Veebirakenduse projekt (ITI0302)

Loeng 2

Your server, Docker, and a deployment by hand · Siim Rebane

Today

1. The problem: an application that runs on your laptop, and an assignment that wants it running somewhere else
2. Your server, and the Linux you need to operate it: SSH, users, files, packages, processes, services
3. Docker: images, containers, a Dockerfile for your Spring application, Compose
4. Deployment by hand: the jar to the server, the container up, nginx in front, a domain name, HTTPS
5. What the lab does with all of this

The problem

Individual assignment 1 asks for three things your laptop cannot give:

- the application **runs on your server**, not on your machine
- it runs **in Docker**
- a **pipeline deploys it on every push**

Today covers the first two, by hand: the server, and the application packaged so that it runs anywhere. The pipeline is the next lecture; everything it automates, you do once by hand today, so that you know what it does.

Your server

You have one virtual machine, provisioned from the SSH key you submitted:

What	Yours
Operating system	Ubuntu Server, no desktop, no browser
Your user	`ubuntu`, with `sudo`
Address	a public IP; no name yet
Reachable from the internet	ports 22 (SSH), 80 (HTTP), 443 (HTTPS). Nothing else
Installed	nothing beyond the operating system
Lifetime	the whole semester

Everything the application needs, you install. Everything you break, you fix, or the machine is reimaged and you start over.

Reaching it: SSH

SSH is a shell on another machine over an encrypted connection, with your key pair as the login:

```
ssh ubuntu@<ip>
```

- First connection: the server's fingerprint is shown once and stored; answer `yes`
- If your key is not in the default place: `ssh -i ~/.ssh/<file> ubuntu@<ip>`
- Windows: PowerShell or Windows Terminal have `ssh` built in; PuTTY is not needed
- `exit` ends the session. The server keeps running; only your shell stops

The private key never leaves your laptop. The server has only the public half, in `~/.ssh/authorized_keys`.

After you log in, there is no screen — there is a line

`ssh` let you in and the server printed a line waiting for you to type:

```
ubuntu@web:~$
```

That line is the **shell**: a program that reads one command, runs it, prints the result, and waits for the next. No windows, no icons, no mouse — this is the whole interface to the machine. Everything the server needs, you type here. The commands below are the ones you will actually use.

Read the prompt before you type — it says four things

```
ubuntu@web:~$
```

Piece	Meaning
`ubuntu`	**who** you are logged in as
`web`	**which machine** — matters once you have more than one open
`~`	**where** you are: `~` is your home directory, `/home/ubuntu`
`$`	the shell is **ready**. A `#` here would mean you are root, the administrator

You type after the `$` and press Enter. Every command in this deck goes there.

Every command has the same three-part shape

```
command  -options  arguments
```

One real example:

```
ls -l /etc/nginx
```

- `ls` — the **command**: the program to run (here, list files)
- `-l` — an **option**: changes how it behaves (here, the long, detailed listing). Options start with `-`
- `/etc/nginx` — an **argument**: what to act on (here, which directory to list)

Options and arguments are often optional: `ls` on its own works. The shape never changes.

Where am I, and what is here

The two commands you will run most, to orient yourself:

```
pwd                 # print working directory — the folder you are standing in
ls                  # list the files and folders here
ls -l               # the same, with detail: permissions, owner, size, date
```

`pwd` right after login prints `/home/ubuntu` — that is what `~` means. On a fresh server `ls` prints almost nothing, because you have installed nothing yet. That is normal, not an error.

Text after `#` is a comment for humans; the shell ignores it. Type it or leave it out.

The filesystem is one tree that starts at `/`

There are no drive letters. Everything hangs off a single root, written `/`:

```
/
├── etc                 # system configuration  → nginx's config lives here
│   └── nginx
├── home
│   └── ubuntu          # your files. "~" means exactly this
└── var
    └── www/html        # nginx's default web page
```

- A path starting with `/` is **absolute**, measured from the root: `/etc/nginx`.
- A path not starting with `/` is **relative**, measured from where you stand now.

You will edit files under `/etc`, so you type absolute paths often.

Moving around: `cd`

`cd` = change directory; its argument is where to go.

```
cd /etc/nginx          # jump to an absolute path
cd sites-enabled       # step into a folder that is right here (relative)
cd ..                  # go UP one level: ".." always means the parent
cd ~                   # go home; "cd" with nothing does the same
cd -                   # back to where you just were
```

If you are unsure where you landed, `pwd`. Getting lost is normal at the start, and `pwd` is how you find yourself again.

Reading a file without changing it

```
cat  compose.yaml                       # print the whole file to the screen
less /etc/nginx/nginx.conf              # scroll a long file: arrows move, q quits
docker compose logs -f                  # follow a container's log live (Ctrl+C stops following)
journalctl -u nginx -n 50               # the last 50 log lines of the nginx service
```

`cat` suits a short file. For a long one, `less` lets you scroll and press **q** to quit. When something is broken, the **log** is the first place to look — an empty response almost always has a reason printed there.

Making, copying, moving, deleting

```
mkdir app                  # make a new directory
cp compose.yaml compose.yaml.bak   # copy: keep the original, make a backup
mv old.jar app.jar         # move OR rename — the same command
rm app.jar                 # remove a file
rm -r app                  # remove a directory and everything in it
```

There is no recycle bin. `rm` deletes immediately and permanently; `rm -r` on the wrong directory is gone with no undo. Read the whole line before Enter — most of all when `sudo` is in front of it.

Editing a file in the terminal: `nano`

You will edit the nginx config on the server. There is no Notepad; you edit inside the terminal, with **nano**, the beginner-friendly editor:

```
sudo nano /etc/nginx/sites-enabled/default
```

Your cursor is in the file; type normally. The shortcuts are printed along the bottom, where `^` means the **Ctrl** key:

Keys	Does
Ctrl+O, then Enter	save (it calls this "write out")
Ctrl+X	exit back to the shell

Online you will see `vim`. It is powerful and famously hard to quit; use nano for this course.

Paste config into nano, never into the prompt

A trap that catches people every year: if you copy a block of several lines and paste it **at the shell prompt**, the shell runs each line as a command the instant it meets the newline — half-finished, in the wrong place.

So when a slide gives you a block for a file: open `nano` first, then paste **inside** the editor. There the newlines are just text, and nothing runs until you save. Pasting single commands at the prompt is fine; pasting file contents is not.

Four keys that save you every session

Tab — auto-complete

Type `cd /etc/ngi` and press **Tab**; the shell finishes `nginx/`. It completes commands and paths and prevents typos. Use it constantly.

↑ up arrow — history

Brings back your last command to rerun or fix, instead of retyping. Keep pressing for older ones.

Ctrl+C — stop

Cancels whatever is running and returns a fresh prompt. Your escape hatch when a command hangs or follows a log.

Ctrl+L — clear

Wipes the screen clutter. Nothing is deleted; you just get a clean prompt.

When a command fails, the message names the fix

The three you will actually meet, and what each means:

Message	Cause	Fix
`command not found`	typo, or the program is not installed	check spelling; `sudo apt install <it>`
`Permission denied`	the file or action belongs to root	put `sudo` in front
`No such file or directory`	wrong path, or you are not where you think	`pwd` to see where you are, `ls` to see what is really there

Read the message. It is short, literal, and almost always tells you the next move. Case and spelling are exact: `App.jar` is not `app.jar`, and one wrong letter in a path is `No such file or directory` — which is what Tab-completion is for.

Copying files from your laptop: `scp`

The application is built on your laptop; it has to reach the server. `scp` is `cp` over SSH, and you run it **on the laptop**, not on the server:

```
scp app.jar Dockerfile compose.yaml ubuntu@<ip>:~/app/
```

- same key and login as `ssh`; the `:` separates the machine from the path
- `~/app/` is the destination directory in your home on the server (`mkdir app` there first)
- direction is left to right: laptop → server. Swap the two sides to copy back

This is how the jar gets from where it is built to where it runs.

Users, root and sudo

- **root** is the administrator; it may do anything, including destroying the system with a typo
- **ubuntu** is your user. It owns your home directory and can run the application, but cannot install software or touch system files
- `sudo <command>` runs one command as root. That is how you install packages and edit system configuration

Never log in as root, never run the application as root. The habit that matters: `sudo` only for the commands that need it.

Files and permissions

Every file has an owner, a group, and three sets of rights: read, write, execute, for the owner, the group, and everyone else. `ls -l` shows them:

```
-rw-------  1 ubuntu ubuntu  411 Sep 10 10:12 id_ed25519
-rw-r--r--  1 ubuntu ubuntu   97 Sep 10 10:12 id_ed25519.pub
-rw-r--r--  1 root   root   1245 Sep 10 09:40 /etc/nginx/nginx.conf
```

- The first line is a private key: readable by its owner only. SSH refuses to use a key that others can read
- The last line belongs to root: reading is fine, editing needs `sudo`
- `chmod 600 <file>` sets owner-read-write-only; the one permission command you need

Packages: apt

Ubuntu installs software from its package repositories with `apt`:

```
sudo apt update              # refresh the list of available packages
sudo apt install -y nginx    # install one
```

- `apt update` first, always; the list on a fresh machine is stale
- A package brings its dependencies with it, and registers its service
- Everything the course needs is a package: `nginx`, `docker.io`, `docker-compose-v2`, `certbot`

Processes and ports

A running program is a **process**. A program that waits for network connections **listens on a port**, and one port belongs to one process at a time.

```
ss -tlnp                          # every port something listens on, and which process
```

- Ports below **1024** can only be opened by root: 80 and 443 are nginx's, running as root
- Your application listens on **8080**, as `ubuntu`, no root needed
- Two things on 8080 is the error *address already in use*: find the first one with `ss`, stop it

Services: systemd

Software that must run all the time, and start again after a reboot, runs as a **service** managed by systemd:

```
sudo systemctl status nginx      # running? since when? last log lines
sudo systemctl restart nginx     # after a configuration change
sudo systemctl enable nginx      # start at boot (apt does this for you)
journalctl -u nginx -n 50        # the service's log
```

nginx and Docker are services. Your application is not: it runs inside Docker, and Docker restarts it.

What is reachable from where

The lab network lets in **22, 80 and 443** and nothing else. So:

```
laptop → http://<ip>:8080/api/teams     refused: 8080 is not open from outside
server → curl localhost:8080/api/teams  works: same machine
laptop → http://<ip>/api/teams          works, once nginx on port 80 forwards to 8080
```

This is not a restriction to work around; it is the setup you want. One entry point, port 80, and a web server there that decides what goes where.

The problem Docker solves

Your application runs on your laptop because your laptop has the right Java, the right paths, the right environment variables, and three months of things you installed and forgot. The server has none of that.

Copying a jar and hoping is how "works on my machine" happens.

Docker's answer: package the application **with everything it needs** into one artefact, an **image**, and run that same image anywhere Docker runs. The server needs Docker and nothing else.

Docker is not a virtual machine

Both isolate your application, so they get confused. The difference is what they carry:

	Virtual machine	Container
Contains	a **whole** guest operating system + your app	your app + only the libraries it needs
Uses	its own kernel, emulated hardware	the **host's** Linux kernel, shared
Size	gigabytes	tens to hundreds of megabytes
Start time	a minute — it boots	under a second — it's just a process

A container is not a small computer; it is **one isolated process** on the host, wrapped so it sees its own files and network. That is why ten of them are cheap and start instantly, and why the image only needs your app, not an operating system.

Image and container

- An **image** is a read-only package: a minimal Linux filesystem, a Java runtime, your jar, and the command to start it. Built once, stored, copied around. `eclipse-temurin:25-jre-alpine` is an image; so will yours be
- A **container** is a running instance of an image: its own process, its own filesystem view, its own network. Start ten from one image and they do not see each other

The relationship is exactly **class and object**: the image is the class, a container is one instance. One image, many containers, each running independently.

An image is built in layers

Each line in the recipe adds a **layer** — a read-only slice stacked on the one before:

```
eclipse-temurin:25-jre-alpine     ← base layer (Java + tiny Linux), ~180 MB
  + your user                     ← a few KB
  + WORKDIR, app.jar copied in    ← the size of your jar
```

- Docker **caches** each layer. Rebuild after changing only your jar → every layer below the copy is reused, only the top is redone. Seconds, not minutes
- The base layer is downloaded once and shared by every image that starts `FROM` it
- **Consequence for the recipe:** put what changes rarely first, what changes often last. Your jar changes every build, so it is copied near the end

Where images live: registries

You built an image on one machine; the server needs it. A **registry** is the store images are pushed to and pulled from:

- **Docker Hub** — the public default. `docker run nginx` silently pulls `nginx` from here
- **GitLab Container Registry** — one per project, private; where your pipeline will push your image
- `docker pull <name>` fetches, `docker push <name>` uploads (after `docker login`)

An image name is really `registry/name:tag` — `docker.io/library/nginx:latest` in full. Leave parts out and Docker fills in Docker Hub and `latest`. Today you skip the registry entirely: you copy the jar and build on the server. The registry is how a pipeline does it instead.

Installing Docker on the server

```
sudo apt update
sudo apt install -y docker.io docker-compose-v2
sudo usermod -aG docker ubuntu
```

Log out and in again after the last line: it lets `ubuntu` use Docker without `sudo`, and the change applies to new sessions only. Then:

```
docker run hello-world
```

pulls a tiny image and runs it. If that prints a greeting, Docker works and your user may use it.

Running a container

```
docker run -d -p 8080:80 --name web nginx
```

- `nginx` is the image; Docker pulls it from Docker Hub if it is not local
- `-d` runs it in the background
- `-p 8080:80` maps the **server's** port 8080 to the **container's** port 80; the container's own ports are invisible until mapped
- `--name web` names the container so the next commands can refer to it

```
docker ps                  # running containers
docker logs web            # what it printed
docker stop web            # stop; docker start web starts it again
docker rm web              # remove the stopped container
```

Port mapping: `-p host:container`

A container's network is its own. A program inside it listening on a port is **invisible from the server** until you connect the two with `-p`:

```
-p 8080:80
   │     └── the port INSIDE the container (where the program listens)
   └──────── the port on the SERVER (what the outside can reach)
```

- Left is the server's port, right is the container's. The order trips up everyone once
- `-p 8080:8080` is common because Spring listens on 8080 inside, and you expose 8080 outside — same number, but still two different sides
- No `-p`, no reach: the container runs, `curl localhost:8080` on the server gets nothing, and it is not a bug

A container forgets when it is removed

A container's filesystem starts as a fresh copy of the image every time. Anything it writes lives only until `docker rm`:

- Rebuild and replace the container → whatever it wrote to disk is **gone**. This is a feature: every run starts from the known image, not from an accumulated mess
- So two things must never live inside the container: **data you want to keep**, and **configuration that differs per machine** (a real feature from the day there's a database — a mounted volume keeps the data; A1 has none)
- Your jar is fine inside: it is part of the image, rebuilt every time on purpose

The rule: a container is disposable. If losing it would lose something, that something belongs outside it.

Your application as an image

What your application needs at run time: a Java runtime, the jar, a command. That is the whole **Dockerfile**, the recipe an image is built from. In the project root:

```
FROM eclipse-temurin:25-jre-alpine
RUN addgroup -S app && adduser -S app -G app
WORKDIR /app
COPY app.jar app.jar
RUN chown app:app app.jar
USER app
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

The Dockerfile, line by line

Line	What it does
`FROM eclipse-temurin:25-jre-alpine`	start from an image that has Java 25's runtime on a tiny Linux. Runtime only: the jar is compiled before this
`RUN addgroup … adduser …`	create a user with no rights; the application will run as it, not as root
`WORKDIR /app`	the directory the following lines work in
`COPY app.jar app.jar`	copy the jar from the build context (the directory `docker build` runs in) into the image
`RUN chown app:app app.jar`	the jar belongs to that user
`USER app`	from here on, everything runs as `app`
`EXPOSE 8080`	documentation: this image listens on 8080
`ENTRYPOINT ["java", "-jar", "app.jar"]`	the command a container runs when it starts

The base image tag matters: `25-jre-alpine` is Java 25, runtime only, on Alpine Linux. The finished image is about 380 MB on disk, most of it Java; your jar is the last 20.

The jar first

The Dockerfile copies `app.jar`, so build it first, on your laptop, and give it that name:

Maven

```
./mvnw clean package -DskipTests
cp target/*.jar app.jar
```

Gradle

```
./gradlew clean bootJar
cp build/libs/*.jar app.jar
```

- `-DskipTests` today, because tests take time and this is about packaging. The pipeline runs them
- Gradle's `build/libs` may hold two jars; `bootJar` alone produces only the runnable one. If both exist, the runnable one is the larger, without `-plain` in its name
- `app.jar` is a build product: add it to `.gitignore`

Build the image, run the container

```
docker build -t iti0302-app .
```

`-t` names the image; `.` is the build context, the current directory, where `Dockerfile` and `app.jar` are. The first build pulls the base image; later builds reuse every layer above the `COPY`.

```
docker run -d -p 8080:8080 --restart unless-stopped --name app iti0302-app
curl http://localhost:8080/api/teams
```

- `-p 8080:8080` because Spring listens on 8080 inside the container
- `--restart unless-stopped`: Docker starts the container again after a crash or a reboot, unless you stopped it yourself. That is what keeps A1 running while you are not looking

Configuration: nothing baked in

An image should not contain anything that differs between your laptop and the server, and never a password. Two ways to hand configuration to a container:

- **Environment variables**: `docker run -e SERVER_PORT=8080 …`. Spring reads `SERVER_PORT` as `server.port`; every property has an environment name, upper case with underscores
- **A properties file** on the server, mounted into the container: `-v /home/ubuntu/app/application.properties:/app/application.properties` and `--spring.config.additional-location=file:/app/application.properties` on the command line

A1 has no database and no secrets, so it needs neither. The rule matters from the day the application has a password: **it lives on the server, not in Git, not in the image**.

Compose: the run command as a file

The `docker run` line above has six options and will grow. **Docker Compose** keeps it in a file, `compose.yaml`, next to the Dockerfile:

```yaml
services:
  app:
    build: .
    image: iti0302-app
    container_name: app
    ports:
      - "8080:8080"
    restart: unless-stopped
```

```
docker compose up -d --build     # build the image if needed, start, in the background
docker compose ps                # running?
docker compose logs -f           # follow the log; Ctrl+C stops following, not the app
docker compose down              # stop and remove the container
```

`docker compose` with a space; the old `docker-compose` command is a different, retired program.

Getting the application onto the server

Three ways to put an image on a server:

1. **Build on the server** from the source: needs Git, a JDK and enough memory for a build there. The small VM makes that slow
2. **Push the image to a registry, pull on the server**: the right way, and what a pipeline does. It needs a registry account and a login on both sides
3. **Copy the jar over and build the image there**: the base image is small, the build takes seconds, and it needs nothing but Docker on the server

Today is the third way, by hand. It is also the way to check something quickly on the server for the rest of the semester.

Deployment by hand: the steps

On the laptop, after building `app.jar`:

```
ssh ubuntu@<ip> mkdir -p app
scp app.jar Dockerfile compose.yaml ubuntu@<ip>:~/app/
```

On the server:

```
cd ~/app
docker compose up -d --build
docker compose logs -f           # until "Started … in 2.3 seconds", then Ctrl+C
curl http://localhost:8080/api/teams
```

An empty list, `[]`, is the right answer: the application runs, with nothing in memory yet. The same URL from your laptop does not answer, and it is not supposed to.

What it looks like when it works

Real output from the course test server, so you know what right looks like:

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  STATUS         PORTS                    NAMES
1ba391529bfb   iti0302-app    "java -jar app.jar"      Up 8 minutes   0.0.0.0:8080->8080/tcp   app

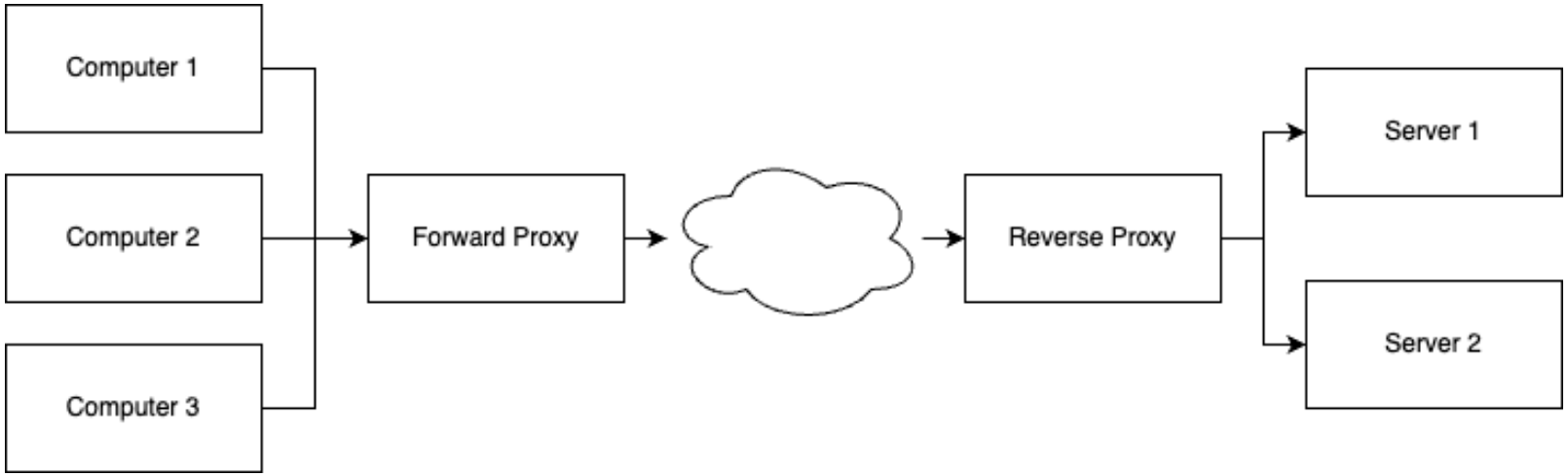
$ docker images
IMAGE                 ID             DISK USAGE   CONTENT SIZE
iti0302-app:latest    2e0a84cd8b24         381MB          112MB

$ docker compose logs | grep Started
app  | ... Started Lab1SampleApplication in 5.043 seconds
```

- `STATUS Up` : the container runs. `Exited (1)` means it started and died; the log says why
- `PORTS 0.0.0.0:8080->8080/tcp` : the mapping is in place; without it, nothing reaches the app
- `Started … in 5 seconds` : Spring is up. On a 1-CPU server it takes a few seconds longer than on your laptop

nginx: one entry point

A **reverse proxy** is a web server that accepts requests on the public port and forwards them to the application listening elsewhere on the same machine:



nginx is that server. It owns port 80, decides by path where a request goes, and hides the application's port. One nginx can do two jobs at once: serve a frontend's files on `/` and forward `/api` to a backend.

Why a web server in front, and not the app on port 80

Spring can listen on 80 itself. You still put nginx in front, and here is what it buys you:

Job	Without nginx	With nginx
The public port	the app is the only thing on 80	nginx on 80, the app private on 8080
HTTPS	every app implements TLS itself	nginx terminates it once, for everything
Frontend + backend	two public ports, or bundle them	`/` serves files, `/api` to the app, one address
Restart the app	the site is down while it boots	nginx still answers; the gap is one path
Ports under 1024	the app must run as root to open 80	nginx opens 80 as root; your app stays unprivileged on 8080

One front door, and the application behind it stays a plain process on a high port.

nginx: install and check

```
sudo apt install -y nginx
```

nginx starts as a service, on port 80. From the laptop, `http://<ip>/` in a browser shows nginx's default site, served from `/var/www/html`; the configuration that serves it is `/etc/nginx/sites-enabled/default`.



nginx: forward /api to the application

Edit the default site, `sudo nano /etc/nginx/sites-enabled/default`, and inside the `server { … }` block add:

```
location /api {
    proxy_pass http://localhost:8080;
}
```

```
sudo nginx -t                    # syntax check; never restart without it
sudo systemctl reload nginx
```

Then from the laptop:

```
curl http://<ip>/api/teams
```

Every request whose path starts with `/api` goes to port 8080 with the path unchanged. Everything else still shows the default page, until a frontend replaces it.

A name instead of a number

An IP address is what the machine has; a name is what people and certificates use. A1's URL may be an IP; the team project's cannot be, and HTTPS needs a name.

You create one **A record**: a name that points at your server's IP.

Where	What you get	Cost
GitHub Student Developer Pack, education.github.com/pack	a real domain for a year, with a DNS console	free with student status
freedns.afraid.org, DuckDNS	a subdomain under a shared domain; you manage the record	free
a domain of your own	any registrar; full control	about 10–20 € a year

After the record exists, `http://<name>/api/teams` answers like the IP did. Changes to DNS take minutes to spread; a name that does not resolve yet is not an error on your server.

HTTPS: a certificate for the name

HTTP is plain text; every password sent over it is readable on the way. HTTPS needs a **certificate** that proves the name is yours. Let's Encrypt issues them for free, and `certbot` does the whole exchange:

```
sudo apt install -y certbot python3-certbot-nginx
sudo certbot --nginx -d <name>
```

certbot proves to Let's Encrypt that the name points at this machine, gets a certificate valid for 90 days, rewrites the nginx configuration to use it and to redirect port 80 to 443, and installs a timer that renews it. Port 443 is open on your server, so this works.

A1's check accepts plain `http://`. Every real service is HTTPS, and once it is set up there is nothing to maintain.

Every change, by hand

A change to the application means, every time:

1. build the jar
2. copy it to the server
3. build the image there
4. stop the old container, start the new one
5. check the log and the URL

Five steps, two machines, each a chance to forget one. A pipeline runs exactly these steps, on every push, the same way each time, and A1 requires one. Today's version by hand is so that the pipeline is no mystery.

Common mistakes

Symptom	Cause
`permission denied` on `docker …`	the group change needs a new login; log out and in
`http://<ip>:8080` does not answer from the laptop	correct: 8080 is not open from outside. Go through nginx on 80
`COPY failed: app.jar not found`	the jar was not copied next to the Dockerfile, or has another name
`address already in use` on 8080	an older container or a stray `java` still runs; `docker ps`, `ss -tlnp`
nginx returns 502	the application is not running on 8080; `docker compose ps`, `docker compose logs`
`nginx -t` fails	a missing `;` or `}` in the file you edited
SSH `Permissions … are too open`	`chmod 600` on the private key

Lab 2

- SSH into your server; the Linux commands from this deck, on a real machine
- Install nginx and Docker; `docker run hello-world`
- Your A1 application from lab 1, as `app.jar`, copied to the server, built into an image, running behind nginx
- `http://<ip>/api/<your resource>` answers from your laptop
- If time: a name for the server, and HTTPS

Bring the laptop with your A1 project on it. The deck's commands are meant to be copied.



Questions