Build a Home Server Course

Flavio Copes

Updated August 3, 2026

Contents

Plan, install, secure, and operate an Ubuntu home server with private remote access, durable storage, containers, HTTPS, backups, and recovery notes.

What you'll learn: Build a private home server that survives routine updates and power loss and can be restored from documented backups.

Plan and install

Choose hardware and workloads, install Ubuntu Server, assign a stable address, and record the machine.

Choose a home server job

Start from one useful workload and derive CPU, memory, storage, network, noise, and power needs.

A home server is useful when it runs a service you understand and maintain. Start with one job such as file sync, backups, a private dashboard, or a development service.

One service, done properly, teaches you more than five half-configured containers ever will.

Why one job first

Every service you add brings its own updates, storage growth, and failure modes. Start with three at once and you will debug all three with no working baseline to compare against.

Pick the workload that solves a real problem for you today. You can add more later, on a host you already trust.

Write the requirement down

Before you look at any hardware, write a one-page requirement:

```
service: private file sync
users: 2
storage now: 200 GB
storage in two years: 500 GB
downtime tolerance: one day
network: home LAN and Tailscale
backup target: encrypted offsite copy
```

Each line answers a question the hardware will force on you anyway. Two users syncing files need almost no CPU. A media transcoder needs a lot. ”Downtime tolerance: one day” tells you a single cheap machine is fine and you don't need redundant anything.

Derive the hardware from the job

For a small workload like this, a used mini PC or an old laptop is enough. Check that it has room for the storage you projected, not just the storage you need today.

Power matters because the machine runs all day, every day:

```
idle draw: 10 W
one year: 10 W x 8760 h = 87.6 kWh
at 0.30 EUR/kWh: about 26 EUR per year
```

An old desktop tower idling at 60 W costs six times that. Noise matters too if the server lives near where you sleep or work.

The mistake everyone makes

The common failure is buying hardware first, then hunting for jobs to justify it. That path ends with an expensive machine running nothing, or running twelve services nobody depends on.

Choose hardware only after the workload, growth, recovery, and power constraints are visible.

And one hard rule: do not put irreplaceable data on the new machine before the backup and restore path works. If family photos land on it in week one and the disk dies in week three, no amount of planning after the fact will bring them back.

Prepare installation and recovery

Verify the installer image, save required configuration, and keep another device available for recovery.

Installation can erase a disk or interrupt network access. Resolve the exact target hardware and preserve anything needed to recover before you boot an installer.

Verify the installer image

Download Ubuntu Server from ubuntu.com over HTTPS, then record image integrity before writing media:

```
shasum -a 256 ubuntu-server.iso
# Linux commonly uses: sha256sum ubuntu-server.iso
```

Compare the hash with Ubuntu's published checksum over a trusted HTTPS path. The two long hex strings must match character for character.

A corrupted download produces confusing installer failures halfway through, which you will otherwise blame on the hardware. Thirty seconds of checking saves an evening of that.

Write the verified image to a USB stick, and double-check you picked the stick and not a data drive. This is the first place people erase the wrong device.

Identify the target disk

Boot the machine from the USB stick. Before confirming any destructive step, identify the install target by model and size, not by device name:

```
lsblk -o NAME,SIZE,MODEL
```

Label the target disk in your notes: "Samsung 870 EVO, 500 GB, system disk". Never choose an install disk by a broad wildcard or assumption. Disconnect unrelated removable drives when practical, so the wrong choice isn't even on the menu.

If the disk currently holds anything you might want, copy it off now. The installer will not ask twice.

Keep a recovery path open

Assume the first attempt fails halfway. You want, ready before you start:

- a second device (laptop or phone) to look up documentation
- keyboard and monitor access to the server itself
- your router's admin address and credentials
- a written plan: hostname, username, and disk layout

The failure mode to respect here is writing the installer to the wrong disk. You notice when a drive that held data suddenly shows as empty. There is no fix after the fact, only the copy you made beforehand.

Install a minimal server

Install Ubuntu Server with one administrator, SSH support, deliberate storage, and no unnecessary services.

A smaller initial system has fewer services and fewer choices to maintain. Add workloads after the base host is updated and observable, not during the install.

What minimal means in practice

Walk through the Ubuntu Server installer and make these choices:

- one administrator account, with a strong password you store in a password manager
- install the OpenSSH server when the installer offers it, so you can manage the machine remotely from day one
- skip the optional featured snaps. Every one of them is a service you'd have to maintain
- accept the default storage layout unless your plan from the previous lessons says otherwise

The installer finishes, the machine reboots, and you log in at the console.

Check what you actually got

After first login, record the release and pending updates:

```
cat /etc/os-release
sudo apt update
apt list --upgradable
```

`os-release` confirms you installed the version you meant to install. The upgrade list is usually long right after an install, and that's fine; you'll patch in a later lesson.

Then confirm the identity and network basics:

```
hostnamectl
timedatectl
ip a
```

Confirm the hostname, administrator account, timezone, storage layout, and network interface before adding applications. A wrong timezone makes every log timestamp misleading. A wrong hostname follows you into DNS, backups, and the runbook.

Test SSH before walking away

From another device on the same network:

```
ssh admin@192.168.1.50
```

Use the address `ip a` showed on the server. If the connection is refused, check on the console that the service is running with `systemctl status ssh`.

Keep console or physical recovery access until SSH and networking have been tested from another device. The classic mistake is unplugging the monitor and keyboard right after install, then discovering SSH was never reachable, and dragging everything back out again.

Give the server a stable address

Reserve an address through the router and verify the server keeps correct gateway and DNS configuration.

Services need a stable destination. If the server's address changes after a power cut, every bookmark, mount, and SSH config pointing at it breaks.

A **DHCP reservation** is the clean fix. The router keeps managing addresses centrally, but always hands this one machine the same address. You configure nothing on the server itself.

Record the current state

Record the current lease and route:

```
ip address
ip route
resolvectl status
```

Note four things: the interface name (something like `enp3s0`), the current address, the default gateway from `ip route`, and the DNS servers from `resolvectl status`. The MAC address appears in the `ip address` output as `link/ether`, and it's what the router uses to recognize this machine.

Create the reservation

Log into the router's admin interface and find the DHCP or LAN settings. Create a reservation binding the server's MAC address to an address, for example `192.168.1.50`.

Then make the server pick it up. The simplest reliable way is a reboot in a maintenance window:

```
sudo reboot
```

When it comes back, verify the reservation took effect:

```
ip address
ip route
```

You want the reserved address on the interface, and the same gateway and DNS as before. Then confirm from another device that `ssh admin@192.168.1.50` still works. If nothing changed, the old lease may not have expired yet; renew or reboot again after it does.

Why not just set a static address?

You can configure a static address directly on the server, but there's a trap. Avoid placing an arbitrary static address inside the DHCP pool. The router doesn't know you took it, and will eventually hand the same address to another device.

Duplicate addresses create intermittent failures: connections that work, then stall, then work again, depending on which machine answered last. It's one of the most confusing things to debug on a home network. The reservation avoids the whole category, because the router stays the single owner of the address plan.

Check your understanding: plan and install

Check the commands, decisions, safety boundaries, and failure cases from plan and install.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Access and security

Update the host, use personal SSH keys, restrict network exposure, and add private remote access.

Patch and reboot the base

Apply package updates, identify reboot requirements, and confirm every expected service returns afterward.

Patch the base before adding data and services. A fresh install ships with weeks or months of pending fixes, and this is the cheapest moment to apply them: nothing depends on the machine yet.

The reboot that follows matters just as much. It proves the host can go down and come back without you standing at a keyboard.

Apply updates

```
sudo apt update
sudo apt upgrade
```

`apt update` refreshes the package lists. `apt upgrade` installs the new versions and shows you the list before touching anything. Read it once; on a fresh server it's mostly base system packages.

Ubuntu Server also runs `unattended-upgrades` by default, which installs security fixes automatically in the background. That's a good default. It covers security patches between your manual sessions, but it doesn't replace them.

Check whether a reboot is needed

Some updates, like a new kernel, only take effect after a reboot. Ubuntu tells you explicitly:

```
test -f /var/run/reboot-required && cat /var/run/reboot-required
```

If the file exists you'll see `*** System restart required ***`. No output means no reboot needed, but during this lab, reboot anyway. You want to rehearse it now, not during a real outage.

Reboot and verify the return

```
sudo reboot
```

Wait a minute, reconnect over SSH, and check the state:

```
uname -r
systemctl --failed
ip a
findmnt
```

`uname -r` should show the new kernel if one was installed. `systemctl --failed` should print `0 loaded units listed`. The address should match your reservation, and every expected filesystem should still be mounted.

The failure mode

A remote-only machine that doesn't come back is the risk here. Maybe the bootloader broke, maybe the network config didn't survive. You find out only when SSH times out.

Keep a recovery route before rebooting a machine you can't reach physically: console access, or someone who can press the power button. On a server in the same room this is a small thing. Make it a habit anyway, because it won't always be in the same room.

Use personal SSH keys

Install a personal public key, verify host identity, and keep a second session open during access changes.

SSH keys separate user identities and avoid sharing an administrator password. Each person gets their own key pair, so you can later revoke one person's access without touching anyone else's.

Generate a key on your client

If you don't already have one, create a key pair on your laptop, not on the server:

```
ssh-keygen -t ed25519 -C "flavio@laptop"
```

Accept the default location and set a passphrase. The private key stays on the laptop forever. Only the `.pub` half travels.

Install the public key

Copy a public key through an authorized existing login:

```
ssh-copy-id admin@homeserver.local
ssh admin@homeserver.local
```

`ssh-copy-id` appends your public key to `~/.ssh/authorized_keys` on the server with the right permissions. The second command should now log you in without asking for the account password. If your key has a passphrase, you'll type that instead, and an agent can remember it.

Verify the host key, once

The first connection shows a fingerprint prompt. Most people blindly type yes. Do it properly once: on the server's console, print the real fingerprint and compare:

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

If it matches what the client showed, you've verified you're talking to your server and not something impersonating it. The client remembers the key after that, and will warn loudly if it ever changes.

Change access with a safety line

Whenever you touch SSH configuration, keep the current session open and test a second session before closing the first. The open session is your undo button.

The classic self-lockout goes like this: key login half works, you disable password authentication anyway, close the terminal, and now nothing gets in. So the rule is firm: do not disable password login until key login and recovery access are proven.

If key login fails with `Permission denied (publickey)`, check permissions on the server: `~/.ssh` must be `700` and `authorized_keys` must be `600`, both owned by the user. SSH silently ignores the file otherwise.

And never share private keys. One person, one device, one key.

Allow only needed firewall traffic

Define inbound service requirements, enable the firewall safely, and verify allowed and denied paths.

A home LAN is not automatically trusted. A compromised laptop, a guest's phone, a cheap IoT device: anything on your network can reach anything that listens. The host firewall limits which services accept connections and from where.

On Ubuntu that firewall is **UFW**, and the right posture is deny by default, then allow exactly what you listed.

Know your subnet before writing rules

Check what the local subnet actually is:

```
ip route
# default via 192.168.1.1 dev enp3s0
# 192.168.1.0/24 dev enp3s0 proto kernel scope link src 192.168.1.50
```

Here the LAN is `192.168.1.0/24`. Substitute the real local subnet after verifying it. A wrong rule can lock you out.

Deny by default, allow SSH first

Order matters. Allow SSH from the local subnet before enabling UFW:

```
sudo ufw default deny incoming
sudo ufw default allow outgoing
sudo ufw allow from 192.168.1.0/24 to any port 22 proto tcp
sudo ufw enable
sudo ufw status verbose
```

The status output should show `Default: deny (incoming), allow (outgoing)` and your SSH rule. Rules added before `enable` take effect the moment the firewall turns on, which is why SSH goes in first.

Verify both directions

Keep the current SSH session open and test a new one from another device. The old session is your recovery path if the new one fails.

Then compare what listens with what you allowed:

```
sudo ss -tlnp
```

Every listener in that output should either have a matching allow rule or be bound to `127.0.0.1`, where the firewall isn't needed. Scan only your own server from an authorized client if you want an outside view; scanning other people's machines is not a lab exercise.

If you do lock yourself out

It happens: a typo in the subnet, or an allow rule on the wrong port. The symptom is a new SSH connection that hangs and times out while the old one still works.

From the console, `sudo ufw status numbered` shows the rules, `sudo ufw delete <n>` removes the bad one, and `sudo ufw disable` turns the firewall off entirely while you regroup. This is exactly why console access stays available until the rules are proven.

Add private remote access

Connect the server through Tailscale and avoid forwarding an administration port from the home router.

You'll want to reach the server from outside the house. The tempting shortcut is forwarding port 22 on the router to the server. Don't.

A forwarded SSH port is visible to the entire Internet, and scanners find it within minutes. From then on, your home server's security depends on that one port being perfectly configured, forever.

An outbound-established private overlay can provide remote access without exposing SSH directly to the public Internet. **Tailscale** builds a WireGuard-based private network (a tailnet) between your devices. Every connection is outbound from the server's point of view, so the router needs no inbound rules at all.

Install and join the tailnet

Install through the current official Tailscale path:

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

`tailscale up` prints a login URL. Open it on any browser and authenticate; the server joins your tailnet. Then inspect the node:

```
tailscale status
tailscale ip -4
```

`status` lists the devices in your tailnet and their state. `ip -4` prints the server's tailnet address, something in the `100.x.y.z` range.

Connect from another device

Install Tailscale on your laptop or phone, log into the same tailnet, and connect using the tailnet address:

```
ssh admin@100.101.102.103
```

Try it from outside your home network, on mobile data, to prove the path really doesn't depend on your LAN. Then confirm the router has no SSH port forward. If one exists from earlier experiments, remove it now.

What this does and doesn't solve

Review tailnet access rules and device expiry in the Tailscale admin console. By default device keys expire and need re-authentication; that's a feature, not a bug. A stolen laptop that stays in your tailnet forever is the failure mode you're preventing.

When a node's key expires you'll see it marked in `tailscale status`, and connections to it fail. Fix it by running `sudo tailscale up` again on that machine.

Private connectivity still needs user authorization and host security. The firewall rules and SSH keys from the previous lessons stay exactly as they are. Tailscale decides which machines can reach the server; SSH still decides who gets in.

Check your understanding: access and security

Check the commands, decisions, safety boundaries, and failure cases from access and security.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Storage and data

Attach storage, mount by stable identity, set service ownership, inspect health, and separate live data from backups.

Inventory storage devices

Identify disks, partitions, filesystems, mount points, capacity, and models before changing anything.

Before you format, partition, or mount anything, you need a complete picture of what storage the machine has and what each device is for.

Linux storage names can change between boots or attachments. The disk called `/dev/sdb` today can become `/dev/sdc` after you plug in a USB stick. That's why the inventory records stable identifiers, and why mounts later use filesystem **UUIDs** instead of device names.

List storage without modifying it

All of these commands are read-only:

```
lsblk -o NAME,SIZE,TYPE,FSTYPE,UUID,MOUNTPOINTS,MODEL
findmnt
df -h
```

`lsblk` is the main view: every disk, its partitions, sizes, filesystem types, UUIDs, and the drive model. `findmnt` shows the tree of what's currently mounted where. `df -h` adds how full each mounted filesystem is.

A healthy small server might show:

```
NAME     SIZE TYPE FSTYPE UUID                                  MOUNTPOINTS MODEL
sda    465.8G disk                                                           Samsung SSD 870
 sda1     1G part vfat   9F1C-42A0                             /boot/efi
 sda2 464.8G part ext4   6d5f0a2e-...                          /
sdb      3.6T disk                                                           WDC WD40EFRX
 sdb1   3.6T part ext4   3f6c9d81-...
```

Match every device to a purpose

Now write the mapping down: `sda` is the Samsung SSD holding the operating system, `sdb` is the 4 TB data drive. Mark the system disk clearly. It's the one mounted at `/`, and it's the one you must never format by accident.

Save the inventory before partitioning or formatting:

```
lsblk -o NAME,SIZE,TYPE,FSTYPE,UUID,MOUNTPOINTS,MODEL > ~/storage-inventory.txt
```

That file becomes part of your runbook later, and it's your reference when a drive fails and you need to know which physical unit to pull.

The mistake this prevents

The classic storage disaster is running a format command against the wrong device, because "it was `sdb` yesterday". The wrong disk is erased in seconds and there is no undo.

So the rule: never run a formatting command until the exact target has been resolved, right now, in this boot, by size, model, and UUID, and its data is recoverable. If the sizes in `lsblk` don't match what you expect, stop and figure out why before typing anything destructive.

Mount a data filesystem

Create a mount point, mount an existing practice filesystem by UUID, and test startup configuration safely.

A mount attaches a filesystem to one directory in the visible tree. Until a filesystem is mounted, the data on it exists but nothing can reach it.

For a data drive on a server, you want two things: mount it by UUID so device-name shuffling can't hit the wrong disk, and make the mount survive reboots without being able to break the boot.

Mount it by hand first

Take the UUID of your practice filesystem from the inventory you saved in the previous lesson, then:

```
sudo mkdir -p /srv/data
sudo mount UUID=YOUR-LAB-UUID /srv/data
findmnt /srv/data
```

Do not copy the placeholder UUID. Use the real one from `lsblk` on your machine.

`findmnt` confirms the mount: it prints the target, the source device, the filesystem type, and the options. If it prints nothing, the mount didn't happen. You should now be able to `ls /srv/data` and see the filesystem's contents.

Make it survive a reboot

Startup mounts belong in `/etc/fstab` with deliberate failure behavior. Add one line:

```
UUID=3f6c9d81-real-uuid-here /srv/data ext4 defaults,nofail 0 2
```

The option that matters is `nofail`. Without it, a missing or dead data drive stops the whole boot and drops the server into emergency mode. That's a terrible trade for a headless machine: you'd lose SSH access because a data disk died. With `nofail`, the system boots, and you fix the disk over SSH.

Validate before rebooting

A typo in `fstab` is the classic way to break a boot. Test the file while the system is running:

```
sudo umount /srv/data
sudo mount -a
findmnt /srv/data
```

`mount -a` mounts everything listed in `fstab` that isn't mounted yet, using the same parsing the boot process relies on. If your line is wrong, you get an error here, in a working shell, instead of at a boot prompt. If systemd warns that `fstab` changed, run `sudo systemctl daemon-reload` and repeat.

Only after `mount -a` works cleanly, reboot once and check `findmnt /srv/data` again. Now the mount is proven, not assumed.

Assign service ownership

Give each service a narrow data directory and account instead of opening storage permissions globally.

Filesystem permissions should express which process owns and reads each dataset. When you look at `/srv/data` in a year, the owner and mode of each directory should tell you exactly which service the data belongs to.

World-writable directories hide ownership mistakes. Everything works, right up until you can't tell what wrote a file, or a misbehaving service scribbles over another service's data.

Create a dedicated account

Each service gets its own system account, with no login shell and no password:

```
sudo adduser --system --group appsvc
```

`--system` creates a low-UID account meant for services, and `--group` gives it a matching private group. Nobody logs in as `appsvc`; it exists only so files and processes can carry its identity.

Create a dedicated service directory

```
sudo install -d -o appsvc -g appsvc -m 0750 /srv/data/app
```

`install -d` creates the directory with owner, group, and mode in one step. The mode `0750` means: the service account has full access, members of its group can read, and everyone else gets nothing.

Test allowed and denied paths

Run the service as its account and test create, read, and delete operations:

```
sudo -u appsvc touch /srv/data/app/probe.txt
sudo -u appsvc rm /srv/data/app/probe.txt
```

Both should succeed silently. Then prove the boundary holds. An unrelated account should be denied:

```
sudo -u nobody touch /srv/data/app/denied.txt
# touch: cannot touch '/srv/data/app/denied.txt': Permission denied
```

That error is the test passing. If the write succeeds, the directory is more open than you think; check its mode and parents with `ls -ld /srv /srv/data /srv/data/app`.

The chmod 777 trap

At some point a service or container will fail with a permission error, and the Internet will tell you to run `chmod -R 777` on the data directory. Do not solve container or service errors with recursive `chmod 777`. It makes the error disappear by erasing the ownership model entirely, and every process on the machine can now modify that data.

Resolve the actual UID, GID, and required access instead. Find out what identity the process runs as, for a container with `docker exec app id`, then either run the service as the owning account or `chown` the directory to match. The fix is one precise line, not a blanket permission.

Monitor capacity and disk health

Track space, inodes, disk errors, and hardware health before the service reaches failure.

Storage fails in two ways, and both announce themselves in advance if you look. A filesystem can run out of bytes or inodes. A disk can also report media errors before complete failure. Services on a full filesystem don't crash politely — they corrupt their own state mid-write, which turns a capacity problem into a restore problem.

Capture a baseline

```
df -h
df -i
sudo dmesg --level=err,warn | tail -50
sudo smartctl -H /dev/sdX
```

`df -h` shows used space per filesystem. `df -i` shows **inodes** — the metadata slots filesystems use, one per file. A service writing millions of tiny files can hit 100% inode usage while `df -h` still shows free space, a genuinely confusing failure until you know to check.

The `dmesg` filter surfaces kernel-level I/O errors: lines mentioning `I/O error` or a specific device resetting are early hardware warnings. `smartctl` (from the `smartmontools` package, `sudo apt install smartmontools`) queries the disk's own health assessment. Replace `sdX` with your real device only after resolving the exact disk against your inventory — you want `PASSED`:

```
SMART overall-health self-assessment test result: PASSED
```

Anything else, and it's time to plan a replacement while the disk still reads.

Watch growth, not snapshots

A single reading tells you little; the trend is what predicts trouble. Review growth by directory when a filesystem fills faster than expected:

```
sudo du -xh --max-depth=1 /srv/data | sort -h
```

The biggest line at the bottom is usually a log directory or a cache nobody rotates. Set alerts below full capacity — 80% is a good threshold, because it leaves you time to act on a weekend rather than during the failure. Re-run the baseline weekly at first; a later lesson folds this into one observation checklist.

The mistake

Treating a healthy SMART status as a safety guarantee. SMART data is one signal, not a backup. A drive can fail without warning — controller deaths in particular skip the polite degradation phase entirely. Monitoring buys you early warnings most of the time; the backup module is what covers the rest.

Check your understanding: storage and data

Check the commands, decisions, safety boundaries, and failure cases from storage and data.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Run services

Deploy one containerized service, expose it privately, add local naming, and put HTTPS at the edge.

Deploy with Docker Compose

Define one service, persistent storage, restart behavior, and a loopback-only published port.

You could start containers with long `docker run` commands, but those live in your shell history and nowhere else. **Compose** records the desired container, image, volumes, ports, and restart policy in one reviewable file. The file is the deployment: you can read it, diff it, back it up, and reproduce the service on a rebuilt host.

Define one small service

Create a directory like `/opt/services/whoami` and inside it a `compose.yaml` for a small service image you trust:

```
services:
  web:
    image: traefik/whoami:v1.11.0
    restart: unless-stopped
    ports:
      - '127.0.0.1:8081:80'
```

Three deliberate choices here. The image tag is pinned to `v1.11.0`, not `latest`, so today's working deployment is still the same deployment next month. `restart: unless-stopped` makes Docker bring the container back after a crash or a reboot — unless you stopped it on purpose. And the port mapping binds to `127.0.0.1`, so the service answers on the server's loopback interface only. Nothing on the LAN can reach it directly; the HTTPS proxy lesson adds the deliberate entry point later.

Start it and verify

```
cd /opt/services/whoami
docker compose up -d
docker compose ps
```

```
docker compose logs web
curl http://127.0.0.1:8081
```

`docker compose ps` should show the container `Up`. The `curl` from the server itself returns the whoami response — hostname, headers, addresses.

Now verify the boundary from your laptop:

```
curl --max-time 3 http://192.168.1.20:8081
# curl: (28) Connection timed out ...
```

The timeout is a success. It proves the loopback binding works: the service runs, and the network can't touch it. If your laptop gets a response instead, the port line lost its `127.0.0.1:` prefix and the service is published to the whole LAN.

The failure mode

Treating images casually. A container image is executable supply-chain input, not a harmless data file — you're running someone else's code as a long-lived service on your network. Pin reviewed image versions, prefer official or well-known publishers, and read the image's page before the first `up`. An unpinned `latest` tag also breaks reproducibility: the rollback lesson depends on knowing exactly which version you were running.

Persist container data

Map important service data to a named volume or host directory and prove it survives recreation.

A container's writable layer is disposable. Every file a process writes inside the container, outside a mounted volume, is deleted the moment the container is recreated — and updates recreate containers. Persistent application state needs an explicit storage location and backup plan, declared in the Compose file where you can see it.

Declare the volume

First find where the application keeps its state — the image's documentation states it (databases often use `/var/lib/<name>`, apps often use `/data` or `/config`). Add a named volume where the chosen lab service expects data:

```
services:
  app:
    volumes:
      - app-data:/var/lib/app

volumes:
  app-data:
```

The top-level `volumes:` key declares a **named volume** called `app-data`; the service line mounts it at `/var/lib/app` inside the container. Docker manages the volume's lifecycle separately from the container, which is the entire point. The alternative is a bind mount like `/srv/data/app:/var/lib/app`, which puts the files on a host path you own — pair it with the ownership lesson's UID setup if you go that way.

Prove it survives recreation

Never trust persistence you haven't tested. Create test data, remove and recreate the container, and confirm the data remains:

```
docker compose exec app sh -c 'echo probe > /var/lib/app/probe.txt'
docker compose down
docker compose up -d
docker compose exec app cat /var/lib/app/probe.txt
# probe
```

`docker compose down` removes the container entirely (named volumes survive by default). If `probe` comes back after the recreate, the state lives in the volume. If the file is gone, the app writes its data somewhere you didn't mount — go back to the image docs and fix the path.

Then locate and document the volume:

```
docker volume inspect whoami_app-data
```

The `Mountpoint` field shows where the bytes live on the host, typically under `/var/lib/docker/volumes/`. Put that in your notes; the backup lesson needs it.

The failure mode

Confusing surviving a restart with surviving anything else. Persistence is not backup. Deletion, corruption, and host failure can affect both container and local volume — the volume is on the same disk, in the same house. A volume protects you from container recreation, nothing more. The operate-and-recover module handles the rest.

Add a local service name

Give the home service a stable local hostname through controlled DNS instead of teaching users an address and port.

Nobody in your house wants to remember `192.168.1.20:8081`. Names make services easier to move, too: when the service migrates to new hardware, you update one DNS record instead of re-teaching every person and reconfiguring every client.

The pieces stack: the router reservation keeps the server's address stable, and local DNS can map a private hostname to the server address on top of it.

Pick a name you control

Use a private zone like `lab.test` — the `.test` top-level domain is reserved and will never exist on the public Internet. Avoid inventing names under public domains you do not control: if you name your server `homeserver.somestartup.com` and that company later serves something at that name, your clients will resolve to the wrong side depending on which resolver answers first.

So the target record is:

```
homeserver.lab.test  ->  192.168.1.20
```

Configure it where your clients look

The name must be answered by the DNS server your LAN clients actually use. That's usually one of:

- the router's DNS settings, if it supports custom local records
- a self-hosted resolver such as Pi-hole or Unbound, if you already run one and point DHCP at it

Configure the name in the local DNS system you control. As a last resort, `/etc/hosts` entries on each client work, but they don't scale past two devices and they're exactly the kind of hidden state runbooks forget.

Verify from two clients

Inspect the intended answer from a client:

```
dig homeserver.lab.test
curl -I http://homeserver.lab.test
```

The `dig` answer section should show `192.168.1.20`. The `curl -I` result depends on where you are in the course: while the service is still bound to loopback, a refused or timed-out connection from your laptop is correct — the name resolves, and the next lesson's proxy provides the actual entry point.

Then verify answers from two clients and document the authoritative source. Two clients, because one device may carry a stale cache or a manual hosts entry; two matching answers mean the DNS server is really doing the work.

The failure mode

Split DNS must have clear ownership and recovery. If the record lives in a Pi-hole nobody documented, then when that box dies, every service name on the LAN dies with it — and whoever debugs it will chase "the server is down" when the server is fine. Write down which device answers `lab.test` queries, and put that fact in the runbook.

Add a private HTTPS proxy

Terminate HTTPS at a reverse proxy and keep the application container bound to loopback.

Right now the service answers on `127.0.0.1:8081` and nothing on the LAN can reach it. That was deliberate. Now we add the one deliberate entry point: a reverse proxy that terminates HTTPS on the LAN and forwards to the loopback port. The proxy gives users one named HTTPS entry point while the application remains private on its local port.

Caddy fits this job well because it can act as its own certificate authority for private names. Public certificate authorities can't issue certificates for `homeserver.lab.test` — the name doesn't exist on the Internet — so Caddy creates a local CA and issues its own.

Configure the route

Install Caddy from its official Ubuntu repository, then create a Caddy lab route in `/etc/caddy/Caddyfile`:

```
homeserver.lab.test {
  tls internal
  reverse_proxy 127.0.0.1:8081
}
```

`tls internal` tells Caddy to issue the certificate from its local CA instead of trying a public one (which would fail for a `.test` name). `reverse_proxy 127.0.0.1:8081` forwards each request to the whoami container. Reload and allow HTTPS from the LAN:

```
sudo systemctl reload caddy
sudo ufw allow from 192.168.1.0/24 to any port 443 proto tcp
```

Trust the local CA on your devices

Your laptop doesn't trust Caddy's private CA yet, so browsers will warn. Export the root certificate from the server — Caddy stores it at `/var/lib/caddy/.local/share/caddy/pki/authorities/local/root.crt` — and install the local CA certificate only on devices you own. Never ask guests to install it: whoever holds a trusted root CA can impersonate any website to that device.

Verify from a client

From your laptop, request the service and inspect both proxy and container logs:

```
curl -I https://homeserver.lab.test
# HTTP/2 200
```

On the server, `journalctl -u caddy -n 20` shows the TLS handshake and the proxied request, and `docker compose logs web` shows the request arriving on the container side. Seeing it in both places confirms the full path: client, TLS at Caddy, loopback, container.

Boundaries to keep

Protect the local root key and document client trust removal — note in your runbook which devices trust this CA and how to remove it, because that trust outlives the server.

And keep the layers straight: HTTPS does not replace application login. The proxy encrypts the path; it says nothing about who is allowed in. The most common mistake at this step is ”fixing” a browser warning by clicking through it forever instead of trusting the CA properly. Warnings you train yourself to ignore stop protecting you everywhere else.

Check your understanding: run services

Check the commands, decisions, safety boundaries, and failure cases from run services.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Operate and recover

Back up data, observe services, update deliberately, handle power loss, and maintain a recovery runbook.

Back up home server data

Inventory configuration and state, copy them off the server, and restore into a disposable location.

The server is replaceable; its unique data, configuration, and secrets are not. The operating system reinstalls in twenty minutes, containers pull again from the registry — but the file sync data, the Compose files, and the keys exist nowhere else. Back up only what you can identify and restore.

Inventory before tooling

Create an inventory before selecting a tool:

```
printf '%s\n' \
  /etc \
  /srv/data \
  /opt/services/compose.yaml \
  'database dumps' \
  'encryption keys stored separately'
```

Walk through it honestly. `/etc` covers host configuration (the Caddyfile, fstab, SSH config). `/srv/data` is the service state. The Compose files under `/opt/services` are the deployment itself. If a service runs a database, back up a **dump** — an export the database wrote while consistent — not the raw files underneath a running engine, which may be mid-write and restore as corruption.

The last line is the one people miss. If the backup is encrypted (it should be), the encryption keys stored separately rule is absolute: a key stored only on the server dies with the server, taking every backup with it. Print it, or keep it in your password manager.

Copy it off the machine, encrypted

Use the Backup and Restore course to create an encrypted offsite copy — it builds the full restic workflow. The shape of it:

```
restic -r sftp:backup@othermachine:/backups/homeserver backup /etc /srv/data /opt/services
```

"Off the server" is the requirement; a second directory on the same disk protects against nothing that matters.

Restore into a disposable location

A backup you haven't restored is a hope, not a backup. Perform a file and service restore before trusting it:

```
restic -r sftp:backup@othermachine:/backups/homeserver restore latest \
  --target /tmp/restore-drill --include /opt/services
diff -r /opt/services /tmp/restore-drill/opt/services
```

An empty `diff` means the round trip preserved the bytes. Then go one step further once: start the service from the restored copy on a scratch path and check it answers. That's the drill the runbook lesson rehearses in full.

The false comfort to avoid

RAID, snapshots, and local copies do not protect against every deletion, compromise, theft, or site failure. RAID mirrors your mistakes in real time. Snapshots share the disk they protect. The test for a real backup: if the server were stolen tonight, could you restore tomorrow? Only a copy that lives somewhere else, with keys that live somewhere else again, answers yes.

Observe services and host

Check failed units, containers, logs, capacity, memory, temperature, and network reachability from one checklist.

Monitoring starts with the failures users care about, then connects them to host and service evidence. Nobody in your house cares about CPU load; they care that file sync stopped. So the first check is always the user-visible one, and everything else exists to explain it.

The local checklist

Run a compact local check:

```
systemctl --failed
docker compose ps
df -h
free -h
ip route
```

Each line answers one question. `systemctl --failed` — did any host service die? You want `0 loaded units listed`. `docker compose ps` (run in your service directory) — are the containers `Up`, and not restart-looping? `df -h` — is any filesystem creeping past 80%? `free -h` — is memory exhausted, with swap climbing? `ip route` — does the host still have its default route, or did the network config quietly break?

When something looks wrong, logs are the next layer down:

```
journalctl -p err --since -1h
docker compose logs --since 1h web
```

The first shows host-level errors from the last hour; the second, the service's own output.

Check from the outside

Every check so far runs on the server, and a down server can't report itself. Add one external check from another authorized device:

```
# from your laptop:
curl -I --max-time 5 https://homeserver.lab.test
# HTTP/2 200
```

That single request exercises DNS, the network path, the proxy, TLS, and the container. It's the closest thing to what a real user experiences. Run it from your laptop when you think of it, or schedule it on any always-on device you own — even a cron job on another machine that pings you when the check fails.

Record thresholds and where alerts go when you are away. "Disk over 80% → email me" written in the runbook beats a perfect dashboard nobody looks at.

The boundary

Do not expose monitoring dashboards without authentication. They reveal names, versions, and internal state — a map of your setup, drawn for whoever finds it. Keep dashboards on the tailnet or behind the LAN-only proxy, never port-forwarded. And prefer the boring failure mode: if the checklist takes more than a minute to run, you'll stop running it. Five commands you actually type beat thirty you don't.

Update with a rollback plan

Review changes, preserve configuration and data, update one layer, test it, and retain a recovery target.

Home servers fail when unattended changes combine operating system, container, configuration, and data migrations at once. Four things changed, something broke, and now you're bisecting instead of using the service. The discipline: update one layer at a time, know what you're running before you change it, and keep a way back.

Know the current state first

Record the current container image before updating:

```
docker image inspect --format "{{index .RepoDigests 0}}" traefik/whoami:v1.11.0
# traefik/whoami@sha256:200689790a0a... (your exact digest will differ)
```

That **digest** pins the exact bytes you're running today — more precise than the tag, which a publisher can move. Paste it into your notes. If the new version misbehaves, this line is your rollback target.

Then read release notes for migrations and breaking changes. The word to search for is "migration": a version that rewrites its database on first start often can't be downgraded by swapping the image back. For those, take a fresh backup of the service's volume first — the backup becomes the rollback.

Update and test

Bump the pinned tag in `compose.yaml` (say, to the next release), then:

```
docker compose pull
docker compose up -d
docker compose logs --since 5m web
```

`pull` fetches the new image, `up -d` recreates the container on it, and the logs show the first minute of the new version's life — where migration output and startup errors appear.

Smoke-test through the real hostname and inspect logs:

```
# from your laptop:
curl -I https://homeserver.lab.test
# HTTP/2 200
```

Test through `https://homeserver.lab.test`, not the loopback port, because users take the full path — DNS, proxy, TLS, container — and any layer can be the one the update broke.

Keep the way back open

Keep the prior image or backup until the new version proves stable — a few days of real use, not one successful curl. If you need to roll back and there was no data migration, restore the old pin and recreate:

```
docker compose up -d   # after restoring the previous tag in compose.yaml
```

The failure mode is the floating tag. Do not use an unbounded floating tag for critical stateful services: with `image: app:latest`, a routine `pull` can silently jump major versions, migrate your data, and leave you unable to say what you were running before. Pinned tags make updates boring, and boring is the goal.

Write and rehearse the runbook

Document inventory, access, startup, shutdown, backup, restore, update, and hardware-replacement procedures.

A home server should remain recoverable when you are tired, away, or replacing failed hardware. Six months from now you won't remember which UUID goes in fstab or where the Caddy root certificate lives. The **runbook** holds operational memory outside the machine — written for a future you who remembers nothing.

What goes in it

Create a checklist with no embedded secrets:

```
inventory
network and DNS
accounts and key locations
service start and stop
backup and restore
update and rollback
power-loss recovery
complete rebuild
```

Each heading becomes a short section of commands and facts, in the order you'd need them. Inventory is the hardware and disk map from the storage module. Network and DNS records the reserved address, the subnet, and which device answers `lab.test`. Accounts and key locations says *where* keys and recovery material live — the password manager entry, the drawer — never the secrets themselves, because the runbook will be copied to places with weaker protection than your secrets deserve.

The entries should be commands, not prose:

```
## service start and stop
cd /opt/services/whoami
docker compose up -d      # start
docker compose down       # stop
verify: curl -I https://homeserver.lab.test  ->  HTTP/2 200
```

Every step states how to verify it worked. A runbook that says ”start the service” without the check line still requires the knowledge it was supposed to replace.

Rehearse it

An unrehearsed runbook is fiction — pleasant to have written, wrong in the details. Follow it to rebuild one disposable service from configuration and backup. Pick the whoami service, remove it completely, and rebuild using only what the document says:

```
docker compose down --volumes
# now rebuild from the runbook, restoring config from backup
```

The rule during the drill: if you have to remember something, the runbook failed. Fix every missing assumption you discover — the unstated directory, the firewall rule you forgot existed, the ”oh, right, the CA certificate” moment. Those gaps are the entire value of rehearsing.

Keep it reachable

Store a copy away from the server and keep secret recovery material in an appropriate protected location. A runbook stored only on the machine it describes is unreachable during the exact failures it exists for. A printout in a drawer, a copy in your password manager's notes, a file in your synced documents — any of these works. Re-run one drill whenever the setup changes; the runbook is only current until the next lesson you apply.

Check your understanding: operate and recover

Check the commands, decisions, safety boundaries, and failure cases from operate and recover.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/home-server/.