



MITS.RO / HANDBOOK 07

# VII

---

## *The Machine.*

Run your own things. A complete course, from your first command to a server you trust for years.

HANDBOOK

ASSUMES NOTHING

FREE

*mits.ro*

## CONTENTS

# What is *inside*.

|      |                                  |        |
|------|----------------------------------|--------|
| I    | Before you touch <i>anything</i> | p. 3   |
| II   | The <i>terminal</i>              | p. 6   |
| III  | The <i>machine</i>               | p. 27  |
| IV   | The operating <i>system</i>      | p. 37  |
| V    | The <i>network</i>               | p. 46  |
| VI   | Getting <i>in</i>                | p. 53  |
| VII  | Docker in <i>practice</i>        | p. 58  |
| VIII | The <i>tunnel</i>                | p. 72  |
| IX   | Git and <i>GitHub</i>            | p. 80  |
| X    | Running it for <i>years</i>      | p. 85  |
| XI   | The rented <i>machine</i>        | p. 93  |
| XII  | <i>Reference</i>                 | p. 115 |

### READ THIS FIRST

This book assumes you have never opened a terminal. Every command is explained. Every flag is explained. Every word that sounds like jargon gets a plain definition the first time it appears.

You do not need to read it in order. But if you are starting from zero, Part II is the part that makes everything else make sense. Do not skip it.

## *How to read the boxes*

| BOX     | WHAT IT MEANS                                                  |
|---------|----------------------------------------------------------------|
| Jargon  | A word gets a plain definition. Safe to skip once you know it. |
| Try it  | Something to actually type. Do these.                          |
| Careful | A way to break something. Read before you run.                 |
| Anatomy | A command taken apart piece by piece.                          |

PART ONE

# *Before you touch **anything.***

What a server actually is. What you are signing up for. And the five problems every self-hosted machine has to solve, whether you plan for them or not. If you would rather rent a machine than run one at home, part eleven covers that.

1. What you are building
2. Why do this at all
3. The five jobs
4. What it costs you

## CHAPTER 1

## *What you are **building**.*

A server is a computer that stays on and answers requests. That is the whole idea. There is nothing else to it.

The laptop in front of you is already capable of this. So is a ten-year-old office PC. So is a small box the size of a paperback. What makes something a server is not the hardware. It is the job you give it and the discipline you apply.

Your phone asks a question. The machine answers. That question might be "show me the shopping list" or "what is for dinner" or "here is a photo of a receipt, keep it". The machine sits somewhere, plugged in, and answers.

**JARGON**

**Self-hosting.** Running software on a computer you own, instead of paying a company to run it for you. Your data sits in your house. Nobody reads it. Nobody sells it. Nobody switches it off because a business plan changed.

## *Why do this at **all***

There are four honest reasons. Take the ones that apply to you and ignore the rest.

**You own the data.** Family photos, receipts, notes, medical records. On somebody else's server, those are terms of service. On yours, they are files.

**It does not disappear.** Services close. Prices triple. Features you rely on get removed. A thing running on your own machine keeps running.

**It is cheap over time.** Ten pounds a month for cloud storage is 120 a year, every year. A used mini PC costs 200 once and runs for six.

**You learn how computers actually work.** This one sneaks up on you. After a year of running your own machine you will understand networking, storage and process management better than most people who do this for a living.

**CAREFUL**

Here is the honest cost. You become the support desk. When it breaks at 11pm, nobody else is fixing it. If that sounds like a hobby you would enjoy, this book is for you. If it sounds like a chore, pay someone. That is a fine answer.

## *The five jobs*

Every self-hosted machine has to solve five problems. Most people solve two of them and are surprised when the other three bite.

|                                                  |                                                       |                                                 |                                                           |                                                  |
|--------------------------------------------------|-------------------------------------------------------|-------------------------------------------------|-----------------------------------------------------------|--------------------------------------------------|
| <p>1</p> <p>Stay powered</p> <p>or it sleeps</p> | <p>2</p> <p>Keep an address</p> <p>or links break</p> | <p>3</p> <p>Let you in</p> <p>from anywhere</p> | <p>4</p> <p>Let others in, safely</p> <p>not everyone</p> | <p>5</p> <p>Survive a mistake</p> <p>backups</p> |
|--------------------------------------------------|-------------------------------------------------------|-------------------------------------------------|-----------------------------------------------------------|--------------------------------------------------|

Skip 1 and the machine sleeps when you close the lid.

Skip 2 and every bookmark breaks the day the router reboots.

Skip 3 and you are stuck the first time you leave the house.

Skip 4 or 5 and you lose the data. There is no undo.

### THE FIVE JOBS, AND WHAT SKIPPING EACH ONE COSTS

Most home servers die from job 1 or job 3. Neither is technical. Both are discipline. This book spends a lot of pages on them for that reason.

## PART TWO

# *The terminal.*

A black window with text in it. It looks unfriendly and it is not. Every command follows the same three-part shape, and once you see the shape you can read commands you have never met.

1. What a shell is
2. The anatomy of a command
3. Flags, and what the letters mean
4. Getting help without asking anyone
5. Moving around
6. Reading and writing files
7. Permissions and sudo
8. Pipes and redirection
9. Processes and services

## CHAPTER 2

# What a shell *is*.

The terminal is a window. The shell is the program running inside it that reads what you type and does it.

You have used one before without knowing. When you double-click an icon, something underneath turns that click into "run this program". The shell removes the icon and lets you say it directly.

### JARGON

**Terminal.** The window. It draws text and takes your keystrokes. That is all it does.

**Shell.** The program inside that window which reads a line of text and runs it. On Ubuntu the shell is called `bash`. On a modern Mac it is `zsh`. They are 95% the same for everything in this book.

**Command line.** The whole idea of typing instructions instead of clicking them. Also called CLI, short for command line interface.

## The *prompt*

When the shell is ready it prints a prompt and waits. On your server it looks like this.

```
sam@mits-server:~$
```

Four pieces of information, before you have typed anything.

| PIECE       | MEANING                                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------------------------------|
| sam         | Who you are logged in as. This book uses sam as an example name throughout. Yours will be whatever you chose at install. |
| mits-server | Which machine you are on. Worth checking before you delete anything.                                                     |
| ~           | Which folder you are in. The squiggle means your home folder.                                                            |
| \$          | You are a normal user. A # here means you are root, and root can destroy the machine.                                    |

## IN THIS BOOK

Commands are shown without the prompt, so you can copy them cleanly. When you see output, it follows the command.

*Nothing happening is good*

This trips up everyone at the start. Most Unix commands say nothing when they succeed. You copy a file, you get no message, you get the prompt back. That silence is success.

If something goes wrong you get a line of red-ish text explaining it. So the rule is simple. Silence means it worked. Text means read the text.

## CHAPTER 3

# The anatomy of a *command*.

Every command you will ever type is built from the same four parts. Learn the shape once and you can read a command you have never seen before.

One real command, split into its four parts. The spaces are what separate them.

| docker                                     | run                                                      | -it --rm                                                 | ubuntu bash                                                                |
|--------------------------------------------|----------------------------------------------------------|----------------------------------------------------------|----------------------------------------------------------------------------|
| <b>1 PROGRAM</b>                           | <b>2 SUBCOMMAND</b>                                      | <b>3 OPTIONS</b>                                         | <b>4 ARGUMENTS</b>                                                         |
| The tool you are running.<br>Always first. | Which of the tool's jobs you want. Some tools have none. | How to do it. They start with a dash. Also called flags. | What to do it to. Here: which image, and which program to start inside it. |

Read it aloud as a sentence, left to right:

"Docker. Run something. Interactively, and delete it afterwards. Use the ubuntu image, and start bash."

Not every command has all four parts. `ls` has only a program. `ls -l` has a program and one option.

`git commit -m "fixed it"` has all four.

EVERY COMMAND, TAKEN APART

## Order *matters*, mostly

The program comes first. Always. After that, most programs accept flags and arguments in either order, but some are strict. Put flags before arguments and you will never hit the problem.

```
ls -l /home      # works everywhere
ls /home -l      # works on Linux, fails on some systems
```

## Spaces are *separators*

The shell splits your line on spaces. So a file called `my notes.txt` looks like two arguments, `my` and `notes.txt`. Two ways to fix it.

```
cat "my notes.txt"  # quote the whole thing
cat my\ notes.txt   # or escape the space with a backslash
```

#### HABIT WORTH FORMING

Do not put spaces in filenames on a server. Use a dash or an underscore. Every script you ever write will be simpler.

## CHAPTER 4

# Flags, and what the letters *mean*.

A flag changes how a command behaves. They look cryptic because they are abbreviations. Once you know what the letter stands for, it stops being magic.

## Two *shapes*

| SHAPE | EXAMPLE | NOTES                                                      |
|-------|---------|------------------------------------------------------------|
| Short | -l      | One dash, one letter. Fast to type.                        |
| Long  | --long  | Two dashes, a whole word. Slower, but readable in scripts. |

Most flags come in both. `-h` and `--help` usually do the same thing. Use short ones when typing and long ones when writing something another person will read.

## Combining *short* flags

Short flags stack. These three lines are identical.

```
ls -l -a -h
ls -la -h
ls -lah
```

This is why you see things like `tar -xvzf`. That is four separate flags glued together, not one weird word.

## Flags that take a *value*

Some flags need something after them.

```
git commit -m "a message"    # -m takes the message
docker run -p 8080:80 nginx  # -p takes a port mapping
tail -n 50 app.log           # -n takes a number
```

A flag that takes a value cannot be stacked in the middle. `-lan 50` would be read as the letters l, a, n. Put the value-taking flag last, or write it separately.

## The letters you asked *about*

Here are the ones that come up constantly, with what they actually stand for.

| FLAG    | STANDS FOR                      | WHERE YOU MEET IT                                                                                                                                                                                          |
|---------|---------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -i      | interactive                     | <code>docker run -i</code> keeps input open so you can type into it. <code>rm -i</code> asks before each delete. <code>ssh -i</code> key is different, it means identity file.                             |
| -t      | tty, a terminal                 | <code>docker run -t</code> gives the container a proper terminal so text is formatted and colours work.                                                                                                    |
| -it     | both of the above               | The pair you always see. It means "give me a shell I can actually type in".                                                                                                                                |
| -B      | depends entirely on the program | In <code>grep -B 3</code> it means show 3 lines <b>before</b> each match. In <code>ls -B</code> it means ignore backup files. In <code>cp -b</code> it means make a backup. There is no universal meaning. |
| -d      | detach, or directory, or delete | <code>docker run -d</code> runs in the background. <code>ls -d</code> shows the folder itself not its contents.                                                                                            |
| -v      | verbose, or volume, or version  | <code>ssh -v</code> prints what it is doing. <code>docker run -v</code> mounts a folder. <code>node -v</code> prints the version.                                                                          |
| -r / -R | recursive                       | Apply to a folder and everything inside it. <code>cp -r</code> , <code>rm -r</code> , <code>chmod -R</code> .                                                                                              |
| -f      | force, or file, or follow       | <code>rm -f</code> do not ask. <code>tar -f</code> the file to use. <code>tail -f</code> keep watching.                                                                                                    |
| -a      | all                             | <code>ls -a</code> includes hidden files. <code>docker ps -a</code> includes stopped containers.                                                                                                           |
| -l      | long                            | <code>ls -l</code> one file per line with details.                                                                                                                                                         |
| -h      | human, or help                  | <code>ls -lh</code> shows sizes as 4.2G instead of 4509715660.                                                                                                                                             |
| -p      | port, or parents, or preserve   | <code>docker run -p</code> publish a port. <code>mkdir -p</code> create parent folders too.                                                                                                                |
| -e      | environment                     | <code>docker run -e KEY=value</code> passes a setting in.                                                                                                                                                  |
| -q      | quiet                           | Print less. <code>docker ps -q</code> prints only IDs, handy for scripting.                                                                                                                                |
| -y      | yes                             | <code>apt install -y</code> answers yes to the confirmation.                                                                                                                                               |

### CAREFUL

There is no committee. The same letter means different things in different programs. Never assume. When you meet an unfamiliar flag, run the command with `--help` and read that program's own list. Chapter 5 shows you how.

## *The lonely double dash*

A bare `--` means "no more flags after this point". It exists for files whose names start with a dash.

```
rm -- -weird-file.txt
```

Without the `--`, `rm` would read `-weird-file.txt` as a pile of flags and complain.

## CHAPTER 5

# Getting help without asking *anyone*.

Every command on the machine can explain itself. This is the single most useful habit in this book.

## The fast *one*

```
ls --help
docker run --help
systemctl --help
```

Prints a summary and the full flag list, right there. Start here every time.

## The thorough *one*

```
man ls
```

### JARGON

**man.** Short for manual. Opens the full documentation for a command in a reader.

Inside a man page you are in a pager, so normal typing does nothing. These keys work.

| KEY   | DOES                                      |
|-------|-------------------------------------------|
| Space | Next page                                 |
| b     | Back a page                               |
| /word | Search for "word". Press Enter.           |
| n     | Next search result                        |
| g / G | Jump to the start / the end               |
| q     | Quit. This is the one people cannot find. |

## How to read a man page *synopsis*

Near the top there is a line like this.

## SYNOPSIS

```
cp [OPTION]... SOURCE DEST
```

Three conventions, used everywhere.

| NOTATION        | MEANING                                        |
|-----------------|------------------------------------------------|
| Square brackets | Optional. You can leave it out.                |
| CAPITALS        | Replace this with your own value.              |
| Three dots      | You can repeat this as many times as you like. |

So that line reads "cp takes any number of optional flags, then a source, then a destination".

*When you forget the command **itself***

```

apropos "copy files"    # search man pages by description
type ls                 # what is this thing, exactly
which docker            # where does this program live
  
```

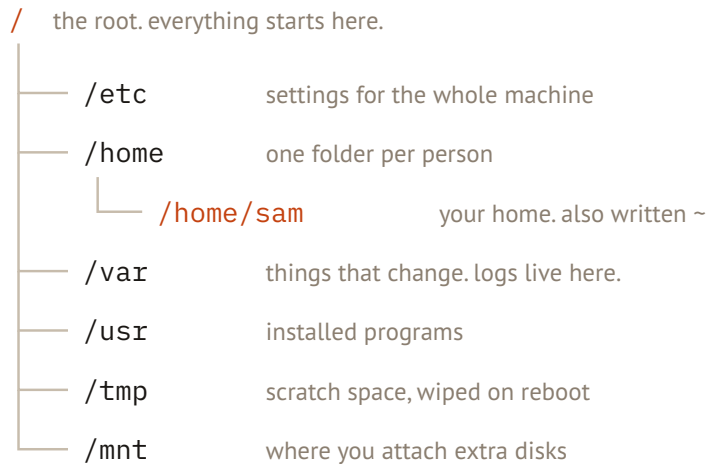
## WORTH INSTALLING

`sudo apt install tldr` then `tldr tar`. It shows the five commands people actually use, instead of the 400-line manual. It is the friendliest thing on a server.

## CHAPTER 6

# Moving *around*.

The filesystem is one tree. Everything hangs off a single root. Once you can picture the tree, navigation is three commands.



No drive letters. One tree. A USB stick becomes a folder inside it.

THE FILESYSTEM IS ONE TREE

## The three *commands*

```

pwd      # print working directory. where am I?
ls       # list. what is here?
cd /var/log  # change directory. go there.
  
```

## Shortcuts that save *hours*

| TYPE THIS    | MEANS                                               |
|--------------|-----------------------------------------------------|
| ~            | Your home folder                                    |
| .            | The folder you are in right now                     |
| ..           | The folder above this one                           |
| cd alone     | Go home                                             |
| cd -         | Go back to where you just were. Like a back button. |
| - in general | Often means "the previous thing"                    |

### JARGON

**Absolute path.** Starts with a slash. `/home/sam/apps`. Means the same thing no matter where you are standing.

**Relative path.** Does not start with a slash. `apps/wiki`. Means "from here". Faster to type, but only correct if you are in the right place. Run `pwd` when confused.

## Tab completion, the most important *key*

Start typing a name and press Tab. The shell finishes it. Press Tab twice to see all the options.

This is not a convenience. It is a correctness tool. If Tab does not complete, the thing you are typing does not exist, and you just caught a typo before running a command on the wrong path.

## Looking at what *ls* tells you

```
ls -lah

drwxr-xr-x  5 sam sam 4.0K Aug 21 14:02 apps
-rw-r--r--  1 sam sam  1.2K Aug 20 09:31 notes.txt
```

## ANATOMY

d rwx r-x r-x 5 sam sam 4.0K Aug 21 14:02 apps

d The type. d is a directory, - is a normal file, l is a shortcut to somewhere else.

rwx r-x r-x Permissions, in three groups of three. Chapter 8 takes these apart.

5 How many links point at it. You can ignore this.

sam sam The owner, then the group.

4.0K The size. Readable because of the -h flag.

Aug 21 14:02 When it last changed.

apps The name.

## CHAPTER 7

# Reading and writing *files*.

On Linux almost everything is a file. Settings are files. Logs are files. Even some hardware appears as a file. So reading and editing files is most of the job.

## Reading

| COMMAND                      | USE IT WHEN                                                          |
|------------------------------|----------------------------------------------------------------------|
| <code>cat file</code>        | The file is short. Dumps the whole thing.                            |
| <code>less file</code>       | The file is long. Scroll with arrows, quit with q.                   |
| <code>head -n 20 file</code> | First 20 lines.                                                      |
| <code>tail -n 20 file</code> | Last 20 lines. Usually what you want in a log.                       |
| <code>tail -f file</code>    | Watch it live. New lines appear as they are written. Ctrl+C to stop. |
| <code>wc -l file</code>      | Count the lines.                                                     |

### THE ONE TO REMEMBER

`tail -f` is how you watch a server do something. Start it, then trigger the thing in another window, and read what happens as it happens.

## Editing with *nano*

There are two famous editors on Linux. `vim` is powerful and has a steep first hour. `nano` is simple and takes 30 seconds. Use nano. You are editing config files, not writing a novel.

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

You get the file and a menu at the bottom. The `^` symbol means Control.

| KEYS   | DOES                                                   |
|--------|--------------------------------------------------------|
| Arrows | Move. There is no mouse.                               |
| Ctrl+O | Save. It asks for the filename, press Enter to accept. |
| Ctrl+X | Exit. It offers to save if you have not.               |
| Ctrl+W | Search.                                                |
| Ctrl+K | Cut the current line.                                  |
| Ctrl+U | Paste it back.                                         |
| Alt+U  | Undo.                                                  |

#### CAREFUL

Editing a file in `/etc` needs `sudo`. If you open it without, nano will let you type happily and then refuse to save. Ten minutes lost. Open it as `sudo nano /etc/...` from the start.

## Copying, moving, *deleting*

```

cp notes.txt backup.txt      # copy
cp -r apps/ apps-backup/    # -r for a whole folder
mv old.txt new.txt          # rename
mv notes.txt ~/documents/    # or move
mkdir -p apps/fam/data       # -p creates every missing parent
rm notes.txt                 # delete a file
rm -r old-folder/            # delete a folder and contents

```

#### CAREFUL

There is no recycle bin. `rm` is permanent and immediate.

Never type `rm -rf /`. Never type `rm -rf $VARIABLE` unless you are certain the variable is set, because if it is empty the command becomes `rm -rf /`. This has destroyed real companies.

The safe habit: run `ls` with the same path first. If `ls` shows what you expect, change `ls` to `rm`.

## CHAPTER 8

# Permissions and *sudo*.

Every file says who may read it, change it, and run it. Once you can read the ten characters, a whole class of confusing error disappears.

These are the first ten characters of one line of `ls -l` output.

|   |     |     |     |
|---|-----|-----|-----|
| - | rwx | r-x | r-- |
|---|-----|-----|-----|

### 1 TYPE

1 character.

- is a file.

d is a folder.

l is a shortcut.

### 2 OWNER

3 characters.

What the account that

owns this file may do.

### 3 GROUP

3 characters.

What accounts in the

file's group may do.

### 4 EVERYONE ELSE

3 characters.

What every other

account may do.

EACH GROUP OF THREE IS ALWAYS `r` THEN `w` THEN `x`

`r` Read the file. Counts as 4. A dash here means not allowed.

`w` Change or delete the file. Counts as 2.

`x` Run the file as a program. On a folder, enter it. Counts as 1.

Add the numbers inside each group. `rw` is  $4+2+1 = 7$ . `r-x` is  $4+0+1 = 5$ . `r--` is  $4+0+0 = 4$ .

So this file is also written as the number 754. That is what `chmod 754` sets.

THE TEN CHARACTERS, DECODED

## Changing *them*

```
chmod +x script.sh      # make it runnable
chmod 600 secret.env    # only the owner may read or write
chmod -R 755 public/    # -R applies to everything inside
chown sam:sam file      # change owner and group
```

| NUMBER | MEANS                                        | USE FOR                           |
|--------|----------------------------------------------|-----------------------------------|
| 600    | Owner reads and writes. Nobody else sees it. | Secrets, .env files, private keys |
| 644    | Owner writes, everyone reads.                | Normal files                      |
| 700    | Owner does everything, nobody else enters.   | Private folders, .ssh             |
| 755    | Owner writes, everyone reads and runs.       | Scripts, program folders          |

## What *sudo* actually does

Your normal account cannot change the machine itself. That is protection, not an obstacle. `sudo` means "do this one command as the administrator".

### JARGON

**root.** The administrator account. It can do absolutely anything, including deleting the operating system while it is running. You never log in as root. You borrow its power one command at a time.

**sudo.** Short for "substitute user do". It asks for **your** password, not root's, then runs that single command with full power.

```
apt install htop          # Permission denied
sudo apt install htop     # works
```

### A TINY TRICK

Forgot the sudo? Type `sudo !!` and press Enter. The `!!` means "the last command I ran". It saves retyping a long line.

### CAREFUL

Pause before every `sudo`. The safety net is off for that line. Most catastrophic server mistakes are a `sudo` command run in the wrong folder or on the wrong machine. Check your prompt says the machine you think it does.

## CHAPTER 9

# Pipes and *redirection*.

This is the idea that makes Unix powerful. Small programs that each do one job, joined together into something bigger.

## The *pipe*

The `|` character takes the output of the command on the left and feeds it in as the input of the command on the right.

```
docker ps | grep postgres
```

Read it as "list the containers, and from that list keep only the lines mentioning postgres".



Each program does one job. The pipe joins them. Chain as many as you like. Nothing is written to disk. It all happens in memory, instantly.

HOW A PIPE WORKS

## *Redirection, to and from files*

| SYMBOL          | DOES                                           | WARNING                                                             |
|-----------------|------------------------------------------------|---------------------------------------------------------------------|
| >               | Send output into a file                        | Wipes the file first. No warning.                                   |
| >>              | Add output to the end of a file                | Safe. Use this by default.                                          |
| <               | Feed a file in as input                        | Rare in daily use.                                                  |
| 2>              | Send only errors into a file                   |                                                                     |
| 2>&1            | Send errors to the same place as normal output | The one people memorise without understanding.                      |
| >/dev/null<br>1 | Throw output away                              | /dev/null is a special file that discards everything written to it. |

### JARGON

Every program has three channels. **stdin** is what goes in. **stdout** is normal output, numbered 1. **stderr** is error output, numbered 2. They are separate so you can keep results and errors apart.

That is why `2>&1` reads as "send channel 2 to wherever channel 1 is going".

## *The tools you will pipe into*

```

grep "error" app.log           # lines containing "error"
grep -i "error" app.log        # -i ignores capitals
grep -v "healthy" app.log      # -v inverts. lines NOT matching.
grep -c "error" app.log        # -c counts instead of printing
grep -B 3 -A 3 "error" app.log # 3 lines Before and After each hit
grep -r "password" .           # -r searches every file below here

sort                           # alphabetical
uniq -c                        # collapse duplicates and count them
wc -l                          # count the lines
head / tail                    # first or last few
awk '{print $1}'               # print just the first column
  
```

A real example. "Which IP addresses hit my server most?"

```
cat access.log | awk '{print $1}' | sort | uniq -c | sort -rn | head -n 10
```

Read right to left in meaning. Take the log, pull the first column, sort it, count duplicates, sort by count descending, show the top ten. Six small tools, one useful answer.

## CHAPTER 10

# Processes and *services*.

A program that is running is called a process. A program that the machine starts on its own and keeps running is called a service. Your server is mostly services.

## Looking at what is *running*

```

htop          # the good one. install it: sudo apt install htop
ps aux        # everything, one line each
ps aux | grep node # just the node processes
top           # always installed, uglier than htop

```

In `htop`, press `F6` to sort by memory, `F10` to quit, and `F4` to filter by name.

## Stopping *something*

```

kill 4821      # ask process 4821 to stop politely
kill -9 4821   # force it. it gets no chance to save.
pkill -f "node app" # kill by name instead of number

```

### CAREFUL

Try `kill` before `kill -9`. A polite stop lets a database finish writing. A forced stop can corrupt it. Reach for `-9` only when the polite version has failed twice.

## *systemd*, the thing that starts everything

### JARGON

**systemd.** The manager that starts services at boot, restarts them if they crash, and collects their logs. On Ubuntu it runs everything. You talk to it with `systemctl` and read its logs with `journalctl`.

```

systemctl status docker      # is it running? why not?
sudo systemctl start docker  # start it now
sudo systemctl stop docker   # stop it now
sudo systemctl restart docker # stop then start
sudo systemctl enable docker # start automatically at boot
sudo systemctl disable docker # do not start at boot
systemctl is-active docker   # prints just: active

```

#### THE DISTINCTION THAT CATCHES PEOPLE

`start` means now. `enable` means at every boot from now on. They are separate. A service you started but never enabled will be gone after a power cut.

`sudo systemctl enable --now docker` does both.

## Reading the *logs*

```
journalctl -u docker           # everything docker has ever said
journalctl -u docker -n 50     # last 50 lines
journalctl -u docker -f        # follow live
journalctl -u docker --since "1 hour ago"
journalctl -p err -b           # errors only, this boot only
journalctl --disk-usage         # how much space are logs eating
```

The `-u` means unit, which is systemd's word for a service. `-b` means this boot.

#### THE DEBUGGING PAIR

When a service will not start, these two lines answer it almost every time.

`systemctl status thing` then `journalctl -u thing -n 50`.

## PART THREE

# *The machine.*

What hardware to buy, and the idea that confuses everyone at the start. Virtual machines, containers, LXC, Docker. They sound like the same thing. They are not, and the difference decides how you build.

1. Laptop, mini PC or tower
2. What virtualization actually is
3. Bare metal, virtual machines, containers
4. LXC explained
5. Docker explained
6. Which one to pick

## CHAPTER 11

# Laptop, mini PC or tower.

An old laptop is a better server than most people think. It has a screen for when things go wrong, a keyboard, and a battery that is a free power backup.

A desk PC dies the instant the power flickers. A laptop keeps running for hours on its own battery. For a machine that has to stay up, that is a real feature and not a small one.

## What each one costs to run

| MACHINE                  | POWER      | BUY      | NOTES                                         |
|--------------------------|------------|----------|-----------------------------------------------|
| A laptop you already own | 10 to 20W  | Nothing  | Battery and screen included                   |
| Low-power mini PC        | 6 to 12W   | Low      | Silent, small, no battery                     |
| Desktop tower            | 40 to 90W  | Varies   | Loud, hot, expensive to run                   |
| Single-board computer    | 5 to 8W    | Low      | Weak for databases. Memory cards wear out.    |
| Used rack server         | 80 to 200W | Very low | Loud enough that you will not keep it indoors |

Work out your own running cost. The formula is short.

$$\text{watts} \div 1000 \times 24 \times 365 \times \text{your price per kWh} = \text{cost per year}$$

A 15W machine on a tariff of 0.20 per unit costs about 26 a year. A 90W tower on the same tariff costs about 158 a year. That gap is roughly the price of a small low-power machine, every single year.

Find your own price per unit on an electricity bill. It is printed there. Put your own number into the formula and the answer is exact instead of approximate.

## What actually matters

1. **An SSD, not a spinning disk.** Databases are unusable on old mechanical drives. This matters more than the processor.
2. **8 GB of memory minimum.** 16 GB if you plan to run several things or any virtual machines.
3. **Wired ethernet if the cable can reach.** Wifi drops. Cable does not.

4. **Any processor from the last ten years.** You are serving a household, not a public website. Processor speed is almost never the thing that limits a home server.

## *What does **not** matter*

Graphics card. Screen quality. Battery health. A laptop with a dead battery still works as a server, you just lose the free backup power.

### IF YOU ARE BUYING

Look for a used low-power mini PC. Buy on four numbers: 4 processor cores, 16 GB of memory, a 500 GB SSD, and a rated power draw under 15W.

That specification is silent, draws roughly 8W in normal use, and runs a dozen containers without complaining. Buy on those four numbers rather than on a model name. Model names change every year. The numbers do not.

## CHAPTER 12

# What virtualization actually *is*.

One physical computer, pretending to be several. That is the whole concept. Everything else is detail about how convincing the pretence is.

Why would you want that? Because a machine running one thing wastes almost everything it has. Your server sits at 3% processor use most of the day. Virtualization lets you run eight separate things on it, each believing it has its own computer, each unable to break the others.

### JARGON

**Host.** The real physical machine.

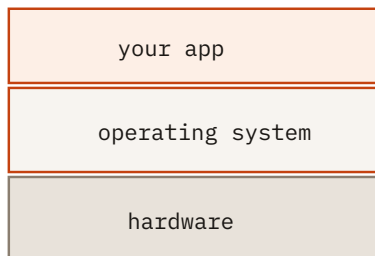
**Guest.** A pretend machine running on it.

**Hypervisor.** The software that does the pretending and shares the real hardware between guests.

**Isolation.** The guarantee that one guest cannot see or damage another. The whole point.

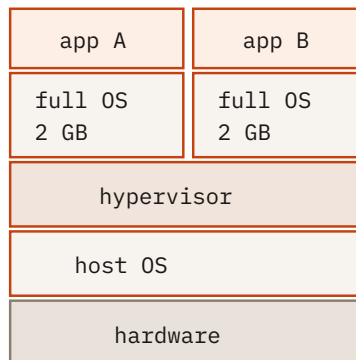
## Three levels of *pretending*

### BARE METAL



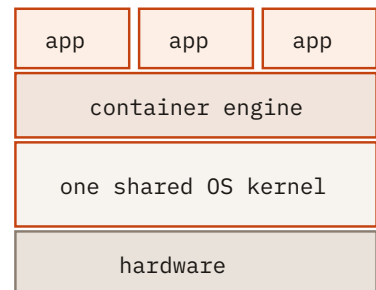
One app. All the power.  
No isolation at all.  
Anything breaks everything.

### VIRTUAL MACHINES



Each guest carries a whole operating system. Heavy.  
Boots in 30 seconds.  
Isolation is total.

### CONTAINERS



No second OS. They share the one underneath.  
Starts in under a second.  
Isolation is good, not total.

The difference is the middle layer. A virtual machine fakes the hardware, so each guest needs its own OS. A container fakes only the operating system, so they all share one kernel. That is why containers are small and fast.

### THREE WAYS TO SHARE ONE COMPUTER

## The *kernel*, in one paragraph

### JARGON

**Kernel.** The core of an operating system. It talks to the hardware and decides which program gets the processor next. Everything else, the commands, the desktop, your apps, sits on top of it and asks it for things.

This single fact explains the whole table above. A virtual machine has its own kernel, so it can run Windows on a Linux host. A container borrows the host's kernel, so it can only run Linux on Linux. That is the trade. You give up the ability to run a different operating system, and you get speed and size in return.

## Two kinds of *hypervisor*

| TYPE               | RUNS ON                  | EXAMPLES                            | USE                                                 |
|--------------------|--------------------------|-------------------------------------|-----------------------------------------------------|
| Type 1, bare metal | Directly on the hardware | Proxmox, ESXi, XCP-ng               | Servers. Faster, because there is no OS in the way. |
| Type 2, hosted     | Inside a normal OS       | VirtualBox, VMware Workstation, UTM | Your laptop. Easy, slower.                          |

## LXC explained.

LXC gives you something that behaves like a whole computer but costs almost nothing to run. It is the middle ground between a virtual machine and Docker.

### JARGON

**LXC.** Linux Containers. A container that holds a complete Linux system, with its own users, its own services, its own package manager. You log into it like a small server. It shares the host's kernel, so it starts in about a second and uses very little memory.

**LXD.** The friendlier manager built on top of LXC. Nowadays often packaged as **Incus**.

**Proxmox.** A complete operating system whose whole job is running virtual machines and LXC containers, managed from a web page. The most common way people meet LXC.

## The two *machinery* words

Containers are built from two Linux features. You do not need to configure either, but knowing the names makes the rest make sense.

| FEATURE | WHAT IT DOES |
|---------|--------------|
|---------|--------------|

|                   |                                                                                                                                                                       |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Namespaces</b> | Control what a process can <b>see</b> . Its own process list, its own network, its own filesystem view. Inside the container, its main process thinks it is number 1. |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                |                                                                                                                                         |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <b>cgroups</b> | Control what a process can <b>use</b> . Cap it at 2 processor cores and 1 GB of memory and it cannot take more, no matter what it does. |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------|

Put plainly: namespaces decide what a process is allowed to look at, and cgroups decide how much hardware it is allowed to consume. A container is a normal Linux process with both of those applied to it. There is no virtual hardware involved and no second operating system.

## Trying it

```
# on Ubuntu, without Proxmox
sudo apt install -y lxd
sudo lxd init --minimal
sudo lxc launch ubuntu:24.04 test1 # use the current LTS number
sudo lxc list
sudo lxc exec test1 -- bash      # you are now inside it
exit
sudo lxc stop test1
sudo lxc delete test1
```

### ANATOMY

```
sudo lxc exec test1 -- bash
```

**exec** Run a command inside a container that is already running.

**test1** Which container.

**--** Everything after this belongs to the container, not to lxc. Without it, lxc would try to interpret the rest as its own flags.

**bash** The command to run inside. A shell, so you get a prompt.

## On Proxmox

Proxmox is a Debian-based system you install instead of Ubuntu. You then manage everything from a browser at <https://your-ip:8006>.

| STEP                     | WHERE                                          |
|--------------------------|------------------------------------------------|
| Download a template      | Storage, CT Templates, Templates               |
| Create the container     | Create CT, top right                           |
| Give it 2 cores and 1 GB | In the wizard. Change it later freely.         |
| Start and enter it       | Select it, Console                             |
| Back it up               | Backup, Backup now. Whole container, one file. |

That last row is why people love Proxmox. A whole running system becomes one file you can restore in two minutes.

#### CAREFUL

Running Docker inside an LXC container works but needs the container marked as privileged or specially configured. It is a known source of strange failures. If you want both, the cleaner shape is a proper virtual machine for Docker, and LXC for everything else.

## Docker explained.

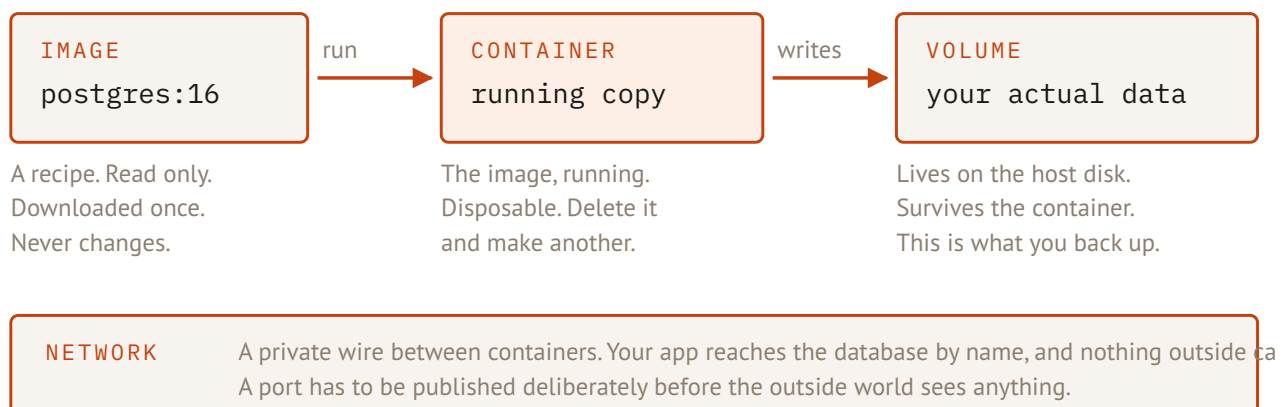
LXC gives you a small computer. Docker gives you a packaged application. Same underlying technology, completely different intent.

The difference is how you treat them over time. With LXC you log in, install things, edit config files by hand, and keep that same container for years.

With Docker you never log in to change anything. When you want a new version you delete the container and start a fresh one from an updated image. Any change you had made by hand inside it would be lost, which is the intended behaviour rather than a fault.

This is why Docker images are called immutable. An image never changes after it is built. Every change is made by building a new image.

### The four words



Delete a container and you lose nothing. Delete a volume and the data is gone forever.  
If you remember one thing about Docker, remember that sentence.

IMAGE, CONTAINER, VOLUME, NETWORK

### Why anyone bothers

**It runs the same everywhere.** The image contains the app and every library it needs, at the exact versions it was built with. It cannot work on your laptop and fail on the server.

**Uninstalling is complete.** Delete the container and the image. Nothing is left behind. No stray config in six folders.

**Versions do not fight.** One app needs an old database, another needs a new one. On a normal machine that is a bad afternoon. In containers they simply do not meet.

**Someone already packaged it.** Almost every open source tool ships an official image. Running it is one line instead of an install guide.

## LXC or Docker

| QUESTION                         | LXC                          | DOCKER                          |
|----------------------------------|------------------------------|---------------------------------|
| Feels like                       | A small computer             | A packaged app                  |
| You log in and configure it      | Yes, that is the point       | No, you rebuild instead         |
| Runs several services at once    | Yes, naturally               | One per container by convention |
| Survives a reboot with its state | Yes                          | Only what is in volumes         |
| Ready-made software              | You install it yourself      | Thousands of official images    |
| Backup                           | Snapshot the whole container | Back up the volumes             |
| Best for                         | A machine you tend           | An app you ship                 |

### THE HONEST RECOMMENDATION

For a home server running apps, use **Docker**. Almost everything you want is already packaged, updates are one command, and backups are a folder.

Reach for **LXC** when you want a whole second machine to tinker in, or when you are running Proxmox anyway and want cheap isolated systems.

You do not have to choose forever. Many people run Proxmox, put one virtual machine on it for Docker, and use LXC for the rest.

PART FOUR

# *The operating system.*

Which Linux, and why. Then the install, screen by screen, and the thirty minutes afterwards that decide whether this machine is still running in three years.

1. Why Ubuntu Server LTS
2. The alternatives, honestly
3. Installing, screen by screen
4. The first thirty minutes
5. Making it stay on

## Why Ubuntu Server *LTS*.

Because when something breaks at midnight, the answer you find online will be written for Ubuntu. That is the entire argument, and it is a good one.

### JARGON

**Distribution, or distro.** Linux itself is just the kernel. A distribution wraps it with a package manager, default settings and a release schedule. Ubuntu, Debian and Fedora are all Linux with different wrapping.

**LTS.** Long Term Support. Five years of security updates with no forced upgrades. The non-LTS releases get nine months, which is a bad deal for a server.

**Package manager.** The program that installs software. On Ubuntu it is `apt`. It handles downloading, dependencies and updates.

## Server, not *Desktop*

Pick the Server edition. No graphical interface means less software to attack, less to update, and a few hundred megabytes of memory back. You will drive it over SSH anyway.

## The alternatives, *honestly*

| OPTION                   | WHAT IT IS                                  | PICK IT WHEN                                                       |
|--------------------------|---------------------------------------------|--------------------------------------------------------------------|
| <b>Ubuntu Server LTS</b> | The default answer                          | You want the largest amount of documentation on earth. Start here. |
| Debian Stable            | Ubuntu's parent. Slower, more conservative. | You want fewer surprises and do not mind older package versions.   |
| Proxmox VE               | Debian plus a virtualization control panel  | You want virtual machines and LXC managed from a browser.          |
| TrueNAS Scale            | Storage first, apps second                  | The machine is mainly a file store with many disks.                |
| Unraid                   | Paid, very friendly, mixed disk sizes       | You want it to just work and will pay for that.                    |
| CasaOS or Umbrel         | An app-store layer over Docker              | You want clicking, not typing. Harder to fix when it breaks.       |
| Fedora Server            | Newest everything                           | You enjoy upgrades. Not a fit for a set-and-forget box.            |
| Arch                     | Build it yourself                           | Never, for a first server. It is a hobby, not a foundation.        |

### THE SHORTCUT

If you want one machine that runs apps, install **Ubuntu Server LTS** and use Docker.

If you want to slice one machine into several, install **Proxmox** and put Ubuntu inside it.

Both are correct. The second is more machine to learn.

## CHAPTER 16

# Installing, screen by *screen*.

Twenty minutes, most of it waiting. The only screens that matter are the five below.

## Making the USB *stick*

1. Download the Server LTS image from [ubuntu.com](https://ubuntu.com). It is about 2 GB and ends in `.iso`.
2. Download Balena Etcher, or use the Raspberry Pi Imager. Both are free and work on Mac and Windows.
3. Write the image to a USB stick of 4 GB or more. Everything on that stick is erased.
4. Put the stick in the server, power on, and press the boot menu key. It is F12 on most machines, F9 on HP, F2 or Del on others. The screen tells you for one second.

## The five screens that *matter*

| SCREEN            | CHOOSE                               | WHY                                             |
|-------------------|--------------------------------------|-------------------------------------------------|
| Installation type | Ubuntu Server (minimized)            | Fewer packages, smaller attack surface          |
| Storage layout    | Use entire disk. <b>Uncheck LVM.</b> | LVM adds a layer you will never use on one disk |
| Encryption        | Skip it                              | See the warning below. This one matters.        |
| Profile           | Short username, no spaces            | You will type it for years                      |
| SSH setup         | <b>Tick "Install OpenSSH server"</b> | Without it you cannot log in remotely at all    |

### CAREFUL

**Do not encrypt the disk on a server.** It sounds safer. What it means in practice is that after a power cut the machine boots to a password prompt and sits there. Nobody is home. Your apps are down until someone walks over and types.

Encryption is right for a laptop you carry. It is wrong for a machine that must come back on its own.

## The GitHub *trick*

At the SSH screen Ubuntu offers to import your public keys from GitHub. Type your GitHub username and it pulls the keys you already use.

You are then key-only from the very first boot, with no password to manage or leak. If that means nothing yet, Part VI explains it.

## *First boot*

Pull the USB stick out and let it boot. You get a text login. Type your username and password. You are in.

Find its address so you can leave the keyboard behind.

```
ip a
```

Look for a line starting `inet` that is not `127.0.0.1`. It will look like `192.168.1.42/24`. That number is the machine. From your laptop, on the same wifi, you can now do this and never touch the server keyboard again.

```
ssh yourname@192.168.1.42
```

## CHAPTER 17

# *The first thirty minutes.*

Everything here runs once, right after install. Skipping it is how machines get owned.

## *Update everything*

```
sudo apt update && sudo apt upgrade -y
sudo reboot
```

### ANATOMY

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

**apt update** Refresh the list of what is available. Downloads no software.

**&&** Run the next command only if this one succeeded. A single **&** would mean something else entirely.

**apt upgrade** Actually install the newer versions.

**-y** Answer yes to the confirmation, so it does not wait for you.

## *Turn on automatic security updates*

```
sudo apt install -y unattended-upgrades
sudo dpkg-reconfigure --priority=low unattended-upgrades
```

This installs security patches on its own, forever. On a machine you will forget about for months, it is the single highest value command in this book.

## *Name it and set the clock*

```
sudo hostnamectl set-hostname mits-server
timedatectl list-timezones | grep -i madrid # find yours
sudo timedatectl set-timezone Europe/Madrid # then set it
```

Replace Madrid with your own city. The list uses Region/City names, so search it for a large city near you.

Get the timezone wrong and every log line, backup filename and scheduled job lands at the wrong hour. You will chase that for an evening.

## Lock down *SSH*

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

Find these two lines, remove any `#` in front, and set them.

```
PasswordAuthentication no  
PermitRootLogin no
```

```
sudo systemctl restart ssh
```

### CAREFUL

**Open a second terminal and test the login before you close the first one.**

If you have turned off passwords and your key does not work, and you have closed your only session, you are walking to the machine with a keyboard. Test first, every time.

## Firewall and *fail2ban*

```
sudo ufw allow OpenSSH  
sudo ufw enable  
sudo ufw status  
  
sudo apt install -y fail2ban
```

### JARGON

**ufw.** Uncomplicated Firewall. Blocks everything by default and lets through only what you allow. Note the order above. Allow SSH **before** enabling, or you lock yourself out.

**fail2ban.** Watches for repeated failed logins and bans the source address. Set it and forget it.

Apps published through a tunnel need no open ports at all, because the tunnel dials out. That is Part IX.

## CHAPTER 18

# *Making it stay **on**.*

This is where home servers actually die. Not hacked. Asleep.

## *Stop the lid from sleeping **it***

A laptop suspends when you close the lid. A suspended server is an offline server.

```
sudo nano /etc/systemd/logind.conf
```

Set these three lines, removing any `#` in front.

```
HandleLidSwitch=ignore
HandleLidSwitchDocked=ignore
HandleLidSwitchExternalPower=ignore
```

```
sudo systemctl restart systemd-logind
```

Then close the lid and check it still answers. Do not assume.

## *Stop it sleeping when **idle***

```
sudo systemctl mask sleep.target suspend.target hibernate.target hybrid-sleep.target
```

### JARGON

**mask** is stronger than **disable**. Disabled means it will not start on its own. Masked means nothing can start it at all, even if another program asks. For sleep on a server, masked is what you want.

## *Come back after a power **cut***

The power returns at 3am. Does the machine turn itself on?

For a desktop or mini PC this is a BIOS setting, not a Linux one. Reboot, press Del or F2 to enter setup, and look for **Restore on AC Power Loss** or **After Power Failure**. Set it to **Power On**. The factory default is usually Stay Off, which is wrong for a server.

Laptops mostly handle this themselves, because the battery never lets them fully lose power.

## Protect the *battery*

A laptop held at 100% forever will swell. Many machines can cap the charge.

```
sudo apt install -y tlp
sudo nano /etc/tlp.conf      # set STOP_CHARGE_THRESH_BAT0=80
sudo systemctl enable --now tlp
```

On ThinkPads and many HP models this works out of the box. On others it does not, and that is fine. Just check every few months that the battery is not physically bulging. A swollen battery will lift the trackpad off the chassis. Remove it and run on mains.

### TRY IT

#### PROVE THE MACHINE SURVIVES

Close the lid. Pull the power cable. Plug it back in. Walk away for ten minutes.

Then check the app answers. If it does, job 1 is solved. If it does not, you have found the fault at a time of your choosing, rather than at 3am on a Sunday.

Do this once. It is the difference between hoping and knowing.

PART FIVE

# *The network.*

Addresses, ports and names. Four ideas that explain almost every "I cannot reach my server" problem you will ever have, including the one that will catch you first.

1. Addresses in plain words
2. Ports
3. DNS, names into numbers
4. NAT, and why nobody can reach you
5. Checking your network

## Addresses in plain *words*.

Every device on a network has a number. There are two kinds, and confusing them is the most common self-hosting mistake there is.

| KIND    | LOOKS LIKE                             | REACHABLE FROM                                       |
|---------|----------------------------------------|------------------------------------------------------|
| Private | 192.168.1.42<br>10.0.0.5<br>172.16.x.x | Only inside your house. Meaningless anywhere else.   |
| Public  | 188.24.144.61                          | The whole internet. You get one, from your provider. |

### THE MISTAKE EVERYONE MAKES ONCE

**A 192.168 address only exists inside your house.** It is not a real internet address.

Your server at 192.168.1.42 works on your own wifi. From a hotel, a cafe or a friend's house, that address reaches nothing at all. Millions of other networks also use 192.168.1.42, so the number carries no meaning outside the one network it belongs to. Nothing is broken. You have left the only network where that address exists.

This is not something you can configure your way out of. Private ranges are reserved by international agreement so that they are never carried across the public internet. Part eight covers the two working ways to reach a machine that has no public address.

## The */24* at the end

You will see addresses written as 192.168.1.42/24. The /24 says how much of the address is the street and how much is the house.

One address, as it is written on a Linux machine.

192.168.1.

42

/24

#### 1 THE NETWORK

Every device on this network shares these first three numbers.

#### 2 THE DEVICE

Yours alone, on this network. No two devices share it.

#### 3 THE SPLIT

Says how many bits belong to the network. 24 of 32 here.

An address has 32 bits, written as 4 numbers of 8 bits each.

/24 means the first 24 bits are the network. That leaves 8 bits, which is 256 combinations.

Two are reserved, so 254 devices fit. That is why home networks are almost always /24.

Two devices can talk directly only when their network part matches exactly.

#### READING AN ADDRESS

## Give the server a fixed *address*

By default your router hands out addresses temporarily and can change them. Your server needs one that never moves. There are two ways and one is right.

#### METHOD

#### VERDICT

Static IP set on the machine

Avoid. The machine can claim an address the router later gives to something else. Two devices, one address, a bad evening.

**DHCP reservation on the router**

**Do this.** The router always hands this one machine the same address. One place to look, no conflicts.

To set one, find the server's hardware address, then bind it in the router.

```
ip link show
```

```
# the MAC address, like d4:b7:09:e9:0b:08
```

Then open your router at `192.168.1.1`, find DHCP, then Address Reservation or Static Lease, and bind that MAC to something above the normal pool. `192.168.1.200` is a good choice, because guest phones never reach that high.

#### JARGON

**MAC address.** A permanent serial number burned into a network card. Six pairs of hex digits. Unlike an IP address it never changes, which is why the router uses it to recognise a machine.

**DHCP.** The service in your router that hands out addresses automatically. A **reservation** tells it "this machine always gets that number".

# *Ports, DNS and NAT.*

Three more ideas and the network is no longer mysterious.

## *Ports*

An address identifies the machine. A port number identifies which program on that machine should receive the traffic.

One machine runs many programs at once. Without port numbers, arriving data would have no way of saying which program it belongs to. So every connection carries an address and a port, and the operating system hands the data to whichever program claimed that port number.

| PORT       | USUALLY                         |
|------------|---------------------------------|
| 22         | SSH                             |
| 80         | Web, unencrypted                |
| 443        | Web, encrypted. The normal one. |
| 5432       | PostgreSQL                      |
| 3306       | MySQL                           |
| 3000, 8080 | Whatever you are building       |

Only one program may hold a port at a time. "Address already in use" means something is already there. Find it.

```
sudo ss -tulpn | grep 3000
```

#### ANATOMY

`sudo ss -tulpn`

|                 |                                                                     |
|-----------------|---------------------------------------------------------------------|
| <code>ss</code> | Socket statistics. What is listening on this machine.               |
| <code>-t</code> | TCP connections.                                                    |
| <code>-u</code> | UDP too.                                                            |
| <code>-l</code> | Only things listening, not existing connections.                    |
| <code>-p</code> | Show which program it is. Needs sudo to see other users' processes. |
| <code>-n</code> | Numbers, not names. Faster, and shows the real port.                |

## *DNS, names into numbers*

Computers reach each other by number. People do not remember numbers. DNS is the service that takes a name and returns the address behind it.

Every time you open a web page your machine asks a DNS server for the address first, then connects to that address. The lookup and the connection are two separate steps. That is why a site can fail to load for two completely different reasons.

```
dig +short cristea.mits.ro
nslookup cristea.mits.ro
```

#### RECORD

#### MEANS

|       |                                                   |
|-------|---------------------------------------------------|
| A     | This name points at this IPv4 address             |
| AAAA  | Same, for IPv6                                    |
| CNAME | This name is an alias for another name            |
| MX    | Send this domain's email here                     |
| TXT   | Free text. Used for verification and email rules. |

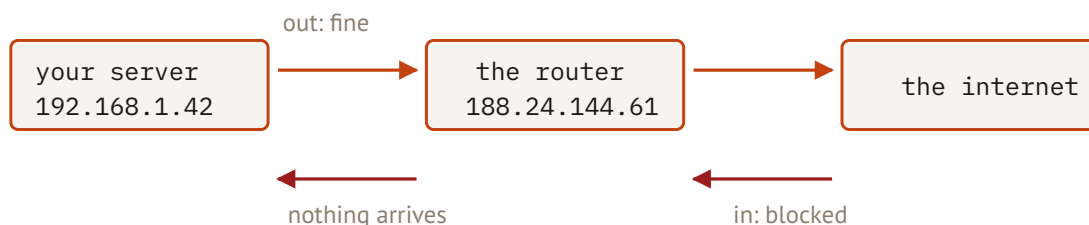
#### THE OLDEST JOKE IN THE TRADE

"It is not DNS. There is no way it is DNS. It was DNS."

When something worked yesterday and does not today, check the name resolves before you check anything else. It takes five seconds.

## NAT, and why nobody can reach you

You have one public address and fifteen devices. NAT is the trick that makes that work. Your router rewrites outgoing traffic so it all appears to come from the one public address, and remembers who asked for what so replies get back to the right device.



Outbound works because the router remembers you asked. Inbound has no such record, so it is dropped. This is why a home server is invisible from outside by default. It is a feature, not a fault. Part IX turns this on its head: the tunnel dials OUT, so inbound is never needed.

### WHY THE WORLD CANNOT REACH YOUR SERVER

#### JARGON

**CGNAT.** Carrier grade NAT. Your provider does the same trick again, sharing one public address between hundreds of customers. If you have this, port forwarding cannot work at all, no matter what you configure.

To check: compare the address your router reports as its WAN address with what `curl ifconfig.me` says. If they differ, you are behind CGNAT. Addresses in the range 100.64 to 100.127 are a giveaway.

## Checking your network

```

ip a                # my addresses
ip route            # where traffic goes by default
ping 192.168.1.1    # can I reach the router
ping 1.1.1.1        # can I reach the internet by number
ping google.com     # can I resolve names
curl ifconfig.me    # what is my public address
traceroute 1.1.1.1  # every hop along the way
sudo ss -tulpn      # what is listening here
  
```

### THE ORDER MATTERS

Run those first four in that exact order. The first one that fails tells you where the problem is.

Router fails, it is your cable or wifi. 1.1.1.1 fails, it is your provider. Names fail but numbers work, it is DNS. All four pass and the app is still unreachable, the problem is the app, not the network.

PART SIX

# *Getting in.*

SSH is how you drive a machine with no screen. Keys instead of passwords, a config file that saves you typing forever, and a method for the day it stops working.

1. What SSH is
2. Keys, not passwords
3. The config file
4. Copying files
5. When SSH fails

## Keys, not *passwords*.

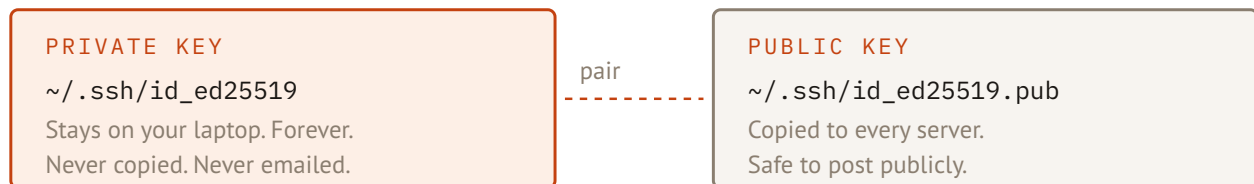
SSH gives you a shell on another machine, encrypted. Everything in this book happens through it.

```
ssh sam@192.168.1.42
```

Username, then `@`, then the machine. You get a prompt on the other computer, and every command runs there.

### How keys *work*

A key is two files that belong together. This is the part worth understanding properly, because it removes all fear of using it.



The server sends a puzzle that only the matching private key can solve.  
Your laptop solves it and sends back the answer. The private key never leaves your machine.  
Nothing reusable crosses the network, so there is no password to steal, guess or reuse.

This is why key-only login is not just more convenient. It is strictly safer.

#### WHY A KEY BEATS A PASSWORD

### Making *one*

```
ssh-keygen -t ed25519 -C "codrut laptop"
```

#### ANATOMY

```
ssh-keygen -t ed25519 -C "codrut laptop"
```

**-t ed25519** The type of key. Ed25519 is modern, short and fast. Use it unless something very old refuses, in which case `-t rsa -b 4096`.

**-C** A comment. It gets appended to the public key so you can tell which laptop it came from years later.

It asks where to save it. Press Enter. It asks for a passphrase. Setting one is safer, because a stolen laptop then does not hand over your servers. Your operating system will remember it after the first use.

### *Putting it on the **server***

```
ssh-copy-id sam@192.168.1.42
```

One command. It asks for your password one final time, appends your public key to the server's `~/.ssh/authorized_keys`, and from then on you log in with no password at all.

#### CAREFUL

Permissions on `.ssh` are strict on purpose, and SSH silently refuses keys if they are wrong. If your key is ignored for no visible reason, this is almost always why.

```
chmod 700 ~/.ssh and chmod 600 ~/.ssh/authorized_keys.
```

### *The config file that saves you **forever***

Typing `ssh sam@192.168.1.42 -i ~/.ssh/some_key -p 2222` gets old immediately. Write it down once instead.

```
nano ~/.ssh/config
```

```
Host mits
  HostName 192.168.1.42
  User sam
  IdentityFile ~/.ssh/id_ed25519

Host mits-remote
  HostName ssh-mits.mits.ro
  User sam
  ProxyCommand cloudflared access ssh --hostname %h
```

Now `ssh mits` is the whole command. The same names work for `scp` and `rsync`.

## Copying *files*

```

scp notes.txt mits:~/                # one file up
scp mits:~/backup.sql ./              # one file down
scp -r ./site mits:~/apps/           # -r for a folder

rsync -avz ./site/ mits:~/apps/site/ # better for repeats

```

### ANATOMY

```
rsync -avz ./site/ mits:~/apps/site/
```

**-a** Archive. Keeps permissions and timestamps, and recurses into folders.

**-v** Verbose. Print what it is doing.

**-z** Compress during transfer. Faster over a slow link.

**trailing /** Matters. `site/` means "the contents of site". `site` means "the folder itself". Get it wrong and you nest a folder inside itself.

`rsync` only sends what changed. On the second run of a big folder it finishes in a second.

## When *SSH fails*

```

ssh -v mits      # show me what you are trying
ssh -vvv mits    # show me everything

```

## MESSAGE

## ALMOST ALWAYS MEANS

|                               |                                                                                                                         |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| Connection refused            | The machine is up but nothing is listening on port 22. SSH is not running, or the firewall blocks it.                   |
| Connection timed out          | Nothing answered at all. Wrong address, machine off, or you are on a different network.                                 |
| No route to host              | The address does not exist on this network. Usually you left the building.                                              |
| Permission denied (publickey) | You reached the server. It does not accept your key. Check permissions, check you copied the right key.                 |
| Host key verification failed  | The server's identity changed. Fine after a reinstall, alarming otherwise. Remove the old line from ~/.ssh/known_hosts. |

## THE LESSON THAT COST A NIGHT

Never let a private address be your only way in. Set up a second route while you are standing next to the machine, not on the evening you need it.

Tailscale takes three minutes and removes this entire class of problem. `curl -fsSL https://tailscale.com/install.sh | sh` then `sudo tailscale up --ssh`. Install it on your laptop too and the two machines find each other from anywhere, with no ports and no DNS.

PART SEVEN

# *Docker in practice.*

The concepts from Part three, now as commands. Every flag on `docker run` explained, how to look inside something that is running, and what to do when a container will not start.

1. Installing it
2. `docker run`, flag by flag
3. Looking inside
4. Compose
5. Where your data lives
6. Debugging containers
7. Cleaning up

## CHAPTER 22

# Installing and first *run*.

```
curl -fsSL https://get.docker.com | sudo sh
sudo usermod -aG docker $USER
```

Log out and back in, then check.

```
docker ps
```

An empty table means it works. An error about permissions means the group change has not taken effect yet, so log out properly.

### ANATOMY

```
curl -fsSL https://get.docker.com | sudo sh
```

- f** Fail quietly on server errors instead of downloading an error page.
- s** Silent. No progress bar.
- S** But do show errors. `-sS` together means quiet unless something breaks.
- L** Follow redirects. Without it you often download nothing.
- | sudo sh** Pipe the downloaded script into a shell as root.

### CAREFUL

Piping a URL straight into a shell runs whatever that server sends, as root. It is fine for Docker's official domain and it is a terrible habit everywhere else.

When in doubt, download first and read it: `curl -fsSL https://... -o s.sh`, then `less s.sh`, then `sudo sh s.sh`.

### THE DOCKER GROUP, HONESTLY

Being in the `docker` group is effectively root on that machine. Docker can mount any folder into a container, including the whole disk.

On a single-user home server where you already have full `sudo`, this changes almost nothing. On a machine with other people on it, think harder.

## *Your first container*

```
docker run -it --rm ubuntu bash
```

You are now inside a fresh Ubuntu, on your server, in about a second. Type `ls`, poke around, then `exit`. It is gone completely, and your server is untouched.

That is the thing worth feeling early. A container is cheap and disposable. You cannot break the host by experimenting inside one.

## *docker run, flag by flag.*

This one command has more flags than any other you will meet. Here are the ones that matter, in the order you will need them.

### ANATOMY

```
docker run -d --name web -p 8080:80 -v data:/var/lib/x -e KEY=val -
-restart unless-stopped nginx:alpine
```

- d** Detach. Run in the background and give the prompt back. Without it, the container owns your terminal until you stop it.
- name web** A name you choose. Without it Docker invents one like `nostalgic_hoover`, and you will not remember which is which.
- p 8080:80** Publish a port. **Left is the host, right is the container.** So visitors hit 8080 on your server and arrive at 80 inside. Get the order backwards and nothing works.
- v data:/var/lib/x** Mount storage. Left is the volume name or a host folder, right is the path inside. This is the difference between keeping your data and losing it.
- e KEY=val** Pass a setting in as an environment variable. How almost all containers are configured.
- restart** What to do if it stops. `unless-stopped` means bring it back after a crash or reboot, unless I stopped it deliberately. Put this on everything.
- nginx:alpine** The image, then a colon, then the tag. The tag is the version.

## The rest, in one *table*

| FLAG            | DOES                                                                |
|-----------------|---------------------------------------------------------------------|
| -it             | Interactive plus a terminal. For shells you type into.              |
| --rm            | Delete the container when it exits. For throwaway experiments.      |
| --network mynet | Put it on a named network so it can reach other containers by name. |
| --env-file .env | Read many settings from a file instead of ten -e flags.             |
| -u 1000:1000    | Run as this user id, not root.                                      |
| --memory 512m   | Cap its memory. It cannot take more.                                |
| --cpus 1.5      | Cap its processor share.                                            |
| --health-cmd    | Tell Docker how to check it is truly alive, not just running.       |
| -w /app         | Start in this folder inside the container.                          |

## Tags, and the word *latest*

```

postgres:16      # good. a real version.
postgres:16.2    # better. exact.
postgres:latest  # trouble.
postgres         # same as :latest
  
```

### CAREFUL

`latest` is not a version. It is a moving label. Pull it again in six months and you get a different database, possibly one that cannot read your files.

Pin a real version on anything holding data. Databases especially.

## Publishing a port, *safely*

```

-p 8080:80      # the whole network can reach it
-p 127.0.0.1:8080:80 # only this machine can
  
```

The second form is what you want for databases and for anything that sits behind a tunnel. Nothing outside the machine can touch it, and the tunnel or the reverse proxy reaches it locally.

#### THE MISTAKE THAT LEAKS DATABASES

Writing `-p 5432:5432` on a database publishes it to your whole network, and if your router forwards that port, to the internet. Every open Postgres on earth gets found within hours.

Bind it to `127.0.0.1` or do not publish it at all. Containers on the same Docker network reach each other without any published port.

## CHAPTER 24

# Looking *inside*.

Six commands cover almost every question you will have about a running container.

```

docker ps                # what is running
docker ps -a             # including stopped ones
docker logs web           # what has it printed
docker logs -f --tail 100 web # last 100 lines, then follow live
docker exec -it web bash  # get a shell inside it
docker inspect web        # every setting, as JSON
docker stats              # live memory and processor use
docker top web            # processes inside it
  
```

### ANATOMY

`docker exec -it web bash`

**exec** Run an extra command inside a container that is already running. Different from `run`, which starts a new one.

**-it** Interactive with a terminal, so you get a usable prompt.

**web** Which container, by name or id.

**bash** What to run. On small images bash may not exist, so use `sh` instead.

# Starting, stopping, *removing*

```

docker stop web          # polite stop, 10 seconds to finish
docker start web          # start it again
docker restart web
docker rm web             # delete the container. must be stopped.
docker rm -f web          # stop and delete in one go
  
```

Deleting a container is safe as long as your data is in a volume. That is the whole reason volumes exist.

## Where your data actually *lives*

| WRITTEN AS                            | CALLED          | LIVES AT                                         |
|---------------------------------------|-----------------|--------------------------------------------------|
| -v<br>pgdata:/var/lib/postgresql/data | Named volume    | Docker manages it, under /var/lib/docker/volumes |
| -v ./data:/app/data                   | Bind mount      | A real folder you can see and back up directly   |
| nothing                               | Container layer | <b>Gone when the container is deleted</b>        |

```

docker volume ls
docker volume inspect pgdata
docker system df           # how much space is all this using

```

### WHICH TO USE

Use **bind mounts** for things you want to see and back up with ordinary tools. Use **named volumes** for database internals, where performance matters and you never open the files by hand.

Either way, that folder is what you back up. Not the container.

## *Compose, the way you will actually **run things**.*

Nobody types long `docker run` lines twice. You write the whole setup in one file and start it with two words.

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

```
services:
  db:
    image: postgres:16
    restart: unless-stopped
    environment:
      POSTGRES_USER: fam
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_DB: fam_cristea
    volumes:
      - db-data:/var/lib/postgresql/data
    ports:
      - "127.0.0.1:5433:5432"

  app:
    build: .
    restart: unless-stopped
    depends_on:
      - db
    environment:
      DATABASE_URL: postgres://fam:${DB_PASSWORD}@db:5432/fam_cristea
    volumes:
      - app-data:/data
    ports:
      - "127.0.0.1:3001:3000"

volumes:
  db-data:
  app-data:
```

## Reading that *file*

| KEY         | MEANS                                                                                                   |
|-------------|---------------------------------------------------------------------------------------------------------|
| services    | The containers. Each name under it becomes one.                                                         |
| image       | Use a ready-made image from the internet.                                                               |
| build: .    | Build from a Dockerfile in this folder instead.                                                         |
| restart     | Come back after a crash or reboot. Put it on everything.                                                |
| environment | Settings passed in. <code>\${DB_PASSWORD}</code> is read from a <code>.env</code> file beside this one. |
| volumes     | Storage that survives. Listed again at the bottom to declare them.                                      |
| ports       | Host port, then container port. Bound to 127.0.0.1 so only this machine reaches them.                   |
| depends_on  | Start order only. It does <b>not</b> wait for the database to be ready, just started.                   |

### THE QUIET SUPERPOWER

Notice `@db:5432` in the app's database URL. Compose puts every service on one private network and each is reachable **by its service name**.

So the app finds the database at `db`. No IP addresses, no published port needed. Nothing outside can reach it at all.

## Running *it*

```

docker compose up -d           # start everything, in the background
docker compose up -d --build   # rebuild images first
docker compose ps              # what is running
docker compose logs -f         # follow all logs
docker compose logs -f app     # just one service
docker compose restart app
docker compose down            # stop and remove containers
docker compose down -v         # AND DELETE THE VOLUMES
docker compose exec app sh     # shell inside a service
docker compose pull            # fetch newer images
    
```

#### CAREFUL

`docker compose down -v` deletes your volumes. That is your database. There is no confirmation and no undo.

`down` on its own is safe and is what you almost always want.

## Updating an *app*

```
cd ~/apps/wiki
git pull
docker compose up -d --build
docker compose logs -f --tail 50
```

Four lines. Compose rebuilds what changed, replaces the containers, and leaves the volumes alone. Write it into a small script so you cannot get it wrong at 1am.

## Debugging *containers*.

A container that will not start is the most common problem you will have. It is also the most methodical to solve.

### *The order*

1. **Is it running?** `docker compose ps`. Look at the STATUS column. "Restarting" over and over means it crashes on boot.
2. **What did it say?** `docker compose logs --tail 100 app`. The answer is here about 80% of the time. Read the **first** error, not the last.
3. **Did it exit with a code?** `docker ps -a` shows "Exited (1)". Code 0 means it finished normally. Anything else is an error. 137 means it was killed, usually out of memory.
4. **Can you get inside?** `docker compose exec app sh`. If it will not stay up long enough, override the command: `docker run -it --rm --entrypoint sh yourimage`.
5. **Is the config what you think?** `docker compose config` prints the file with every variable filled in. Empty values show up immediately.
6. **Can it reach the database?** From inside the app container, `ping db` or install a client and try a real connection.

## Messages you will actually *see*

| MESSAGE                       | MEANS                                             | FIX                                                                                   |
|-------------------------------|---------------------------------------------------|---------------------------------------------------------------------------------------|
| port is already allocated     | Something else holds that port                    | <code>sudo ss -tulpn   grep 3000</code> and stop it, or use another port              |
| no such file or directory     | A bind mount points at a path that does not exist | Check the left side of your <code>-v</code>                                           |
| Exited (137)                  | Killed, almost always out of memory               | <code>free -h</code> , add a memory limit, or add swap                                |
| connection refused to db      | App started before the database was ready         | Add a healthcheck, or let the app retry. <code>depends_on</code> alone does not wait. |
| permission denied on a volume | The container user cannot write to that folder    | <code>chown</code> the host folder, or run with <code>-u</code>                       |
| manifest unknown              | That image tag does not exist                     | Check the spelling and the version on the registry                                    |

## Cleaning *up*

Docker fills disks. Images pile up, old volumes linger, logs grow. This is the most common slow death of a home server.

```
docker system df          # what is using space
docker image prune        # delete untagged images. safe.
docker container prune    # delete stopped containers. safe.
docker system prune       # both, plus unused networks. safe.
docker system prune -a --volumes # DANGEROUS. see below.
```

### CAREFUL

`--volumes` deletes every volume not currently attached to a running container. If your app is stopped while you clean up, that includes your database.

Run `docker system prune` without `-a --volumes`. It recovers most of the space and cannot destroy data.

Check the disk monthly. Prune when it passes 80%.

```
df -h                                # disk space, human readable
du -sh /var/lib/docker              # how big is docker specifically
du -sh * | sort -h                  # biggest things in this folder
```

PART EIGHT

# *The tunnel.*

How to let the outside world reach one app on your home server without opening a single port. Then how to run several, how to protect them, and how to work out why one has stopped.

1. The three doors
2. What a tunnel is
3. Setting one up
4. Running several
5. Locally or remotely managed
6. Locking it down
7. Debugging

## The three *doors*.

There are three ways to reach a home server from the internet. They are not equal, and one of them is usually not even available to you.

|                         | PORT<br>FORWARDING | CLOUDFLARE<br>TUNNEL | TAILSCALE             |
|-------------------------|--------------------|----------------------|-----------------------|
| Cost                    | Free               | Free                 | Free for personal use |
| Opens a port            | <b>Yes</b>         | No                   | No                    |
| Works behind CGNAT      | <b>No</b>          | Yes                  | Yes                   |
| Others can use it       | Yes                | Yes                  | No, only your devices |
| Hides your home address | No                 | Yes                  | Yes                   |
| HTTPS certificate       | You set it up      | Automatic            | Not needed            |
| Risk                    | <b>High</b>        | Low                  | Very low              |

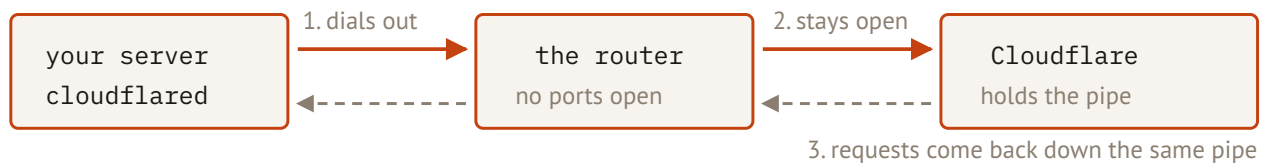
### Port *forwarding*

You tell the router to send outside traffic on a port to a machine inside. It is the old way. Your service is then exposed to the entire internet and scanned within minutes of going live.

It also simply fails if your provider uses CGNAT, which most do now. Many providers additionally block ports 80 and 443 on home connections.

### What a tunnel *actually is*

A small program on your server dials **out** to Cloudflare and holds that connection open. Visitors arrive at Cloudflare, and Cloudflare passes their request back down the pipe your server already opened.



Because your server opened the connection, the router allows the replies. Exactly like loading a web page.  
 There is no port to scan and no address to attack. Your home IP is never published.  
 This works behind CGNAT, on mobile broadband, anywhere with outbound internet.

A TUNNEL DIALS OUT, SO NOTHING NEEDS TO BE OPEN

#### USE TWO DOORS, NOT ONE

**Cloudflare Tunnel** for anything other people use. Your family should not have to install software to open a web page.

**Tailscale** for administration. SSH, databases, the things only you touch. It is also your safety net. If a tunnel config breaks, Tailscale still lets you in to fix it.

## Setting a tunnel *up*.

Five commands and one config file. You need a domain on Cloudflare, which is free apart from the domain itself.

```

curl -L https://pkgs.cloudflare.com/cloudflared-linux-amd64.deb -o cf.deb
sudo dpkg -i cf.deb
cloudflared tunnel login
cloudflared tunnel create mits
cloudflared tunnel list
  
```

`login` opens a browser and links this server to your account. `create` makes the tunnel and writes a credentials file with a long id in the name. Keep that id, you will need it.

## The config *file*

```
nano ~/.cloudflared/config.yml
```

```

tunnel: mits
credentials-file: /home/sam/.cloudflared/<TUNNEL-ID>.json

ingress:
  - hostname: app.mits.ro
    service: http://localhost:3001
  - hostname: notes.mits.ro
    service: http://localhost:3002
  - hostname: ssh-mits.mits.ro
    service: ssh://localhost:22
  - service: http_status:404
  
```

### CAREFUL

Rules are matched **top to bottom**, and the catch-all must be **last**. Anything written after `http_status:404` is dead and will never match.

Every new app is one more pair of lines, added above the catch-all.

## DNS and *start*

```

cloudflared tunnel route dns mits app.mits.ro
cloudflared tunnel route dns mits notes.mits.ro
sudo cloudflared service install
sudo systemctl enable --now cloudflared
sudo systemctl status cloudflared
  
```

If the DNS command fails, usually a missing certificate file, add the record by hand in the Cloudflare dashboard under DNS. Type CNAME, name `app`, target `<TUNNEL-ID>.cfargotunnel.com`, proxied.

### THE CERTIFICATE DEPTH TRAP

Cloudflare's free certificate covers `mits.ro` and `*.mits.ro`. It does **not** cover anything deeper.

`ssh-app.mits.ro` works. `ssh.app.mits.ro` is two levels down, gets no certificate, and fails at the handshake with an error you will waste an hour on.

Keep every hostname one level deep. Use dashes, not extra dots.

## Running *several* tunnels

One tunnel can serve unlimited hostnames, so most people never need a second. You want a second when the apps belong to genuinely different places, for example one tunnel per physical machine, or when you want to hand one to someone else without giving them the rest.

```

cloudflared tunnel create home
cloudflared tunnel create office
sudo mkdir -p /etc/cloudflared
  
```

Give each its own config file, then its own service.

```

sudo nano /etc/cloudflared/home.yml
sudo nano /etc/cloudflared/office.yml

sudo cloudflared --config /etc/cloudflared/home.yml service install
# then copy the unit for the second one
sudo cp /etc/systemd/system/cloudflared.service /etc/systemd/system/cloudflared-office.service
sudo nano /etc/systemd/system/cloudflared-office.service # point it at office.yml
sudo systemctl daemon-reload
sudo systemctl enable --now cloudflared-office
  
```

```
systemctl status cloudflared
systemctl status cloudflared-office
cloudflared tunnel list          # both should show as healthy
```

#### BEFORE YOU BUILD TWO

Ask what problem the second tunnel solves. If the answer is "more hostnames", you do not need it. Add ingress rules instead. Two tunnels means two configs, two services and two things to forget about.

## CHAPTER 29

# *Locking it down, and working out *what broke.**

## *Locally or remotely *managed**

This distinction decides whether you can fix things from a phone.

|                 | LOCALLY MANAGED                  | REMOTELY MANAGED             |
|-----------------|----------------------------------|------------------------------|
| Config lives in | A file on the server             | Cloudflare's dashboard       |
| Started with    | <code>--config config.yml</code> | <code>--token eyJh...</code> |
| Add a route     | Only at the machine              | From any browser             |
| Config in git   | Yes                              | No                           |

The dashboard tells you which you have. If it says "This tunnel is locally managed", the Routes tab is read only and nothing you click there will apply.

### WORTH CONSIDERING

For a home server you are often away from, remotely managed is usually the better trade. You lose config in version control. You gain the ability to publish a new app, or an SSH route, from a phone in another country.

## *Access *policies**

A published hostname is public to the entire world unless you put a policy in front of it. For a family app that is fine. For an admin panel or SSH it is not.

Cloudflare, then Zero Trust, then Access, then Applications, then Add, then Self-hosted.

| FIELD              | VALUE                     |
|--------------------|---------------------------|
| Application domain | admin.mits.ro             |
| Policy action      | Allow                     |
| Include            | Emails, then your address |

Now Cloudflare demands a one-time email code before a single packet reaches your server. While you are there, look for **Browser rendering**, **SSH**. If it is offered, you get a terminal in a browser tab with no client at all.

## Debugging a tunnel

```

cloudflared tunnel list           # is it registered and healthy
cloudflared tunnel info mits     # which edges is it connected to
cloudflared tunnel ingress validate # is my config file valid
cloudflared tunnel ingress rule https://app.mits.ro # which rule would match
systemctl status cloudflared
journalctl -u cloudflared -n 100 --no-pager
journalctl -u cloudflared -f      # watch live
  
```

That fourth command is the one people never find. It tells you exactly which ingress rule a given URL hits, which settles most "why is it 404" arguments in one line.

| SYMPTOM                      | USUALLY                                                                                             |
|------------------------------|-----------------------------------------------------------------------------------------------------|
| Error 1033                   | The tunnel is not connected. Check the service is running on the server.                            |
| 502 Bad Gateway              | The tunnel is fine. The app behind it is not. Check <code>curl localhost:3001</code> on the server. |
| 404 from the tunnel          | No ingress rule matched, so it fell through to the catch-all. Check the hostname spelling.          |
| Certificate error            | Hostname is more than one level deep. See the trap in chapter 28.                                   |
| Works locally, not publicly  | The app is bound to <code>127.0.0.1</code> and cloudflared runs elsewhere, or DNS is not proxied.   |
| Dashboard changes do nothing | The tunnel is locally managed. Edit the file on the server.                                         |

### THE TWO-SIDED CHECK

When a site is down, decide which half is broken before touching anything.

On the server, `curl -s localhost:3001/`. If that answers, the app is fine and the problem is the tunnel. If it does not, the tunnel is innocent and you have an app problem. This one habit halves your debugging time.

PART NINE

# *Git and GitHub.*

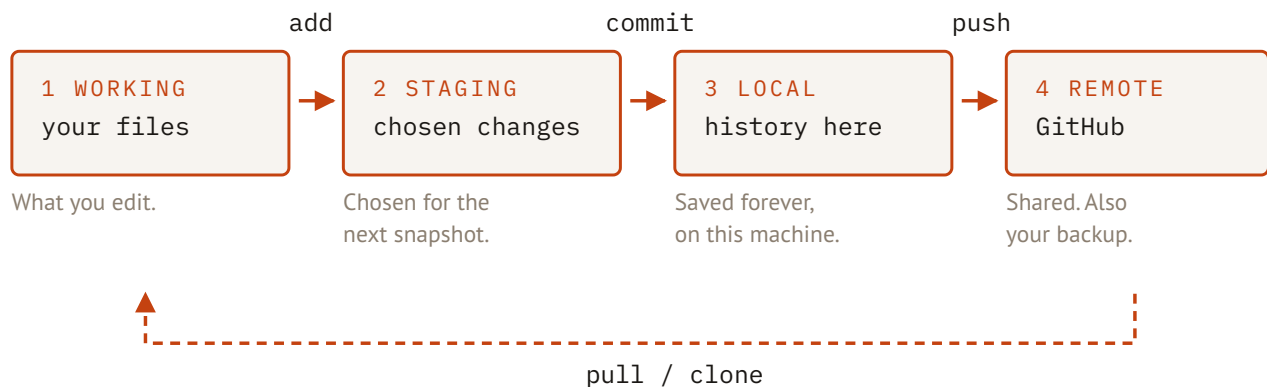
Version control is a time machine for your files. It is also how code gets from your laptop to your server without a USB stick. Ten commands cover 95% of it.

1. What it is for
2. The four places your code lives
3. The everyday commands
4. Connecting to GitHub
5. Keeping secrets out
6. Undoing things

## CHAPTER 30

# The four *places*.

Git confuses people because a file can be in four states at once. Once you can name the four, every command makes sense.



Nothing is shared until you push. Nothing is saved until you commit.

A file you edited but never added is in place 1 only. That is what "unstaged changes" means.

### THE FOUR PLACES A FILE CAN BE

#### JARGON

**Repository, or repo.** A folder that git is watching, plus its whole history.

**Commit.** A saved snapshot with a message attached. The unit of history.

**Branch.** A parallel line of work. `main` is the usual default.

**Remote.** A copy somewhere else, usually GitHub. Called `origin` by default.

**Clone.** Download a repo with all its history.

## The everyday *commands*

```
git status           # what has changed. run this constantly.
git add file.txt     # mark one file for the next commit
git add .            # everything changed here and below
git commit -m "fix the login" # save a snapshot with a message
git push             # send it to GitHub
git pull             # fetch and apply what others pushed
git log --oneline -10 # last 10 commits, one line each
git diff             # what changed, not yet staged
git diff --staged    # what is marked for the next commit
git checkout -b new-thing # make and switch to a branch
git switch main      # go back to main
```

### THE HABIT

`git status` before and after everything. It tells you which of the four places your work is in and usually suggests the exact next command. It is impossible to overuse.

## Connecting to *GitHub*

Use SSH, not a password. You already understand keys from Part VI.

```
ssh-keygen -t ed25519 -C "server"
cat ~/.ssh/id_ed25519.pub
```

Copy that whole line. On GitHub, Settings, then SSH and GPG keys, then New SSH key, paste, save. Then test.

```
ssh -T git@github.com      # expect: Hi name! You've successfully authenticated
```

```
git clone git@github.com:you/your-repo.git
cd your-repo
git remote -v              # where does this push to
```

### DEPLOY KEYS

For a server that only needs to **read** one repo, use a deploy key instead of your personal key. On GitHub it is per repository, under Settings, Deploy keys. Leave "Allow write access" unticked.

If that server is stolen, the key opens exactly one repo, read only.

## Keeping secrets *out*

This is the most common real world leak there is. Passwords committed to a public repository get found by automated scanners within minutes.

```
nano .gitignore
```

```
.env
*.key
*.pem
node_modules/
backups/
data/
```

### CAREFUL

`.gitignore` only stops files that are **not already tracked**. If you committed a secret once, it is in the history forever, even after you delete the file.

If that happens, the only correct response is to **change the password**. Assume it is public. Rewriting history is possible and does not help, because copies already exist.

## Undoing *things*

| SITUATION                                 | COMMAND                                                                                   |
|-------------------------------------------|-------------------------------------------------------------------------------------------|
| Discard changes to one file               | <code>git restore file.txt</code>                                                         |
| Unmark a file you had staged              | <code>git restore --staged file.txt</code>                                                |
| Fix the last commit message               | <code>git commit --amend</code>                                                           |
| Undo the last commit, keep the work       | <code>git reset --soft HEAD~1</code>                                                      |
| Undo the last commit, throw the work away | <code>git reset --hard HEAD~1</code>                                                      |
| Undo a commit that is already pushed      | <code>git revert &lt;hash&gt;</code>                                                      |
| Put changes aside for a minute            | <code>git stash</code> then <code>git stash pop</code>                                    |
| I have broken everything                  | <code>git reflog</code> , find the good state, <code>git reset --hard &lt;hash&gt;</code> |

### THE RULE ABOUT HISTORY

`reset` rewrites history. `revert` adds a new commit that undoes an old one.

On work that is only on your machine, either is fine. On anything already pushed, use `revert`. Rewriting shared history breaks it for everyone else.

`git reflog` is the safety net almost nobody knows about. Git remembers where you have been for 90 days, even after a hard reset. Very little is truly lost.

PART TEN

# *Running it for **years.***

Anyone can get a server working for a week. The difference between that and a machine you still trust in 2031 is backups you have tested, monitoring that reaches you, and a method for when something breaks.

1. Several apps on one box
2. Backups
3. Monitoring
4. A method for debugging
5. The security floor

## CHAPTER 31

# Several apps on one *box*.

One machine can host a dozen things. It needs a system, or you will collide with yourself in six months.

## Keep a port *register*

Write it down. Keep it in a file in a repo, not in your head.

| PORT | APP             | NOTES                          |
|------|-----------------|--------------------------------|
| 3000 | photos          | Already taken. Leave it alone. |
| 3001 | wiki            | The notes app                  |
| 5432 | System Postgres | Unrelated                      |
| 5433 | wiki database   | Docker, localhost only         |

## One folder per *app*

```

/home/sam/apps/wiki/
/home/sam/apps/photos/

```

Each with its own `docker-compose.yml`, its own `.env` and its own deploy script. Never share a compose file between projects. When you want to delete an app, you delete a folder.

## One deploy script per *app*

```
nano ~/apps/wiki/deploy.sh
```

```
#!/usr/bin/env bash
set -euo pipefail
cd "$(dirname "$0")"
git pull --ff-only
docker compose up -d --build
for i in $(seq 1 60); do
  curl -fsS localhost:3001/api/health >/dev/null 2>&1 && { echo "up"; exit 0; }
  sleep 2
done
echo "did not come up"; docker compose logs --tail 40; exit 1
```

```
chmod +x ~/apps/wiki/deploy.sh
```

#### ANATOMY

`set -euo pipefail`

- `-e` Stop the whole script the moment any command fails. Without this a failed `git pull` would happily continue and deploy the old code.
- `-u` Treat an unset variable as an error. This is what stops `rm -rf $EMPTY` becoming `rm -rf /`.
- `-o pipefail` A pipeline fails if any part fails, not just the last one.

Those three letters at the top of every script you write. They turn silent wrong behaviour into a loud stop.

## *A reverse proxy, when you need one*

If you serve several web apps on one machine without a tunnel, you want one program listening on 443 that forwards to each app by hostname. That is a reverse proxy. Caddy is the friendliest, because it gets certificates by itself.

```
# /etc/caddy/Caddyfile
app.mits.ro {
  reverse_proxy localhost:3001
}
notes.mits.ro {
  reverse_proxy localhost:3002
}
```

With Cloudflare Tunnel you usually do not need this at all. The tunnel's ingress rules do the same job, and the certificates are handled for you.

# *Backups, and the restore you have never **tested**.*

A backup you have never restored is a guess. This chapter is short because the rules are short, and it is the most important chapter in the book.

## *3, 2, 1*

**3** copies, on **2** kinds of media, with **1** somewhere else entirely. For a home server that means the live data, a copy on the same machine, and a copy out of the house.

The off-site one is not paranoia. A fire, a flood or a burglary takes the machine and the backup sitting next to it in the same moment.

## *What to back **up***

| ITEM                 | MATTERS         | WHY                                    |
|----------------------|-----------------|----------------------------------------|
| Database dumps       | <b>Critical</b> | The actual content                     |
| Uploaded files       | <b>Critical</b> | Photos, PDFs, attachments              |
| .env files           | <b>Critical</b> | Not in git. Impossible to reconstruct. |
| Docker volumes       | High            | Where the above usually live           |
| Compose files        | Medium          | Usually in git already                 |
| Application code     | Low             | It is on GitHub                        |
| The operating system | Low             | Reinstalling is faster than restoring  |

## A *real script*

```
#!/usr/bin/env bash
set -euo pipefail
STAMP=$(date +%F_%H%M%S)
DEST=~/backups
mkdir -p "$DEST"

# database, compressed
docker compose exec -T db pg_dump -U fam fam_cristea | gzip >
"$DEST/db-$STAMP.sql.gz"

# uploaded files
tar czf "$DEST/files-$STAMP.tar.gz" -C /var/lib/docker/volumes/app-data/_data .

# keep 14, delete the rest
ls -lt "$DEST"/db-*.sql.gz | tail -n +15 | xargs -r rm --
```

### ANATOMY

```
tar czf archive.tar.gz -C /path .
```

- c** Create an archive. The opposite is **x**, extract.
- z** Compress it with gzip.
- f** The filename comes next. This flag must be last of the group, because it takes a value.
- C /path** Change to that folder first, so the archive does not contain the whole path.
- .** Everything in that folder.

That is why the famous `tar -xzvf` looks like nonsense. It is four separate flags: extract, gzip, verbose, file.

## Schedule *it*

```
crontab -e
```

```
0 3 * * * /home/sam/apps/wiki/backup.sh >> /home/sam/backup.log 2>&1
```

#### ANATOMY

0 3 \* \* \* command

- 0 Minute. On the hour.
- 3 Hour. 3am.
- \* Day of month. Every one.
- \* Month. Every one.
- \* Day of week. Every one.

The `>>` appends output to a log, and `2>&1` sends errors to the same place. Without those, a failing backup fails silently forever.

### *Get one copy out of the **house***

```
rclone sync ~/backups remote:server-backups # to cloud storage
rsync -avz ~/backups/ other-machine:~/backups/ # to another address
```

#### TRY IT

##### RESTORE, ONCE, ON PURPOSE

Once a quarter, take the newest dump and load it into a scratch database. Look at the data. Confirm the rows are really there.

Ten minutes. It is the entire difference between having backups and believing you have backups. Most people who lose data had backups running the whole time.

## *Monitoring, and a method for when it **breaks**.*

You will not notice a server going down. Nothing beeps. You find out when someone asks why the app is broken.

### *Give every app a health **endpoint***

One URL that answers only when the app truly works, database included.

```
curl -s https://app.mits.ro/api/health
```

### *Watch it from **outside***

Use a free uptime monitor. Point it at the health URL, every five minutes, alerting to email and phone.

It has to run **outside your house**. A monitor on the same machine cannot tell you the machine is gone. Cloudflare can also email you when a tunnel goes unhealthy, which catches problems the app itself cannot report.

### *Watch the **disk***

The most common slow death. Logs and images fill the disk, then everything stops at once.

```
df -h                # how full is it
du -sh /var/lib/docker # is docker the culprit
journalctl --disk-usage # are logs the culprit
sudo journalctl --vacuum-time=14d
```

### *The **method***

When something breaks, resist the urge to change things. Work down this list in order. The first failure tells you which half of the system to look at.

1. Is the machine on and reachable at all?

`ping the server`

2. Is the container running, or restarting?

`docker compose ps`

3. What did it say when it failed?

`logs --tail 100`

4. Does it answer on the machine itself?

`curl localhost:3001`

5. Is the tunnel connected?

`cloudflared tunnel list`

6. Does the name still resolve?

`dig +short app.mits.ro`

Change one thing at a time. Write down what you changed. Most damage is done during the fix, not the fault.

WORK DOWN. DO NOT SKIP.

## The security *floor*

### CONTROL

### HOW

|                               |                                              |
|-------------------------------|----------------------------------------------|
| Key-only SSH                  | PasswordAuthentication no                    |
| No root login                 | PermitRootLogin no                           |
| Firewall on                   | <code>sudo ufw enable</code>                 |
| Automatic security updates    | <code>unattended-upgrades</code>             |
| Ban repeat offenders          | <code>fail2ban</code>                        |
| Access policy on private apps | Cloudflare Zero Trust                        |
| Databases on localhost only   | <code>127.0.0.1:5433:5432</code>             |
| Secrets out of git            | <code>.env</code> in <code>.gitignore</code> |
| Long unique passwords         | A password manager                           |

What you do not need: running your own VPN server, exposing a database for any reason at all, or putting a home server on a company network without asking.

PART ELEVEN

# *The rented machine.*

A VPS is the same skills, in someone else's building. What you are actually renting, how to size it without guessing, how to tell an overloaded box from badly written code, and how to grow it without losing anything.

1. What a VPS actually is
2. Rent or self-host
3. Choosing specs without guessing
4. The first hour on a new box
5. Is it overloaded? Reading the numbers
6. Bigger box, or better code
7. Resizing and migrating
8. Running several

## What a VPS actually *is*.

A virtual machine, running on somebody else's hardware, in a building with better power and internet than your house. That is all. Part three already explained the technology. This is the same thing, rented by the month.

### JARGON

**VPS.** Virtual Private Server. One slice of a large physical machine, with its own operating system, its own address, and root access. You are one of perhaps 20 tenants on that hardware.

**Provider.** The company that owns the physical machine and rents you a slice. Well-known ones include Hetzner, DigitalOcean, Vultr, OVH and Contabo. Prices and plans change constantly, so check current ones rather than trusting any figure in print.

**Instance.** Another word for one VPS. Cloud providers prefer it.

**Datacentre.** The building. Redundant power, redundant internet, cooling, and someone on site at 3am.

## What you are actually *renting*

### ONE PHYSICAL SERVER IN A DATACENTRE

64 cores · 256 GB · NVMe

tenant

tenant

YOUR VPS  
2 vCPU

tenant

tenant

Each slice has its own OS, its own root, its own IP. None can see the others.  
All of them share the same physical processors, disk and network card.

That last line is the whole catch. Your 2 vCPU are two shares of a queue, not two dedicated cores. When a neighbour goes berserk, your machine waits. Linux has a number for exactly that wait, called steal. A dedicated plan costs more and removes the sharing. For most work, shared is fine and you check steal occasionally. Chapter 38 shows you how to read it.

YOU ARE RENTING A SLICE, NOT A COMPUTER

## The words on the pricing *page*

| TERM                       | WHAT IT REALLY MEANS                                                                                               |
|----------------------------|--------------------------------------------------------------------------------------------------------------------|
| <b>vCPU</b>                | A share of a physical core. Two customers can hold shares of the same core. Not the same as two cores of your own. |
| <b>Dedicated vCPU</b>      | That share is yours alone. Costs 2 to 3 times more. Worth it only when you have measured steal.                    |
| <b>RAM</b>                 | This one is real and yours. It is also the thing you run out of first.                                             |
| <b>NVMe</b>                | Fast modern storage. Prefer it. Plain "SSD" is usually fine too.                                                   |
| <b>Traffic or transfer</b> | How much data may leave per month. 20 TB is generous, 1 TB is not. Incoming is usually free.                       |
| <b>Egress</b>              | Data leaving. The thing cloud providers charge painfully for.                                                      |
| <b>IPv4</b>                | A real public address. Increasingly charged as a small monthly extra.                                              |
| <b>Snapshot</b>            | A frozen copy of the whole disk, stored by the provider. Useful. <b>Not a backup.</b>                              |

## Not all VPS are the same *underneath*

| TYPE                | MEANS                                                  | VERDICT                                                                                                             |
|---------------------|--------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| <b>KVM</b>          | A real virtual machine with its own kernel             | <b>Want this.</b> You can load kernel modules, run Docker properly, use swap normally.                              |
| LXC or OpenVZ based | A container sharing the host kernel                    | Cheaper. Cannot change the kernel, swap often fake, some Docker features fail. Avoid unless price rules everything. |
| Shared hosting      | Not a server at all. A folder on someone's web server. | No root, no Docker, no SSH worth having. Different product.                                                         |
| Dedicated server    | The whole physical machine                             | No neighbours, no steal. Several times the price of a small VPS.                                                    |
| Cloud, AWS or GCP   | A VPS with 200 other services attached                 | Powerful, and the bill is a skill of its own. Overkill for one app.                                                 |

#### HOW TO CHECK WHAT YOU HAVE

On the box, `systemd-detect-virt`. It prints `kvm`, `lxc`, `openvz` or `none`. One word, and it tells you which rules apply.

## CHAPTER 35

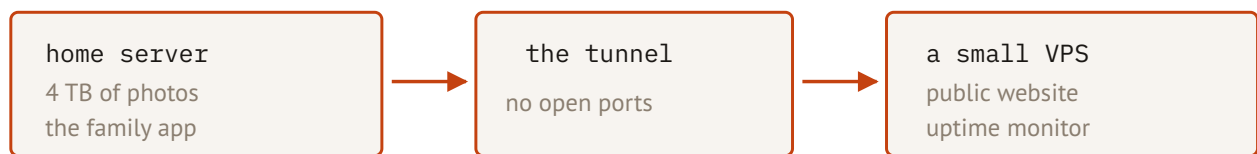
*Rent or self-host.*

This is not a loyalty question. The two are good at different things, and the honest answer for most people is both.

|                            | HOME SERVER                                                  | VPS                                                         |
|----------------------------|--------------------------------------------------------------|-------------------------------------------------------------|
| Cost                       | Hardware once, then the electricity formula in chapter 11    | A monthly bill, forever                                     |
| Your data sits             | In your house                                                | On someone else's disk                                      |
| Uptime                     | Your power and your internet                                 | Redundant power, redundant network                          |
| Public address             | Usually none, thanks to CGNAT                                | Always, and it is static                                    |
| Upload speed               | Whatever your provider gives you                             | 1 Gbit or better                                            |
| Storage                    | Cheap. A 4 TB disk costs about one month of a mid-sized VPS. | Expensive per gigabyte                                      |
| Someone fixes the hardware | You                                                          | Them, at 3am, without telling you                           |
| Sending email              | Effectively impossible from home                             | Possible, still painful                                     |
| Good for                   | Photos, files, media, family apps, anything large or private | Anything the public touches, anything that must not go down |

*The shape most people end up with*

Big and private stays home. Small and public gets rented. The two talk over a tunnel, so the home machine still never opens a port.



The big private data never leaves the house. The public front is cheap, replaceable and always up.  
 The VPS is also a fine place to run the monitor that watches the house, because it is somewhere else.  
 The cheapest plan a provider sells is usually enough for both of those jobs.

#### THE HYBRID MOST PEOPLE ARRIVE AT

##### ONE RULE WORTH KEEPING

Anything you would be upset to lose, and anything large, stays at home where storage is cheap and the disk is yours.

Anything the public touches goes on rented hardware, where a bad day is someone else's 3am.

## Choosing specs without *guessing*.

Do not pick a plan by feel. Measure what your thing actually uses, then add headroom. If it is not built yet, start at the bottom and grow.

### *Measure first*

Run the app on any machine you already have, including your laptop, put a realistic amount of work through it, and read four numbers.

```
free -h          # memory. read the "available" column.
nproc            # how many cores it can see
uptime           # load average, explained in chapter 38
df -h /          # disk used
docker stats --no-stream # memory and CPU per container
```

#### THE SIZING RULE

Take the memory it actually uses under load. **Double it.** Round up to the next plan.

Memory is the resource that kills a box outright. Processor shortage makes things slow, which is annoying. Memory shortage makes the kernel start killing your processes, which is an outage.

## Starting points if you have nothing to *measure*

| WHAT YOU ARE RUNNING            | START WITH           | NOTES                                            |
|---------------------------------|----------------------|--------------------------------------------------|
| Static site, or a reverse proxy | 1 vCPU, 1 GB, 20 GB  | The cheapest plan is genuinely enough            |
| Small app, no database          | 1 vCPU, 2 GB, 40 GB  | Node and Python are memory hungry at idle        |
| App plus PostgreSQL             | 2 vCPU, 4 GB, 80 GB  | The common answer. Give Postgres 25% of RAM.     |
| Several apps and a database     | 4 vCPU, 8 GB, 160 GB | Where a home server starts to win on price       |
| Anything with video             | Dedicated cores      | Transcoding is the one job that really needs CPU |
| A tunnel endpoint or a monitor  | 1 vCPU, 1 GB         | Almost idle. The smallest plan forever.          |

## Disk is the one you cannot easily take *back*

On almost every provider, growing a disk is instant and shrinking it is impossible. So do not buy 500 GB "to be safe". Buy what you need plus half, and grow it later in five minutes.

Work out what you need honestly. The operating system takes about 4 GB. Docker images take 2 to 10. Then your data, then room for backups if you keep them locally, then 20% free so the disk never fills.

## Where the box *sits*

Latency is distance. Pick a datacentre near the people using it, not near you.

```
ping -c 5 <the test address the provider publishes>
```

Every provider publishes a speed-test address for each location, usually on a page called "network" or "looking glass". Ping the ones you are considering and compare the average.

Under 30 milliseconds feels instant. Over 100 milliseconds feels sluggish on every click.

As a rough guide: a datacentre in your own country is usually under 20, one on the same continent is 20 to 50, and one across an ocean is 100 or more. Measure it rather than guess. Every provider publishes a test address you can ping.

## Providers, *honestly*

| PROVIDER     | GOOD AT                                       | WATCH FOR                                                                                                                 |
|--------------|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| Hetzner      | Among the cheapest in Europe. Solid hardware. | Strict signup checks. Fewer datacentre locations.                                                                         |
| DigitalOcean | Excellent documentation. Simple pricing.      | Costs noticeably more than the budget European hosts for the same specification.                                          |
| Vultr        | Many locations. Hourly billing.               | Performance varies by location.                                                                                           |
| OVH          | Cheap. Datacentres across Europe.             | Support is thin. Has lost customer data to a datacentre fire before, which is a reminder that a snapshot is not a backup. |
| Contabo      | Enormous specs for the money.                 | Oversold. Steal time and slow disks are common complaints.                                                                |
| AWS or GCP   | Everything, at scale                          | The bill. Egress charges. Not for one app.                                                                                |

### START SMALLER THAN YOU THINK

Vertical growth takes about five minutes and one reboot. Paying for a year of unused capacity takes twelve months.

Take the cheapest plan that fits your measured number, and watch it for a fortnight.

## CHAPTER 37

# *The first hour on a new **box**.*

Everything from chapter 17 applies, plus four things that only matter when the machine is genuinely on the public internet. It is being scanned already. Assume that.

## *Before you do anything **else***

```
ssh root@203.0.113.10      # the address the provider emailed you

adduser sam                # make a normal account
usermod -aG sudo sam       # let it use sudo
mkdir -p /home/sam/.ssh
cp ~/.ssh/authorized_keys /home/sam/.ssh/
chown -R sam:sam /home/sam/.ssh
chmod 700 /home/sam/.ssh
chmod 600 /home/sam/.ssh/authorized_keys
```

Now open a **second terminal** and confirm `ssh sam@203.0.113.10` works and `sudo whoami` prints root. Only then continue.

```
sudo nano /etc/ssh/sshd_config
# PermitRootLogin no
# PasswordAuthentication no
sudo systemctl restart ssh
```

### CAREFUL

A brand new VPS with a root password gets thousands of login attempts within the first day. This is not a story, it is the log file.

Do the account and key steps in the first ten minutes, not tomorrow.

# *The firewall matters more **here***

At home your router hides everything by accident. On a VPS every port you open is open to the planet.

```
sudo ufw default deny incoming
sudo ufw default allow outgoing
sudo ufw allow OpenSSH
sudo ufw enable
sudo ufw status verbose
```

Add web ports only when something is actually serving.

```
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
```

#### TWO FIREWALLS, ONE GOTCHA

Most providers also have their own firewall in the control panel, outside the machine. It is worth using, because it blocks traffic before it ever reaches you.

The gotcha: when a port is unreachable, check **both**. Half of all "I opened the port and it still does not work" cases are the panel firewall, not `ufw`.

Docker is a third trap. Publishing a port with `-p` writes its own rules and can bypass `ufw` entirely. Bind to `127.0.0.1` and let a proxy or tunnel handle the outside.

## Add *swap*, because there usually is none

Most VPS images ship with no swap at all. Without it, a brief memory spike kills a process instead of slowing down for a second.

```
sudo fallocate -l 2G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
echo '/swapfile none swap sw 0 0' | sudo tee -a /etc/fstab
free -h
```

Two gigabytes is a sensible amount on a small box. Swap is a safety net, not extra memory. If the machine is using it constantly you need a bigger plan, and chapter 38 shows you how to tell.

## Then the *usual*

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y unattended-upgrades fail2ban htop
sudo hostnamectl set-hostname web-01
timedatectl list-timezones | grep -i madrid # find yours
sudo timedatectl set-timezone Europe/Madrid # then set it
```

Replace Madrid with your own city. The list uses Region/City names, so search it for a large city near you.

- ☐ Non-root user with sudo, tested in a second terminal
- ☐ Key-only login, root login disabled
- ☐ ufw enabled, deny by default
- ☐ Provider firewall configured too
- ☐ Swap file created and in fstab
- ☐ unattended-upgrades and fail2ban installed
- ☐ Hostname and timezone set
- ☐ A snapshot taken, so you can return to this clean state

## CHAPTER 38

# *Is it overloaded? Reading the **numbers**.*

Four resources can run out: processor, memory, disk and network. Each has a number, and each number is misread in a predictable way. This chapter is the one to come back to.

## *Load average, and what it really **means***

```
uptime
```

```
16:22:07 up 41 days, 3:11, 1 user, load average: 1.94, 2.11, 1.87
```

Three numbers: the average over 1, 5 and 15 minutes. They are **not** percentages. They count processes that are running or waiting for their turn.

### THE RULE EVERYONE GETS WRONG

Compare load to your **core count**, not to 100.

`nproc` says how many you have. Load 2.0 on 2 cores means fully busy and coping. Load 2.0 on 8 cores means bored. Load 8.0 on 2 cores means a queue is forming and everything feels slow.

Rough guide: under 0.7 per core is comfortable, 1.0 per core is fully used, over 2.0 per core is a problem you can feel.

Read the three numbers as a direction. Rising to the left means it is getting worse right now. Falling to the left means the spike has passed.

## *The one line that explains a slow **VPS***

```
top      # then look at the %Cpu(s) line
```

```
%Cpu(s):  4.2 us,  1.1 sy,  0.0 ni, 88.9 id,  4.1 wa,  0.0 hi,  0.2 si,  1.5 st
```

| FIELD | MEANS                                                           | WHEN TO WORRY                                                                               |
|-------|-----------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| us    | User. Your own programs.                                        | High is normal and healthy. It is doing work.                                               |
| sy    | System. The kernel.                                             | Over 30% sustained means something odd.                                                     |
| id    | Idle.                                                           | You want this high.                                                                         |
| wa    | <b>I/O wait.</b> Waiting for the disk.                          | Over 10% means the disk is your bottleneck, not the processor.                              |
| st    | <b>Steal.</b> Time the host gave to a neighbour instead of you. | <b>Over 5% sustained means you have a noisy neighbour.</b> Nothing you can fix in software. |

#### STEAL IS THE VPS-ONLY NUMBER

You will never see steal on your home server, because nobody is sharing it.

Sustained steal above 5% is the one problem you cannot code around. The answers are: move to a dedicated-vCPU plan, rebuild the instance so it lands on a different host, or change provider. Take a screenshot of the number first, because support will ask.

## Memory, and the column people misread

```
free -h
```

|       | total | used  | free  | shared | buff/cache | available |
|-------|-------|-------|-------|--------|------------|-----------|
| Mem:  | 3.8Gi | 1.9Gi | 180Mi | 21Mi   | 1.7Gi      | 1.7Gi     |
| Swap: | 2.0Gi | 0.0Gi | 2.0Gi |        |            |           |

#### READ AVAILABLE, NOT FREE

`free` being small is not a problem. Linux deliberately fills spare memory with cached disk contents, because unused memory is wasted memory. It hands that cache back the instant a program needs it.

**available is the real number.** If available is comfortable, you have enough memory, no matter how alarming `free` looks.

| SIGNAL                       | MEANS                                                                                            |
|------------------------------|--------------------------------------------------------------------------------------------------|
| available under 10% of total | Genuinely tight. Plan an upgrade.                                                                |
| Swap used, but not changing  | Fine. Linux parked something idle.                                                               |
| Swap used and climbing       | Real pressure. Act.                                                                              |
| si and so busy in vmstat 5   | <b>Thrashing.</b> Constantly swapping in and out. The machine will feel dead. Upgrade or reduce. |
| Processes vanishing          | The out-of-memory killer. Confirm with <code>dmesg -T   grep -i oom</code> .                     |

## Disk, in two directions

```

df -h           # space
df -i           # inodes. the forgotten one.
du -sh /var/lib/docker
sudo iotop -o    # which process is hitting the disk right now

```

### JARGON

**Inodes.** A separate budget for the **number** of files, independent of their size. Millions of tiny files can exhaust inodes while `df -h` still shows plenty of free space. The error says "No space left on device" and makes no sense until you run `df -i`.

Over 80% full is the point to act. Over 90% and some databases refuse to write at all.

## Network, and the monthly cap

```

vnstat -m       # monthly totals. install it first.
sudo iftop      # live, by connection
ss -s           # how many connections are open

```

Most providers do not cut you off at the cap, they slow you to a crawl or bill you. Either is unpleasant to discover on the 28th. Install `vnstat` on day one, because it can only report from when it started watching.

## *The tools, in one **place***

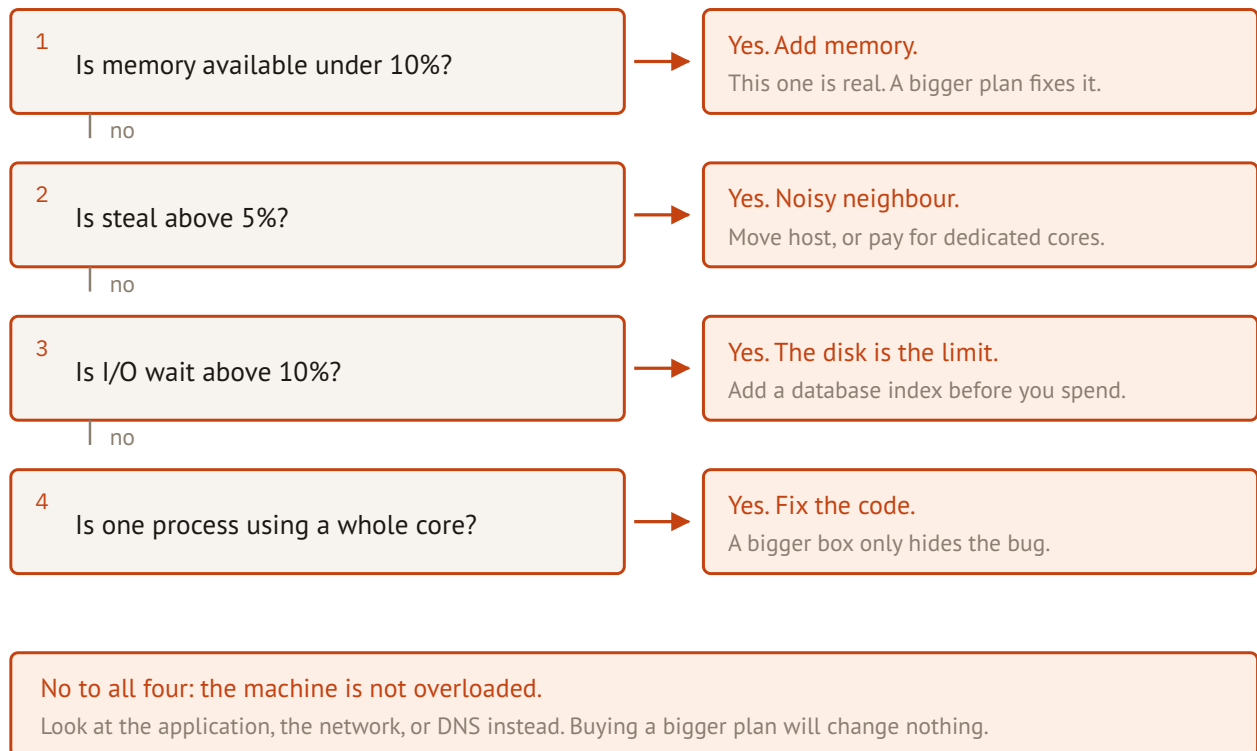
| COMMAND         | ANSWERS                                        |
|-----------------|------------------------------------------------|
| htop            | What is using the machine right now            |
| uptime          | Is there a queue forming                       |
| vmstat 5        | Is it swapping, is it waiting on disk          |
| iostat -xz 5    | Which disk is busy, and how busy               |
| iotop -o        | Which process is doing the reading and writing |
| pidstat 5       | Per-process usage over time                    |
| docker stats    | Which container is the problem                 |
| dmesg -T   tail | Did the kernel kill something                  |
| sar -u 1 5      | Historical usage, if sysstat is installed      |

```
sudo apt install -y htop iotop sysstat vnstat
```

## Bigger box, or better *code*.

Upgrading is the easy answer and often the wrong one. A box that is genuinely too small behaves differently from a box running something badly written.

Ask these four questions in order. Stop at the first yes.



Only the first two are ever solved by paying more money.

BIGGER BOX, OR BETTER CODE

## Symptoms, causes, *fixes*

| SYMPTOM                         | USUALLY                                   | DO THIS FIRST                                                            |
|---------------------------------|-------------------------------------------|--------------------------------------------------------------------------|
| Slow, load high, memory fine    | Real processor shortage or a runaway loop | htop, sort by CPU. One process at 100% is a bug, not a capacity problem. |
| Slow, load high, I/O wait high  | Disk bound, usually an unindexed query    | Find the slow query. An index is free and instant.                       |
| Processes killed at random      | Out of memory                             | dmesg -T   grep -i oom. Add swap now, add memory next.                   |
| Everything crawls, swap busy    | Thrashing                                 | Upgrade, or cap container memory so one app cannot eat it all.           |
| Fine, then terrible every night | A backup or cron job                      | crontab -l. Move it, or slow it with nice and ionice.                    |
| Steady creep over months        | A memory leak                             | Restart it and watch. If a restart fixes it for a week, it is a leak.    |
| Disk full, nothing obvious      | Docker or logs                            | docker system prune, journalctl --vacuum-time=14d                        |
| Fast for you, slow for others   | Not the server at all                     | Latency or bandwidth. Check where the box is.                            |

## Free wins before you *pay*

1. **Add swap.** Two gigabytes turns some crashes into a brief pause.
2. **Cap each container.** `--memory 512m` stops one app taking the machine down with it.
3. **Rotate logs.** `journalctl --vacuum-size=200M` and a `logrotate` rule.
4. **Prune Docker.** Old images are usually the largest thing on a small disk.
5. **Index the database.** The single biggest win available, and it costs nothing.
6. **Put a cache in front.** Cloudflare caching static files can cut the work by most of it.
7. **Fewer workers.** Many frameworks default to one worker per core and each holds memory. Two is often plenty.

### THE HONEST TEST

Before upgrading, ask: what will the new box do that the old one could not?

If the answer is "run the same inefficient thing with more room", you have bought time, not a fix. Sometimes buying time is exactly right. Just know which one you are doing.

# Resizing, migrating, and running *several*.

## Growing the *box*

Vertical scaling means a bigger box. Horizontal means more boxes. For one app, vertical is right until it is genuinely not.

In the control panel: power off, resize, power on. Two to five minutes. Then, because the disk grew but the filesystem did not notice, tell it.

```

df -h                                # still the old size?
lsblk                                # the disk is bigger, the partition is not
sudo growpart /dev/sda 1             # grow partition 1
sudo resize2fs /dev/sda1            # then the filesystem (ext4)
# on XFS instead: sudo xfs_growfs /
df -h                                # now it is bigger

```

### CAREFUL

Note the space in `growpart /dev/sda 1`. The device and the partition number are two arguments, not one. `growpart /dev/sda1` fails with a confusing message.

Also: on nearly every provider, disks grow and never shrink. Choosing 500 GB is a permanent decision. Choosing 80 GB is not.

## Snapshots are not *backups*

|                                     | SNAPSHOT                           | BACKUP                  |
|-------------------------------------|------------------------------------|-------------------------|
| Lives                               | With your provider                 | Somewhere else entirely |
| Survives your account closing       | No                                 | Yes                     |
| Survives the provider having a fire | No                                 | Yes                     |
| Restore one file                    | Awkward                            | Easy                    |
| Good for                            | "I am about to do something risky" | Everything else         |

Take a snapshot before every upgrade or risky change. Keep real backups as well, off that provider. Chapter 32 is still the rule.

## Moving to a new *box*

The order matters, and the first step happens a day early.

1. **Lower the DNS TTL to 300 seconds, a day before.** Otherwise the world caches the old address for hours and half your visitors see the dead box.
2. Build the new machine. Full setup from chapter 37.
3. Copy the data. `rsync -avz` for files, a fresh dump for the database.
4. Start it and test using the IP address directly, before touching DNS.
5. Stop writes on the old box. Take a final dump. Copy it over.
6. Switch the DNS record.
7. Watch both. Traffic drains from the old one over a few minutes.
8. **Keep the old box for a week.** It is the cheapest rollback you will ever buy.
9. Put the TTL back up. Then destroy the old box.

### IF YOU USE A TUNNEL, THIS GETS EASIER

With Cloudflare Tunnel there is no DNS switch at all. Run `cloudflared` on the new box with the same tunnel, confirm both are connected, then stop the old one. Traffic moves with no cached-address problem.

## Running *several*

Two boxes are twice the work unless you make them the same. Three rules keep it manageable.

**Name them by job, not by whim.** `web-01`, `db-01`, `build-01`. You will thank yourself at 2am.

**Put every one in `~/.ssh/config`.** Then a loop can reach all of them.

```
for h in web-01 web-02 db-01; do
  echo "=== $h ==="
  ssh $h 'uptime; df -h / | tail -1; free -h | head -2 | tail -1'
done
```

**Keep one inventory file** in a repo. Address, provider, plan, monthly cost, what runs on it, when it was last rebooted. Review it quarterly and delete what nobody uses. Unused VPSes are the most common quiet expense there is.

| NAME     | PROVIDER              | PLAN                | COST              | RUNS                           |
|----------|-----------------------|---------------------|-------------------|--------------------------------|
| web-01   | Provider A, Frankfurt | 2 vCPU, 4 GB, 40 GB | the monthly price | Public site, tunnel            |
| build-01 | Provider A, Frankfurt | 2 vCPU, 4 GB, 40 GB | the monthly price | Build runner. Off at weekends. |

#### THE QUARTERLY HABIT

Once every three months, open the billing page and read every line. Ask of each one: what breaks if I delete this today?

If you cannot answer, you have found something to delete. Take a snapshot first, wait a month, then delete for real.

PART TWELVE

# *Reference.*

The checklist to work down for any new machine, every command in this book on two pages, and a glossary for the day a word stops making sense.

1. The build checklist
2. Command reference
3. Glossary
4. Where to go next

# *The build checklist.*

Work down this for any new machine. Nothing here takes more than five minutes.

## *Hardware*

- ☐ SSD fitted, not a mechanical disk
- ☐ 8 GB of memory or more
- ☐ Ethernet cable, if one can reach
- ☐ Power-on-after-outage set in the BIOS

## *Operating system*

- ☐ Ubuntu Server LTS, minimized install
- ☐ No LVM, no disk encryption
- ☐ OpenSSH installed, keys imported from GitHub
- ☐ apt update and upgrade, then reboot
- ☐ Hostname set
- ☐ Timezone set
- ☐ unattended-upgrades installed

## *Stays on*

- ☐ Lid switch set to ignore
- ☐ Sleep targets masked
- ☐ Tested: lid closed, power pulled, still answers

## *Network*

- ☐ DHCP reservation set on the router
- ☐ Address written down somewhere findable

## *Access*

- ☐ Tailscale installed and tested from mobile data
- ☐ Key-only SSH, root login off
- ☐ ufw enabled, fail2ban installed
- ☐ SSH tested from outside the house before leaving it

## *Apps*

- ☐ Docker installed, account in the docker group
- ☐ restart: unless-stopped on every service
- ☐ Port register written down
- ☐ Databases bound to 127.0.0.1
- ☐ One folder and one deploy script per app
- ☐ Real version tags, not latest, on anything with data

## *Publishing*

- ☐ Tunnel running as a service, enabled at boot
- ☐ Access policy on every private hostname
- ☐ All hostnames one level deep

## *Survival*

- ☐ Nightly backup in cron, with output logged
- ☐ One copy off-site
- ☐ Restore tested at least once
- ☐ Uptime monitor running outside the house
- ☐ Tunnel health alert switched on

## CHAPTER 42

# Command *reference.*

## Getting around

|                                   |                                                  |
|-----------------------------------|--------------------------------------------------|
| <code>pwd</code>                  | Where am I                                       |
| <code>ls -lah</code>              | List everything, with details and readable sizes |
| <code>cd -</code>                 | Back to the previous folder                      |
| <code>find . -name "*.log"</code> | Find files by name from here down                |
| <code>du -sh *   sort -h</code>   | What is taking the space here                    |
| <code>df -h</code>                | How full are the disks                           |

## Files

|                                            |                                                       |
|--------------------------------------------|-------------------------------------------------------|
| <code>tail -f app.log</code>               | Watch a log live                                      |
| <code>grep -rn "text" .</code>             | Find text in every file below here, with line numbers |
| <code>chmod 600 secret.env</code>          | Only the owner may read it                            |
| <code>chown -R user:user dir</code>        | Change ownership of a whole tree                      |
| <code>tar czf out.tar.gz -C /path .</code> | Compress a folder                                     |
| <code>tar xzf in.tar.gz</code>             | Extract it again                                      |

## System

|                                           |                                       |
|-------------------------------------------|---------------------------------------|
| <code>systemctl status thing</code>       | Is it running, and why not            |
| <code>systemctl enable --now thing</code> | Start it and keep starting it at boot |
| <code>journalctl -u thing -n 50</code>    | Last 50 log lines for that service    |
| <code>journalctl -p err -b</code>         | Errors only, this boot                |
| <code>htop</code>                         | What is using the machine             |
| <code>free -h</code>                      | Memory in use                         |
| <code>uptime</code>                       | How long since the last reboot        |

## Network

|                                   |                                      |
|-----------------------------------|--------------------------------------|
| <code>ip a</code>                 | My addresses                         |
| <code>sudo ss -tulpn</code>       | What is listening, and which program |
| <code>curl ifconfig.me</code>     | My public address                    |
| <code>dig +short name</code>      | What does this name resolve to       |
| <code>curl -I https://site</code> | Headers only, is it answering        |

## Docker

|                                                |                                        |
|------------------------------------------------|----------------------------------------|
| <code>docker compose up -d --build</code>      | Build and start everything             |
| <code>docker compose logs -f --tail 100</code> | Follow the logs                        |
| <code>docker compose exec app sh</code>        | Shell inside a service                 |
| <code>docker compose config</code>             | Show the file with variables filled in |
| <code>docker compose down</code>               | Stop and remove. Volumes survive.      |
| <code>docker ps -a</code>                      | Every container, including stopped     |
| <code>docker system df</code>                  | What is using disk                     |
| <code>docker system prune</code>               | Reclaim space, safely                  |

## Cloudflare

|                                                  |                                 |
|--------------------------------------------------|---------------------------------|
| <code>cloudflared tunnel list</code>             | Are my tunnels healthy          |
| <code>cloudflared tunnel info name</code>        | Which edges is it on            |
| <code>cloudflared tunnel ingress validate</code> | Is my config valid              |
| <code>cloudflared tunnel ingress rule URL</code> | Which rule would match this URL |

## Git

|                                    |                                           |
|------------------------------------|-------------------------------------------|
| <code>git status</code>            | Run this constantly                       |
| <code>git log --oneline -10</code> | Recent history                            |
| <code>git diff</code>              | What changed                              |
| <code>git restore file</code>      | Throw away changes to one file            |
| <code>git reflog</code>            | Everywhere you have been. The safety net. |

# Glossary.

**apt** Ubuntu's package manager. Installs and updates software.

**Bind mount** A real folder on the host, mounted into a container.

**cgroups** The Linux feature that limits what a process may use.

**CGNAT** Your provider sharing one public address between many customers. Port forwarding cannot work behind it.

**CLI** Command line interface. Typing instead of clicking.

**Container** An isolated process that shares the host's kernel. Small and fast.

**Cron** The scheduler. Runs a command on a timetable.

**DHCP** The router service that hands out addresses. A reservation pins one.

**DNS** Turns names into numbers.

**Distro** A packaged version of Linux. Ubuntu, Debian, Fedora.

**Docker** A tool for packaging an app with everything it needs.

**Guest** A pretend machine running on a host.

**Host** The real physical machine.

**Hypervisor** The software that runs virtual machines.

**Image** A read-only recipe a container is started from.

**Kernel** The core of an operating system. Talks to the hardware.

**LTS** Long Term Support. Five years of security updates.

**LXC** Linux Containers. A container holding a whole small system.

**MAC address** A permanent serial number on a network card.

**man** The built-in manual. `man ls`.

**Namespaces** The Linux feature that limits what a process may see.

**NAT** Sharing one public address between many devices.

**Package manager** The program that installs software properly.

**Port** A numbered door on a machine. One program per port.

**Private address** 192.168.x.x and friends. Only exists inside your network.

**Prompt** The line the shell prints when it is ready for you.

**Proxmox** An operating system for running virtual machines and LXC from a browser.

**Repository** A folder git is watching, plus its history.

**Reverse proxy** One program on port 443 that forwards to several apps by hostname.

**root** The administrator account. Can do anything, including destroy everything.

**Shell** The program that reads your commands. Usually bash.

**SSH** An encrypted shell on another machine.

**stdout / stderr** Normal output and error output. Separate channels, 1 and 2.

**sudo** Run one command as the administrator.

**systemd** The manager that starts services and collects their logs.

**Tag** The version part of an image name, after the colon.

**Tailscale** A private network joining your own devices, anywhere.

**Tunnel** An outbound connection that lets traffic in without opening a port.

**ufw** The simple firewall. Blocks all, allows what you name.

**Volume** Storage that outlives a container. What you back up.

## CHAPTER 44

# Where to go *next*.

You now know enough to run a machine properly. Here is the order to do it in, and what to add once it is boring.

## The first *three* things

1. **Install Tailscale.** Three minutes. It ends the "I cannot reach my server" problem permanently, and it is the thing people wish they had done first.
2. **Schedule the backup.** One line in cron. Do it the day you put real data on the machine, not later.
3. **Add an uptime monitor.** Five minutes. You will stop finding out from your family.

Everything else on the checklist is improvement. Those three are insurance.

## Good next *projects*

| THING                     | WHY IT IS A GOOD SECOND STEP                                             |
|---------------------------|--------------------------------------------------------------------------|
| A file sync tool          | Replaces a cloud drive. Teaches you volumes and backups properly.        |
| A password manager        | Small, important, and forces you to take backups seriously.              |
| A read-later or notes app | Low risk. If it breaks, nothing is lost.                                 |
| A network-wide ad blocker | Teaches you DNS by making you responsible for it.                        |
| A media server            | Teaches you storage and transcoding, and everyone notices when it works. |

### OFFICIAL DOCUMENTATION

#### THE FOUR SITES WORTH BOOKMARKING

- [docs.docker.com](https://docs.docker.com) — the Docker reference, genuinely well written
- [ubuntu.com/server/docs](https://ubuntu.com/server/docs) — the server guide
- [Cloudflare Tunnel docs](https://cloudflare.com/tunnel/docs) — every ingress option
- [git-scm.com/book](https://git-scm.com/book) — free, and the only git explanation that ever stuck

## A closing *thought*

*The machine does not care that you are new. It only does what you told it. Every problem in this book has a cause, and the cause is findable.*

Change one thing at a time. Write down what you changed. Read the first error, not the last one. Test the restore.

That is the whole discipline. The rest is just commands, and commands are in chapter 42.

### ONE LAST HABIT

Keep a plain text file on your laptop called `server.md`. Every time you change something on the machine, add one line with the date and what you did.

In two years that file will be worth more than this book.