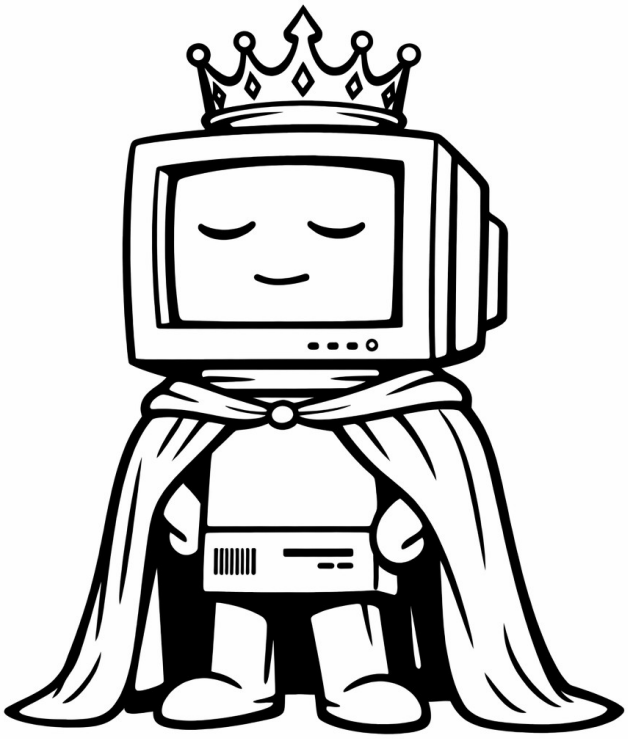
Digital Sovereignty in Practice

Self-hosted services with Docker: Photos, Music, Local AI, and Private Drive



Mattia Coccolo

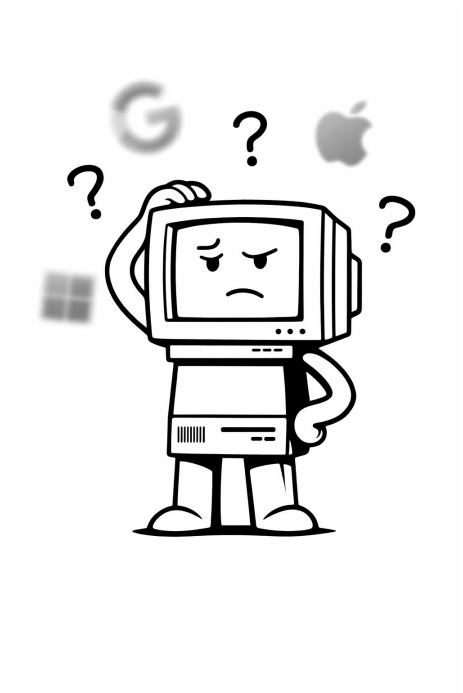
6 March 2026

Version: v1.0.0
Archived version DOI: https://doi.org/10.5281/zenodo.18890858
Latest reading version: https://mattispin.github.io/digital-sovereignty-manual/

License

Contents

Prologue



This manual is a practical, step-by-step guide to building your own small digital ecosystem: a private photo library, a music server, local AI tools, and a personal cloud drive—all running on your own machine, under your control, with Docker as the backbone.

The motivation is simple: most of our digital lives live somewhere else. Photos, documents, messages, research notes, calendars, passwords, and even our work routines are increasingly mediated by services we do not operate. This can be convenient, but convenience often comes with hidden costs: lock-in, recurring subscriptions, opaque data practices, and a quiet dependence on decisions made far from our needs and values.

The goal here is not to reject the internet or to chase an unrealistic ideal of "total independence". The goal is to show that a meaningful level of *digital sovereignty* is achievable with accessible tools: electricity, internet, a computer, and a bit of patience. You can keep your core data at home (or at your institution), expose only what you choose, and still enjoy modern usability.

This manual is written for people who want to:

- understand what they run and where their data lives,
- reduce dependence on external platforms without sacrificing daily convenience,
- create a modular setup that can evolve over time,
- document the process so it can be reproduced by colleagues, friends, or students.

A note on scope

We focus on a pragmatic stack:

- **Docker + Linux** as the foundation,
- **PhotoPrism** for photos,
- **Navidrome** for music,
- **Local AI** (Ollama + a web UI) for private, on-device workflows,
- **A private drive** (e.g., Seafile) for syncing and organizing files.

We will also briefly discuss operational concerns: storage layout, backups, updates, and basic security choices. Whenever something depends on a specific environment (home server vs. university server, VPN vs. public exposure), we explicitly say so.

How to use this manual

Each chapter follows a predictable pattern:

- **Goal**: what you will have at the end,
- **Prerequisites**: what you need beforehand,
- **Steps**: the actions to perform,
- **Verification**: how to confirm it works,
- **Troubleshooting**: the common failure modes and fixes.

If you already have a Docker host running, you can skip directly to the service chapters (PhotoPrism, Navidrome, Local AI, and the Private Drive). If you only want one service, you can treat chapters as independent modules.

Philosophy: small, modular, reproducible

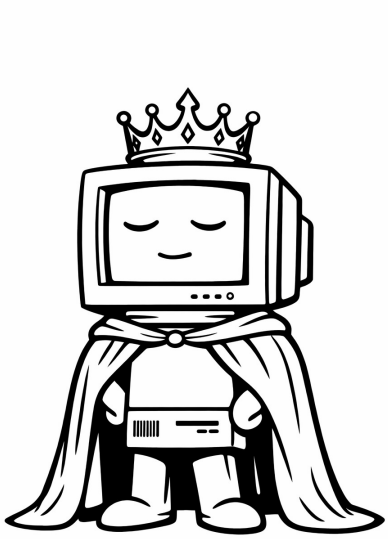
This manual aims for:

- **Small steps**: avoid "big bang" setups. Make one thing work, then add the next.
- **Modularity**: each service lives in its own container(s), with clear volumes and ports.
- **Reproducibility**: all configuration should be portable via compose files and a documented folder structure.

Done well, the result feels less like "running servers" and more like owning your own digital workspace.

Chapter 1

Digital Sovereignty: what it is, and why it matters



1.1 A working definition

Digital sovereignty is the ability to *choose* how your digital life works:

- where your data is stored,
- who can access it (and under what conditions),
- how services evolve over time,
- what happens if a provider changes rules, prices, or policies.

It is not a binary state. It is a spectrum. The question is not “sovereign or not”, but *how much control you want* and *where it is worth the effort.*

1.2 Why it matters (even if you are not “technical”)

There are four recurring reasons people move toward more sovereignty:

1.2.1 Control and resilience

Platforms change. Features disappear. Prices rise. Accounts get locked. Terms of service shift. A self-hosted core makes your workflow more resilient: if one component fails, you still own your data and can migrate.

1.2.2 Privacy and confidentiality

For researchers, professionals, and anyone handling personal archives, privacy is not a luxury. Local-first tools reduce the need to upload sensitive content to third parties by default.

1.2.3 Cost transparency

Many cloud services start free and become expensive at scale (storage, team accounts, add-ons). With self-hosting, costs are mostly upfront (hardware, disks) and predictable (electricity, backups). That trade-off often becomes favorable once you have non-trivial archives (photos, videos, datasets).

1.2.4 Freedom from lock-in

When your data and workflows depend on proprietary formats or closed ecosystems, switching becomes painful. Using standard tools and keeping raw files under your control makes migration a choice, not a crisis.

1.3 Threat model: what are we protecting against?

Before installing anything, define what you care about. A minimal threat model answers:

- **Loss**: disk failure, accidental deletion, ransomware.
- **Exposure**: services reachable from the public internet without safeguards.
- **Lock-in**: dependence on one provider or one app.
- **Downtime**: updates breaking the stack with no rollback plan.

This manual assumes a reasonable baseline: protect your core data, keep the system maintainable, and avoid unnecessary public exposure. If you want internet-facing services, we will highlight the extra steps (reverse proxy, TLS, hardening, monitoring) rather than treating them as default.

1.4 Hardware requirements (practical guidance)

You do not need a data center. The minimum viable setup is surprisingly modest.

1.4.1 Baseline (photos + music + private drive)

- **CPU**: 4 cores (8 threads recommended)
- **RAM**: 8–16 GB (16 GB is comfortable)
- **Storage**: SSD for system + Docker volumes; HDD/SSD for media/archive
- **Network**: stable LAN; optional VPN for remote access

1.4.2 When you want local AI

Local AI is the only part that may require specialized hardware.

- **GPU (optional)**: strongly recommended for larger models and good latency.
- **VRAM**: the practical limiter. More VRAM allows larger models and higher context windows.
- **RAM**: 16 GB minimum; 32 GB recommended if you also run other services heavily.

Without a GPU you can still run small models on CPU, but responses will be slower and the experience changes.

1.4.3 A real-world example (from the author's Docker host)

Table 1.1 shows a snapshot of the kind of resource usage you might see while running a small stack.

Table 1.1: Example: container resource usage snapshot on a 24-CPU host.

Metric	Example value
Container CPU usage	0.79% / 2400% (24 CPUs available)
Container memory usage	4.53 GB / 15.22 GB

The "2400%" upper bound means Docker reports CPU as a sum across cores: 100% roughly corresponds to one fully utilized core, and 2400% corresponds to 24 cores.

1.5 Choose your installation path (read this once)

This manual offers two ways to build the server foundation:

- **Path A (recommended): Linux server + Docker.** This is the simplest and most reliable option for an always-on machine.
- **Path B: Windows + WSL2 + Docker.** This is a beginner-friendly path if you already use Windows and want to learn step by step, but it adds one extra layer (Windows + Linux).

Pick one path and follow it. After the foundation is ready, the rest of the manual (PhotoPrism, Navidrome, Local AI, Private Drive) is the same in both cases.

1.6 What you will build

By the end of this manual you will have:

- a Linux host with Docker and a clean folder layout for persistent data,
- PhotoPrism indexing and serving your photo library,
- Navidrome serving your music library,
- a local AI service and a simple web interface to use it privately,
- a private drive service for file sync and collaboration,
- a basic operational routine: updates, backups, and sanity checks.

1.7 What this manual intentionally avoids

To keep the path accessible, we do *not* start with advanced infrastructure topics such as:

- **Kubernetes clusters** (container orchestration across many machines),
- **mandatory public exposure** (domain name, reverse proxy, TLS/HTTPS),
- **heavy monitoring stacks** (full-time metrics, dashboards, and alerting).

If some of the words above look like mysterious tech jargon, do not worry. This section exists so that, when you encounter these terms in more complex books or manuals, you can understand them without feeling lost. We do not use them in this manual—but you can tell your friends you know what they are and casually show off ;-) ...and sound surprisingly competent at dinner

Kubernetes (we do not use it here)

Kubernetes is a system for running containers at large scale: many machines, many services, automatic scheduling, and automatic recovery. It is powerful and widely used in companies, but it adds a lot of complexity. In this manual, we use Docker Compose because it is simpler, easier to learn, and more than enough for a personal or small-group server.

Public exposure, domains, reverse proxies, and TLS (optional later)

Making a service accessible from the public internet safely usually involves several extra pieces:

- a **domain name** (a human-friendly address),
- a **reverse proxy** (a gateway that routes traffic to the correct service),
- **TLS/HTTPS** (encryption so passwords and data are not sent in plain text).

These are valuable, but they also create more ways to misconfigure a system. In this manual, the default is **local network access or VPN access**, which is simpler and safer for beginners. Later, if you want your services publicly reachable, you can add these components deliberately.

Heavy monitoring stacks (nice, but not the first step)

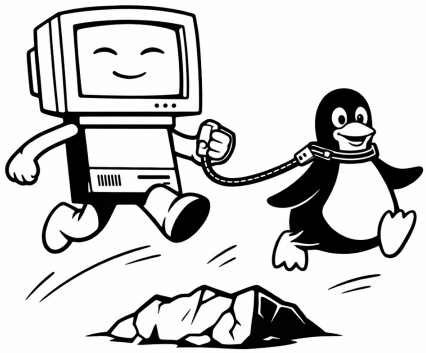
Monitoring stacks are tools that collect metrics, logs, and alerts (CPU usage, memory, disk space, failures, response times). They are great when you operate many systems or need strict uptime. For a personal server, you can start with simple checks and basic logs. Once your core system is stable and you know what "normal" looks like, adding monitoring becomes much easier and more meaningful.

The principle

You can add advanced layers later. First, make the core system reliable: **clear storage, reproducible configuration, and backups**.

Chapter 2

Server Path A (recommended): Linux + Docker



2.1 Goal (what you will achieve)

This chapter sets up a **real Linux server foundation** for the rest of the manual.
By the end, you will have:

- a Linux machine you can administer (locally or via SSH),
- a working Docker installation,
- Docker Compose (`docker compose`) available,
- a clean and safe data layout under `/srv`,
- basic security hygiene (optional but strongly recommended).

2.2 Who should choose this path?

Choose **Path A** if any of the following is true:

- you want an always-on machine (a "real server"),
- you want the most reliable and standard setup,
- you prefer fewer layers (no Windows + WSL complexity),
- you plan to run multiple services long-term.

If you are on Windows and want a guided learning approach first, use Path B (Windows + WSL2).

2.3 What is a terminal, and why do we use it?

A **terminal** is a text-based interface to your system. You type commands; the system executes them.

Many server tasks are easier and more precise in the terminal:

- installing software,
- configuring services,
- checking logs and system status,
- repeating steps reliably (copy/paste).

2.4 What is `sudo`?

`sudo` means "run as administrator" on Linux.

- Installing software and changing system configuration requires admin rights.
- `sudo` grants those rights for one command at a time.

2.5 Step 1 — Install Linux (high-level, beginner-friendly)

If Linux is already installed and you can open a terminal, skip to Step 2.

2.5.1 Recommended distributions

For beginners, choose one of these:

- **Ubuntu Server LTS** (very common, lots of tutorials),
- **Debian** (very stable, slightly more conservative).

This manual uses commands compatible with both Ubuntu and Debian.

2.5.2 During installation: minimal checklist

When the installer asks questions:

- Create a user account (this will be your admin user).
- Enable **OpenSSH server** if offered (recommended).
- Choose a hostname (e.g., `sovereignty-server`).
- Keep disk layout simple (we will store our data under `/srv`).

2.5.3 After installation: how to open the Linux terminal

You have two common options:

Option A: local login (keyboard + screen). You will see a login prompt. After logging in, you are already in a terminal.

Option B: remote login via SSH (recommended). From another computer on the same network, open a terminal (or PowerShell on Windows) and type:

```
ssh username@SERVER_IP
```

After login, you are inside the server terminal.

2.5.4 Quick check: do you have admin privileges?

Run:

```
sudo whoami
```

If it prints `root`, your user can run admin commands (good).

2.6 Step 2 — Update Linux (important before installing Docker)

If this is your first time using a terminal: perfect. You do not need to "understand everything" today. Just follow the steps and you will learn naturally as you go.

2.6.1 2.1 Refresh package lists

```
sudo apt update
```

What this does: downloads the latest list of available packages.
Why you do it: you want current versions and security fixes.

2.6.2 2.2 Upgrade installed packages

```
sudo apt upgrade -y
```

What this does: upgrades installed software.
Why you do it: reduces bugs and security issues.
About `-y`: automatically answers "yes" to prompts.

2.6.3 2.3 Install basic tools (recommended)

```
sudo apt install -y curl ca-certificates gnupg lsb-release unzip
```

What these are:

- `curl`: download files from the internet (we use it to fetch Docker keys),
- `ca-certificates`: makes HTTPS downloads verifiable and safe,
- `gnupg`: used to verify signed packages (security),
- `lsb-release`: helps detect your Linux version automatically,
- `unzip`: useful for downloaded archives.

2.7 Step 3 — Create a clean data layout

A common beginner mistake is to store important data *inside* containers. Do **not** do that.

The idea (in one sentence)

Containers are disposable; your data should live in *normal folders on disk*.

Choose a single root folder for persistent data

Two common options are:

- `/srv` (very common on servers),
- `/opt` (also used for application data).

In this manual we use `/srv`.

Create the folders

```
sudo mkdir -p /srv/{docker,photos,music,drive,ai,backups}
sudo mkdir -p /srv/docker/compose
```

What this does:

- `mkdir` creates directories,
- `-p` means "create parents if needed" and "do not error if it already exists",
- `{...}` creates multiple folders quickly.

Why you do it: a consistent structure makes backups, upgrades, and troubleshooting much easier.

Give your user ownership

```
sudo chown -R $USER:$USER /srv
```

What this does:

- changes ownership of `/srv` to your user,
- `-R` applies to everything inside.

Why you do it: you can manage configs and volumes without constantly using `sudo`.

2.7.1 Why this matters

When you upgrade or recreate containers, your data remains safe in `/srv`. Containers can be disposable; your files should not be.

2.8 Step 4 — Install Docker Engine (from the official repository)

The safest approach is to install Docker from Docker's official repository. This gives you up-to-date packages and security fixes.

2.8.1 4.1 Add Docker's official GPG key

```
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/
    ↪ apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

What this does:

- creates a folder for trusted repository keys,
- downloads Docker's signing key and stores it for APT,
- makes it readable by the package manager.

Why you do it: APT uses the key to verify that packages are authentic (not modified).

This step looks scary because it mentions keys and repositories. You are not "hacking" anything: you are simply telling Linux to trust the official Docker source. Copy/paste exactly as shown.

2.8.2 4.2 Add the Docker repository (choose Ubuntu or Debian)

Important: pick the correct block for your system.

Ubuntu

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
      ↪ https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Debian

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
      ↪ https://download.docker.com/linux/debian \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

What this does:

- tells APT where to download Docker packages from,
- automatically detects your CPU architecture (e.g., amd64),
- automatically detects your distro codename (e.g., `jammy`, `bookworm`),
- restricts trust to packages signed by Docker's key.

Why you do it: so `apt` can install Docker from the official source.

2.8.3 4.3 Install Docker Engine and plugins

```
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-
    ↪ compose-plugin
```

What this installs:

- `docker-ce`: Docker Engine (Community Edition),
- `docker-ce-cli`: Docker command-line tool,
- `containerd.io`: the container runtime used by Docker,
- `docker-buildx-plugin`: modern image-building features,
- `docker-compose-plugin`: Compose integration (`docker compose`).

Why you do it: you get Docker + Compose in a modern, supported configuration.

2.8.4 4.4 Enable and start Docker

```
sudo systemctl enable docker
sudo systemctl start docker
```

What this does:

- starts Docker now,
- ensures Docker starts automatically after a reboot.

Why you do it: your services should come back after restarts.

2.9 Step 5 — Allow running Docker without `sudo` (recommended)

By default, Docker commands require `sudo`. For everyday use, add your user to the `docker` group:

```
sudo usermod -aG docker $USER
```

What this does: allows your user to communicate with the Docker daemon without admin privileges.
Why you do it: it makes daily work smoother.
Then log out and log back in (or reboot) so the group membership applies.

2.10 Step 6 — Verify the installation

2.10.1 Docker version

```
docker --version
```

Why this matters: confirms Docker is installed and accessible.

2.10.2 Compose version

```
docker compose version
```

Why this matters: confirms Compose is available for multi-service setups.

2.10.3 Hello-world test

```
docker run --rm hello-world
```

What this does:

- downloads a small test image (if needed),
- runs it in a container,
- deletes the container after exit (`-rm`).

Why this matters: proves Docker can pull images and run containers correctly.

If you saw the hello-world success message: congratulations. You have Docker working. From here on, it is mostly repeating the same pattern.

2.11 Step 7 — (Optional) Basic firewall and SSH hygiene

If you only use services inside your home network or behind a VPN, you can keep this simple. If the server is exposed to the internet, do **not** skip this.

2.11.1 Enable UFW (simple firewall)

```
sudo apt install -y ufw
sudo ufw default deny incoming
sudo ufw default allow outgoing
sudo ufw allow OpenSSH
sudo ufw enable
sudo ufw status
```

What this does:

- blocks incoming connections by default,
- allows outgoing traffic,
- allows SSH so you do not lock yourself out,
- enables the firewall and shows status.

Why this matters: it reduces your attack surface.

2.11.2 SSH: disable password login (recommended)

Once you can log in via SSH keys, disable password login:

```
sudo nano /etc/ssh/sshd_config
```

What this does: opens the SSH server configuration file.
Why this matters: password logins can be brute-forced; keys are much safer.
Find the line (or add it) and set:

```
PasswordAuthentication no
```

Then restart SSH:

```
sudo systemctl restart ssh
```

What this does: applies the new SSH configuration.
Why this matters: changes require a restart/reload to take effect.

2.12 Verification checklist

Before moving on, confirm:

- `docker run -rm hello-world` works,
- `docker compose version` prints a version,
- `/srv` exists and contains your data folders,
- you understand the core principle: **containers are disposable; data is not**.

2.13 Troubleshooting (common issues)

2.13.1 "Permission denied" when running Docker

If Docker commands fail without `sudo`, your user may not yet be in the `docker` group, or the group membership has not been applied.

Fix:

```
sudo usermod -aG docker $USER
```

Then log out and log back in (or reboot).

2.13.2 Docker service not running

Check status:

```
sudo systemctl status docker
```

Start it:

```
sudo systemctl start docker
```

2.13.3 Compose not found

Install the Compose plugin:

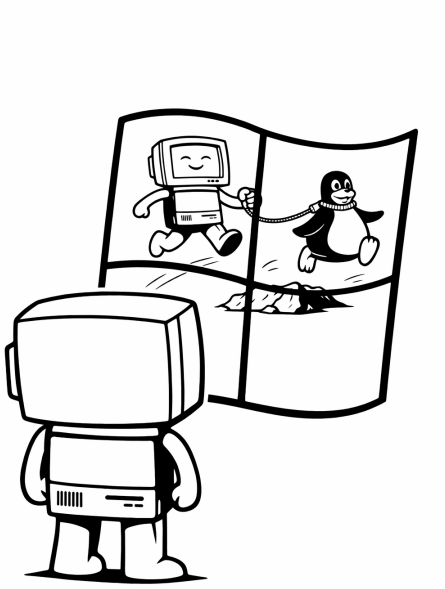
```
sudo apt install -y docker-compose-plugin
```

2.14 Next step

Next, we will deploy the first real service (PhotoPrism) using Docker Compose. We will reuse the same principles: **clear volumes**, **documented configuration**, and **easy upgrades**.

Chapter 3

Server Path B: Windows + WSL2 + Docker (beginner-friendly)



3.1 Goal (what you will achieve)

This chapter helps you build the same foundation as Path A, but starting from Windows.
By the end, you will have:

- Windows prepared with WSL2 (a real Linux environment inside Windows),
- an Ubuntu Linux terminal you can open from the Start menu,
- Docker installed and working *via Docker Desktop* (recommended for beginners),
- Docker Compose available,
- a clean data layout (persistent folders) so containers can be upgraded safely.

3.2 Who should choose this path?

Choose **Path B** if:

- you already use Windows and want to start without reinstalling your machine,
- you are not comfortable installing Linux directly yet,
- you want a gentle, guided learning path.

Good to know: this path adds one extra layer (Windows + Linux). It works well for learning and small self-hosted stacks, but a dedicated Linux server is usually simpler long-term.

3.3 Big picture (in plain language)

3.3.1 What is WSL2?

WSL2 stands for **Windows Subsystem for Linux 2**. It lets Windows run a real Linux system in the background.

Think of it like this:

- Windows is your main computer environment.
- WSL2 is a small Linux computer *inside* Windows.
- You open an **Ubuntu app** to get a Linux terminal.

3.3.2 What is Docker Desktop? (and why we use it here)

Docker Desktop is the easiest way for beginners to run Docker on Windows. It integrates with WSL2 so you can run Docker commands from the Ubuntu terminal.

In this manual, **Path B assumes Docker Desktop**, because it avoids advanced system configuration.

Mental model: you type Docker commands in the Ubuntu terminal, but the Docker engine is running in Docker Desktop on Windows. Ubuntu is your control panel; Docker Desktop is the motor.

3.4 Before you start (requirements)

- Windows 10/11 with administrator access.
- Internet connection.
- Enough disk space (at least 30–50 GB free is a comfortable start).

3.5 Step 1 — Install WSL2

3.5.1 1.1 Open PowerShell as Administrator

How:

- Click Start.
- Type `PowerShell`.

- Right-click **Windows PowerShell** and choose **Run as administrator**.

This looks more serious than it is. "Run as administrator" just gives permission to enable Windows features. You are not breaking anything—you are switching on the Linux subsystem that Windows already supports.

3.5.2 1.2 Install WSL with Ubuntu

In the admin PowerShell window, run:

```
wsl --install
```

What this does:

- enables the Windows features required for WSL,
- installs WSL2,
- installs a default Linux distribution (usually Ubuntu).

Why you do it: it creates the Linux environment where we will run server commands.

3.5.3 1.3 Reboot if Windows asks you to

WSL installation often requires a restart.

3.5.4 1.4 Confirm WSL2 is active

Open a normal PowerShell (not necessarily admin) and run:

```
wsl --status
```

You want to see that WSL is installed and that the default version is 2.

If needed: set WSL2 as default.

```
wsl --set-default-version 2
```

3.6 Step 2 — Open your Linux terminal (Ubuntu)

This is the key beginner step: **your Linux terminal is an app.**

If you have never used Linux: perfect. In this path, Linux is just an app you open and close. You can learn it safely, step by step, without reinstalling your computer.

3.6.1 2.1 Open Ubuntu from the Start menu

- Click Start.
- Type `Ubuntu`.
- Click the Ubuntu app.

A terminal window opens. **This is Linux.**

3.6.2 2.2 Create your Linux username and password

The first time you open Ubuntu, it will ask you to create:

- a Linux username,
- a Linux password.

Important: this password is for Linux inside WSL, not your Windows password.

Two normal things that confuse everyone the first time: (1) this is a new password for Ubuntu, not your Windows password, and (2) when you type it, you will not see any characters on screen. That is normal—type carefully and press Enter.

3.7 Step 3 — Update Ubuntu inside WSL

Now that you are in the Ubuntu terminal, run:

3.7.1 3.1 Refresh package lists

```
sudo apt update
```

What this does: updates the list of available packages.
Why you do it: ensures you install current versions.

3.7.2 3.2 Upgrade installed packages

```
sudo apt upgrade -y
```

What this does: upgrades installed software.
Why you do it: reduces bugs and security issues.

3.7.3 3.3 Install basic tools

```
sudo apt install -y curl ca-certificates gnupg lsb-release unzip
```

3.8 Step 4 — Install Docker (recommended: Docker Desktop)

On Windows, there are multiple ways to run Docker. For beginners, the best approach is:

- install **Docker Desktop** on Windows,
- enable WSL integration,
- run Docker commands from the Ubuntu terminal.

3.8.1 4.1 Install Docker Desktop on Windows

What to do:

- Download and install Docker Desktop (official installer).
- During setup, keep the option “Use WSL 2 instead of Hyper-V” enabled (recommended).

Why we do it: Docker Desktop handles the Windows-side details for you.

If the installer offers many options: do not overthink it. The default settings are fine. We only care that Docker Desktop runs and WSL integration is enabled.

3.8.2 4.2 Enable WSL integration

Open Docker Desktop and go to:

- **Settings → Resources → WSL Integration**

Enable integration for your Ubuntu distribution.
What this does: allows the Ubuntu terminal to use the Docker engine managed by Docker Desktop.

3.9 Step 5 — Verify Docker from the Ubuntu terminal

In the Ubuntu terminal (WSL), run:

3.9.1 5.1 Docker version

```
docker --version
```

3.9.2 5.2 Compose version

```
docker compose version
```

3.9.3 5.3 Hello-world test

```
docker run --rm hello-world
```

If this works, Docker is correctly installed and integrated.

Nice. This is the big milestone. From here on, you are mostly repeating the same pattern: define a service in Compose, start it, and open it in your browser.

3.10 Step 6 — Create a clean data layout (WSL-friendly)

We use the same principle: containers are disposable; data must persist.

3.10.1 6.1 Create folders inside Linux

In the Ubuntu terminal:

```
sudo mkdir -p /srv/{docker,photos,music,drive,ai,backups}
sudo mkdir -p /srv/docker/compose
sudo chown -R $USER:$USER /srv
```

Why inside WSL? It is faster and more reliable to keep Docker volumes on the Linux filesystem used by WSL.

3.10.2 6.2 Important note about Windows drives

You can access Windows folders from WSL under `/mnt/c/`, `/mnt/d/`, etc. However, for Docker data and databases, using the Linux filesystem (inside WSL) is usually more stable and faster.

3.11 Step 7 — (Optional) Remote access

If you want to access your services from another device on your network, you have two options:

- Use Windows networking (Docker Desktop publishes ports on Windows).
- Use a VPN (recommended for access outside your home).

Beginner advice: start local first (use your browser on the same PC), then expand to network access once everything works.

3.12 Verification checklist

Before moving on, confirm:

- You can open Ubuntu from the Start menu (WSL terminal).
- `docker run -rm hello-world` works inside Ubuntu.
- You have `/srv` created inside WSL.

3.13 Troubleshooting (common issues)

3.13.1 "wsl is not recognized" in PowerShell

WSL is not installed or Windows is too old. Update Windows and try again.
If WSL refuses to install, two common causes are:

- Windows is outdated.
- Virtualization is disabled in BIOS/UEFI.

Update Windows first; if the problem persists, check that virtualization is enabled.

3.13.2 Ubuntu asks for a password, but nothing appears when typing

This is normal: Linux passwords do not show characters while you type. Type carefully and press Enter.

3.13.3 Docker command not found inside Ubuntu

Either Docker Desktop is not installed, or WSL integration is not enabled. Check Docker Desktop settings and enable integration for Ubuntu.

3.13.4 Hello-world fails with permission errors

Try running once with sudo:

```
sudo docker run --rm hello-world
```

If that works, then your user is missing Docker permissions in WSL. Restart Docker Desktop, close and reopen Ubuntu, and try again.

3.14 Next step

Next, you can install the first service (PhotoPrism) using Docker Compose. From this point onward, the service chapters are the same for Path A and Path B.

3.15 Optional advanced notes (read only if you need them)

This section is intentionally optional. If your Docker + WSL2 setup works, you can skip it.

3.15.1 A) Do I need Docker Desktop? (alternatives)

For most beginners, **Docker Desktop is the simplest path** on Windows. However, there are alternatives:

- **Docker Engine inside WSL2 (without Docker Desktop).** This can work, but it requires extra configuration (often systemd support, networking details, and careful permissions).
- **A dedicated Linux machine or Linux VM.** This often ends up simpler than fighting Windows-specific quirks if you want a permanent server.

Recommendation: Start with Docker Desktop. If you later decide you want a cleaner long-term server, move to Path A (Linux) using the same Compose files and the same data-layout principles.

3.15.2 B) About `systemctl` and systemd inside WSL2

On a classic Linux server, services are managed by `systemd` and the command `systemctl`. On WSL2, systemd support depends on your Windows/WSL configuration.

If you run:

```
systemctl status docker
```

and it fails, that does **not** automatically mean Docker is broken. In the Docker Desktop workflow, Docker is managed by Docker Desktop on Windows, not by systemd inside WSL.

What to do instead:

- open Docker Desktop and confirm it is running,
- in Ubuntu, run `docker ps` to see whether Docker responds.

3.15.3 C) Where should I store data on Windows + WSL2?

You can store files in:

- the Linux filesystem (inside WSL): e.g., `/srv/...`
- Windows drives mounted into WSL: e.g., `/mnt/c/...`, `/mnt/d/...`

Practical rule:

- For Docker volumes, databases, and indexes: prefer the Linux filesystem (`/srv`) for performance and fewer permission issues.
- For large media archives you already have on Windows disks: you can mount/read them from `/mnt/d/...`, but test performance.

3.15.4 D) Accessing your services from another device

In most Docker Desktop setups, when you publish a port (e.g., `8080:8080`), Windows exposes that port on your local network interface.

Typical workflow:

- First test locally: open your browser on the same PC and use `http://localhost:PORT`.
- Then test from another device on the same network: use `http://WINDOWS_IP:PORT`.

If it does not work from another device:

- check the Windows firewall rules,
- ensure Docker Desktop is running,
- verify the container really publishes the port (`docker ps` shows port mappings).

3.15.5 E) WSL2 resource limits (CPU/RAM/disk)

WSL2 uses Windows resources. On some systems, you may want to limit its maximum RAM/CPU usage, especially if your laptop becomes slow.

This can be configured via a `.wslconfig` file in your Windows user home directory. If you do not have performance problems, ignore this. If you do, this is a good first tuning knob.

3.15.6 F) Backups on Windows + WSL2 (simple advice)

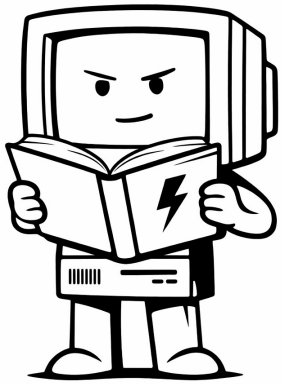
Even in a beginner setup, plan backups early:

- keep your persistent folders under a single root (we use `/srv`),
- periodically copy that folder to an external drive or another machine.

Important: if you store your Docker data in the Linux filesystem inside WSL, make sure your backup method can access it reliably (e.g., exporting/copying from inside the Ubuntu terminal).

Chapter 4

How to read this manual: YAML and Docker Compose (just enough theory)



4.1 Goal (what you will achieve)

This chapter gives you the minimum theory needed to follow the rest of the manual confidently. By the end, you will:

- understand what a `docker-compose.yml` file is and why we use it,
- recognize the most common YAML patterns (indentation, lists, key/value),
- know what `services`, `ports`, `volumes`, and `environment` mean,
- understand what `.env` is and why we keep passwords there,

- know what MariaDB is and when you need a database,
- avoid the most common beginner mistakes (especially indentation and permissions).

If YAML looks like mysterious hieroglyphs: that is normal. After this chapter, you will be able to read Compose files like a recipe. You do not need to become a chef.

4.2 What is YAML?

YAML is a human-friendly text format used to describe structured information. In this manual, we use YAML because Docker Compose uses it.

Three rules that explain 90% of YAML

1. **Indentation matters.** YAML uses spaces to show hierarchy.
2. **Key: value** pairs define settings.
3. **Lists use dashes** (`- item`).

Example: key/value and indentation

```
person:
  name: Mattia
  role: Admin
```

Here, `name` and `role` belong "inside" `person` because they are indented.

Example: a list

```
shopping_list:
  - pasta
  - tomatoes
  - coffee
```

What does # mean?

A `#` starts a **comment**. Comments are ignored by the computer. They are there for humans.

```
# This is a comment. It does nothing, but it helps you understand the file.
```

YAML is picky, but not evil. If something breaks, the most common cause is indentation (missing spaces), not your life choices.

4.3 What is Docker Compose (in plain language)?

Docker Compose is a tool that reads a file called `docker-compose.yml` and starts a set of containers for you.

Think of it like a playlist:

- the file lists which services you want (PhotoPrism, Navidrome, etc.),
- Compose starts them in a consistent way,
- you can stop them, restart them, and upgrade them reliably.

Why we use Compose in this manual

Without Compose, you would type long Docker commands every time. With Compose, the setup becomes:

- **reproducible** (you can move it to another machine),
- **documented** (the file explains what you run),
- **upgrade-friendly** (pull new images and restart).

4.4 A Compose file is a "service recipe"

Most of our Compose files follow the same pattern:

```
services:
  some-service:
    image: some/image:tag
    ports:
      - "1234:1234"
    environment:
      KEY: "value"
    volumes:
      - /host/path:/container/path
```

Let's translate the important words.

4.5 Core blocks you will see everywhere

4.5.1 `services`

A `service` is one container (or sometimes a group of containers with one role). In our manual:

- PhotoPrism is a service,
- MariaDB (database) is another service,
- Navidrome is a service,
- Ollama and Open WebUI are services.

4.5.2 `image`

`image` tells Docker what software to run.

- An **image** is like a packaged application template.
- A **container** is a running instance of that image.

Example:

```
image: photoprism/photoprism:latest
```

4.5.3 `ports`

Ports let your browser or apps reach a service.
Example:

```
ports:
  - "2342:2342"
```

How to read it:

- left side = host (your machine) port,
- right side = container port (inside the service).

So `"2342:2342"` means:

- open port 2342 on your machine,
- forward it to port 2342 inside the container.

4.5.4 `volumes`

Volumes are the most important concept for data safety.
Example:

```
volumes:
  - /srv/photos/originals:/photoprism/originals
```

How to read it:

- left side = real folder on your disk (persistent),
- right side = folder inside the container (where the app expects files).

Why it matters:

- You can delete/recreate containers without losing data.
- Backups become simple: you back up `/srv/...`.

Repeat this mantra: containers are disposable; data is not. If you get volumes right, you win.

4.5.5 `environment`

Environment variables are configuration settings given to the application.
Example:

```
environment:
  PHOTOPRISM_ADMIN_PASSWORD: "a-strong-password"
```

Why we use them:

- They keep configuration close to the Compose file.
- They are the standard way many containers are configured.

4.6 What is a `.env` file (and why we use it)?

A `.env` file sits next to `docker-compose.yml` and stores variables.

Example:

```
PHOTOPRISM_ADMIN_PASSWORD=CHANGE_THIS
MARIADB_PASSWORD=CHANGE_THIS_TOO
```

Then, in YAML, we reference them like:

```
PHOTOPRISM_ADMIN_PASSWORD: ${PHOTOPRISM_ADMIN_PASSWORD}
```

Why we do it:

- keeps passwords out of the YAML (cleaner),
- makes it easier to share Compose files without leaking secrets,
- centralizes configuration in one place.

4.7 What is MariaDB, and why does PhotoPrism need it?

MariaDB is a database server (similar to MySQL). A database is a structured storage system for:

- indexes,
- metadata,
- search tables,
- application state.

Some apps can store everything in simple files; others work better with a database. PhotoPrism can use SQLite or MySQL/MariaDB, but for larger libraries and better performance, MariaDB is commonly used.

4.8 Useful words you will see later (and what they mean)

This manual tries to stay beginner-friendly, but some technical words are unavoidable. Here are the ones that appear in later chapters, explained in plain language.

4.8.1 MySQL vs MariaDB

MySQL and **MariaDB** are very similar database servers. MariaDB started as a community-developed alternative to MySQL and is widely used. In practice, many applications treat them as interchangeable for common use.

What matters for you:

- The "database service" is the container that stores structured data (metadata, indexes, users, settings).
- When a config says `DB_HOST` or `MYSQL_HOST`, it means: *the name of the database service on the Docker network.*

Example:

```
MYSQL_HOST: db
```

This means: "connect to the container named `db` (service name), not to an external server."

4.8.2 Redis and Memcached (fast helpers)

Redis and **Memcached** are small services used as memory caches. They help performance and reduce database load.

In this manual:

- Nextcloud uses Redis for locking/caching (helps stability, especially with many files).
- Seafile often uses Memcached for performance.

You do not "use" them directly as a user; they are internal helpers.

4.8.3 Cron and "background jobs"

Some apps need periodic tasks: cleanup, indexing, sending notifications, updating shares, etc. This is often called **background jobs**.

In Nextcloud, running background jobs properly matters a lot. That is why we include a dedicated `cron` container.

4.8.4 HTTP vs HTTPS (TLS): the safety layer

HTTP is plain web traffic. **HTTPS** is encrypted web traffic (HTTP + TLS).

Practical rule:

- On your local network (LAN) and especially when using VPN, HTTP is often acceptable for beginners.
- If you expose services to the public internet, you must use HTTPS (TLS) and a proper reverse proxy.

If you are unsure: do not expose your services publicly yet. Use VPN first. It is safer and simpler.

4.8.5 Domains, hostnames, and "trusted domains"

A **hostname** is a human-friendly name for a machine (e.g., `server`, `nextcloud`). A **domain** is a public name you can buy (e.g., `yourname.com`).

Trusted domains are a security feature in apps like Nextcloud: they refuse requests from unknown hostnames to prevent certain attacks. That is why you must list the addresses you use (e.g., `localhost`, `SERVER_IP`, or your domain).

4.8.6 UID/GID (why permissions sometimes hurt)

Linux permissions are based on user IDs:

- **UID**: user ID
- **GID**: group ID

Some containers run as a specific internal user. If file permissions do not match, you may see "permission denied" errors. That is why some services support setting `UID/GID` explicitly.

4.8.7 JWT (a shared secret for trusted communication)

JWT stands for JSON Web Token. In our context, you do not need to learn the standard. You only need the practical meaning:

- Some integrations (like OnlyOffice) require a **shared secret**.
- Both services must use the *same secret* to trust each other.
- This is often configured as `JWT_SECRET`.

Practical rule: choose a long random string and keep it consistent across the services that need it.

4.8.8 Reverse proxy (optional, advanced)

A **reverse proxy** is a gateway in front of your services. It can:

- route traffic to the correct container,
- provide HTTPS/TLS,
- centralize security headers and access rules.

We do not require a reverse proxy in this manual, because it is optional and adds complexity. If you later want public access with a domain, a reverse proxy becomes important.

4.9 Finding your server address (IP): the one thing you need for mobile access

Many chapters say: open `http://SERVER_IP:PORT`. This section explains what `SERVER_IP` means and how to find it.

What is `SERVER_IP`?

It is the local network address of the machine running your services (PhotoPrism, Navidrome, etc.). If your phone is on the same Wi-Fi, it can reach that address directly.

You do not need to understand networking deeply. You only need to find the IP once, then reuse it everywhere.

4.9.1 Linux (Path A)

In a Linux terminal, run one of these:

```
hostname -I
```

or:

```
ip a
```

Look for an address like `192.168.x.x` or `10.x.x.x`. That is usually your LAN IP.

4.9.2 Windows (Path B)

Open PowerShell and run:

```
ipconfig
```

Look for `IPv4 Address` under your active network adapter (Wi-Fi or Ethernet).

4.9.3 Quick sanity check

From another device on the same Wi-Fi (phone/tablet/laptop), open a browser and try:

```
http://SERVER_IP:2342
```

If PhotoPrism loads, you found the right IP.

Common confusion: multiple IP addresses

Sometimes you will see more than one IP address (Ethernet + Wi-Fi + VPN). Use the one that matches the network you are currently using (for example, the Wi-Fi subnet).

4.10 VPN: the safe way to access your server from outside

Sooner or later you will want to access your services when you are not at home (or not on the same Wi-Fi). There are two broad approaches:

- **Expose services publicly** (open ports on the router, set up a domain, reverse proxy, TLS).
- **Use a VPN** (recommended for beginners).

4.10.1 What is a VPN (plain language)?

A VPN (Virtual Private Network) creates an encrypted "tunnel" between your device (phone/laptop) and your home network (or server). When connected, your device behaves *as if it were inside your home network*. That means:

- you can use the same addresses as at home, e.g., `http://SERVER_IP:2342`,
- you do not need to expose every service to the public internet.

4.10.2 Why this manual prefers VPN first

Public exposure can be done safely, but it has more moving parts and more ways to misconfigure things. A VPN is often:

- simpler,
- safer by default,
- easier to reason about ("only my devices can enter").

If your goal is "access my stuff from anywhere", a VPN is usually the shortest path that does not turn into a security project.

4.10.3 How VPN fits into this manual

Our default workflow is:

- **Local first:** make everything work on your local network (LAN/Wi-Fi).
- **Then remote:** add a VPN so you can access the same services while travelling.

4.10.4 Do we teach VPN setup here?

We give the concept here to avoid confusion, but a full VPN setup depends on your environment (router, ISP restrictions, CGNAT, existing university VPN, etc.).

For that reason, the step-by-step VPN setup is placed in a separate optional appendix: Appendix C.

4.11 Common beginner mistakes (and how to avoid them)

4.11.1 Mistake 1: indentation errors in YAML

If Compose fails with a YAML parsing error:

- check that indentation uses spaces (not tabs),
- check that lists use `-` and align correctly,
- when in doubt, compare to a known working example from this manual.

4.11.2 Mistake 2: editing the wrong file in the wrong folder

When we say:

```
cd /srv/docker/compose/photoprism
```

we mean: **go into that folder first**, then run Compose there. Otherwise, Docker might run a different project than you think.

4.11.3 Mistake 3: storing data inside the container

If your `volumes:` are missing or wrong, your data may disappear when the container is recreated. Always check that important folders map to `/srv/...` on the host.

4.11.4 Mistake 4: confusing host ports and container ports

If you map `"1234:2342"`, you must open `http://localhost:1234` in your browser. The left side is the port you use from outside.

4.11.5 Mistake 5: permission issues on mounted folders

If an app cannot write to a volume:

- ensure the folder exists,
- ensure your user owns it (or the container is configured with matching UID/GID),
- on Linux, a common fix is:

```
sudo chown -R $USER:$USER /srv
```

4.12 The three Compose commands you will use all the time

Start services

```
docker compose up -d
```

Stop services

```
docker compose down
```

See what is running

```
docker ps
```

Read logs (your best friend when things fail)

```
docker compose logs -f --tail=100
```

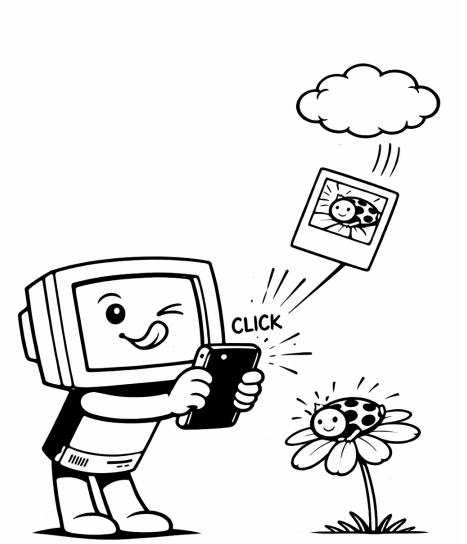
When something fails: do not guess. Check logs. Logs look intimidating, but they usually contain the exact clue you need.

4.13 Next step

Now you can read the Compose files in the service chapters like a recipe. In Appendix B, you will also find all Compose files collected in one place for easy copy/paste.

Chapter 5

PhotoPrism: your private photo library



5.1 Goal (what you will achieve)

In this chapter you will install PhotoPrism, a self-hosted photo management app that helps you:

- browse your photo archive from any browser on your network,
- import and index photos without handing them to a third party,
- search, tag, and organize your library with modern features.

This chapter is where the project starts to feel real. Once PhotoPrism opens in your browser, you are no longer "installing Docker"—you are building your own digital home.

5.2 What is PhotoPrism (plain language)

PhotoPrism is a web app you host yourself. It scans your photo folders, builds an index, and lets you browse and search your library from a clean web interface. It runs well in Docker, and the recommended way to run it is with Docker Compose. :contentReference[oaicite:1]index=1

5.3 Prerequisites

- Docker and Docker Compose working (Chapter 2 or 3).
- A folder where your photos will live (we use `/srv/photos`).
- A bit of patience for the first indexing (it can take time on large libraries).

5.4 Folder layout (what goes where)

We will keep things simple and predictable:

- `/srv/photos/originals` — your original photo/video files (the important part!)
- `/srv/photos/import` — optional folder for importing new files into PhotoPrism
- `/srv/photos/storage` — PhotoPrism internal storage (thumbnails, index, cache)
- `/srv/docker/compose/photoprism` — Compose file and environment file

Create folders (Linux path; on WSL2, run the same inside Ubuntu):

```
mkdir -p /srv/docker/compose/photoprism
mkdir -p /srv/photos/{originals,import,storage,mariadb}
```

Do not store your originals inside the container. Keep them in /srv/photos/originals so you can back them up and move them whenever you want.

5.5 Step 1 — Create the Compose project

Go to the PhotoPrism Compose folder:

```
cd /srv/docker/compose/photoprism
```

5.5.1 1.1 Create an environment file

Create `.env` (this keeps passwords out of the YAML):

```
nano .env
```

Paste and edit:

```
# --- PhotoPrism admin login ---
PHOTOPRISM_ADMIN_USER=admin
PHOTOPRISM_ADMIN_PASSWORD=CHANGE_THIS_TO_A_LONG_PASSWORD

# --- Database (MariaDB) ---
MARIADB_DATABASE=photoprism
MARIADB_USER=photoprism
```


```
MARIADB_PASSWORD=CHANGE_THIS_DB_PASSWORD
MARIADB_ROOT_PASSWORD=CHANGE_THIS_ROOT_PASSWORD

# --- Optional: match your Linux user (helps avoid permission issues) ---
# On Linux, these can often be left as-is if your user is UID/GID 1000.
# On Windows/macOS, setting them explicitly can help if you see permission errors.
PHOTOPRISM_UID=1000
PHOTOPRISM_GID=1000
```

Why this matters:

- PhotoPrism needs an admin password.
- MariaDB needs credentials.
- Keeping secrets in `.env` is cleaner and safer than hardcoding them in YAML.

Yes, set real passwords. Future-you will thank present-you.

5.5.2 1.2 Create `docker-compose.yml`

Create the Compose file:

```
nano docker-compose.yml
```

Paste:

```
services:
  mariadb:
    image: mariadb:11
    container_name: photoprism-mariadb
    restart: unless-stopped
    environment:
      MARIADB_DATABASE: ${MARIADB_DATABASE}
      MARIADB_USER: ${MARIADB_USER}
      MARIADB_PASSWORD: ${MARIADB_PASSWORD}
      MARIADB_ROOT_PASSWORD: ${MARIADB_ROOT_PASSWORD}
    volumes:
      - /srv/photos/mariadb:/var/lib/mysql

  photoprism:
    image: photoprism/photoprism:latest
    container_name: photoprism
    restart: unless-stopped
    depends_on:
      - mariadb
    ports:
      - "2342:2342"
    environment:
      PHOTOPRISM_ADMIN_USER: ${PHOTOPRISM_ADMIN_USER}
      PHOTOPRISM_ADMIN_PASSWORD: ${PHOTOPRISM_ADMIN_PASSWORD}

      # Database config (PhotoPrism supports sqlite or mysql; MariaDB uses mysql driver
          ↪ )
      PHOTOPRISM_DATABASE_DRIVER: "mysql"
      PHOTOPRISM_DATABASE_SERVER: "mariadb:3306"
      PHOTOPRISM_DATABASE_NAME: ${MARIADB_DATABASE}
      PHOTOPRISM_DATABASE_USER: ${MARIADB_USER}
```

```
    PHOTOPRISM_DATABASE_PASSWORD: ${MARIADB_PASSWORD}

    # File permissions (often helps on Linux; may be useful on Windows/macOS too)
    PHOTOPRISM_UID: ${PHOTOPRISM_UID}
    PHOTOPRISM_GID: ${PHOTOPRISM_GID}

    # Optional quality-of-life settings:
    PHOTOPRISM_SITE_URL: "http://localhost:2342/"
    PHOTOPRISM_ORIGINALS_LIMIT: 5000

  volumes:
    - /srv/photos/originals:/photoprism/originals
    - /srv/photos/import:/photoprism/import
    - /srv/photos/storage:/photoprism/storage
```

Notes:

- PhotoPrism's default port is `2342`. :contentReference[oaicite:2]index=2
- Database variables follow PhotoPrism config options for MySQL/MariaDB. :contentReference[oaicite:3]index=3

If YAML feels fragile: it is. Indentation matters. If something fails, it is usually one missing space, not a life problem.

5.6 Step 2 — Start PhotoPrism

Start the stack:

```
docker compose up -d
```

Check containers:

```
docker ps
```

5.6.1 View logs (first debugging tool)

If something does not start:

```
docker compose logs -f --tail=100
```

5.7 Step 3 — Open PhotoPrism in your browser

On the same machine:

```
http://localhost:2342
```

From another device on the same network:

```
http://SERVER_IP:2342
```

Log in with the admin user/password from your `.env` file.

If the login page loads: you are already winning. Most "server projects" fail before this moment.

If it works on `localhost` but not from your phone, the usual culprit is the firewall (especially on Windows). This is normal – see the "Access from mobile" section and the VPN appendix (Appendix C).

5.8 Step 4 — Add photos and index

5.8.1 Option A: put files directly in originals

Copy photos into:

```
/srv/photos/originals
```

Then in PhotoPrism, use the web interface to index (or re-index) the library.

5.8.2 Option B: use the import folder

Copy files into:

```
/srv/photos/import
```

Then use PhotoPrism's import workflow in the UI.

5.9 Verification checklist

You are done when:

- `docker ps` shows `photoprism` and `photoprism-mariadb` running,
- `http://localhost:2342` loads the PhotoPrism login page,
- you can log in and see the PhotoPrism dashboard,
- after adding photos, indexing completes and photos appear.

5.10 Common pitfalls (and quick fixes)

5.10.1 1) "Cannot connect to database"

Usually the database is not ready yet, or credentials do not match.

- Check logs: `docker compose logs -f -tail=100`
- Ensure `PHOTOPRISM_DATABASE_*` matches the MariaDB variables in `.env`.
- Restart after fixes: `docker compose down` then `docker compose up -d`

PhotoPrism troubleshooting notes commonly point to network/database connectivity and correct host/port. :contentReference[oaicite:4]index=4

5.10.2 2) Permission problems on mounted folders

Symptoms: PhotoPrism cannot write to `/photoprism/storage` or fails creating files.

- Make sure `/srv/photos/storage` exists and is writable.
- On Linux, run: `sudo chown -R $USER:$USER /srv/photos`
- If needed, set `PHOTOPRISM_UID` and `PHOTOPRISM_GID` explicitly. :contentReference[oaicite:5]index=5

5.10.3 3) "It is slow" on first run

The first indexing generates thumbnails and builds the database. This is normal.

- Let it run.
- Keep the machine on.
- Consider indexing overnight for large libraries.

Slow is not broken. The first scan is the hardest; later updates are much faster.

5.10.4 4) Wrong URL/port

Default is `http` on port `2342`. :contentReference[oaicite:6]index=6

- Local: `http://localhost:2342`
- Network: `http://SERVER_IP:2342`

5.10.5 5) Exposing PhotoPrism to the public internet too early

Do not expose PhotoPrism publicly until you understand reverse proxies and TLS. Start with LAN or VPN access. (We keep public exposure optional in this manual.)

5.11 Access from mobile and tablet

You do not need a special app to use PhotoPrism. A browser is enough.

On the same Wi-Fi (local network)

- Find the IP address of the machine running PhotoPrism (the "server").
- On your phone/tablet browser, open:

```
http://SERVER_IP:2342
```

Outside your home (recommended: VPN)

If you want access while travelling, the safest beginner approach is:

- connect your phone to your VPN,
- then use the same address:

```
http://SERVER_IP:2342
```

Start with local Wi-Fi first. Once it works locally, adding VPN access is usually just one extra step, not a whole new project.

Make it feel like an app (Add to Home Screen)

Most browsers let you add the PhotoPrism page to your home screen. This creates an icon and opens it in an "app-like" full-screen mode. Look for **Share** → **Add to Home Screen** (wording may vary).

5.11.1 Uploading photos from a phone

PhotoPrism itself does not upload photos directly from a mobile phone. Images must first arrive on the server filesystem.

There are several possible ways to achieve this.

One option is to use synchronization tools such as **Syncthing**, which can automatically send photos from a phone to a folder on the server. However, Syncthing operates outside the Docker stack used in this manual, so it is not covered here.

In this guide we instead rely on solutions that are already part of the infrastructure described later in the book:

- **Nextcloud**, which provides automatic photo upload through its mobile application.
- **Seafile**, which offers similar synchronization capabilities.

Both options integrate naturally with the server setup presented in the following chapters.

Practical tip: For large photo collections, using a dedicated sync tool such as Syncthing can be extremely efficient. Advanced users may want to explore this approach independently.

5.12 Next step

Next, we install Navidrome (music server). It is simpler than PhotoPrism: one container, one library folder, one web UI.

Chapter 6

Navidrome: your private music server

6.1 Goal (what you will achieve)

In this chapter you will install Navidrome, a lightweight self-hosted music server. By the end, you will have:

- a web music library you can access from a browser on your network,
- a clean folder layout for your music files and Navidrome data,
- a simple Compose setup you can start/stop/upgrade reliably.

This one is a morale boost: Navidrome is usually fast to deploy and feels instantly useful.

6.2 What is Navidrome (plain language)

Navidrome is a music streaming server you run yourself. You point it to your music folder, it builds an index, and it serves a modern web UI. It also supports Subsonic-compatible clients (many mobile apps can connect to it), but you can start with the browser and keep it simple.

6.3 Prerequisites

- Docker and Docker Compose working (Chapters 2A or 2B).
- A folder with your music library (we use `/srv/music/library`).

6.4 Folder layout

We will use:

- `/srv/music/library` — your music files (the important part)
- `/srv/music/data` — Navidrome database and cache (persistent)
- `/srv/docker/compose/navidrome` — Compose files

Create folders (Linux path; on WSL2, run inside Ubuntu):

```
mkdir -p /srv/docker/compose/navidrome
mkdir -p /srv/music/{library,data}
```

Same principle as always: your music lives on disk (`/srv/music/library`), not "inside Docker".

6.5 Step 1 — Create the Compose project

Go to the Navidrome compose folder:

```
cd /srv/docker/compose/navidrome
```

6.5.1 1.1 Create a simple environment file (recommended)

Create `.env`:

```
nano .env
```

Paste (you can keep defaults for now):

```
# --- Navidrome basics ---
ND_SCANSCHEDULE=1h
ND_LOGLEVEL=info

# --- Optional: set timezone for correct timestamps ---
TZ=Europe/Madrid
```

What this does:

- `ND_SCANSCHEDULE` tells Navidrome how often to scan for new files,
- `TZ` helps with correct time display and logs.

6.5.2 1.2 Create `docker-compose.yml`

Create the Compose file:

```
nano docker-compose.yml
```

Paste:

```
services:
  navidrome:
    image: deluan/navidrome:latest
    container_name: navidrome
    restart: unless-stopped
    ports:
      - "4533:4533"
    environment:
      ND_SCANSCHEDULE: ${ND_SCANSCHEDULE}
      ND_LOGLEVEL: ${ND_LOGLEVEL}
      TZ: ${TZ}
    volumes:
      - /srv/music/data:/data
      - /srv/music/library:/music:ro
```

How to read it:

- Port `4533` will be the web UI.
- `/srv/music/data` stores Navidrome's database (persistent).
- `/srv/music/library` is mounted as `:ro` (read-only) to protect your originals.

Mounting the music folder as read-only is a nice beginner safety belt: the server can index and stream, but it cannot accidentally modify your files.

6.6 Step 2 — Start Navidrome

Start the container:

```
docker compose up -d
```

Check status:

```
docker ps
```

If something looks wrong, check logs:

```
docker compose logs -f --tail=100
```

6.7 Step 3 — Open Navidrome in your browser

On the same machine:

```
http://localhost:4533
```

From another device on the same network:

```
http://SERVER_IP:4533
```


6.7.1 First login

On the first run, Navidrome typically asks you to create an admin user in the web UI.

If you see the login/setup screen: you are done with the hard part. From here it is just adding music.

6.8 Step 4 — Add music and let it scan

Copy your music into:

```
/srv/music/library
```

Navidrome will scan automatically according to `ND_SCANSCHEDULE`. If you add a lot of files, give it some time to index them.

6.9 Verification checklist

You are done when:

- `docker ps` shows `navidrome` running,
- `http://localhost:4533` loads the Navidrome UI,
- you can create/login to the admin user,
- after adding music, albums and tracks appear.

6.10 Common pitfalls (and quick fixes)

6.10.1 1) No music appears

Most common causes:

- The library folder is empty, or you copied files to the wrong path.
- The container cannot see your files because the volume path is wrong.

Quick checks:

- Confirm your files are in `/srv/music/library` on the host.
- Confirm the container mount exists:

```
docker exec -it navidrome ls -la /music | head
```

6.10.2 2) Permission problems

Because we mount the music library read-only, permission issues are usually about the data folder:

- Ensure `/srv/music/data` exists.
- Ensure it is writable by your user (and thus by Docker):

```
sudo chown -R $USER:$USER /srv/music
```

6.10.3 3) Port already in use

If port 4533 is already used by another service, change the left side of the port mapping.

Example: use port 4534 on the host:

```
ports:
  - "4534:4533"
```

Then access:

```
http://localhost:4534
```

6.10.4 4) Access from another device does not work

If it works on `localhost` but not from other devices:

- check Windows firewall (Path B),
- confirm you are using the correct IP address,
- ensure both devices are on the same network (or VPN).

6.11 Access from mobile and tablet

Navidrome works in a browser and also supports many mobile music apps.

On the same Wi-Fi (local network)

On your phone/tablet browser, open:

```
http://SERVER_IP:4533
```

Outside your home (recommended: VPN)

Connect your device to your VPN, then open the same address:

```
http://SERVER_IP:4533
```

Option: use a dedicated music app (Subsonic-compatible)

Navidrome supports the Subsonic API, which means many mobile music players can connect to it. If you want that experience later, you will typically need:

- server address: `http://SERVER_IP:4533`
- username/password: the Navidrome user you created

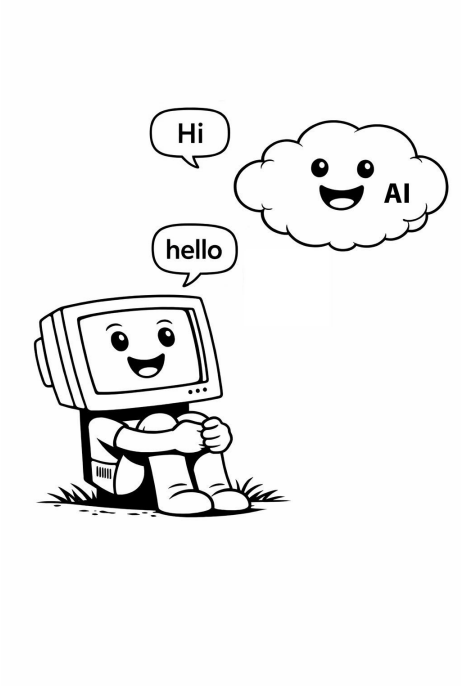
Browser access is the simplest path. Once everything works, dedicated apps are a nice upgrade—not a requirement.

6.12 Next step

Next, we will add **Local AI**. This is the most hardware-dependent part of the manual: it can run on CPU, but a GPU makes it much faster and more pleasant to use.

Chapter 7

Local AI: private assistants on your own machine



7.1 Goal (what you will achieve)

This chapter explains what "local AI" means and how to run it privately on your own hardware. It also gives you a practical mental model of different types of AI models (chat, code, writing, vision), so you can choose the right tool without getting lost in hype.

By the end, you will:

- understand what it means to run an AI model locally (and why it matters),
- know the difference between the most common model families (general chat vs. coding vs. writing),
- have a simple local setup in Docker: **Ollama** (model runner) + **Open WebUI** (a friendly interface),

- be able to load at least one model and run your first private conversation.

If you have ever thought "AI is useful, but I do not want to upload my private documents", this chapter is for you. Local AI is not magic. It is just software running on your computer.

7.2 What is "local AI" (plain language)

Most people use AI through a website. Your text is sent to a remote server, processed there, and the answer comes back.

Local AI means:

- the model runs on your own machine (server, desktop, workstation),
- prompts and data stay local (unless you explicitly send them out),
- you control upgrades, storage, and access.

This can be slower than cloud AI if your hardware is modest, but the privacy and control are often worth it.

7.3 Hardware expectations (honest and beginner-friendly)

You can run local AI on many machines, but performance depends mostly on:

- **GPU and VRAM** (biggest difference for speed and model size),
- **RAM** (helps with multitasking and larger contexts),
- **CPU** (fine for small models; slower for bigger ones).

A practical rule of thumb

- **No GPU**: run smaller models, expect slower responses.
- **GPU with 8–12 GB VRAM**: a comfortable entry point for many "7B–8B" models.
- **GPU with 16–24 GB VRAM**: more room for larger models or higher quality settings.

You do not need a monster GPU to start. The goal is usefulness, not winning a benchmark contest.

7.4 Model types: which model for which job?

AI models are not all the same. Many are specialized.

7.4.1 General chat models ("good at many things")

These are the everyday assistants: summarization, brainstorming, explanations, planning. Good when you want a balanced model.

Typical examples (names change over time, but the idea is stable):

- **Llama (Instruct)** family: general-purpose chat, often strong overall.
- **Qwen (Instruct)** family: strong general reasoning, often good multilingual support.
- **Mistral / Mixtral** family: fast, efficient, often good quality for their size.

7.4.2 Coding models ("better at code than prose")

These are tuned to write, debug, and explain code. They are especially useful for:

- programming assistance,
- reading logs and config files,
- generating Docker Compose files and scripts,
- refactoring.

Typical examples:

- **Qwen2.5-Coder**: strong for code generation and reasoning about code.
- **DeepSeek-Coder**: often strong for coding tasks and technical instructions.
- **Code Llama**: a coding-tuned variant of Llama models.

If your main goal is "help me with code", start with a coding model. A general chat model can code, but a coding model usually feels more confident and consistent.

7.4.3 Writing-focused models ("tone, clarity, structure")

Some models feel especially good at:

- rewriting text,
- improving clarity,
- producing professional tone,
- drafting letters, proposals, or documentation.

In practice, many "general chat" models already do this well, but you may prefer models that feel smoother or more stylistically consistent. (You can also tune outputs via prompts: "be concise", "be friendly", etc.)

7.4.4 Reasoning vs. speed (small models vs. larger models)

Model size often trades off with speed:

- smaller models: faster, cheaper, sometimes less reliable on complex reasoning,
- larger models: slower, more memory, often better reasoning and writing quality.

A good setup is to have:

- one **fast small model** for quick tasks,
- one **stronger model** for harder tasks (research, long writing, complex debugging).

7.5 Our local AI stack: Ollama + Open WebUI

To keep the manual simple, we use:

- **Ollama**: downloads and runs local models,
- **Open WebUI**: a browser interface to chat with those models.

This stack is popular because it is:

- easy to deploy with Docker,
- easy to update,
- flexible: you can add models later without changing the infrastructure.

7.6 Folder layout

We keep AI data in one place:

- `/srv/ai/ollama` — model files
- `/srv/ai/openwebui` — Open WebUI data
- `/srv/docker/compose/local-ai` — Compose files

Create folders:

```
mkdir -p /srv/docker/compose/local-ai
mkdir -p /srv/ai/{ollama,openwebui}
```

7.7 Step 1 — Create the Compose project

Go to the folder:

```
cd /srv/docker/compose/local-ai
```

7.7.1 1.1 Create `docker-compose.yml`

Create the file:

```
nano docker-compose.yml
```

Paste:

```
services:
  ollama:
    image: ollama/ollama:latest
    container_name: ollama
    ports:
      - "11434:11434"
    volumes:
      - /srv/ai/ollama:/root/.ollama
    restart: unless-stopped

  open-webui:
    image: ghcr.io/open-webui/open-webui:main
    container_name: open-webui
```


```
    ports:
      - "3000:8080"
    environment:
      OLLAMA_BASE_URL: "http://ollama:11434"
    volumes:
      - /srv/ai/openwebui:/app/backend/data
    depends_on:
      - ollama
    restart: unless-stopped
```

7.7.2 GPU note (optional)

If your server has an NVIDIA GPU and you want to use it, you may need to enable GPU support in Docker. This depends on your system (Linux vs Windows, driver setup). We keep GPU configuration out of the default Compose file to avoid breaking beginner installs.

Start CPU-only first. Once everything works, adding GPU acceleration is an upgrade, not a requirement.

7.8 Step 2 — Start the local AI stack

Start:

```
docker compose up -d
```

Check:

```
docker ps
```

Logs (if needed):

```
docker compose logs -f --tail=100
```

7.9 Step 3 — Open the web interface

Open in a browser on the same machine:

```
http://localhost:3000
```

From another device on the same network:

```
http://SERVER_IP:3000
```

First-time setup

Open WebUI typically asks you to create the first user (admin). Do that in the browser.

If you see the WebUI login screen, you are done with infrastructure. The rest is just choosing models.

7.10 Step 4 — Download your first model

You have two easy options:

Option A: download from the WebUI

Open WebUI usually provides a model management section where you can pull models.

Option B: download from the terminal (always works)

In your terminal:

```
docker exec -it ollama ollama pull llama3.1:8b
```

What this does: downloads a general-purpose "8B" model (good balance for many tasks). **Why this is a good first model:** it is usually small enough to run on many machines, but strong enough to be useful.

The first model download can take a while. That is normal. After that, chatting is instant compared to downloading.

7.11 Step 5 — Try three small practical prompts

Try these in Open WebUI to validate your setup:

General assistant

Summarize the main idea of digital sovereignty in 5 bullet points.

Writing help

Rewrite this paragraph to be clearer and friendlier, without changing meaning: [paste text].

Coding help

Explain what this Docker Compose service does and how to troubleshoot it: [paste YAML].

7.12 Verification checklist

You are done when:

- `docker ps` shows `ollama` and `open-webui` running,
- `http://localhost:3000` loads the Open WebUI page,
- you pulled at least one model successfully,
- you can chat with the model and get responses.

7.13 Common pitfalls (and quick fixes)

7.13.1 1) "WebUI loads, but no models appear"

- Pull a model explicitly:

```
docker exec -it ollama ollama pull llama3.1:8b
```

- Then refresh the WebUI.

7.13.2 2) "Ollama connection error"

Check the containers:

```
docker ps
```

Check logs:

```
docker compose logs -f --tail=100
```

Make sure the WebUI points to:

```
http://ollama:11434
```

7.13.3 3) Slow responses

This usually means CPU inference or a model that is too large for your hardware.

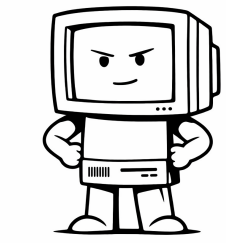
- Try a smaller model.
- If you have a GPU, consider enabling GPU support later.

Slow is not broken. Start small, get a win, then scale up.

7.13.4 4) Running out of disk space

Models can be large. If disk fills up:

- remove unused models,
- move `/srv/ai/ollama` to a larger disk and remap the volume.



7.14 Optional upgrade: enable NVIDIA GPU acceleration (big speed boost)

If you have an NVIDIA GPU, enabling it can make local AI dramatically faster. This section is optional: your setup works fine on CPU-only, but GPU is the difference between "usable" and "wow" for many models.

```
CPU inference
-------------
User -> Ollama container -> CPU


GPU inference
-------------
User -> Ollama container -> NVIDIA GPU
```

7.14.1 Goal (what you will achieve)

By the end of this section, you will:

- confirm your system sees the NVIDIA GPU,
- enable GPU access inside Docker for the `ollama` container,
- verify that Ollama is actually using the GPU.

If you already have a GPU: you are one small configuration step away from a huge upgrade.

7.14.2 Prerequisites (what must be true first)

- Your `ollama + open-webui` stack already works on CPU (you can open WebUI and chat).
- You have an NVIDIA GPU with drivers installed on the host system.

7.14.3 GPU Step A — Verify the GPU is visible on your system

Linux (Path A) Run on the host:

```
nvidia-smi
```

If you see a table with your GPU name and driver version, the host driver is OK.

Windows + WSL2 (Path B) First, verify in Windows (PowerShell or CMD):

```
nvidia-smi
```

Then verify inside Ubuntu (WSL terminal):

```
nvidia-smi
```

If `nvidia-smi` works in WSL too, WSL GPU support is active.

7.14.4 GPU Step B (Linux) — Install the NVIDIA Container Toolkit (one-time setup)

If you already have NVIDIA drivers working (GPU Step A), this is the missing piece that lets Docker containers see the GPU.

This looks long, but it is a one-time setup. Once done, every GPU-enabled container becomes dramatically faster.

Quickstart (Ubuntu/Debian)

Run the following commands on the server:

```
# Add NVIDIA Container Toolkit repository key
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey \
  | sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg

# Add the repository list
curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-
    ↪ toolkit.list \
  | sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list

# Install the toolkit
sudo apt update && sudo apt install -y nvidia-container-toolkit

# Configure Docker runtime and restart Docker
sudo nvidia-ctk runtime configure --runtime=docker
sudo systemctl restart docker
```

What this does (in one line each):

- Adds NVIDIA's package repository (so you can install the toolkit with `apt`).
- Installs the toolkit (`nvidia-container-toolkit`).
- Configures Docker so containers can use the NVIDIA runtime.
- Restarts Docker to apply the new configuration.

If this fails: NVIDIA occasionally changes repository details. In that case, follow the official NVIDIA Container Toolkit installation guide for your distribution.

This looks scary, but it is a standard requirement: "drivers on the host + toolkit for containers".

Windows + WSL2 (Path B): Docker Desktop settings

In Docker Desktop:

- Ensure Docker Desktop is running.
- Ensure WSL integration is enabled for your Ubuntu distro.

On most modern setups, GPU pass-through works automatically once WSL GPU support is enabled.

7.14.5 GPU Step C — Update the Compose file to request the GPU

Edit your local AI Compose file:

```
cd /srv/docker/compose/local-ai
nano docker-compose.yml
```

Inside the `ollama:` service, add the following block:

```
  deploy:
    resources:
      reservations:
        devices:
          - driver: nvidia
```

```
            count: all
            capabilities: [gpu]
```

So `ollama` becomes:

```
  ollama:
    image: ollama/ollama:latest
    container_name: ollama
    ports:
      - "11434:11434"
    volumes:
      - /srv/ai/ollama:/root/.ollama
    restart: unless-stopped
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [gpu]
```

YAML is indentation-sensitive. If it fails, it is usually one missing space, not your GPU.

7.14.6 GPU Step D — Restart the stack

Apply the change:

```
docker compose down
docker compose up -d
```

7.15 Verification checklist (GPU)

You are done when:

- the containers are running: `docker ps`
- you can still open WebUI: `http://localhost:3000`
- GPU appears inside the Ollama container (see below).

Verification command (works everywhere)

Run:

```
docker exec -it ollama nvidia-smi
```

You should see the same GPU table that appeared on the host. If you see the GPU table *from inside the container*, GPU pass-through is working.

If `docker exec ... nvidia-smi` works, you have GPU acceleration. That is the whole game.

7.16 Common pitfalls (GPU)

1) `nvidia-smi` works on host, but not inside container

On Linux, this usually means the NVIDIA container runtime/toolkit is missing or Docker was not restarted.

2) Windows works, but WSL `nvidia-smi` fails

That means WSL GPU support is not active yet. Update NVIDIA drivers and ensure WSL2 is enabled.

3) Docker Compose ignores the GPU block

Make sure you are using `docker compose` (plugin) and not an outdated legacy tool. Check:

```
docker compose version
```

4) Still slow even with GPU

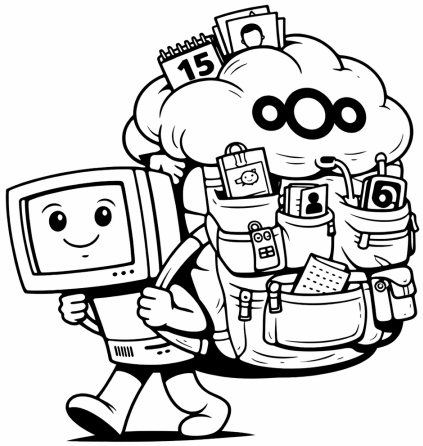
Some models are simply heavy. Try a smaller model first (7B/8B), then scale up once you confirm the GPU path works.

7.17 Next step

Next, we set up the private drive (file sync and organization). Later, you can also integrate local AI with your documents (RAG-style search over your own files), but first we build the stable storage layer.

Chapter 8

Private Drive (Option A): Nextcloud



8.1 Goal (what you will achieve)

In this chapter you will deploy Nextcloud as your private "Drive": file sync, sharing links, and a web interface for your files. Optionally, you can add OnlyOffice to edit documents directly in the browser.

By the end, you will have:

- a working Nextcloud instance reachable on your local network (or VPN),
- persistent storage under `/srv` (so upgrades do not delete your data),
- a clear understanding of the "web address" (URL) Nextcloud expects,
- **optional**: OnlyOffice document editing inside Nextcloud.

Nextcloud can look like a big ecosystem, but the core is simple: one web app + one database + your storage. We will build it step by step.

8.2 What is Nextcloud (plain language)

Nextcloud is a self-hosted file platform. You can:

- upload and organize files in the browser,
- sync files with desktop/mobile clients,
- share folders and links,
- optionally add apps (calendar, contacts, office editing, etc.).

It runs well in Docker, and the official Docker image supports basic auto-configuration and trusted domains. :contentReference[oaicite:0]index=0

8.3 Read this first: what to skip if you do not want online editing

This chapter is split into:

- **Core Drive (recommended baseline):** install Nextcloud and sync files.
- **Optional Office:** add OnlyOffice for browser-based document editing.

If you do **not** want online editing, follow Sections 8.5–8.8 and skip Section 8.12.

8.4 Prerequisites

- Docker and Docker Compose working (Chapter 2 or 3).
- A persistent data root (we use `/srv`).
- Local access (LAN/Wi-Fi). Optional: VPN for remote access (Appendix C).

8.5 The "web address" Nextcloud asks for

At some point, Nextcloud will ask for (or assume) a **web address** (URL). This is not "external IP". It simply means: *what address will you type in the browser to reach Nextcloud?*

This matters because Nextcloud uses it for link generation and security checks (trusted domains).

What should you put there?

Use the address you will actually use most of the time:

- **Local-only on the same machine:** `http://localhost:8080`
- **On your home LAN/Wi-Fi:** `http://SERVER_IP:8080`
- **On VPN (recommended for travel):** the same address you use on LAN, or your VPN IP/name

- **Public internet (advanced):** `https://cloud.yourdomain.com` (only after reverse proxy + TLS)

Start LAN/VPN first. Public internet access is optional and comes later, once everything works locally.

8.6 Folder layout

We keep Nextcloud predictable:

- `/srv/docker/compose/nextcloud` — Compose files
- `/srv/drive/nextcloud/html` — Nextcloud application data (apps/config)
- `/srv/drive/nextcloud/data` — user files (the important part)
- `/srv/drive/nextcloud/db` — database data
- `/srv/drive/nextcloud/redis` — redis data (optional, but recommended)

Create folders:

```
mkdir -p /srv/docker/compose/nextcloud
mkdir -p /srv/drive/nextcloud/{html,data,db,redis}
```

8.7 Core Drive install

8.7.1 Step 1 — Create the Compose project

Go to the Nextcloud compose folder:

```
cd /srv/docker/compose/nextcloud
```

8.7.2 Step 2 — Create `.env` (recommended)

Create:

```
nano .env
```

Paste and edit:

```
# --- Database (MariaDB) ---
MYSQL_DATABASE=nextcloud
MYSQL_USER=nextcloud
MYSQL_PASSWORD=CHANGE_THIS_DB_PASSWORD
MYSQL_ROOT_PASSWORD=CHANGE_THIS_ROOT_PASSWORD

# --- Nextcloud admin ---
NEXTCLOUD_ADMIN_USER=admin
NEXTCLOUD_ADMIN_PASSWORD=CHANGE_THIS_TO_A_LONG_PASSWORD

# --- The addresses you will use to access Nextcloud ---
# Space-separated. Add at least one: localhost, your LAN IP, and/or a local hostname.
NEXTCLOUD_TRUSTED_DOMAINS=localhost 127.0.0.1 SERVER_IP

# Timezone (optional)
TZ=Europe/Madrid
```


Why this matters:

- Nextcloud needs an admin user for first setup.
- The database needs credentials.
- Trusted domains avoid the "untrusted domain" warning and are supported by the official image. :contentReference[oaicite:1]index=1

Yes, set real passwords. Nextcloud becomes your personal data hub. Treat it like one.

8.7.3 Step 3 — Create `docker-compose.yml`

Create:

```
nano docker-compose.yml
```

Paste:

```
services:
  db:
    image: mariadb:11
    container_name: nextcloud-db
    restart: unless-stopped
    command: --transaction-isolation=READ-COMMITTED --binlog-format=ROW
    environment:
      MYSQL_DATABASE: ${MYSQL_DATABASE}
      MYSQL_USER: ${MYSQL_USER}
      MYSQL_PASSWORD: ${MYSQL_PASSWORD}
      MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
      TZ: ${TZ}
    volumes:
      - /srv/drive/nextcloud/db:/var/lib/mysql

  redis:
    image: redis:7-alpine
    container_name: nextcloud-redis
    restart: unless-stopped
    command: redis-server --appendonly yes
    volumes:
      - /srv/drive/nextcloud/redis:/data

  app:
    image: nextcloud:apache
    container_name: nextcloud
    restart: unless-stopped
    depends_on:
      - db
      - redis
    ports:
      - "8080:80"
    environment:
      # Auto-config on first start:
      MYSQL_HOST: db
      MYSQL_DATABASE: ${MYSQL_DATABASE}
      MYSQL_USER: ${MYSQL_USER}
      MYSQL_PASSWORD: ${MYSQL_PASSWORD}
      NEXTCLOUD_ADMIN_USER: ${NEXTCLOUD_ADMIN_USER}
```

```
    NEXTCLOUD_ADMIN_PASSWORD: ${NEXTCLOUD_ADMIN_PASSWORD}

    # Trusted domains (space-separated):
    NEXTCLOUD_TRUSTED_DOMAINS: ${NEXTCLOUD_TRUSTED_DOMAINS}

    TZ: ${TZ}
  volumes:
    - /srv/drive/nextcloud/html:/var/www/html
    - /srv/drive/nextcloud/data:/var/www/html/data

cron:
  image: nextcloud:apache
  container_name: nextcloud-cron
  restart: unless-stopped
  depends_on:
    - app
  entrypoint: /cron.sh
  volumes:
    - /srv/drive/nextcloud/html:/var/www/html
    - /srv/drive/nextcloud/data:/var/www/html/data
```

What this gives you:

- **Nextcloud** web app on port 8080.
- **MariaDB** database (recommended for real use).
- **Redis** cache/locking (helps performance and file locking).
- **Cron** container for background jobs (important for a healthy Nextcloud).

YAML is sensitive to indentation. If Compose refuses to start, it is usually one missing space, not a disaster.

8.7.4 Step 4 — Start Nextcloud

Start:

```
docker compose up -d
```

Check:

```
docker ps
```

Logs (first debugging tool):

```
docker compose logs -f --tail=120
```

8.7.5 Step 5 — Open Nextcloud (first login)

On the same machine:

```
http://localhost:8080
```

From another device on the same network:

```
http://SERVER_IP:8080
```


If auto-configuration works, you should land directly in the Nextcloud UI with your admin account. If you see an installer screen, follow it (database host is `db`).

If the page loads, you are already 80% done. Most of the pain is before the first web page appears.

8.8 Verification checklist (core)

You are done with the core when:

- `docker ps` shows `nextcloud`, `nextcloud-db`, `nextcloud-redis`, and `nextcloud-cron` running,
- `http://localhost:8080` loads,
- you can upload a file and download it again,
- Nextcloud does not complain about background jobs (cron).

8.9 Access from mobile and tablet

On the same Wi-Fi (local network)

Open:

```
http://SERVER_IP:8080
```

Outside your home (recommended: VPN)

Connect your device to VPN (Appendix C), then use the same address. This avoids exposing Nextcloud publicly.

Make it feel like an app

You can add Nextcloud to your home screen from your browser ("Add to Home Screen").

8.10 Common pitfalls (core)

8.10.1 1) "Accessing from an untrusted domain"

Add the address you are using to `NEXTCLOUD_TRUSTED_DOMAINS` in `.env` (space-separated), then restart:

```
docker compose down
docker compose up -d
```

Trusted domains are supported by the official image. :contentReference[oaicite:2]index=2

8.10.2 2) Permission issues on mounted folders

If Nextcloud cannot write to its data:

- confirm the folders exist under `/srv/drive/nextcloud/...`,
- ensure Docker can write to them.

A common Linux fix:

```
sudo chown -R $USER:$USER /srv/drive/nextcloud
```

8.10.3 3) Works on localhost but not from phone

Most common cause: firewall (especially on Windows in Path B). Verify you can reach `SERVER_IP` from your phone on the same Wi-Fi. If you need remote access, use VPN (Appendix C).

8.10.4 4) Background jobs warnings

Make sure the `cron` container is running:

```
docker ps
```

If needed, check logs:

```
docker logs --tail=100 nextcloud-cron
```

8.11 Remote access options (recommended order)

- **LAN only:** simplest and safest for beginners.
- **VPN:** safest way to access from anywhere (Appendix C).
- **Public internet:** advanced; requires domain + reverse proxy + TLS and careful hardening.

Do not rush to public exposure. A private cloud that works reliably on LAN/VPN is already a big win.

8.12 Optional: OnlyOffice integration (skip if you do not want online editing)

OnlyOffice adds browser-based editing of office documents inside Nextcloud. The common approach is:

- run OnlyOffice Docs (Document Server) as a separate container,
- install the OnlyOffice integration app inside Nextcloud,
- configure the connection (often with a shared JWT secret).

OnlyOffice provides an official integration guide for Nextcloud. :contentReference[oaicite:3]index=3

Optional means optional. Your Drive works perfectly without this. Add it only if you actually want browser editing.

8.12.1 Step O1 — Add OnlyOffice Docs to the Compose file

Edit your `docker-compose.yml` and add this service:

```
onlyoffice:
  image: onlyoffice/documentserver:latest
  container_name: onlyoffice
  restart: unless-stopped
  environment:
    TZ: ${TZ}
    JWT_ENABLED: "true"
    JWT_SECRET: "CHANGE_THIS_TO_A_LONG_RANDOM_SECRET"
  ports:
    - "8081:80"
```

Important:

- Choose a strong JWT secret and keep it consistent with Nextcloud's OnlyOffice settings.
- We expose OnlyOffice on port 8081 (LAN/VPN). Do not expose it publicly unless you know what you are doing.

Restart:

```
docker compose down
docker compose up -d
```

8.12.2 Step O2 — Install the OnlyOffice app in Nextcloud

In the Nextcloud web UI:

- go to **Apps**,
- search for **ONLYOFFICE**,
- install the official connector (app).

8.12.3 Step O3 — Configure the connector

In Nextcloud settings for ONLYOFFICE:

- Document Server address: `http://onlyoffice/` (internal Docker network) *or* `http://SERVER_IP:8081/` (LAN)
- JWT secret: the same value you set in the OnlyOffice container

Practical tip:

- If you use `http://onlyoffice/`, Nextcloud talks to OnlyOffice internally (recommended).
- From your browser, you still use Nextcloud normally on port 8080.

8.12.4 Verification (OnlyOffice)

You are done when:

- you can create a new document in Nextcloud,
- it opens in the OnlyOffice editor,
- saving works without errors.

8.12.5 Common pitfalls (OnlyOffice)

OnlyOffice “not available” / connection errors

Most common causes:

- wrong Document Server URL (internal vs external),
- JWT secret mismatch,
- mixed HTTP/HTTPS assumptions (if you later put Nextcloud behind HTTPS).

Self-signed certificates

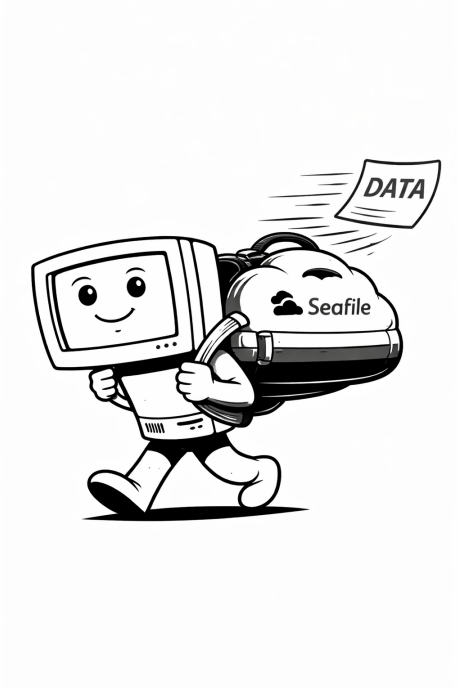
If you later move to HTTPS with self-signed certificates, OnlyOffice may reject them unless configured accordingly. :contentReference[oaicite:4]index=4

8.13 Next step

Next, we build the alternative Drive option: **Seafile** (Chapter 9). Then you can choose which one matches your needs best (or keep both as documented options).

Chapter 9

Private Drive (Option B): Seafile



9.1 Goal (what you will achieve)

In this chapter you will deploy Seafile as a fast, reliable private "Drive" focused on file syncing and libraries. Optionally, you can add OnlyOffice so documents can be edited in the browser.

By the end, you will have:

- a working Seafile instance reachable on your local network (or VPN),
- persistent storage under `/srv` (so upgrades do not delete your data),
- a clear understanding of the "web address" concept (what users type in their browser),
- **optional**: OnlyOffice document editing for Seafile.

If your main goal is "files and sync, without a huge ecosystem", Seafile is often the calm, efficient choice.

9.2 What is Seafile (plain language)

Seafile is a self-hosted file sync and sharing platform. Its core idea is **libraries**: you organize content into libraries, sync them to devices, and share them with others. It tends to feel lighter and faster than full-suite platforms when your focus is file syncing.

9.3 Read this first: what to skip if you do not want online editing

This chapter is split into:

- **Core Drive (recommended baseline):** install Seafile and sync files.
- **Optional Office:** add OnlyOffice for browser-based document editing.

  If you do **not** want online editing, follow Sections 9.5–9.8 and skip Section 9.11.

9.4 Prerequisites

- Docker and Docker Compose working (Chapter 2 or 3).
- Local access (LAN/Wi-Fi). Optional: VPN for remote access (Appendix C).
- A persistent data root (we use `/srv`).

9.5 The "web address" Seafile needs

Seafile needs to know how users will reach it in the browser. This affects link generation and some integrations.

What should you put there?

Use the address you will actually type:

- **On your home LAN/Wi-Fi:** `http://SERVER_IP:8000`
- **On VPN:** the same address you use on LAN, or a VPN IP/name
- **Public internet (advanced):** `https://drive.yourdomain.com` (only after reverse proxy + TLS)

Start with LAN/VPN first. If you later add a domain + HTTPS, you can update the URL settings.

9.6 Folder layout

We keep Seafile predictable:

- `/srv/docker/compose/seafile` — Compose files
- `/srv/drive/seafile` — Seafile persistent data (libraries, config)
- `/srv/drive/seafile-db` — database data

  Create folders:

```
mkdir -p /srv/docker/compose/seafile
mkdir -p /srv/drive/{seafile,seafile-db}
```


9.7 Core Drive install

9.7.1 Step 1 — Create the Compose project

Go to the folder:

```
cd /srv/docker/compose/seafile
```

9.7.2 Step 2 — Create `.env` (recommended)

Create:

```
nano .env
```

Paste and edit:

```
# --- Seafile web address (what users type) ---
SEAFILE_SERVER_HOSTNAME=SERVER_IP

# --- Admin account ---
SEAFILE_ADMIN_EMAIL=admin@example.com
SEAFILE_ADMIN_PASSWORD=CHANGE_THIS_TO_A_LONG_PASSWORD

# --- Database (MariaDB) ---
MYSQL_ROOT_PASSWORD=CHANGE_THIS_ROOT_PASSWORD
MYSQL_LOG_CONSOLE=true

# Timezone (optional)
TZ=Europe/Madrid
```

Why this matters:

- Seafile needs an admin account on first start.
- Seafile uses a database for metadata.
- Keeping secrets in `.env` is cleaner than hardcoding them in YAML.

Yes, real passwords. Drives tend to grow into "critical infrastructure" faster than you expect.

9.7.3 Step 3 — Create `docker-compose.yml`

Create:

```
nano docker-compose.yml
```

Paste (core Seafile + MariaDB):

```
services:
  db:
    image: mariadb:11
    container_name: seafile-db
    restart: unless-stopped
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
      MYSQL_LOG_CONSOLE: ${MYSQL_LOG_CONSOLE}
      TZ: ${TZ}
    volumes:
```

```
    - /srv/drive/seafile-db:/var/lib/mysql

  memcached:
    image: memcached:1.6-alpine
    container_name: seafile-memcached
    restart: unless-stopped

  seafile:
    image: seafileltd/seafile-mc:latest
    container_name: seafile
    restart: unless-stopped
    depends_on:
      - db
      - memcached
    ports:
      - "8000:80"
    environment:
      DB_HOST: db
      DB_ROOT_PASSWD: ${MYSQL_ROOT_PASSWORD}
      TIME_ZONE: ${TZ}

      SEAFILE_ADMIN_EMAIL: ${SEAFILE_ADMIN_EMAIL}
      SEAFILE_ADMIN_PASSWORD: ${SEAFILE_ADMIN_PASSWORD}

      # The hostname (or domain) users will use:
      SEAFILE_SERVER_HOSTNAME: ${SEAFILE_SERVER_HOSTNAME}
    volumes:
      - /srv/drive/seafile:/shared
```

What this gives you:

- Seafile web UI exposed on port 8000.
- MariaDB for metadata.
- Memcached for performance (recommended).
- All persistent data under `/srv/drive/...`.

If YAML feels fragile: it is. Indentation matters. If something fails, it is usually one missing space, not a life problem.

9.7.4 Step 4 — Start Seafile

Start:

```
docker compose up -d
```

Check:

```
docker ps
```

Logs (first debugging tool):

```
docker compose logs -f --tail=120
```


9.7.5 Step 5 — Open Seafile (first login)

On the same machine:

```
http://localhost:8000
```

From another device on the same network:

```
http://SERVER_IP:8000
```

Log in with the admin email/password from `.env`.

If the login page loads: you are already winning. Most "server projects" fail before this moment.

9.8 Verification checklist (core)

You are done with the core when:

- `docker ps` shows `seafile`, `seafile-db`, and `seafile-memcached` running,
- `http://localhost:8000` loads,
- you can create a library and upload/download a file,
- a share link works from another device on the same network.

9.9 Access from mobile and tablet

On the same Wi-Fi (local network)

Open:

```
http://SERVER_IP:8000
```

Outside your home (recommended: VPN)

Connect your device to VPN (Appendix C), then use the same address.

Make it feel like an app

You can add the Seafile page to your home screen ("Add to Home Screen").

9.10 Common pitfalls (core)

9.10.1 1) Seafile starts, but the web UI is not reachable

Check:

- `docker ps` (is the container running?)
- published ports (do you see `0.0.0.0:8000->80/tcp`?)
- logs:

```
docker compose logs -f --tail=120
```

9.10.2 2) Wrong hostname / bad links

If share links point to the wrong place, update `SEAFILE_SERVER_HOSTNAME` in `.env` and restart:

```
docker compose down
docker compose up -d
```

9.10.3 3) Works on localhost but not from phone

Most common cause: firewall (especially on Windows in Path B). If you need access outside home, use VPN (Appendix C).

9.10.4 4) Database container problems

If Seafile complains about the database:

- confirm the DB container is running,
- ensure the root password matches (`MYSQL_ROOT_PASSWORD` vs `DB_ROOT_PASSWD`).

9.11 Optional: OnlyOffice integration (skip if you do not want online editing)

OnlyOffice adds browser-based editing of office documents alongside your Drive.

The typical approach is:

- run OnlyOffice Docs (Document Server) as a separate container,
- configure Seafile to use it (usually with a shared secret / JWT),
- verify by opening a document from the Seafile UI.

Optional means optional. Your Drive works perfectly without this. Add it only if you actually want browser editing.

9.11.1 Step O1 — Add OnlyOffice Docs to the Compose file

Edit `docker-compose.yml` and add this service:

```
  onlyoffice:
    image: onlyoffice/documentserver:latest
    container_name: seafile-onlyoffice
    restart: unless-stopped
    environment:
      TZ: ${TZ}
      JWT_ENABLED: "true"
      JWT_SECRET: "CHANGE_THIS_TO_A_LONG_RANDOM_SECRET"
    ports:
      - "8001:80"
```

Restart:

```
docker compose down
docker compose up -d
```


9.11.2 Step O2 — Tell Seafile where OnlyOffice lives

Seafile needs to know the OnlyOffice Document Server address and the shared secret.

In many Seafile setups, this is configured in Seafile's config files under:

```
/srv/drive/seafile/conf/
```

You will typically set:

- the OnlyOffice URL (for Seafile-to-OnlyOffice communication),
- the JWT secret (must match the container).

Integration settings vary slightly between Seafile versions and deployment styles. The good news: once you know where the config lives (`/srv/drive/seafile/conf/`), you are in control.

Step O2a — Locate the Seafile config file

The Seafile configuration folder is mounted on the host at:

```
/srv/drive/seafile/conf/
```

The file we want is usually:

```
/srv/drive/seafile/conf/seahub_settings.py
```

Step O2b — Edit `seahub_settings.py`

Open it:

```
nano /srv/drive/seafile/conf/seahub_settings.py
```

Add (or update) the OnlyOffice settings below.

Important rules:

- `JWT_SECRET` must match the one used by the OnlyOffice container.
- Use the **internal Docker name** `onlyoffice` for container-to-container communication.

```
# --- OnlyOffice integration (example) ---
ENABLE_ONLYOFFICE = True

ONLYOFFICE_APIJS_URL = "http://onlyoffice/web-apps/apps/api/documents/api.js"
ONLYOFFICE_FILE_EXTENSION = (
  ".doc", ".docx", ".ppt", ".pptx", ".xls", ".xlsx",
  ".odt", ".odp", ".ods"
)

# JWT shared secret (must match the OnlyOffice container)
ONLYOFFICE_JWT_SECRET = "CHANGE_THIS_TO_THE_SAME_JWT_SECRET"
```

Step O2c — Restart Seafile to apply changes

From your Seafile Compose folder:

```
cd /srv/docker/compose/seafile
docker compose restart seafile
```

Then refresh Seafile in the browser and try opening a `.docx`.

9.11.3 Verification (OnlyOffice)

You are done when:

- OnlyOffice is reachable on `http://SERVER_IP:8001/`,
- Seafile can open a document in the browser editor,
- saving works without errors.

9.11.4 Common pitfalls (OnlyOffice)

Connection errors

Most common causes:

- wrong OnlyOffice URL (internal vs external),
- secret mismatch (JWT),
- HTTP/HTTPS mismatch (if you later add TLS).

Public exposure

Do not expose OnlyOffice publicly until you understand reverse proxies and TLS. Keep it LAN/VPN by default.

9.12 Next step

At this point you have two documented Drive options:

- Nextcloud (more “suite” features),
- Seafile (fast, focused sync/libraries).

9.13 Decision summary: Nextcloud vs Seafile (which one should you keep?)

Both options are valid. The best choice depends on what you want your “Drive” to be.

Choose Nextcloud if you want a platform

Nextcloud is a good fit if you want a broader ecosystem:

- lots of built-in sharing features and an “app” mindset,
- future expansion into calendars, contacts, collaborative apps,
- a familiar “cloud suite” feeling.

Choose Seafile if you want fast, focused syncing

Seafile is a good fit if your priority is file libraries and efficient sync:

- a lighter, often faster experience for file syncing,
- a simpler mental model (libraries + clients),
- fewer moving parts if you do not need the full suite.

And about OnlyOffice

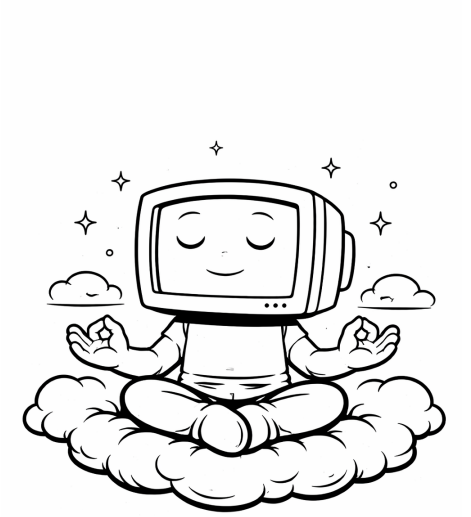
OnlyOffice is optional in both. Add it only if you genuinely want browser-based document editing. Otherwise, keep your setup simpler and edit documents with your local apps.

There is no "wrong" choice here. Pick the one that matches your workflow today. You can always migrate later because your data lives under `/srv` *and your stack is documented.*

Next, we will describe backups and a simple maintenance routine (updates, logs, and safety checks).

Chapter 10

Maintenance: updates, logs, and safety checks



10.1 Goal (what you will achieve)

This chapter helps you keep your stack healthy over time. You will learn a simple routine for:

- updating containers safely,
- checking what is running and whether it is healthy,
- reading logs when something breaks,
- doing basic backups (so a disk failure is not a tragedy).

This chapter is what turns a cool weekend project into something you can trust for years. You do not need to be perfect—you just need a routine.

10.2 A calm mental model: “pets” vs. “cattle”

In old-school server culture, people treated servers like pets: unique, hand-tuned, fragile. Modern self-hosting works better when services are more like “cattle”:

- containers can be destroyed and recreated,
- configuration lives in Compose files,
- **data lives in volumes under `/srv`.**

If you get volumes and backups right, updates stop being scary.

10.3 Your basic toolbox (commands you will reuse)

From inside a Compose folder (e.g., `/srv/docker/compose/photoprism`):

See running containers

```
docker ps
```

See logs

```
docker compose logs -f --tail=100
```

Restart a stack

```
docker compose down
docker compose up -d
```

Update images (safe pattern)

```
docker compose pull
docker compose up -d
```

Clean unused images (optional, disk cleanup)

```
docker image prune
```

10.4 A simple update routine (recommended)

Do updates when you have 10–20 minutes and can verify things afterwards.

10.4.1 Step 1 — Update your Linux packages (Path A)

On Linux servers, run occasionally:

```
sudo apt update
sudo apt upgrade -y
```

10.4.2 Step 2 — Update containers stack-by-stack

For each service folder under `/srv/docker/compose/`:

```
cd /srv/docker/compose/SERVICE
docker compose pull
docker compose up -d
```

Why this is safe:

- `pull` downloads new images without changing your running containers,
- `up -d` recreates containers if needed while keeping volumes intact.

10.4.3 Step 3 — Verify quickly

After updating a service, open its web UI:

- PhotoPrism: `http://SERVER_IP:2342`
- Navidrome: `http://SERVER_IP:4533`
- Open WebUI: `http://SERVER_IP:3000`
- Nextcloud: `http://SERVER_IP:8080`
- Seafile: `http://SERVER_IP:8000`

You are not testing everything. You are doing a quick sanity check: "does it start and does the main page load?" That is enough most of the time.

10.5 Reading logs without losing your mind

Logs look noisy, but you usually want just a few lines around the error.

10.5.1 Per-stack logs (recommended)

From the Compose folder:

```
docker compose logs --tail=200
```

Follow live:

```
docker compose logs -f --tail=100
```

10.5.2 Single container logs

If you know the container name:

```
docker logs --tail=200 CONTAINER_NAME
```


10.5.3 What to look for

Most real failures fall into a few categories:

- **Ports**: "address already in use" (port conflict).
- **Permissions**: "permission denied" (volume ownership).
- **Database**: "cannot connect" (DB not ready or wrong credentials).
- **Disk**: "no space left on device".

When something breaks: do not guess. Read the last 50–200 lines of logs. The clue is usually there.

10.6 Safety checks (weekly or monthly)

This is a short checklist that prevents most disasters.

10.6.1 1) Disk space

Check disk usage:

```
df -h
```

If Docker images are filling the disk:

```
docker system df
docker image prune
```

10.6.2 2) Are containers restarting repeatedly?

If a container keeps restarting, you will usually see it in:

```
docker ps
```

Look for high restart counts.
Then check its logs.

10.6.3 3) Confirm your backups exist (and are recent)

Backups that do not exist are not backups. Backups you have never tested are "hope".

If you do only one mature thing: make backups boring and automatic.

10.7 Backups (simple, realistic guidance)

A full backup strategy can become complex. Start simple.

10.7.1 What to back up

At minimum:

- **Your originals:** photos/music/files under `/srv/photos`, `/srv/music`, `/srv/drive`
- **Your configs:** Compose folders under `/srv/docker/compose`

These two categories are enough to rebuild the entire system after a failure.

10.7.2 A simple approach: copy `/srv` to an external drive

If you have an external disk mounted at `/mnt/backup`, you can do:

```
sudo rsync -aH --delete /srv/ /mnt/backup/srv/
```

What this does:

- `rsync` copies efficiently (only changes),
- `-delete` keeps the backup mirror clean (be careful),
- you end with a mirror copy of `/srv`.

10.7.3 How often?

- If you add files daily: daily backups.
- If you add files weekly: weekly backups.

10.7.4 A minimal "restore test"

Once in a while, pick one file and restore it from backup to confirm the backup is real.

10.8 Security hygiene (small habits with big impact)

10.8.1 Passwords

Use strong passwords for:

- Nextcloud admin,
- Seafile admin,
- PhotoPrism admin,
- Open WebUI user.

10.8.2 Updates

Outdated services are a common source of security issues. Even a simple monthly update helps.

10.8.3 Remote access

Prefer VPN (Appendix C) over public exposure if you do not want to manage TLS and reverse proxies.

10.9 If something goes wrong: a calm recovery recipe

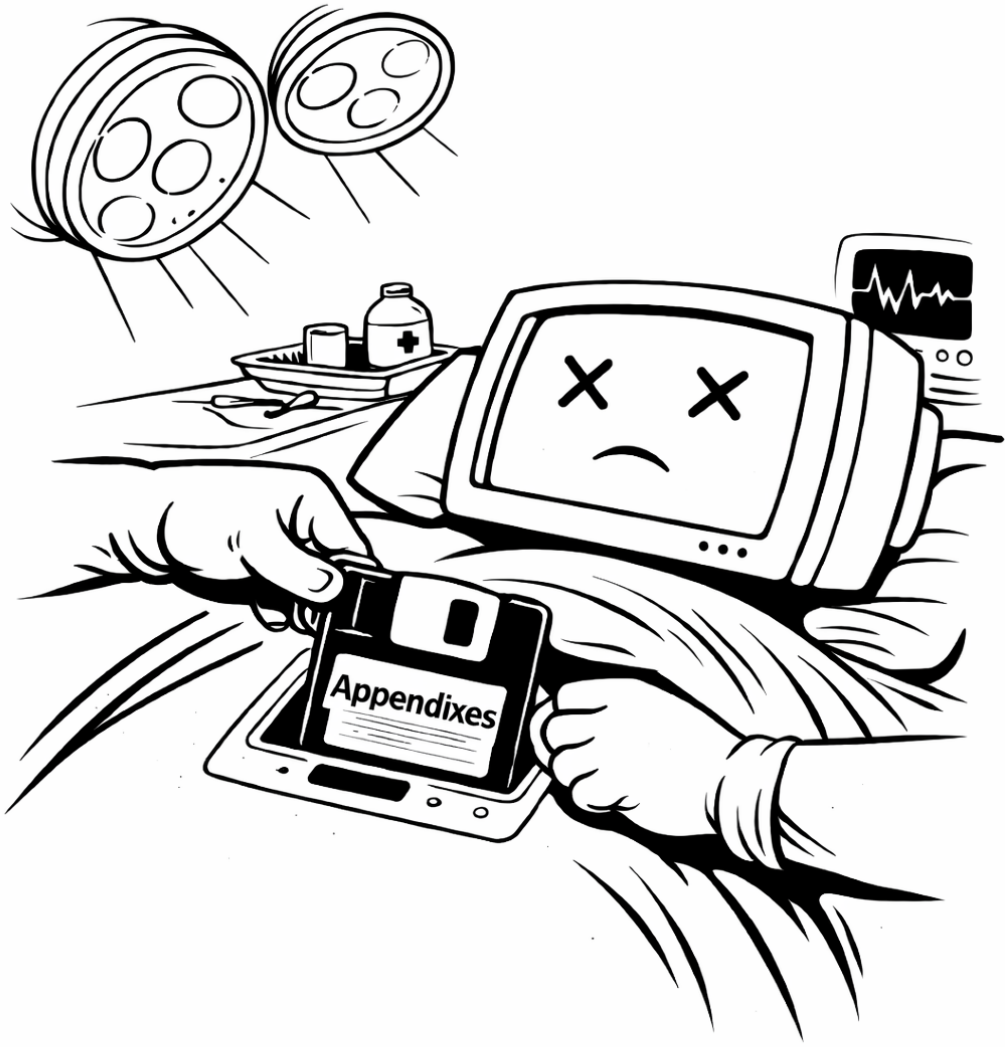
1. Check what is running: `docker ps`
2. Check logs: `docker compose logs -tail=200`
3. If needed, restart the stack: `docker compose down && docker compose up -d`
4. If still broken after an update, consider rolling back by pinning an older image tag (advanced).

Most failures are not catastrophic. They are usually one mis-typed variable, one full disk, or one service that needs a restart. You have the tools now.

10.10 Closing note

You now have a documented self-hosted stack: photos, music, local AI, and a private drive, plus optional VPN access.

The system will never be "set and forget" forever, but it also does not need to be stressful. A small routine—updates, logs, backups—is what makes digital sovereignty sustainable.


Appendixes

Appendix A

Beginner Glossary

This glossary explains common terms used throughout the manual.

Core concepts

Terminal A text-based interface where you type commands for the operating system.

Command A short instruction you type in the terminal (e.g., `sudo apt update`).

sudo "Run as administrator" on Linux. Required for system-level changes.

Package manager (APT) A tool that installs and updates software on Debian/Ubuntu (`apt`).

Repository A source of software packages that `apt` can download from.

GPG key A cryptographic key used to verify that packages come from the real publisher and were not modified.

Docker terms

Docker A system to run applications in containers.

Container A lightweight isolated environment that runs an application with its dependencies.

Image A packaged template used to create containers (like an app "installer", but for containers).

Volume A folder where containers store persistent data on the host disk.

Docker Compose A tool to define and run multi-container setups using a YAML file.

Port A network entry point (e.g., `8080`) used to reach a service from a browser or an app.

Remote access and security

SSH Secure remote login to a Linux machine (terminal access over the network).

Firewall A system that controls which network traffic is allowed in/out.

VPN A private network tunnel that lets you access services securely as if you were on the local network.

Appendix B

Compose Files (copy/paste reference)

B.1 How to use this appendix

This appendix collects the Docker Compose files used throughout the manual in one place.

A small bit of theory (so the files make sense)

- **YAML indentation matters.** Use spaces, not tabs.
- Lines starting with `#` are **comments** (ignored by Docker).
- Many files use a `.env` file in the same folder to store passwords and variables.
- Use this workflow:

```
cd /srv/docker/compose/SERVICE
nano .env
nano docker-compose.yml
docker compose up -d
```

Copy/paste is not cheating. It is how servers are built.

B.2 B.1 PhotoPrism (with MariaDB)

Folder

```
/srv/docker/compose/photoprism
```

.env

```
# --- PhotoPrism admin login ---
PHOTOPRISM_ADMIN_USER=admin
PHOTOPRISM_ADMIN_PASSWORD=CHANGE_THIS_TO_A_LONG_PASSWORD

# --- Database (MariaDB) ---
MARIADB_DATABASE=photoprism
MARIADB_USER=photoprism
MARIADB_PASSWORD=CHANGE_THIS_DB_PASSWORD
MARIADB_ROOT_PASSWORD=CHANGE_THIS_ROOT_PASSWORD
```

```
# --- Optional: match your Linux user (helps avoid permission issues) ---
PHOTOPRISM_UID=1000
PHOTOPRISM_GID=1000
```

docker-compose.yml

```
services:
  mariadb:
    image: mariadb:11
    container_name: photoprism-mariadb
    restart: unless-stopped
    environment:
      MARIADB_DATABASE: ${MARIADB_DATABASE}
      MARIADB_USER: ${MARIADB_USER}
      MARIADB_PASSWORD: ${MARIADB_PASSWORD}
      MARIADB_ROOT_PASSWORD: ${MARIADB_ROOT_PASSWORD}
    volumes:
      - /srv/photos/mariadb:/var/lib/mysql

  photoprism:
    image: photoprism/photoprism:latest
    container_name: photoprism
    restart: unless-stopped
    depends_on:
      - mariadb
    ports:
      - "2342:2342"
    environment:
      PHOTOPRISM_ADMIN_USER: ${PHOTOPRISM_ADMIN_USER}
      PHOTOPRISM_ADMIN_PASSWORD: ${PHOTOPRISM_ADMIN_PASSWORD}

      PHOTOPRISM_DATABASE_DRIVER: "mysql"
      PHOTOPRISM_DATABASE_SERVER: "mariadb:3306"
      PHOTOPRISM_DATABASE_NAME: ${MARIADB_DATABASE}
      PHOTOPRISM_DATABASE_USER: ${MARIADB_USER}
      PHOTOPRISM_DATABASE_PASSWORD: ${MARIADB_PASSWORD}

      PHOTOPRISM_UID: ${PHOTOPRISM_UID}
      PHOTOPRISM_GID: ${PHOTOPRISM_GID}

      PHOTOPRISM_SITE_URL: "http://localhost:2342/"
      PHOTOPRISM_ORIGINALS_LIMIT: 5000
    volumes:
      - /srv/photos/originals:/photoprism/originals
      - /srv/photos/import:/photoprism/import
      - /srv/photos/storage:/photoprism/storage
```

B.3 B.2 Navidrome

Folder

```
/srv/docker/compose/navidrome
```


.env

```
ND_SCANSCHEDULE=1h
ND_LOGLEVEL=info
TZ=Europe/Madrid
```

docker-compose.yml

```
services:
  navidrome:
    image: deluan/navidrome:latest
    container_name: navidrome
    restart: unless-stopped
    ports:
      - "4533:4533"
    environment:
      ND_SCANSCHEDULE: ${ND_SCANSCHEDULE}
      ND_LOGLEVEL: ${ND_LOGLEVEL}
      TZ: ${TZ}
    volumes:
      - /srv/music/data:/data
      - /srv/music/library:/music:ro
```

B.4 B.3 Local AI (Ollama + Open WebUI)

Folder

```
/srv/docker/compose/local-ai
```

docker-compose.yml

```
services:
  ollama:
    image: ollama/ollama:latest
    container_name: ollama
    ports:
      - "11434:11434"
    environment:
      TZ: Europe/Madrid
    volumes:
      - /srv/ai/ollama:/root/.ollama
    restart: unless-stopped

  open-webui:
    image: ghcr.io/open-webui/open-webui:main
    container_name: open-webui
    ports:
      - "3000:8080"
    environment:
      OLLAMA_BASE_URL: "http://ollama:11434"
      TZ: Europe/Madrid
    volumes:
      - /srv/ai/openwebui:/app/backend/data
    depends_on:
```

```
    - ollama
  restart: unless-stopped
```

Optional: enable NVIDIA GPU acceleration (snippet)

If you have an NVIDIA GPU, add the following block **inside the `ollama:` service**, at the same indentation level as `restart`, `ports`, and `volumes`. **Note:** GPU acceleration requires NVIDIA drivers and NVIDIA Container Toolkit (Linux) or Docker Desktop GPU support (Windows).

```
ollama:
  image: ollama/ollama:latest
  container_name: ollama
  ports:
    - "11434:11434"
  environment:
    TZ: Europe/Madrid
  volumes:
    - /srv/ai/ollama:/root/.ollama
  restart: unless-stopped
  deploy:
    resources:
      reservations:
        devices:
          - driver: nvidia
            count: all
            capabilities: [gpu]
```

Then restart the stack:

```
docker compose down
docker compose up -d
```

Verification

First, verify GPU on the host:

```
nvidia-smi
docker ps
```

If available inside the container (not always):

```
docker exec -it ollama nvidia-smi
```

If your Compose ignores `deploy:` on your system, see Chapter 7 (GPU troubleshooting) for the alternative `-gpus all` approach.

B.5 B.4 Nextcloud (core stack)

Folder

```
/srv/docker/compose/nextcloud
```


.env

```
MYSQL_DATABASE=nextcloud
MYSQL_USER=nextcloud
MYSQL_PASSWORD=CHANGE_THIS_DB_PASSWORD
MYSQL_ROOT_PASSWORD=CHANGE_THIS_ROOT_PASSWORD

NEXTCLOUD_ADMIN_USER=admin
NEXTCLOUD_ADMIN_PASSWORD=CHANGE_THIS_TO_A_LONG_PASSWORD

# Space-separated:
NEXTCLOUD_TRUSTED_DOMAINS=localhost 127.0.0.1 SERVER_IP

TZ=Europe/Madrid
```

docker-compose.yml (core)

```
services:
  db:
    image: mariadb:11
    container_name: nextcloud-db
    restart: unless-stopped
    command: --transaction-isolation=READ-COMMITTED --binlog-format=ROW
    environment:
      MYSQL_DATABASE: ${MYSQL_DATABASE}
      MYSQL_USER: ${MYSQL_USER}
      MYSQL_PASSWORD: ${MYSQL_PASSWORD}
      MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
      TZ: ${TZ}
    volumes:
      - /srv/drive/nextcloud/db:/var/lib/mysql

  redis:
    image: redis:7-alpine
    container_name: nextcloud-redis
    restart: unless-stopped
    command: redis-server --appendonly yes
    volumes:
      - /srv/drive/nextcloud/redis:/data

  app:
    image: nextcloud:apache
    container_name: nextcloud
    restart: unless-stopped
    depends_on:
      - db
      - redis
    ports:
      - "8080:80"
    environment:
      MYSQL_HOST: db
      MYSQL_DATABASE: ${MYSQL_DATABASE}
      MYSQL_USER: ${MYSQL_USER}
      MYSQL_PASSWORD: ${MYSQL_PASSWORD}
      NEXTCLOUD_ADMIN_USER: ${NEXTCLOUD_ADMIN_USER}
      NEXTCLOUD_ADMIN_PASSWORD: ${NEXTCLOUD_ADMIN_PASSWORD}
      NEXTCLOUD_TRUSTED_DOMAINS: ${NEXTCLOUD_TRUSTED_DOMAINS}
```

```
    TZ: ${TZ}
  volumes:
    - /srv/drive/nextcloud/html:/var/www/html
    - /srv/drive/nextcloud/data:/var/www/html/data

cron:
  image: nextcloud:apache
  container_name: nextcloud-cron
  restart: unless-stopped
  depends_on:
    - app
  entrypoint: /cron.sh
  volumes:
    - /srv/drive/nextcloud/html:/var/www/html
    - /srv/drive/nextcloud/data:/var/www/html/data
```

Optional: add OnlyOffice Docs (service snippet)

Add inside `services:`:

```
onlyoffice:
  image: onlyoffice/documentserver:latest
  container_name: onlyoffice
  restart: unless-stopped
  environment:
    TZ: ${TZ}
    JWT_ENABLED: "true"
    JWT_SECRET: "CHANGE_THIS_TO_A_LONG_RANDOM_SECRET"
  ports:
    - "8081:80"
```

B.6 B.5 Seafile (core stack)

Folder

```
/srv/docker/compose/seafile
```

.env

```
SEAFILE_SERVER_HOSTNAME=SERVER_IP

SEAFILE_ADMIN_EMAIL=admin@example.com
SEAFILE_ADMIN_PASSWORD=CHANGE_THIS_TO_A_LONG_PASSWORD

MYSQL_ROOT_PASSWORD=CHANGE_THIS_ROOT_PASSWORD
MYSQL_LOG_CONSOLE=true

TZ=Europe/Madrid
```

docker-compose.yml (core)

```
services:
  db:
    image: mariadb:11
    container_name: seafile-db
    restart: unless-stopped
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
      MYSQL_LOG_CONSOLE: ${MYSQL_LOG_CONSOLE}
      TZ: ${TZ}
    volumes:
      - /srv/drive/seafile-db:/var/lib/mysql

  memcached:
    image: memcached:1.6-alpine
    container_name: seafile-memcached
    restart: unless-stopped

  seafile:
    image: seafileltd/seafile-mc:latest
    container_name: seafile
    restart: unless-stopped
    depends_on:
      - db
      - memcached
    ports:
      - "8000:80"
    environment:
      DB_HOST: db
      DB_ROOT_PASSWD: ${MYSQL_ROOT_PASSWORD}
      TIME_ZONE: ${TZ}
      SEAFILE_ADMIN_EMAIL: ${SEAFILE_ADMIN_EMAIL}
      SEAFILE_ADMIN_PASSWORD: ${SEAFILE_ADMIN_PASSWORD}
      SEAFILE_SERVER_HOSTNAME: ${SEAFILE_SERVER_HOSTNAME}
    volumes:
      - /srv/drive/seafile:/shared
```

Optional: add OnlyOffice Docs (service snippet)

Add inside `services:`:

```
  onlyoffice:
    image: onlyoffice/documentserver:latest
    container_name: seafile-onlyoffice
    restart: unless-stopped
    environment:
      TZ: ${TZ}
      JWT_ENABLED: "true"
      JWT_SECRET: "CHANGE_THIS_TO_A_LONG_RANDOM_SECRET"
    ports:
      - "8001:80"
```

B.7 B.6 Notes on placeholders

Replace these placeholders before running:

- `SERVER_IP` with your server IP (see Chapter 4),

- passwords marked `CHANGE_THIS_*` with strong passwords,
- emails like `admin@example.com` with your real admin email.

If you keep your Compose files and `/srv` backups, you can rebuild the entire system. That is the quiet superpower of this manual.

Appendix C

Optional: Remote access with VPN (recommended)

C.1 Goal

This appendix explains how to access your self-hosted services (PhotoPrism, Navidrome, etc.) from outside your home network *without* exposing them publicly to the internet.

By the end, you will:

- understand why VPN is the safest beginner-friendly approach,
- choose a VPN method that fits your environment,
- verify remote access using the same local URLs (e.g., `http://SERVER_IP:2342`).

Remote access is where many projects go off the rails. A VPN keeps it simple: instead of making your services public, you make your device private.

C.2 The key idea (one sentence)

A VPN makes your phone/laptop behave as if it were on your home Wi-Fi, even when you are away.

That means: if PhotoPrism works at home with

```
http://SERVER_IP:2342
```

it should work the same way over VPN.

C.3 Before you start: avoid the "open ports" trap

A common beginner instinct is: "I will just open a port on my router." This can work, but it often leads to:

- security mistakes (public exposure without TLS),
- router/ISP limitations (CGNAT),
- complicated reverse-proxy setups.

VPN is usually the safer, shorter path.

C.4 Choose your VPN approach

There is no single perfect solution for everyone. Pick the one that matches your situation.

C.4.1 Option 1: Use an existing VPN (simplest if you already have one)

Some people already have a VPN provided by their workplace or university.

How it works:

- Connect your phone/laptop to that VPN.
- If your server is reachable through the VPN network, you can access services using the server IP.

When this is enough:

- your server is inside that VPN network (or reachable through it),
- you only need access for yourself (not for many external users).

If you already have a VPN that reaches your server: congrats, you may already be done.

C.4.2 Option 2: Tailscale (step-by-step, least friction)

Tailscale is a mesh VPN built on WireGuard. Many people like it because it works even when:

- you cannot easily configure your router,
- your ISP uses CGNAT,
- you want device-to-device access with minimal networking work.

This is the "I just want it to work" option. Follow the steps in order. If something fails, it is almost always because one device is not logged in, or you are not actually connected.

Step 1 — Create a Tailscale account

On any computer/phone browser:

- Go to the Tailscale website and create an account (for example using Google, Microsoft, GitHub, etc.).

You will manage your devices from the Tailscale admin console.

Step 2 — Install Tailscale on the server

Install it on the machine that runs your services (the same machine that runs Docker).

Linux (Path A). Install using the official Tailscale script (simple and common):

```
curl -fsSL https://tailscale.com/install.sh | sh
```

Then bring the server online:

```
sudo tailscale up
```

What happens now:

- A browser window (or a URL in the terminal) will ask you to log in.
- After login, the server becomes a device in your Tailscale network.

Windows (Path B). Install the Tailscale application on Windows (standard installer). Then sign in with the same account. (You can run Tailscale on Windows even if your Docker workflow uses WSL2.)

Step 3 — Install Tailscale on your phone/tablet/laptop

Install Tailscale on the device you want to use remotely:

- phone/tablet: install the Tailscale app from your app store,
- laptop: install the Tailscale client for your OS.

Log in with the same Tailscale account.

Step 4 — Find your server's Tailscale IP

Each device gets a stable private IP (often in the `100.x.y.z` range).

On the server, run:

```
tailscale ip -4
```

Or check the device in the Tailscale admin console (it shows the IP and device name).

When using Tailscale, you typically use the Tailscale IP (often `100.x.y.z`), not your home Wi-Fi IP (often `192.168.x.x`).

Step 5 — Access services using the Tailscale IP

Once your phone/laptop is connected to Tailscale, open:

```
http://TAILSCALE_IP:2342 % PhotoPrism
http://TAILSCALE_IP:4533 % Navidrome
```

Important: This works from outside your home because your device is now inside your private VPN.

If you can load PhotoPrism over mobile data while connected to Tailscale: you have achieved remote access without exposing anything publicly. That is a big win.

Verification checklist (Tailscale)

You are done when:

- the server appears as "connected" in Tailscale,
- your phone/laptop is connected in the Tailscale app,
- you can access `http://TAILSCALE_IP:2342` from outside your home Wi-Fi.

Common pitfalls (Tailscale)

- **You are still on Wi-Fi and think it works "remotely".** Test properly: disable Wi-Fi (use mobile data), connect Tailscale, then try again.
- **Wrong IP.** Use the Tailscale IP (often `100.x.y.z`), not your home LAN IP (e.g., `192.168.x.x`).
- **Not actually connected.** Open the Tailscale app and confirm "Connected".
- **Firewall blocks.** Rare, but if needed, allow inbound connections on the server for local ports (2342, 4533) on the Tailscale interface.

C.4.3 Option 3 (advanced): WireGuard (classic self-hosted VPN)

WireGuard is a modern VPN protocol known for simplicity and performance.

High-level architecture:

- your server runs a WireGuard VPN endpoint,
- your phone/laptop is a WireGuard client,
- once connected, your device reaches the server via a private VPN subnet.

Why it is "advanced" here:

- you may need router configuration (port forwarding),
- ISP limitations (CGNAT) can complicate inbound connectivity,
- you will manage keys and a config file manually.

WireGuard is worth it if you want maximum control and minimal external dependencies. For this manual, Tailscale remains the recommended default because it has the least friction for beginners.

C.5 Verification checklist (for any VPN option)

You have remote access when:

- your device shows "VPN connected",
- you can reach the server IP (or VPN IP) from your device,
- PhotoPrism/Navidrome load from outside your home network.

A practical test:

- turn off Wi-Fi on your phone (use mobile data),
- connect the VPN,
- open `http://SERVER_IP:2342` (or VPN IP).

C.6 Common pitfalls

It works at home but not outside

Most likely: you are not actually reaching your home network. Check that your VPN is connected *and* that it routes traffic to your home subnet.

CGNAT or ISP restrictions

Some ISPs prevent inbound connections to your home. This makes classic port-forwarding VPN setups harder. In these cases, solutions like Tailscale often work more smoothly.

Firewall blocking

If the VPN connects but services do not load, check:

- server firewall rules,
- Windows firewall (if applicable),
- whether the service is bound to the correct interface (usually OK with Docker port publishing).

Security reminder

VPN access still deserves strong passwords and updates. A VPN is not a substitute for basic hygiene; it is a safer network boundary.

C.7 Where to go next

If you want an even more independent setup, the next upgrade is a full **WireGuard quickstart** (self-hosted and fully under your control). For most users, Tailscale is already an excellent solution.