitalOcean that's a **volume**, a block device you attach to a Droplet and can resize or move to another Droplet later. At home it would be a second SSD. Everything after the attach step is the same.

Add a disk and find it

I created a 10 GB volume and attached it from my Mac with `doctl`, DigitalOcean's command line tool:

```
doctl compute volume create handbook-notes-data --region fra1 --size 10GiB
doctl compute volume-action attach deddea78-eb49-11f1-b370-0afd82f81f7a 624334546 --wait
```

The first command prints the volume ID, and the second attaches that volume to the Droplet with ID `624334546`. I attached it as a plain block device, without a filesystem, because I wanted to set it up myself.

On the server, `lsblk` lists the block devices:

```
lsblk -o NAME,SIZE,TYPE,FSTYPE,MOUNTPOINTS
```

```
NAME     SIZE TYPE FSTYPE  MOUNTPOINTS
sda       10G disk
vda       25G disk
+-vda1    24G part ext4    /
+-vda14    4M part
+-vda15  106M part vfat    /boot/efi
+-vda16  913M part ext4    /boot
vdb      488K disk iso9660
```

`vda` is the main disk, with `/` on `vda1`. The new 10 GB disk showed up as `sda`, with no partitions and no filesystem. The kernel logged it when it appeared:

```
sudo dmesg | grep -i 'sda' | tail -3
```

```
[ 1154.517584] sd 0:0:0:1: [sda] Mode Sense: 63 00 00 08
[ 1154.517633] sd 0:0:0:1: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO or FUA
[ 1154.523588] sd 0:0:0:1: [sda] Attached SCSI disk
```

Names like `sda` are assigned at boot in the order devices are found, so they can change. Before running anything destructive, I make sure `sda` really is the volume I just attached. `/dev/disk/by-id` has stable names that say what the device is:

```
ls -l /dev/disk/by-id/ | grep -i volume
```

```
lrwxrwxrwx 1 root root 9 Nov 28 16:36 scsi-0DO_Volume_handbook-notes-data -> ../../sda
```

That's my volume, by name. Then two checks that it's empty. `blkid` prints nothing and exits with code 2 when it finds no filesystem or other signature, and `wipefs` without options only lists signatures, it doesn't erase them:

```
sudo blkid /dev/sda; echo blkid exit $?
```

```
blkid exit 2
```

```
sudo wipefs /dev/sda
```

No output: nothing on the disk. If either command had printed something, I'd stop and find out what that data is.

The storage stack

Between a raw disk and a folder you can write to, Linux stacks a few layers:

```
/dev/sda                     the DigitalOcean volume
+-- LVM physical volume
    +-- volume group "data"
        +-- logical volume "data/notes"
            +-- ext4 filesystem, label "notes"
                +-- mounted on /var/lib/notes-api
```

You could skip LVM and put a filesystem straight on the disk, or on a partition. I used **LVM** (Logical Volume Manager) because it makes growing painless, and because it gives us snapshots. And I used the whole disk instead of a partition, because a volume that only ever holds LVM doesn't need a partition table, and that's one less layer to grow later.

The three LVM layers are:

- a **physical volume** (PV): a disk or partition handed to LVM
- a **volume group** (VG): a pool of space made of one or more PVs
- a **logical volume** (LV): a slice of the pool, which behaves like a disk

The LVM tools are already on Ubuntu 24.04 (`apt-get install lvm2` changed nothing). Let's create the three layers:

```
sudo pvcreate /dev/sda
```

```
  Physical volume "/dev/sda" successfully created.
```

```
sudo vgcreate data /dev/sda
```

```
  Volume group "data" successfully created
```

```
sudo lvcreate -L 5G -n notes data
```

```
  Logical volume "notes" created.
```

I made the logical volume 5 GB on purpose, half the disk. Free space in the volume group is what lets us grow the volume later, or take a snapshot. `pvs`, `vgs` and `lvs` show the result:

```
sudo pvs; sudo vgs; sudo lvs
```

```
  PV         VG   Fmt  Attr PSize   PFree
  /dev/sda   data lvm2 a--  <10.00g <5.00g
  VG   #PV #LV #SN Attr   VSize   VFree
  data   1   1   0 wz--n- <10.00g <5.00g
  LV    VG   Attr       LSize Pool Origin Data%  Meta%  Move Log Cpy%Sync Convert
  notes data -wi-a----- 5.00g
```

Create the filesystem and move the data

The logical volume is `/dev/data/notes`. I put an ext4 filesystem on it, with the label `notes`:

```
sudo mkfs.ext4 -L notes /dev/data/notes
```

```
...
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done
```

```
lsblk -o NAME,SIZE,TYPE,FSTYPE,LABEL,MOUNTPOINTS /dev/sda
```

```
NAME         SIZE TYPE FSTYPE      LABEL MOUNTPOINTS
sda           10G disk LVM2_member
+-data-notes   5G lvm  ext4        notes
```

To move the data, I mounted the new filesystem in a temporary place, stopped the service so nothing writes while we copy, and copied with `rsync`:

```
sudo mkdir -p /mnt/newdata && sudo mount /dev/data/notes /mnt/newdata && df -h /mnt/newdata
```

```
Filesystem              Size  Used Avail Use% Mounted on
/dev/mapper/data-notes  4.9G   24K  4.6G   1% /mnt/newdata
```

```
sudo systemctl stop notes-api
sudo rsync -aHAX /var/lib/notes-api/ /mnt/newdata/
sudo ls -la /mnt/newdata
```

```
total 28
drwxr-x--- 3 notes notes  4096 Nov 28 16:21 .
drwxr-xr-x 3 root  root   4096 Nov 28 16:37 ..
drwx------ 2 root  root  16384 Nov 28 16:37 lost+found
-rw-rw-r-- 1 notes notes    92 Nov 28 16:21 notes.json
```

`-a` keeps permissions, owners and times, `-H` hard links, `-A` ACLs and `-X` extended attributes. The trailing slash on the source copies the *contents* of the folder, not the folder itself. Notice the owner: `notes`, the service user, exactly like the original.

The `lost+found` folder belongs to ext4. `fsck` puts recovered files there after a crash.

fstab without losing boot

Now the new filesystem has to replace the folder, and come back after every reboot. That's `/etc/fstab`.

I unmounted the temporary mount and got the filesystem's UUID:

```
sudo umount /mnt/newdata && sudo blkid /dev/data/notes
```

```
/dev/data/notes: LABEL="notes" UUID="b59b22a0-97c1-4df2-9b8f-d1ffc46f4724" BLOCK_SIZE="4096" TYPE="ext4
    ↪ "
```

I mount by UUID because it belongs to the filesystem and never changes, while names like `sda` can. Then I appended one line to `fstab`, taking the UUID from `blkid` instead of typing it:

```
echo "UUID=$(sudo blkid -s UUID -o value /dev/data/notes) /var/lib/notes-api ext4 defaults,nofail,x-
    ↪ systemd.device-timeout=10s 0 2" | sudo tee -a /etc/fstab
```

```
UUID=b59b22a0-97c1-4df2-9b8f-d1ffc46f4724 /var/lib/notes-api ext4 defaults,nofail,x-systemd.device-
    ↪ timeout=10s 0 2
```

The six fields are the device, the mount point, the filesystem type, the options, the dump flag (always 0 today) and the `fsck` order (2 means "check after the root filesystem").

The two options I added are what keep a missing disk from ruining your day:

- `nofail` means boot continues even if this disk isn't there. Without it, a detached volume drops the server into emergency mode, on a machine you can only reach over SSH.
- `x-systemd.device-timeout=10s` means systemd waits 10 seconds for the disk instead of the default 90.

Before trusting the file, I check it:

```
sudo findmnt --verify
```

```
0 parse errors, 0 errors, 1 warning
   [W] your fstab has been modified, but systemd still uses the old version;
       use 'systemctl daemon-reload' to reload
```

No errors. The warning is useful: systemd turns `fstab` into mount units, and it hasn't read the new line yet. After `daemon-reload` I can mount by mount point alone, which proves the line works:

```
sudo systemctl daemon-reload && sudo mount /var/lib/notes-api && findmnt /var/lib/notes-api
```

```
TARGET             SOURCE                  FSTYPE OPTIONS
/var/lib/notes-api /dev/mapper/data-notes ext4   rw,relatime
```

Then I started the API and immediately asked it for the notes:

```
sudo systemctl start notes-api && curl -s http://127.0.0.1:3000/notes
```

Nothing came back, and curl exited with code 7, "couldn't connect". For a second I thought the move had broken something. The journal said otherwise:

```
systemctl is-active notes-api; journalctl -u notes-api --no-pager -n 4 -o cat
```

```
active
notes-api.service: Deactivated successfully.
Stopped notes-api.service - Notes API.
Started notes-api.service - Notes API.
notes-api listening on http://127.0.0.1:3000
```

`systemctl start` returned as soon as Node was launched, and curl got there before Node opened the port. That's what `Type=simple` means: systemd doesn't know when the app is ready. Running curl again gave the note back:

```
curl -s http://127.0.0.1:3000/notes
```

```
[{"id":1,"text":"buy milk","createdAt":"2026-11-28T16:21:35.718Z"}]
```

The only real test of `fstab` is a reboot. I rebooted and checked the mount and the API as soon as I could log back in:

```
uptime; findmnt /var/lib/notes-api; curl -s http://127.0.0.1:3000/notes
```

```
 16:38:56 up 0 min,  1 user,  load average: 0.77, 0.23, 0.08
TARGET             SOURCE                  FSTYPE OPTIONS
/var/lib/notes-api /dev/mapper/data-notes ext4   rw,relatime
[{"id":1,"text":"buy milk","createdAt":"2026-11-28T16:21:35.718Z"}]
```

Up for less than a minute, volume mounted, notes there. One thing to know: the old copy of the files is still on the root disk, hidden under the mount point. It stays there, using space, until you unmount the volume and delete it.

Grow it while it's running

This is where LVM pays off. The volume group still has 5 GB free, so I gave 2 GB of it to the notes volume, with the API running:

```
sudo lvextend -r -L +2G data/notes
```

```
  Size of logical volume data/notes changed from 5.00 GiB (1280 extents) to 7.00 GiB (1792 extents).
  Logical volume data/notes successfully resized.
resize2fs 1.47.0 (5-Feb-2023)
Filesystem at /dev/mapper/data-notes is mounted on /var/lib/notes-api; on-line resizing required
old_desc_blocks = 1, new_desc_blocks = 1
The filesystem on /dev/mapper/data-notes is now 1835008 (4k) blocks long.
```

`-r` resizes the filesystem in the same command. Without it you get a bigger volume with the same small filesystem on it, and `df` doesn't change.

```
df -h /var/lib/notes-api
```

```
Filesystem              Size  Used Avail Use% Mounted on
/dev/mapper/data-notes  6.9G   28K  6.5G   1% /var/lib/notes-api
```

When the volume group itself is full, you grow the disk underneath. On DigitalOcean that's one command from my Mac:

```
doctl compute volume-action resize deddea78-eb49-11f1-b370-0afd82f81f7a --size 15 --region fra1 --wait
```

The kernel noticed the new size right away:

```
lsblk -o NAME,SIZE /dev/sda
```

```
NAME         SIZE
sda           15G
+-data-notes   7G
```

But LVM didn't. `pvresize` tells the physical volume to use the whole disk:

```
sudo pvresize /dev/sda && sudo vgs
```

```
  Physical volume "/dev/sda" changed
  1 physical volume(s) resized or updated / 0 physical volume(s) not resized
  VG   #PV #LV #SN Attr   VSize   VFree
  data   1   1   0 wz--n- <15.00g <8.00g
```

And this time I gave the logical volume all the free space:

```
sudo lvextend -r -l +100%FREE data/notes
```

```
...
old_desc_blocks = 1, new_desc_blocks = 2
The filesystem on /dev/mapper/data-notes is now 3931136 (4k) blocks long.
```

```
df -h /var/lib/notes-api; sudo lvs
```

```
Filesystem              Size  Used Avail Use% Mounted on
/dev/mapper/data-notes   15G   28K   14G   1% /var/lib/notes-api
  LV    VG   Attr       LSize   Pool Origin Data%  Meta%  Move Log Cpy%Sync Convert
  notes data -wi-ao---- <15.00g
```

From 5 GB to 15 GB, without stopping the API once. Since there was no partition table on the disk, there was no partition to grow, which is the step that usually trips people up.

Growing is easy. Shrinking is not: ext4 can only shrink while unmounted, and a mistake in the order of the steps (shrink the volume before the filesystem) destroys

data. I treat shrinking as a migration: create a smaller volume, copy the data, switch the mount.

Snapshots need free space

Giving the volume `100%FREE` had a cost I noticed right after: LVM snapshots need free space in the volume group, and I had none left.

An LVM **snapshot** freezes a logical volume at one moment. It starts empty, and every time a block changes on the original, LVM first copies the old block into the snapshot. So the snapshot only needs room for what changes while it exists.

I grew the DigitalOcean volume once more, to 20 GB, and ran `pvresize` again:

```
sudo vgs
```

```
  VG   #PV #LV #SN Attr   VSize   VFree
  data   1   1   0 wz--n- <20.00g 5.00g
```

Then a 1 GB snapshot of the notes volume:

```
sudo lvcreate -s -L 1G -n notes-snap data/notes
```

```
  Logical volume "notes-snap" created.
```

```
sudo lvs
```

```
  LV         VG   Attr       LSize   Pool Origin Data%  Meta%  Move Log Cpy%Sync Convert
  notes      data owi-aos--- <15.00g
  notes-snap data swi-a-s---   1.00g      notes  0.01
```

`Data%` is how full the snapshot is. If it reaches 100%, the snapshot becomes invalid, so size it for the changes you expect while it exists.

I mounted it read-only to look inside:

```
sudo mkdir -p /mnt/snap && sudo mount -o ro /dev/data/notes-snap /mnt/snap && sudo ls -l /mnt/snap
```

```
total 20
drwx------ 2 root  root  16384 Nov 28 16:37 lost+found
-rw-rw-r-- 1 notes notes    92 Nov 28 16:21 notes.json
```

Then I added a note through the API, and counted the notes in the snapshot and in the live file:

```
curl -s -X POST http://127.0.0.1:3000/notes -d '{"text":"call the dentist"}' >/dev/null
sudo cat /mnt/snap/notes.json | grep -c '"id"'
sudo grep -c '"id"' /var/lib/notes-api/notes.json
```

```
1
2
```

The snapshot still had one note, the live data two. That frozen view is perfect for taking a consistent backup of a busy filesystem: snapshot, back up the snapshot, delete it.

```
sudo umount /mnt/snap && sudo lvremove -y data/notes-snap
```

```
  Logical volume "notes-snap" successfully removed.
```

A snapshot is not a backup, though. It lives on the same disk as the original, and if the disk dies, both go.

When the disk is full

"No space left on device" has more than one cause, and `df -h` only shows one of them. I created a small 200 MB test volume to break things on purpose, with ext4 limited to about 1,000 **inodes**. Every file and folder uses one inode, whatever its size:

```
sudo lvcreate -L 200M -n scratch data
sudo mkfs.ext4 -q -N 1000 /dev/data/scratch
sudo mkdir -p /mnt/scratch && sudo mount /dev/data/scratch /mnt/scratch && sudo chown flavio: /mnt/
    ↪ scratch
df -h /mnt/scratch && df -i /mnt/scratch
```

```
Filesystem                Size  Used Avail Use% Mounted on
/dev/mapper/data-scratch  184M   24K  170M   1% /mnt/scratch
Filesystem               Inodes IUsed IFree IUse% Mounted on
/dev/mapper/data-scratch   1024    11  1013    2% /mnt/scratch
```

Then I tried to create 2,000 empty files:

```
for i in $(seq 1 2000); do touch /mnt/scratch/f$i 2>/dev/null || { echo "failed at file $i"; break; };
    ↪ done; touch /mnt/scratch/one-more
```

```
failed at file 1014
touch: cannot touch '/mnt/scratch/one-more': No space left on device
```

```
df -h /mnt/scratch; df -i /mnt/scratch
```

```
Filesystem                Size  Used Avail Use% Mounted on
/dev/mapper/data-scratch  184M   40K  170M   1% /mnt/scratch
Filesystem               Inodes IUsed IFree IUse% Mounted on
/dev/mapper/data-scratch   1024  1024     0  100% /mnt/scratch
```

”No space left on device” with the disk 1% full. `df -h` says everything is fine, `df -i` shows the inodes are gone. On real servers this happens with millions of tiny session files or cache entries, and that's why I always run both commands.

The second case is a file that's gone but still takes space. I recreated the scratch volume with normal inode settings, filled it with one big file, and started a process that keeps the file open (like an app writing its log):

```
dd if=/dev/zero of=/mnt/scratch/big.log bs=1M count=500
setsid python3 -c "import time; f=open('/mnt/scratch/big.log'); time.sleep(600)" </dev/null >/dev/null
    ↪ 2>&1 &
```

`dd` stopped at 157 MiB when the disk was full. Then I deleted the file:

```
rm /mnt/scratch/big.log && ls /mnt/scratch && df -h /mnt/scratch
```

```
lost+found
Filesystem                Size  Used Avail Use% Mounted on
/dev/mapper/data-scratch  172M  158M     0 100% /mnt/scratch
```

The file is gone from the folder, and the disk is still 100% full. `du`, which adds up the files it can see, disagrees with `df`:

```
sudo du -sh /mnt/scratch
```

```
20K /mnt/scratch
```

When `df` and `du` disagree like this, look for deleted files that are still open. `rm` only removes the name. The data stays on disk until the last process holding it closes it. `lsof +L1` lists open files with zero links, which means deleted:

```
sudo lsof +L1 /mnt/scratch
```

```
COMMAND  PID   USER   FD   TYPE DEVICE  SIZE/OFF NLINK NODE NAME
python3 4069 flavio    3r   REG  252,1 164892672     0   12 /mnt/scratch/big.log (deleted)
```

The same information is in `/proc`, under the process's file descriptors:

```
ls -l /proc/4069/fd/3
```

```
lr-x------ 1 flavio flavio 64 Nov 28 16:45 /proc/4069/fd/3 -> /mnt/scratch/big.log (deleted)
```

Ending the process frees the space:

```
kill 4069; sleep 1; df -h /mnt/scratch
```

```
Filesystem                Size  Used Avail Use% Mounted on
/dev/mapper/data-scratch  172M   24K  158M   1% /mnt/scratch
```

On a real server the process is usually your app or a web server, and someone deleted its huge log file to make room. Restarting or reloading the service makes it reopen its files and releases the space. Next time, truncate the log instead of deleting it, or let logrotate handle it.

Then I removed the test volume:

```
sudo umount /mnt/scratch && sudo lvremove -y data/scratch
```

```
  Logical volume "scratch" successfully removed.
```

Backups you can restore

The notes are now on their own volume. That helps with space, and it does nothing for safety: a bad deploy, a typo in a command, a compromised server or a deleted Droplet takes the data with it.

What matters isn't having a backup. It's being able to restore, so this chapter ends with a restore drill.

Two numbers first: RPO and RTO

Before any tool, I decide two things.

The **Recovery Point Objective** (RPO) is how much recent data I can afford to lose. With a backup every night, a disk that dies at 11 PM loses almost a day of notes.

The **Recovery Time Objective** (RTO) is how long recovery may take, from "it's broken" to "it works again". That includes finding the password, getting the data back and checking it, not just running the restore command.

For the notes API I picked an RPO of 24 hours and an RTO of one hour. A nightly backup meets the first, and a written, rehearsed restore meets the second. If the notes were paid orders, 24 hours would be far too much, and a nightly backup would never get there. That's when you look at database replication or continuous archiving.

What has to survive

On this server, that's:

- `/var/lib/notes-api`, the data
- `/etc`, the configuration: the Nginx site, the systemd units, `/etc/notes-api/env`, the Let's Encrypt certificates and account
- the restic password, stored *outside* the server

The application code isn't on the list, because it lives on my laptop and in Git, and I can deploy it again. Same for the packages, which `apt` reinstalls.

The password deserves its own line. Backups are encrypted, so without the password the backup is useless. If the only copy of the password is on the server being backed up, losing the server loses both. It belongs in a password manager, and that password manager must not depend on this repository for its own backup.

3-2-1, and what I tested

The usual rule is **3-2-1**: three copies of the data, on two different kinds of storage, with one copy off-site.

For this book I tested a restic repository on the server itself, in `/srv/restic` on the main disk. Since the data now lives on the separate volume, that copy survives a broken or deleted volume, and a bad command that wipes the notes. It does not survive losing the server, and that's what the second and third copies are for. restic stores repositories on S3-compatible object storage (Backblaze B2, Cloudflare R2, DigitalOcean Spaces, Amazon S3) and on other machines over SFTP, with the same commands we'll use below, and `restic copy` copies snapshots from one repository to another. I didn't run those here, because they need storage and credentials on an account, and I didn't want to put those on a test server. On a real server, the off-site copy is the one I'd set up first.

One more rule for the off-site copy: the server should be able to *add* backups but not *delete* them. Otherwise anyone who gets root on the server can wipe the history too, which is exactly what ransomware does first. Object storage with versioning or object lock, or a separate key for deletions that only you hold, takes care of that.

A restic repository

restic makes encrypted, deduplicated snapshots. Each backup only stores the parts of files that changed, and every snapshot can still be restored on its own. It's in the Ubuntu archive:

```
sudo apt-get install -y restic
restic version
```

```
restic 0.16.4 compiled with go1.22.2 on linux/amd64
```

The repository password goes in a file only root can read. I generate a random one:

```
sudo install -d -m 700 /etc/restic
openssl rand -base64 32 | sudo tee /etc/restic/password >/dev/null
sudo chmod 600 /etc/restic/password
sudo ls -l /etc/restic
```

```
total 4
-rw------- 1 root root 45 Nov 28 16:46 password
```

Before creating the repository, copy that password somewhere outside the server, like a password manager. That's the step people skip.

```
sudo restic init --repo /srv/restic --password-file /etc/restic/password
```

```
created restic repository b1fefce0ff at /srv/restic

Please note that knowledge of your password is required to access
the repository. Losing your password means that your data is
irrecoverably lost.
```

The first backups

The first backup reads everything:

```
sudo restic -r /srv/restic --password-file /etc/restic/password backup /var/lib/notes-api /etc --
    ↪ exclude /etc/restic
```

```
no parent snapshot found, will read all files

Files:         930 new,     0 changed,     0 unmodified
Dirs:          270 new,     0 changed,     0 unmodified
Added to the repository: 2.850 MiB (921.580 KiB stored)

processed 930 files, 2.324 MiB in 0:00
snapshot 5fe8a13b saved
```

I exclude `/etc/restic`, so the key to the repository isn't stored inside the repository. Notice that `/etc` also contains `/etc/shadow` and the app's env file. The repository is encrypted, but whoever has the password can read them, which is one more reason to guard it.

Then I added a third note and ran the same backup again:

```
curl -s -X POST http://127.0.0.1:3000/notes -d '{"text":"renew the domain"}'
```

```
{"id":3,"text":"renew the domain","createdAt":"2026-11-28T16:47:04.727Z"}
```

```
using parent snapshot 5fe8a13b
Files:           0 new,     1 changed,   929 unmodified
Added to the repository: 2.300 KiB (1.533 KiB stored)
snapshot cbd2d8c8 saved
```

One changed file, 2.3 KiB added. That's deduplication at work, and it's why I can afford a snapshot every night.

```
sudo restic -r /srv/restic --password-file /etc/restic/password snapshots --compact
```

```
ID        Time                 Host                 Tags
----------------------------------------------------------
5fe8a13b  2026-11-28 16:47:01  handbook-linux-test
cbd2d8c8  2026-11-28 16:47:04  handbook-linux-test
----------------------------------------------------------
2 snapshots
```

Without `--compact` the list also shows the paths of each snapshot, `/etc` and `/var/lib/notes-api`, on more lines.

Typing the repository and the password file every time gets old. restic reads them from environment variables, and `sudo -E` passes my environment to the command:

```
export RESTIC_REPOSITORY=/srv/restic RESTIC_PASSWORD_FILE=/etc/restic/password
```

Look inside a backup

`restic diff` shows what changed between two snapshots:

```
sudo -E restic diff 5fe8a13b cbd2d8c8
```

```
comparing snapshot 5fe8a13b to cbd2d8c8:

M    /var/lib/notes-api/notes.json

Files:           0 new,     0 removed,     1 changed
Dirs:            0 new,     0 removed
Others:          0 new,     0 removed
Data Blobs:      1 new,     1 removed
Tree Blobs:      4 new,     4 removed
  Added:   2.300 KiB
  Removed: 2.204 KiB
```

And `restic ls` lists the files in a snapshot, here the latest one, filtered to the data folder:

```
sudo -E restic ls latest /var/lib/notes-api
```

```
snapshot cbd2d8c8 of [/var/lib/notes-api /etc] filtered by [/var/lib/notes-api] at 2026-11-28
    ↪ 16:47:04.83581622 +0000 UTC):
/var/lib/notes-api
/var/lib/notes-api/lost+found
/var/lib/notes-api/notes.json
```

Restore one file

Most restores aren't disasters. Someone deleted a file, or broke a config, and wants yesterday's version back. I restore into a separate folder, so I can look at the file before putting it anywhere:

```
sudo -E restic restore latest --target /tmp/restore --include /var/lib/notes-api/notes.json
sudo cat /tmp/restore/var/lib/notes-api/notes.json | grep text
```

```
restoring <Snapshot cbd2d8c8 of [/var/lib/notes-api /etc] at 2026-11-28 16:47:04.83581622 +0000 UTC by
    ↪ root@handbook-linux-test> to /tmp/restore
Summary: Restored 4 / 1 files/dirs (288 B / 288 B) in 0:00
    "text": "buy milk",
    "text": "call the dentist",
    "text": "renew the domain",
```

The file comes back with its full path under the target folder.

Check the repository

`restic check` verifies the repository's structure:

```
sudo -E restic check
```

```
...
[0:00] 100.00%  2 / 2 snapshots

no errors were found
```

By default it checks the metadata, not every byte of data. `restic check --read-data-subset=10%` reads a sample of the actual data too, and I'd run that weekly on a real server.

`restic stats` shows what the repository costs:

```
sudo -E restic stats --mode raw-data
```

```
...
              Total Size:  878.910 KiB
    Compression Progress:  100.00%
       Compression Ratio:  3.36x
Compression Space Saving:  70.26%
```

Mostly text from `/etc`, so it compresses well.

Retention

Without a policy, snapshots pile up forever. `restic forget` removes the ones a policy doesn't keep, and `--prune` deletes the data nothing refers to anymore:

```
sudo -E restic forget --keep-daily 7 --keep-weekly 4 --keep-monthly 6 --prune
```

That keeps the last snapshot of each of the last 7 days, of the last 4 weeks and of the last 6 months. Seventeen snapshots at most, going back half a year.

Be careful with retention. `forget --prune` really deletes, and a policy that's too short means the corruption you discover in three weeks is already in every snapshot you kept.

Run it every night

The backup runs from a systemd service, `/etc/systemd/system/restic-backup.service`:

```
[Unit]
Description=Back up the notes data and /etc with restic
After=network-online.target

[Service]
Type=oneshot
Environment=RESTIC_REPOSITORY=/srv/restic RESTIC_PASSWORD_FILE=/etc/restic/password
ExecStart=/usr/bin/restic backup /var/lib/notes-api /etc --exclude /etc/restic
ExecStartPost=/usr/bin/restic forget --keep-daily 7 --keep-weekly 4 --keep-monthly 6 --prune
```

`ExecStartPost` runs the retention only when the backup succeeded. And a timer, `/etc/systemd/system/restic-backup.timer`:

```
[Unit]
Description=Back up every night

[Timer]
OnCalendar=*-*-* 02:30:00
RandomizedDelaySec=20m
Persistent=true

[Install]
WantedBy=timers.target
```

`Persistent=true` runs a missed backup at the next boot, if the server was off at 02:30. We saw how timers work in the timers chapter.

```
sudo cp restic-backup.service restic-backup.timer /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable --now restic-backup.timer
systemctl list-timers restic-backup.timer --no-pager
```

```
NEXT                         LEFT LAST PASSED UNIT                ACTIVATES
Sun 2026-11-29 02:31:27 UTC   9h -         - restic-backup.timer restic-backup.service
```

I never wait for the first night to find out whether it works. I start the service by hand and read the journal:

```
sudo systemctl start restic-backup.service
journalctl -u restic-backup --no-pager -o cat | grep -E 'snapshot .* saved|Finished|keep'
```

```
snapshot 73ed1248 saved
Applying Policy: keep 7 daily, 4 weekly, 6 monthly snapshots
```

```
keep 2 snapshots:
Finished restic-backup.service - Back up the notes data and /etc with restic.
```

A backup that fails silently is worse than no backup, because you believe you have one. In the monitoring chapter we attach an alert to failing services with `OnFailure=`, and this service gets one too.

The restore drill

Now the real test. I simulated the most common disaster, the data file gone, with the service running normally:

```
sudo systemctl stop notes-api && sudo rm /var/lib/notes-api/notes.json && sudo systemctl start notes-
    ↪ api
sleep 1 && curl -s http://127.0.0.1:3000/notes
```

```
[]
```

Look at that answer. A `200` with an empty list. The API is up, `/health` says `ok`, and every note is gone. Nothing in the logs looks wrong, so the monitoring chapter checks more than "does it answer".

The restore: stop the service so nothing writes, restore the file into its original place, start again:

```
sudo systemctl stop notes-api
sudo -E restic restore latest --target / --include /var/lib/notes-api/notes.json
sudo ls -l /var/lib/notes-api/notes.json
sudo systemctl start notes-api
sleep 1 && curl -s http://127.0.0.1:3000/notes | grep -o '"text":"[^"]*"'
```

```
restoring <Snapshot 73ed1248 of [/var/lib/notes-api /etc] at 2026-11-28 16:47:57.728650304 +0000 UTC by
    ↪ root@handbook-linux-test> to /
Summary: Restored 4 / 1 files/dirs (288 B / 288 B) in 0:00
-rw-r----- 1 notes notes 288 Nov 28 16:47 /var/lib/notes-api/notes.json
"text":"buy milk"
"text":"call the dentist"
"text":"renew the domain"
```

The notes are back, and the file belongs to `notes` again. restic restores ownership and permissions when it runs as root, so the service could read it right away.

The whole drill was five commands, well inside the one-hour RTO. My advice is to write those steps down in a runbook next to the server's other notes, and repeat the drill every few months, and after any change to the backup setup. The bigger drill, restoring everything on a fresh server, is the one that proves the off-site copy and the password are really where you think.

Consistent backups

A backup copies files one at a time. If the app writes in the middle, the backup can catch a half-written file.

The notes API avoids this by design: it writes the new data to `notes.json.tmp` and then renames it over `notes.json`. A rename is atomic, so any reader, restic included, sees either the old file or the new one, never half of each.

Databases are different. Copying the files of a running PostgreSQL or SQLite database can give you something that won't open. Use the database's own tool first, and back up its output:

- PostgreSQL: `pg_dump --format=custom` for each database, plus `pg_dumpall --globals-only` for the roles
- SQLite: the `.backup` command of the `sqlite3` tool

The notes API has no database, but the Vaultwarden container we'll run in chapter 17 keeps everything in SQLite, so I tried it on that file while the container was running:

```
sudo apt install sqlite3
sudo sqlite3 vault/data/db.sqlite3 '.backup /var/backups/vault.db'
sudo sqlite3 /var/backups/vault.db 'PRAGMA integrity_check;'
```

```
ok
```

`.backup` uses SQLite's own locking, so the copy is consistent even while the app writes. Point restic at `/var/backups/vault.db`, not at the live file.

When neither works, stop the service for the few seconds the copy takes, or back up from an LVM snapshot like the one in the previous chapter.

Simpler tools: tar, rsync and checksums

restic isn't always the right tool. For a one-off copy before a risky change, `tar` and a checksum are enough:

```
sudo tar -czf /tmp/etc-nginx.tar.gz -C / etc/nginx
tar -tzf /tmp/etc-nginx.tar.gz | head -4
sha256sum /tmp/etc-nginx.tar.gz
```

```
etc/nginx/
etc/nginx/mime.types
etc/nginx/fastcgi_params
etc/nginx/modules-enabled/
eb1a7db1ab29702ea0f4404239f4304b75452bbb5866f9f5ca9d0cd00dc40c1d  /tmp/etc-nginx.tar.gz
```

`-C /` stores the paths without the leading slash, so extracting doesn't overwrite `/etc/nginx` unless you ask it to. The checksum lets you prove later, after copying the archive somewhere else, that it's the same file.

`rsync` keeps a mirror of a folder:

```
sudo rsync -aHAX --delete --itemize-changes /var/lib/notes-api/ /srv/notes-copy/
```

```
created directory /srv/notes-copy
cd+++++++++ ./
>f+++++++++ notes.json
cd+++++++++ lost+found/
```

`--itemize-changes` prints one line per change (`>f+++++++++` is a new file), so you can see what each run copied. But be careful with `--delete`: a mirror copies deletions too. If the notes file disappears, the next run deletes it from the mirror as well. A mirror protects against a dead disk, and only snapshots like restic's protect against mistakes.

Security, day two

The first hour locked the doors. SSH takes keys only, root can't log in, and UFW drops everything except SSH, HTTP and HTTPS.

Day two is about the people and the programs that are already inside. Who can run what, where the secrets live, what listens where, and how you notice when something changes that you didn't change.

One person, one account

On this server I log in as `flavio`, never as root. That account is in two groups that matter: `sudo`, to run commands as root, and `adm`, to read the system logs without `sudo`:

```
id flavio
```

```
uid=1000(flavio) gid=1000(flavio) groups=1000(flavio),4(adm),27(sudo),100(users)
```

A personal account means every privileged action has a name on it. The journal records each `sudo` with the user who ran it:

```
journalctl -t sudo --no-pager -n 2
```

```
Nov 28 16:22:58 handbook-linux-test sudo[6561]: pam_unix(sudo:session): session opened for user root(
    ↪ uid=0) by flavio(uid=1000)
Nov 28 16:22:58 handbook-linux-test sudo[6561]: pam_unix(sudo:session): session closed for user root
```

If a second person needs access, they get their own account and their own key. A shared `admin` login with a shared key tells you nothing when you read that log, and you can't remove one person without changing the key for everyone.

Service accounts can't log in

The notes API runs as `notes`, a system user with no shell and no password:

```
getent passwd notes
```

```
notes:x:999:988::/var/lib/notes-api:/usr/sbin/nologin
```

That user owns its data directory and nothing else. If someone finds a bug in the API and gets it to run a command, the command runs as `notes`, and `notes` can't read the files that matter:

```
sudo -u notes cat /etc/shadow
```

```
cat: /etc/shadow: Permission denied
```

The systemd sandbox from the services chapter adds more walls on top, like `ProtectSystem=strict` and `ProtectHome=true`. The account is the first wall, and it costs nothing.

Secrets live outside the code

The API token is in `/etc/notes-api/env`, readable by root and by the `notes` group, and by nobody else. The unit loads it with `EnvironmentFile=`.

Notice what `systemctl show` prints for the environment:

```
systemctl show notes-api -p Environment
```

```
Environment=HOST=127.0.0.1 PORT=3000 DATA_DIR=/var/lib/notes-api
```

The values from `Environment=` in the unit file are visible to any user on the machine. The token from the `EnvironmentFile=` isn't there. So the rule I follow is simple: settings go in the unit, secrets go in the file.

Even `ls` on that file fails for my own user, because `flavio` isn't in the `notes` group:

```
ls: cannot access '/etc/notes-api/env': Permission denied
```

That's the permissions doing their job. I read it with `sudo` when I need to.

The secrets themselves don't belong in Git, in the shell history or in a chat with an AI agent. I wrote down how I handle them across my projects in where I keep the secrets of my infrastructure.

Private services stay private

Three things on this server listen only on `127.0.0.1`: the notes API on port 3000, Vaultwarden on 8080 (from the home server chapter), and Caddy's admin API on

2019 when Caddy was running. We checked the API with `ss` in the services chapter. Here are the other two:

```
LISTEN 0      4096       127.0.0.1:8080       0.0.0.0:*    users:(("docker-proxy",pid=9602,fd=7))
LISTEN 0      4096       127.0.0.1:2019      0.0.0.0:*    users:(("caddy",pid=11004,fd=3))
```

Binding to localhost is a second lock behind the firewall. From my Mac, port 443 answered and port 3000 timed out:

```
nc -vz -w 3 203-0-113-42.sslip.io 3000
```

```
nc: connectx to 203-0-113-42.sslip.io port 3000 (tcp) failed: Operation timed out
```

If UFW ever gets disabled by mistake, the API still isn't on the internet. Nginx is the only public door, and it's the one we configured with HTTPS, headers and rate limits.

Caddy's admin API deserves a special mention. It can rewrite the whole configuration with one HTTP request, and it listens on localhost by default. Keep it there.

Who's knocking

I looked at the logs about 35 minutes after the server first booted.

SSH was quiet. I counted the lines that usually mean a failed attempt (invalid users, closed authentications, too many tries) and found three:

```
journalctl -u ssh --since '2026-11-28 16:13' --no-pager | grep -cE 'Invalid user|Connection closed by
    ↪ authenticating user|maximum authentication attempts'
```

```
3
```

Then I counted the distinct source addresses in the whole SSH log, and there was one. Mine. Those three lines were my own root login tests from the hardening chapter.

Nginx was a different story. The access log already had 80 requests, and the most requested paths were these:

```
sudo awk '{print $7}' /var/log/nginx/access.log | sort | uniq -c | sort -rn | head -8
```

```
     26 /
      7 /.well-known/acme-challenge/fImKAGbx_iUmVTGzGbxt1r603h5005Jy6xDybSvL6Q0
      7 /.well-known/acme-challenge/RrYROlS4F1gvo1qXZurgm8R5-r2P63UXtZUWUScu2jk
      4 /favicon.ico
      3 400
      2 /v2/_catalog
      2 /trace.axd
      2 /telescope/requests
```

Some of the 26 requests to `/` are mine. The ACME challenges are Let's Encrypt checking the domain from several places before issuing the certificate. The `400` lines are requests so broken that Nginx couldn't find a path in them.

The last three are scanners:

- `/v2/_catalog` is the API of a Docker registry. An open registry leaks every image you pushed.
- `/trace.axd` is the trace page of ASP.NET apps, which can expose request details.
- `/telescope/requests` is Laravel Telescope, a debug dashboard that people forget to lock down.

None of those exist here, so they got a 404 or a closed connection. But it shows how fast it happens. Within half an hour of getting a public IP, bots were trying debug panels from three different stacks.

`last` lists logins and reboots. Here it only showed reboots:

```
reboot   system boot  6.8.0-151-generi Sat Nov 28 16:38   still running
reboot   system boot  6.8.0-151-generi Sat Nov 28 16:17 - 16:38  (00:20)
reboot   system boot  6.8.0-149-generi Sat Nov 28 16:13 - 16:17  (00:04)
wtmp begins Sat Nov 28 16:13:18 2026
```

My sessions while writing this book ran single commands over SSH, without a terminal, and those don't get a line in `wtmp`. An interactive login does. So `last` is a good place to look for people, and the SSH journal is the complete record.

Noticing what changed

You can't spot an unexpected change if you don't know what's expected. Two checks help on a small server.

The first asks `dpkg` to compare installed files with the checksums in the packages:

```
sudo dpkg --verify nginx openssh-server
```

No output and exit code 0 means every file of those packages matches what Ubuntu shipped. A replaced `sshd` binary would show up here, and so would an edited configuration file. I had edited `/etc/nginx/nginx.conf`, but that file belongs to the `nginx-common` package, so it would show up only when verifying that one.

The second lists the files in `/etc` changed in the last hour:

```
sudo find /etc -newer /etc/hostname -type f -mmin -60 | head -12
```

```
/etc/subuid
/etc/passwd-
/etc/fstab
/etc/hosts
/etc/timezone
```

```
/etc/shadow
/etc/caddy/Caddyfile
/etc/udev/rules.d/99-digitalocean-automount.rules
/etc/nginx/sites-available/notes
/etc/nginx/sites-available/notes.bak
/etc/nginx/sites-available/catch-all
/etc/nginx/nginx.conf
```

The exercise is to give every line an owner. `passwd-`, `shadow` and `subuid` changed when I created the users. `fstab` is the data volume. The Caddy and Nginx files are from the web chapters. `hosts` and `timezone` come from cloud-init, which configures the Droplet at boot.

One line I didn't expect: `/etc/udev/rules.d/99-digitalocean-automount.rules`. DigitalOcean's agent wrote it when I attached the volume. It's harmless, and now I know where it comes from. So the next time that list has a line nobody can explain, it will stand out.

When it might be a compromise

A single odd process can have a boring explanation, so I look for a pattern. These are the signs I'd stop for:

- an account you didn't create
- a key in `authorized_keys` you can't place
- a listener on a port you didn't open
- a process with a name that's almost a system name
- outbound traffic to an address nothing should talk to
- a `sudo` line at an hour nobody was working

The instinct is to fix it right away, delete the key, kill the process, reboot. Don't. A reboot wipes the memory, the running processes and the open connections, which are often the only evidence of what happened, and leaves anything persistent exactly where it was.

Record first, then act. Save the state to files and copy them to another machine:

```
date -u
ps auxf > /tmp/evidence-ps.txt
sudo ss -tnp > /tmp/evidence-net.txt
last -a | head -20
sudo stat /home/flavio/.ssh/authorized_keys
```

On the test server, with nothing wrong, the key file looked like this:

```
  File: /home/flavio/.ssh/authorized_keys
  Size: 101         Blocks: 8          IO Block: 4096   regular file
Access: (0600/-rw-------)  Uid: ( 1000/  flavio)   Gid: ( 1000/  flavio)
Modify: 2026-11-28 16:13:33.257000000 +0000
Change: 2026-11-28 16:18:31.896157705 +0000
```

`Modify` is 16:13, when cloud-init wrote my key at first boot. `Change` is 16:18, when I copied it into `flavio`'s home and changed the owner in chapter 4. Both times have an explanation. A `Modify` time at 3 a.m. on a day nobody touched the server wouldn't.

The modification time from `stat` on a key file tells you when access changed, and the rest of the timeline hangs on that. Check sources the attacker can't edit too, like your provider's activity log and your Git history.

Rebuild from known-good inputs

After someone got root, you can't prove that you found every change they made. Removing the backdoor you saw doesn't mean it was the only one.

So I don't clean a compromised server. I build a new one from inputs the attacker couldn't touch:

```
image:    a fresh Ubuntu 24.04 LTS image from the provider, not a snapshot of the old server
config:   the unit files and Nginx configs from Git, reviewed after the incident
app:      the notes API rebuilt from source
secrets:  all new; the old API token, SSH keys and restic password are burned
data:     restored from a backup that passes restic check, then inspected
```

Everything in this book is written to make that easy. Every unit file and config is short enough to keep in a Git repository, the data has backups we've already restored once, and the restore drill in the backups chapter is the same procedure you'd use here.

Keep the old server isolated (a firewall rule that drops everything, or powered off) instead of deleting it right away. You may need its disk to understand what happened. And don't copy "just one file" from it to the new server. A binary or a config that's annoying to regenerate is exactly where something can hide.

Monitoring and alerts

A server that breaks at 3 AM and tells nobody isn't monitored, it's just running. The minimum I want is one check that tests the service like a visitor would, and one way for its failure to reach me.

Check from the outside

The health check is a oneshot service that asks the public URL for `/health`, through DNS, Nginx and TLS, the same path a real visitor takes:

```
[Unit]
Description=Check that the notes API answers from outside
OnFailure=alert@%n.service

[Service]
Type=oneshot
ExecStart=/usr/bin/curl --fail --silent --show-error --max-time 5 https://203-0-113-42.sslip.io/health
```

`--fail` makes curl exit with an error on any HTTP status of 400 or more. `--max-time 5` turns a hang into a failure instead of a check that never ends. And because it's HTTPS, an expired or broken certificate fails the check too.

A timer runs it every minute:

```
[Unit]
Description=Check the notes API every minute

[Timer]
OnCalendar=minutely

[Install]
WantedBy=timers.target
```

`OnFailure=` is the interesting part. When the check fails, systemd starts `alert@notes-healthcheck.service.service`, a template unit that gets the failed unit's name as `%i`:

```
[Unit]
Description=Alert about %i

[Service]
Type=oneshot
ExecStart=/usr/bin/logger -p user.crit -t alert "%i failed on %H"
```

I installed the three files and turned on the timer:

```
sudo cp notes-healthcheck.service notes-healthcheck.timer alert@.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable --now notes-healthcheck.timer
```

A first run, by hand, while everything works:

```
sudo systemctl start notes-healthcheck.service
journalctl -u notes-healthcheck --no-pager -o cat -n 3
```

```
{"ok":true}
notes-healthcheck.service: Deactivated successfully.
Finished notes-healthcheck.service - Check that the notes API answers from outside.
```

Then I stopped the API and ran the check again:

```
sudo systemctl stop notes-api
sudo systemctl start notes-healthcheck.service
```

```
Job for notes-healthcheck.service failed because the control process exited with error code.
```

The journal tells the whole story, from curl's error to the alert:

```
curl: (22) The requested URL returned error: 502
notes-healthcheck.service: Main process exited, code=exited, status=22/n/a
notes-healthcheck.service: Failed with result 'exit-code'.
Failed to start notes-healthcheck.service - Check that the notes API answers from outside.
notes-healthcheck.service: Triggering OnFailure= dependencies.
```

Nginx was up, the API behind it wasn't, so Nginx answered 502 Bad Gateway, and curl exited with code 22. Then the alert unit ran:

```
journalctl -t alert --no-pager
```

```
Nov 28 16:49:12 handbook-linux-test alert[6640]: notes-healthcheck.service failed on handbook-linux-
    ↪ test
```

Make the alert reach you

That last step needs one more piece. A line in the journal of the server that's in trouble is an alert nobody reads. If the whole server goes down, the check goes down with it.

So the `logger` line is a placeholder for something that leaves the machine. The template unit is the one place to change: replace its `ExecStart` with a command that sends an email, or a `curl` to a push notification service you already have on your phone. Every unit that has `OnFailure=alert@%n.service` then uses it.

For "the whole server is gone", the check has to run somewhere else. An uptime monitoring service, or the same curl on a different machine, like a home server or a tiny VPS in another region. That's the case that matters most, the one where the server can't tell you anything.

Check the data, not only that it's alive

In the backups chapter I deleted `notes.json` to rehearse a restore. The API kept running, `/health` still said `{"ok":true}`, and `GET /notes` returned `[]` with a 200.

Every check in this chapter would have passed. The service was up, and empty.

So a health check should test what users care about. For the notes API that could be "`/notes` returns at least one note", or a `/health` endpoint that reads the data file and fails if it can't. A liveness check tells you the process is there. It doesn't tell you the process is useful.

Capacity over time

`sysstat` was already installed on the Droplet, and its timer was already collecting CPU, memory and disk numbers every 10 minutes when I looked at `systemctl list-timers`. `sar` reads them back.

CPU for today:

```
sar -u
```

```
16:30:04        all      9.07      1.23      4.16      0.90      0.86     83.79
Average:        all      9.07      1.23      4.16      0.90      0.86     83.79

16:38:30     LINUX RESTART  (1 CPU)
```

`LINUX RESTART` marks the reboot I did for the data volume. On a long-running server, that line in the middle of the day is a question worth asking: who rebooted it, and why?

`sar -r` does the same for memory, and `df` covers the disks. With `--output` you get only the columns you want, which makes it easy to put in a script that complains above a threshold:

```
df -h / /var/lib/notes-api --output=target,pcent,avail
```

```
Mounted on         Use% Avail
/                   13%   21G
/var/lib/notes-api   1%   14G
```

These numbers are for trends, not alerts. A disk that fills up over three months, or memory that grows a bit after every deploy, shows up in them weeks before it becomes an outage.

A troubleshooting method

When something breaks, the temptation is to start changing things until it works. Sometimes that's fine. Often it erases the evidence, and the same problem comes back next week.

What works for me is a loop: define the symptom, keep the state, build a timeline, test one idea at a time, fix, then verify from where the user is.

1. Define the symptom and its scope.
2. Preserve the state before changing anything.
3. Build a timeline.
4. Test one hypothesis at a time. If it's wrong, go back to the timeline.
5. Fix, then verify from the outside.

Define the symptom and the scope

"The site is broken" isn't a symptom. "`https://203-0-113-42.sslip.io/notes` returns 502 since 16:49, `/health` too, the static default server is fine" is one.

Write down what fails, since when, for whom, and what still works. The last part narrows things down the most. If Nginx answers but the API doesn't, you can stop looking at DNS and TLS.

Preserve state before changing it

A restart can bring the service back and erase what explained the failure: the stuck process, its open files, the memory numbers. Two minutes of capture first:

```
d=/var/tmp/incident-$(date -u +%Y%m%dT%H%M%S)
mkdir "$d" && cd "$d"
date -u > time.txt
uptime > uptime.txt
systemctl --failed > failed-units.txt
ps auxww --sort=-%cpu > processes.txt
free -h > memory.txt
df -h > disk.txt
sudo ss -tnp state established > sockets.txt
journalctl -u notes-api --since "-30 min" --no-pager > app-log.txt
sudo dmesg --ctime | tail -100 > kernel.txt
```

The first time I ran it, I had `dmesg` and `ss` without `sudo`, and one line of the output was easy to miss:

```
dmesg: read kernel buffer failed: Operation not permitted
```

Ubuntu only lets root read the kernel log, so `kernel.txt` came out empty (0 bytes in `ls -l`), and `ss` without `sudo` doesn't show which process owns each connection. Those are two of the files you want most after an OOM kill or a network problem. So check the folder before you restart anything:

```
ls -l "$d"
```

Then restart if you need to. You're allowed to restart, just not first.

Build a timeline

The journal is the backbone of the timeline because it has everything with timestamps: systemd, the kernel, `sudo`, your app. Boots are the first landmarks:

```
journalctl --list-boots --no-pager
```

```
IDX BOOT ID                          FIRST ENTRY                 LAST ENTRY
 -1 1710cae489654efb8bb97805e92e8cb6 Sat 2026-11-28 16:13:19 UTC Sat 2026-11-28 16:17:24 UTC
  0 0bbfda88b87b44eab59ba89c52b19093 Sat 2026-11-28 16:17:32 UTC Sat 2026-11-28 16:18:15 UTC
```

Then the deploys, the config changes, the alerts. Put them in order and mark each line as a fact or a guess. "The API was killed at 16:49:34" is a fact from the log. "It ran out of memory because of the traffic" is a guess until you check.

One hypothesis at a time

Pick the most likely explanation, find a test that would prove it wrong, run it. If you change two things and the problem goes away, you don't know which one fixed it, and you can't be sure you won't need the other one later.

Here are the cases I hit on the test server while writing this book. Each one has the command that found it.

A service that won't start

We saw this one in the services chapter. `notes-api` failed five times in a row with `Start request repeated too quickly`, while `curl` to port 3000 worked fine.

The journal had the reason, `EADDRINUSE`: something else was already on port 3000. `ss -lntp` showed a process called `MainThread`, the copy of the API I had started by hand and thought I'd stopped. `pkill node` hadn't matched it, because Node 24 names its main thread `MainThread`.

The lesson that stays: read the service's own log before anything else, and find the owner of a port with `ss -lntp`, by PID, not by guessing the process name.

Killed by the OOM killer

To see what a memory problem looks like, I gave the API a memory limit far too small, at runtime, without touching the unit file:

```
sudo systemctl set-property --runtime notes-api MemoryMax=12M
```

Then I sent it 30 requests. A few seconds later:

```
systemctl show notes-api -p NRestarts -p Result
```

```
Result=oom-kill
NRestarts=2
```

`Result=oom-kill` is systemd telling you exactly what happened. The unit's journal says it in words:

```
notes-api.service: A process of this unit has been killed by the OOM killer.
notes-api.service: Main process exited, code=killed, status=9/KILL
notes-api.service: Failed with result 'oom-kill'.
notes-api.service: Scheduled restart job, restart counter is at 1.
```

And the kernel log has the other side:

```
sudo journalctl -k --no-pager --since '2 min ago' | grep -iE 'oom|killed process'
```

```
Nov 28 16:49:34 handbook-linux-test kernel: systemd invoked oom-killer: gfp_mask=0xcc0(GFP_KERNEL),
    ↪ order=0, oom_score_adj=0
```

Notice that `Restart=on-failure` kept restarting it, so from the outside the API looked flaky rather than down. An OOM kill hides well behind a restart policy. `NRestarts` going up is the clue.

`--runtime` puts the setting in `/run`, so it would be gone at the next reboot anyway. `systemctl revert` removed it right away:

```
sudo systemctl revert notes-api
```

```
Removed "/run/systemd/system.control/notes-api.service.d/50-MemoryMax.conf".
Removed "/run/systemd/system.control/notes-api.service.d".
```

Back to `MemoryMax=268435456`, the 256 MB from the unit file.

The CPU is busy

Load average is the first number people look at, and the easiest to misread. It counts processes running or waiting to run. Whether a number is high depends on the CPU count, and this Droplet has one.

I ran `stress-ng` with two CPU workers for 40 seconds. After 25 seconds:

```
 16:50:37 up 12 min,  1 user,  load average: 1.27, 0.50, 0.22
```

1.27 on one CPU means there's more work than the CPU can do. The 1-minute average was still climbing, and the 15-minute one hadn't noticed yet.

`top` sorted by CPU showed who:

```
%Cpu(s): 91.7 us,  8.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
...
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   7729 flavio    20   0   48500   5780   2664 R  54.5   0.6   0:13.51 stress-+
   7730 flavio    20   0   48500   5780   2664 R  54.5   0.6   0:13.51 stress-+
```

Two workers sharing one CPU, each getting half. `0.0 id` means no idle time at all, and `0.0 wa` means nothing was waiting on the disk, so this was pure computation.

The pressure file tells you how much time tasks spent waiting for the CPU:

```
cat /proc/pressure/cpu
```

```
some avg10=95.03 avg60=43.03 avg300=12.88 total=59511196
full avg10=0.00 avg60=0.00 avg300=0.00 total=0
```

`some avg10=95.03` means that in the last 10 seconds, at least one task was waiting for the CPU 95% of the time. That's a number I find easier to reason about than the load average.

`vmstat` shows the same picture, second by second:

```
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st gu
 2  0      0 194364  16640 613500    0    0     0    76 1018 1031 100  0  0  0  0  0
```

`r` is 2 (two runnable processes for one CPU), `us` is 100. Twenty seconds after stress-ng ended, the 1-minute load was already going down, to 1.11.

If `wa` had been high instead of `us`, I'd look at the disks, and if `si` and `so` (swap in and out) weren't zero, at memory.

A clean exit that nobody restarted

This one surprised me more than the others. I sent the API a `SIGTERM`, the signal `systemctl stop` uses, by hand:

```
sudo kill -TERM $(systemctl show notes-api -p MainPID --value)
```

The app did what it's written to do, it closed the server and exited:

```
SIGTERM received, closing
notes-api.service: Deactivated successfully.
```

And it stayed down:

```
systemctl is-active notes-api
systemctl show notes-api -p NRestarts -p Result
```

```
inactive
Result=success
NRestarts=0
```

`Restart=on-failure` only restarts after a failure. A clean exit with code 0 is a success, so systemd did nothing. Anything that sends the wrong signal to your app (a script, a `kill` typed in the wrong terminal, an app that exits 0 when it gives up) takes it offline with no restart.

The health check from the monitoring chapter is what would catch this, within a minute. You can also use `Restart=always` if your app should never stop unless you stop it with `systemctl`. `systemctl stop` still works with `Restart=always`, because systemd knows you asked for it.

File descriptors

"Too many open files" errors come from a limit per process. Here's how to see the limit and the current count for the API:

```
P=$(systemctl show notes-api -p MainPID --value)
grep 'open files' /proc/$P/limits
sudo ls /proc/$P/fd | wc -l
```

```
Max open files            524288               524288               files
22
```

22 open out of 524288 allowed. That limit comes from systemd's default for services (`LimitNOFILE=524288` in `systemctl show`), and it's far higher than what the shell gives you with `ulimit -n`. When you test an app by hand and it hits the limit, while the service doesn't, that difference is the reason.

If the count keeps growing on a real service, something is opening files or sockets and never closing them. `ls -l /proc/$P/fd` shows what each one points to.

The network, layer by layer

When ”it doesn't connect”, I test from the client side, one layer at a time, starting from the bottom. From my Mac:

```
dig +short 203-0-113-42.sslip.io
```

```
203.0.113.42
```

The name resolves to the right address. Next, the TCP port:

```
nc -vz -w 3 203-0-113-42.sslip.io 443
```

```
Connection to 203-0-113-42.sslip.io port 443 [tcp/https] succeeded!
```

Then TLS and HTTP:

```
curl -sv -o /dev/null https://203-0-113-42.sslip.io/health 2>&1 | grep -E 'Connected|SSL connection|<
    ↪ HTTP'
```

```
* Connected to 203-0-113-42.sslip.io (203.0.113.42) port 443
* SSL connection using TLSv1.3 / AEAD-AES256-GCM-SHA384 / [blank] / UNDEF
< HTTP/2 200
```

The first layer that fails tells you where to look:

- a wrong address points at DNS
- a timeout on the port is a firewall, UFW or your provider's
- a refused connection means nothing is listening
- a TLS error is the certificate or the Nginx config
- a 502 is the app behind Nginx

Time and certificates

TLS, logs, backups and timers all rely on the clock. `timedatectl` says if it's synchronized:

```
               Local time: Sat 2026-11-28 16:51:14 UTC
           Universal time: Sat 2026-11-28 16:51:14 UTC
                 RTC time: Sat 2026-11-28 16:51:14
```

```
               Time zone: Etc/UTC (UTC, +0000)
System clock synchronized: yes
              NTP service: active
          RTC in local TZ: no
```

`System clock synchronized: yes` is what you want. Droplets start on UTC and I leave them there, so logs from different machines line up without doing arithmetic.

For certificates, `openssl s_client` piped to `openssl x509 -noout -dates`, like in the Nginx chapter, gives you `notAfter`. An expired certificate shows up as a failed health check too, because curl verifies it.

Processes stuck in D or Z

Two process states deserve a look when the server feels stuck. `D` is uninterruptible sleep, usually waiting on a disk or a network filesystem that doesn't answer. `Z` is a zombie, a process that exited but whose parent never collected it.

```
ps -eo pid,stat,comm --sort=stat | awk 'NR==1 || $2 ~ /^(D|Z)/'
```

```
    PID STAT COMMAND
```

Only the header, so none here. A few `D` processes that come and go are normal. Many that stay there point at storage.

Verify from the outside

When the fix is in, check from where the users are. "`systemctl status` says active" isn't the end. The SIGTERM case had a clean status and no API.

I run the same layered checks from my Mac, `dig`, `nc`, `curl`, and look at the health check's journal for the next couple of runs.

Write the incident review

Within a few days, while you still remember, write one page. It's for the next person who meets the same problem, often you.

```
# Incident review: notes API down, <date>

**Impact**: what users saw, from when to when
**Detection**: how we found out (alert, user, by chance)

## Timeline (UTC)
- facts from the journal, in order

## Root cause
what made it happen [confirmed / inference]

## Contributing factors
```

```
what made it worse or slower to notice

## What worked

## Follow-ups
- [ ] one change, one owner, one date
```

Keep facts and guesses apart, like in the timeline. And look for the conditions that made the mistake easy, not for who typed the command. The SIGTERM outage, for example, is a missing data check and a restart policy that trusts exit codes, not "someone typed kill".

A home server

Everything so far ran on a VPS. A home server is the same Ubuntu with three differences: you own the hardware, it sits behind your home router, and nobody else fixes it when the power goes out.

It's great for storage-heavy things that would be expensive in the cloud, like photos, media, backups, or a build runner for your projects.

Some steps in this chapter only make sense on a machine at home: the router, the LAN address, the power. I couldn't run those on a Droplet, so they're described from the setup I'd follow. The Docker parts I ran on the same test server as the rest of the book.

One job first

A home server earns its place when it runs one service you understand. Pick one job, get it running, backed up and boring, then add the next.

Before looking at hardware, I write the requirement down:

```
service: password manager (Vaultwarden)
users: 2
storage now: under 1 GB
downtime tolerance: one day
network: home LAN and Tailscale
backup target: encrypted offsite copy
```

Each line answers a question the hardware will ask anyway. Two users and a password manager need almost nothing, so an old laptop or a used mini PC is plenty. "Downtime tolerance: one day" says a single cheap machine is fine.

A minimal install and a stable address

Install Ubuntu Server LTS with the minimal options: OpenSSH, no desktop, nothing else. Everything from Part 1 and Part 2 applies, from the personal account to the SSH keys and UFW.

The new part is the address. Your router hands out addresses with DHCP, and they can change after a power cut. Then every bookmark, SSH config and mount that pointed at the server breaks.

The fix is a **DHCP reservation** on the router: you tell it to always give this machine the same address, by its MAC address. First note the current state on the server:

```
ip address
ip route
```

The MAC is the `link/ether` value in `ip address`. In the router's admin page, under DHCP or LAN settings, bind that MAC to an address, like `192.168.1.20`. Reboot the server and check that `ip address` shows the reserved address.

Two more things I'd do before trusting it with data: a small UPS, if the server runs a database, and the BIOS setting that powers the machine back on after a power loss ("Restore on AC Power Loss" or similar). Otherwise a 2-second power cut keeps your server off until you get home.

Run the service with Docker Compose

Most self-hosted apps ship as Docker images, and Compose is the easiest way to keep the configuration in one file. Ubuntu's own packages are enough:

```
sudo apt-get install -y docker.io docker-compose-v2
docker --version
docker compose version
```

```
Docker version 29.1.3, build 29.1.3-0ubuntu3~24.04.2
Docker Compose version 2.40.3+ds1-0ubuntu1~24.04.1
```

I added `flavio` to the `docker` group to run `docker` without `sudo`. Be aware of what that means: the `docker` group can start a container that mounts `/` from the host, so it's root in practice. Give it only to accounts you'd trust with `sudo`.

```
sudo usermod -aG docker flavio
```

The Compose file for Vaultwarden pins a version, publishes the port only on localhost, keeps the data in a folder next to the file, and turns off signups:

```
services:
  vaultwarden:
    image: vaultwarden/server:1.37.4
    restart: unless-stopped
    ports:
      - "127.0.0.1:8080:80"
    volumes:
      - ./data:/data
    environment:
```

```
      SIGNUPS_ALLOWED: "false"
```

```
docker compose up -d
docker compose ps
```

```
NAME                 IMAGE                        COMMAND       SERVICE       CREATED          STATUS
    ↪                            PORTS
vault-vaultwarden-1  vaultwarden/server:1.37.4  "/start.sh"   vaultwarden   10 seconds ago   Up 9
    ↪ seconds (health: starting)   127.0.0.1:8080->80/tcp
```

It answers on localhost:

```
curl -s -o /dev/null -w '%{http_code}\n' http://127.0.0.1:8080/alive
```

```
200
```

The data lives in `./data`, an SQLite database and a key, which is what you back up:

```
-rw-r--r-- 1 root   root   278528 Nov 28 16:53 db.sqlite3
-rw-r--r-- 1 root   root    32768 Nov 28 16:53 db.sqlite3-shm
-rw-r--r-- 1 root   root        0 Nov 28 16:53 db.sqlite3-wal
-rw-r--r-- 1 root   root     1679 Nov 28 16:53 rsa_key.pem
```

A reverse proxy in front (Caddy, from the web chapters) or Tailscale (next chapter) gives you HTTPS. Vaultwarden needs it, because its web vault relies on browser crypto APIs that only work on secure pages.

The Docker port that skips UFW

This is the trap I'd most want you to remember from this chapter.

UFW denies everything incoming except SSH and the web ports. So I published a container on port 8081, on all interfaces this time:

```
docker run -d --rm --name exposed -p 8081:80 nginx:alpine
sudo ufw status | grep -c 8081
```

```
0
```

No UFW rule for 8081. And yet `ss` shows Docker listening on every address:

```
LISTEN 0      4096         0.0.0.0:8081       0.0.0.0:*    users:(("docker-proxy",pid=10046,fd=7))
LISTEN 0      4096            [::]:8081          [::]:*    users:(("docker-proxy",pid=10050,fd=7))
```

From my Mac, over the internet:

```
curl -s -o /dev/null -w "%{http_code}\n" http://203.0.113.42:8081/
```

```
200
```

The firewall said no and the container answered anyway. Docker writes its own iptables rules to forward published ports into containers, and those rules sit in chains that the traffic reaches before UFW's rules. Here's the `DOCKER` chain with the Vaultwarden rule in it:

```
sudo iptables -S DOCKER
```

```
-N DOCKER
-A DOCKER -d 172.18.0.2/32 ! -i br-487a07d16c84 -o br-487a07d16c84 -p tcp -m tcp --dport 80 -j ACCEPT
-A DOCKER ! -i docker0 -o docker0 -j DROP
-A DOCKER ! -i br-487a07d16c84 -o br-487a07d16c84 -j DROP
```

At home, behind a router with no port forwarding, the damage stays on your LAN. On a VPS it's the whole internet.

The fix is the one in the Compose file above: publish on `127.0.0.1:` and put a reverse proxy or Tailscale in front. And check from the outside after every new container, because `ufw status` won't tell you.

Updates with a way back

The rule for home servers: one layer at a time, and know how to go back before you change anything.

For a container, going back means two things: the previous image and the data as it was before the new version touched it. Some apps migrate their database on first start, and the old version may not read it anymore.

So, before updating, I stopped the container and archived the data, with the database closed:

```
docker compose stop
sudo tar -czf ~/vault-data-$(date +%F).tar.gz data
docker compose start
```

```
-rw-r--r-- 1 root root 9.0K Nov 28 16:54 /home/flavio/vault-data-2026-11-28.tar.gz
```

Then I changed the pinned tag to the next release, pulled, and recreated the container:

```
sed -i 's/server:1.37.4/server:1.37.5/' compose.yaml
docker compose pull
docker compose up -d
docker compose ps --format '{{.Image}} {{.Status}}'
```

```
vaultwarden/server:1.37.5 Up 10 seconds (health: starting)
```

Going back is the same with the old tag:

```
sed -i 's/server:1.37.5/server:1.37.4/' compose.yaml
docker compose up -d


vaultwarden/server:1.37.4 Up 8 seconds (health: starting)
```

It was fast because the old image was still on disk:

```
docker image ls vaultwarden/server


IMAGE                         ID             DISK USAGE   CONTENT SIZE   EXTRA
vaultwarden/server:1.37.4     efb3cde96201         402MB         90.4MB   U
vaultwarden/server:1.37.5     4fe17f415f90         416MB         92.8MB
```

Don't run `docker image prune` right after an update. Keep the previous image until the new one has run for a few days. And if the release notes mention a migration, the rollback is the old image plus the tarball, not the old image alone.

I don't use `latest` or auto-updating containers on anything I care about. A tag that changes under you is an update you didn't schedule.

The runbook

Six months from now you won't remember the volume's UUID, where the backup password is, or which container holds what. The runbook holds that, outside the machine:

```
inventory          hardware, disks, what's mounted where
network and DNS    reserved address, names, router login location
accounts and keys  where they are (never the secrets themselves)
services           start, stop, check, for each one
backup and restore the commands, and the date of the last drill
update and rollback
power-loss recovery
complete rebuild
```

I write commands in it, with the check that proves they worked. "`cd ~/vault && docker compose up -d`, then `curl http://127.0.0.1:8080/alive` returns 200" is something tired-you can follow.

Then rehearse one section, on purpose, every few months. A runbook nobody has followed has mistakes in it, and you want to find them on a quiet Sunday.

Reaching it from outside your home

Two rules keep a home server safe without opening ports on the router:

1. Private services, the ones only you use, stay on Tailscale. No port forwarding, and it works even when your provider puts you behind CGNAT.

2. Public services go through a tunnel, like Cloudflare Tunnel, which dials out from your server so your home IP never shows up in DNS. Tailscale is the next chapter.

Tailscale

Everything public on our server goes through Nginx on port 443. But some things should never be public: SSH, an admin panel, Vaultwarden, a database UI. Opening a port for them, even with a password in front, means bots will try that password.

Tailscale puts your devices on one private network. Your Mac, the server and your phone each get an address that only the others can reach, without opening any port.

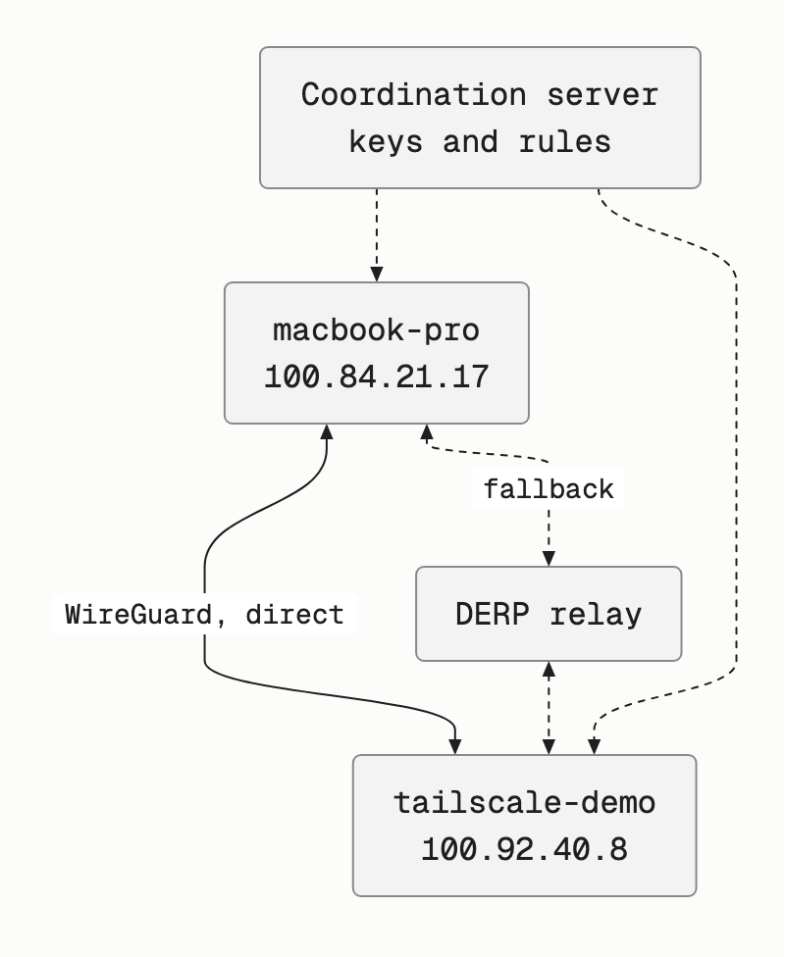
I didn't join this book's test Droplet to a tailnet. The outputs in this chapter come from the Droplet `tailscale-demo` I set up for my Tailscale tutorial in November 2026, also Ubuntu 24.04 in Frankfurt, with the same stand-in addresses.

What a tailnet is

Your private network is a **tailnet**. Every device in it gets an address in the `100.x.y.z` range and a name.

Traffic between two devices goes over WireGuard, encrypted, and directly between them when the networks allow it. When they don't, it goes through a Tailscale relay called a **DERP** server, still encrypted end to end.

Tailscale's coordination server knows your devices, their keys, and who may talk to whom. It hands out keys and rules, and your traffic doesn't go through it.



There's no Tailscale password. You sign in with GitHub, Google, Microsoft, Apple or a passkey, and that login decides which tailnet a device joins.

Connect the server

On the Mac I installed the app and signed in. On the Ubuntu server, Tailscale's install script adds its APT repository and installs the client. Like the NodeSource script in the services chapter, it's worth reading it before running it:

```
curl -fsSL https://tailscale.com/install.sh | sh
```

```
Installation complete! Log in to start using Tailscale by running:

tailscale up
```

`--hostname` sets the server's name in the tailnet:

```
tailscale up --hostname=tailscale-demo
```

A server has no browser, so it prints a link to open on a device that's already signed in:

```
To authenticate, visit:
```

```
    https://login.tailscale.com/a/1f12656a01b843
```

After I clicked **Connect** on that page, `tailscale up` printed `Success.` and returned.

For a server set up by a script, you'd use an auth key instead of the link. Make it one-off, with a short expiry, pass it through an environment variable so it doesn't end up in the shell history, and delete it from the admin console once it's used:

```
sudo tailscale up --auth-key="$TS_AUTH_KEY" --advertise-tags=tag:server
```

Check the path

`tailscale status` lists the devices:

```
$ tailscale status
100.84.21.17  macbook-pro     flaviocopes@  macOS  -
100.92.40.8     tailscale-demo  flaviocopes@  linux  -
```

`tailscale ping` goes through Tailscale, not ICMP, and tells you if the connection is direct or relayed:

```
$ tailscale ping tailscale-demo
pong from tailscale-demo (100.92.40.8) via 203.0.113.42:41641 in 44ms
```

`via 203.0.113.42:41641` is the server's public IP and Tailscale's UDP port, so it's direct WireGuard. My Mac was behind a home router with no port forwarding, and it got a direct path on the first try. A relayed connection says `via DERP(ams)` or similar.

`tailscale netcheck` shows the relays and how far they are:

```
$ tailscale netcheck

Report:
    * UDP: true
    * IPv4: yes, 203.0.113.7:7691
    * IPv6: no, but OS has support
    * MappingVariesByDestIP: true
    * CaptivePortal: false
    * Nearest DERP: Amsterdam
    * DERP latency:
        - ams: 83ms    (Amsterdam)
        - lhr: 83.5ms  (London)
        - waw: 84.1ms  (Warsaw)
        - nue: 84.1ms  (Nuremberg)
        - hel: 84.1ms  (Helsinki)
        - fra: 84.4ms  (Frankfurt)
```

The direct path to Frankfurt was 44 ms, about half of going through the closest relay. A relayed connection still works, only slower. When something feels slow over Tailscale, `tailscale ping` is the first thing I run.

SSH by name

Every device gets a DNS name with **MagicDNS**, on by default, answered by `100.100.100.100` on your devices. So SSH works with the short name and my normal key:

```
$ ssh root@tailscale-demo
tailscale-demo
100.84.21.17 65460 100.92.40.8 22
```

The second line is the server printing `$SSH_CONNECTION`: from the Mac's tailnet address to the server's, without touching the public IP.

This is plain OpenSSH over the tailnet, with the keys in `authorized_keys` in charge. Tailscale also has its own SSH server, Tailscale SSH (`tailscale set --ssh`), which logs you in with Tailscale identities instead. I left it off and kept my SSH setup from Part 2. Tailscale alternatives has the October 2026 Tailscale SSH bulletins, which are part of why.

A service only your devices can reach

A service that listens on the tailnet address is reachable from your devices and from nothing else. A tiny web server on the Droplet, bound to its tailnet address:

```
python3 -m http.server 8080 --bind 100.92.40.8
```

From the Mac, by name, and then on the public IP:

```
$ curl http://tailscale-demo:8080
hello from tailscale-demo
$ curl http://203.0.113.42:8080
curl: (7) Failed to connect to 203.0.113.42 port 8080 after 95 ms: Couldn't connect to server
```

This is how I'd run Vaultwarden from the last chapter, or an admin panel: bound to `127.0.0.1` or to the tailnet address, never to `0.0.0.0`.

Be aware that UFW on that server only allowed ports 22 and 80, and port 8080 still worked over the tailnet. Tailscale adds firewall rules that accept tailnet traffic before UFW sees it. For tailnet traffic, the Tailscale access policy decides, not UFW.

Close every public port

Once SSH works over the tailnet, the server doesn't need port 22 open to the world. I attached a DigitalOcean Cloud Firewall with no inbound rules at all, kept outbound open because Tailscale dials out, and scanned the public IP from my Mac with `nmap`:

Port	Before, UFW only	After, no inbound rules	Over the tailnet
22, SSH	open	filtered	works
80, web	open	filtered	works
5432, Postgres in Docker	open	filtered	works
8080, tailnet-only page	filtered	filtered	works

The Postgres row in the ”Before” column is the Docker trap from the home server chapter, on another server: UFW denied incoming, and the published port answered anyway.

I expected Tailscale to fall back to a relay with no inbound rule left, so I restarted `tailscaled` to force a new path:

```
$ tailscale ping tailscale-demo
pong from tailscale-demo (100.92.40.8) via 203.0.113.42:41641 in 32ms
```

Still direct. Both ends send UDP out to each other, and the stateful firewall counts the other side's packets as replies.

Before you close port 22 on a real server, check that SSH over the tailnet works and that you can get in through your provider's web console. My secure VPS guide goes through the safe order.

For a server that also has a public website, this is the setup I like: ports 80 and 443 open for Nginx, nothing else, and everything admin over the tailnet. Tailscale vs Cloudflare Tunnel vs a cloud firewall compares the options on the same Droplet. Over the tailnet the page loaded in 72 ms (median of 10), against 137 ms on the public IP.

Who can reach what

A new tailnet starts with a policy that lets every device reach every other device on every port:

```
{"src": ["*"], "dst": ["*"], "ip": ["*"]},
```

With one Mac and one server that's fine. Once you add another person, or a machine you trust less (like a home server with a few containers), you want narrow rules. They're called grants.

My demo tailnet had one Mac and one server, so I kept the default policy there. These are the rules I'd start from once a second person or a less trusted machine joins.

First, tags. A server gets a role instead of being owned by your user. The policy declares who may assign the tag:

```
{
  "tagOwners": {
    "tag:server": ["flavio@example.com"]
  }
}
```

Then a grant replaces the wildcard: my user may reach TCP port 22 on servers tagged `tag:server`, and nothing else.

```
{
  "grants": [
    {
      "src": ["flavio@example.com"],
      "dst": ["tag:server"],
      "ip": ["tcp:22"]
    }
  ]
}
```

Test both sides after a change: SSH should work, and another port on the same server, like 5432, should time out. A rule that allows what you want proves nothing until you see it block what you didn't grant.

Edit the policy from a session that doesn't depend on it. If you change the rules over tailnet SSH and the change breaks tailnet SSH, the provider's web console is your way back in.

Subnet routers

Some devices can't run Tailscale: a printer, a NAS, a router's admin page. A **subnet router** is a Linux machine on the same LAN that brings that whole network into the tailnet. A home server is a natural place for it.

It needs IP forwarding on, the route advertised, the route approved in the admin console, and a grant for the destination. Advertise the narrowest prefix that covers what you need, and don't advertise a common one like `192.168.1.0/24` from a remote site, because it clashes with the LAN your laptop is on.

The forwarding settings are the same as for an exit node, below. When a routed address times out, check those four switches in order. Most of the time it's the approval or the grant that's missing.

Use the server as an exit node

An **exit node** forwards all your internet traffic, so sites see its IP instead of yours. It's handy on hotel or café Wi-Fi.

On the server, turn on forwarding and advertise it:

```
echo 'net.ipv4.ip_forward = 1' > /etc/sysctl.d/99-tailscale.conf
echo 'net.ipv6.conf.all.forwarding = 1' >> /etc/sysctl.d/99-tailscale.conf
```

```
sysctl -p /etc/sysctl.d/99-tailscale.conf
tailscale set --advertise-exit-node
```

Advertising only asks. I approved it in the admin console under **Edit route settings**, then picked it on the Mac, checking my public IP before, during and after:

```
$ curl https://ifconfig.me
203.0.113.7
$ tailscale set --exit-node=tailscale-demo
$ curl https://ifconfig.me
203.0.113.42
$ tailscale set --exit-node=
$ curl https://ifconfig.me
203.0.113.7
```

Keep in mind that websites now see your VPS's IP, which is tied to your hosting account. It hides your traffic from the café's network, not from the sites you visit.

Serve and Funnel

`tailscale serve` puts a local service on an HTTPS address that only your tailnet can open, with a real certificate. I use it to open my Mac's dev server on my iPhone:

```
$ tailscale serve --bg 4400
Available within your tailnet:

https://macbook-pro.tail8c22d0.ts.net/
|-- proxy http://127.0.0.1:4400

Serve started and running in the background.
To disable the proxy, run: tailscale serve --https=443 off
```

The first time, it asks you to turn on Serve and HTTPS certificates for the tailnet in the admin console. And be aware that those certificates come from Let's Encrypt, so the machine name and your tailnet's random name (`tail8c22d0` here) end up in the public Certificate Transparency logs. Nobody can connect with them, but rename a machine called `secret-project` before turning HTTPS on.

On a server that's a quick way to put a private dashboard behind HTTPS without touching Nginx.

Funnel is the opposite switch: it publishes a local service to the whole internet on a `ts.net` address. It's off by default and I leave it off. For public traffic I want Nginx, its logs and its rate limits, on a server I control.

What it costs

The Personal plan is $0 for non-commercial use, with up to 6 users and unlimited devices. Standard is $8 per user per month (November 2026 prices). A tailnet created with a GitHub, Google or Apple account starts on Personal.

How I run Linux servers

I'll close with how I use all of this myself, because the right amount of server depends on what you're running.

`flaviocopes.com` isn't on a server I manage. It's static files and a few functions on Cloudflare, and there's no operating system there that I need to patch. For a site like that, a VPS would only add work.

Then there are the tools that want a persistent Linux box. Today two Droplets are running in my DigitalOcean account, besides the test one: `Analytics` in Frankfurt, for Plausible, and `Sendy` in New York, for my newsletter. One server, one job, so I always know what a reboot or an update touches.

On a new server, Caddy is my default reverse proxy, because automatic HTTPS removes a whole category of certificate problems. I reach for Nginx when I want the fine control we used in this book, like rate limits per zone or a catch-all default server.

SSH stays off the public internet when I can manage it, thanks to Tailscale, and UFW keeps the rest closed. Backups leave the server encrypted, with restic, to object storage on another provider's infrastructure, so losing the server doesn't mean losing the backups.

The things that broke while I was writing this book are the things that break for real: the automatic update holding the APT lock right after boot, a copy of the app I forgot running and holding the port, a config line the installed Nginx didn't understand, a Docker port the firewall didn't know about, and an app that exited cleanly and stayed down. None of them were exotic, and the right command found each one.

If you're starting out, my advice is to do what I did for this book. Create the cheapest Droplet you can, give it a name like `test-server`, and go from the first SSH connection to the restore drill in one afternoon, with your own app instead of the notes API. Then destroy it and build it again from your notes. The second time you'll find the steps you forgot to write down, and that's a much better moment to find them than when a real server is down.

So skip the server when you need a static site, a small webhook, or a managed database for a prototype. Rent one, or plug in an old mini PC at home, when you want predictable pricing, real disks, Docker Compose stacks, and long-running processes that serverless limits get in the way of.

The test Droplet for this book, with its volume, ran for a few hours and cost a few cents. I destroyed it when I finished. Every unit file, config and command to build it again is in these pages.