PROF. DR.-ING. MARK SCHUTERA

FULL STACK HANDWERK

UNFINISHED LECTURE NOTES

PUBLISHED BY UNFINISHED LECTURE NOTES

These notes are, by their very nature, unfinished, and they improve with every reader. If you spot an error, disagree with a framing, or want to add a source, a question, or a position, open a pull request at https://github.com/Quillstacks/LectureMaterial/tree/main/lecturenotes/notes_missingsemester. Every contribution is welcome.

2026-06-23 · polished grapefruit Maulwurf

Reverse Proxy & Frontend

2026-06-23 · rosy cantaloupe Antilope

Form follows function

The Why

The only way in today is a `curl` one-liner on port 8000 with a secret header attached. There is no human-friendly interface. The backend is the engine of the service; the frontend is what builds an experience around it and makes it accessible to people. Engineers tend to undervalue the frontend, and pay for it by not being understood.

This block closes that gap. We put Nginx in front of `uvicorn`, serve a small static frontend out of `/var/www/pixelwise/`, move the public surface off the application's port 8000 onto the standard web port 80, and route browser traffic through the same `POST /classify` and `GET /results` handlers we wired up in Block 6.

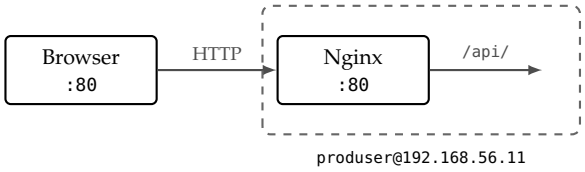

Figure 1:
The browser reaches our reverse proxy Nginx over HTTP on port 80. Nginx serves the static frontend and forwards `/api/` requests inward to the Block 6 service on `localhost:8000`. The dashed box is the `prod` VM, where only Nginx faces the network. The optional HTTPS exercise upgrades the browser connection to TLS on port 443.

Why not expose `uvicorn` to the internet directly? Put the process that classifies digits on the public network and every malformed request, port scan, and slow-connection flood hits your application code, where one crash takes the whole site down. A reverse proxy sits in front instead: It accepts each request, calls `uvicorn` on port 8000 itself, and returns the response, so the client never touches the application. Nginx absorbs the rough traffic at the edge and forwards only clean requests inward, and we can restart, scale, or swap the upstream without the public URL ever moving.

Hands On Experience

THE PRODUCT IS INVISIBLE until you give it a front door. From here on, renting a cheap server is the only step between your project and the world. The gap between a codebase on your laptop and a live service on the internet is huge. Build, share, learn, and have fun, and do not let anyone make your world small and narrow.

THIS SEVENTH BLOCK introduces the public surface for PixelWise. By the end you will have

- installed and configured Nginx as a reverse proxy on `prod`,
- served a static HTML and JavaScript drawing frontend from `/var/www/pixelwise/`,
- and reached the live application in a browser.

An optional section at the end of the block carries the edge further, terminating TLS so the site answers on `https://`.

Optional Exercise: Catch Up to `v0.5`

CATCHING UP VIA TAG. If you are not already at the end of Block 6, rewind both VMs to `v0.5`: PostgreSQL provisioned, the `pixelwise` role and database created, the SQLAlchemy `Prediction` model wired into the FastAPI handlers, and `POST /classify` writing rows that `GET /results` reads back. Both VMs follow the same sequence; run `dev` first, then `prod`.

TAG MAP. `v0.5` marks the end of Block 6, `v0.6` will mark the end of this block. Browse the full set at `https://github.com/schutera/pixelwise/tags`.

ON `dev`. Run the margin sequence. `setup-server.sh` installs PostgreSQL and provisions the `pixelwise` role and database; `python init_db.py` creates the `predictions` table.

SEQUENCE ON `dev`.

```
cd ~/pixelwise
git fetch origin --tags
git reset --hard v0.5
cp .env.example .env
nano .env
source .venv/bin/activate
pip install -r requirements.txt
bash setup-server.sh
python init_db.py
```

Set `DB_PASSWORD` in `.env` and keep the `SECRET_API_KEY` from Block 5. If `.venv/` is missing, run `python3 -m venv .venv` first.

ON `prod`. Same sequence after the `ssh` and `cd`. Verify from `dev`: `curl -s http://192.168.56.11:8000/results` should return `{"results":[]}`. Both machines are now at the end-of-Block 6 state, ready for Block 7.

SEQUENCE ON `prod`.

```
ssh produser@192.168.56.11
cd /opt/pixelwise
git fetch origin --tags
git reset --hard v0.5
cp .env.example .env
nano .env
source .venv/bin/activate
pip install -r requirements.txt
bash setup-server.sh
python init_db.py
```

Use the same `DB_PASSWORD` as on `dev`.

Reverse Proxies and Nginx

A REVERSE PROXY SITS AT THE EDGE. The client opens a connection to the proxy, the proxy opens a connection to the upstream, and the response flows back along the same path.

FORWARD VERSUS REVERSE. A forward proxy sits in front of the client and reaches out to many servers on the client's behalf, the way a corporate web filter does. A reverse proxy sits in front of the server and serves many clients on the server's behalf. Same word, opposite direction.

NGINX is the reverse proxy this course uses.

It is fast, well documented, and the most common piece of software you will find sitting in front of Python and Node services in production. The configuration is a tree of nested blocks: A top-level `http` context, `server` blocks inside it, and `location` blocks inside each `server`. Each `server` block declares a listen port and a hostname; each `location` declares how to handle requests whose paths match a pattern.

Nginx:
https://nginx.org/
Nginx documentation:
https://nginx.org/en/docs/

THREE ROLES IN ONE PROCESS. In the PixelWise stack Nginx wears three hats at once:

- As a static file server, it ships HTML, CSS, and JavaScript from `/var/www/pixelwise/` to the browser as fast as the kernel can read them.
- As a reverse proxy, it forwards every request whose path begins with `/api/` to `uvicorn` on `localhost:8000`.
- As a TLS terminator, it holds the certificate and the private key, so the public connection is encrypted while the internal connection to `uvicorn` stays plain HTTP.

`nginx -t` is the cheapest debugging tool in the chapter. It parses the configuration without applying it, prints `syntax is ok` when the file is well formed, and points at the line and column of the first problem when it is not. Run it before every `systemctl reload nginx` and the daemon will never refuse to start because of a missing semicolon or a typo in a directive name.

WHY A HIGH PORT? On Linux, ports below 1024 are privileged; binding requires either root or a capability granted by root. `uvicorn` runs as the unprivileged `produser`, so it listens on port 8000 where no special permission is needed. Nginx is started by `systemd` as root, binds 80 and 443, then drops its worker processes to an unprivileged user.

Exercise: Install Nginx and the First Proxy Pass

INSTALL NGINX ON `prod`. On a fresh `prod` VM, install the package and confirm the daemon is up and listening on port 80:

```
ssh produser@192.168.56.11
sudo apt update
sudo apt install -y nginx
systemctl status nginx --no-pager
```

The first three lines should report `Active: active (running)`. Visit `http://192.168.56.11` from your host browser; the default Nginx welcome page proves the daemon is reachable on port 80 of the VM over the host-only network.

THE DEFAULT SITE. A fresh install ships an Nginx `Welcome` page from `/etc/nginx/sites-enabled/default`. We will leave it in place until our own site is ready, then disable it in one step so there is no confusion about which configuration is serving requests.

AUTHOR THE SITE CONFIGURATION ON `dev`. The Nginx configuration is part of how the application runs, so it belongs next to the code, the systemd unit, and the setup script. Open the new file at `deploy/pixelwise.nginx` in the repository on `dev`:

```
cd ~/pixelwise
nano deploy/pixelwise.nginx
```

The file below is the first cut, plain HTTP on port 80, the frontend served from disk, every request under `/api/` forwarded to `uvicorn`:

```
# deploy/pixelwise.nginx
server {
    listen 80;
    server_name 192.168.56.11;

    # Serve the static frontend
    location / {
        root /var/www/pixelwise;
        index index.html;
        try_files $uri $uri/ =404;
    }

    # Forward API calls to uvicorn
    location /api/ {
        proxy_pass http://127.0.0.1:8000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
```

```
    }
}
```

COMMIT ON dev, PULL ON prod. The configuration belongs in the repository so a fresh VM can pick it up from a clean clone instead of from somebody's terminal history:

```
cd ~/pixelwise
git add deploy/pixelwise.nginx
git commit -m "Add Nginx reverse proxy config"
git push
```

Then on prod, pull the file into place:

```
ssh produser@192.168.56.11
cd /opt/pixelwise
git pull
```

INSTALL THE SITE ON prod. Nginx looks for active sites under /etc/nginx/sites-enabled/. The conventional pattern is to keep the real files under sites-available/ and symlink the active ones into sites-enabled/, so disabling a site is one rm away:

```
sudo cp deploy/pixelwise.nginx \
    /etc/nginx/sites-available/pixelwise
sudo ln -sf /etc/nginx/sites-available/pixelwise \
    /etc/nginx/sites-enabled/pixelwise
sudo rm -f /etc/nginx/sites-enabled/default
sudo nginx -t
sudo systemctl reload nginx
```

CREATE THE FRONTEND DIRECTORY. The location / block points at /var/www/pixelwise/ as the document root. That directory does not exist yet on prod, and the next exercise will fill it. For now, drop a placeholder so the proxy pass has something to serve when the path is /:

```
sudo mkdir -p /var/www/pixelwise
echo "PixelWise frontend stub" | \
    sudo tee /var/www/pixelwise/index.html
```

DID THE PROXY TAKE? From dev, hit the proxied API and the static path back to back:

```
curl -s http://192.168.56.11/
curl -s http://192.168.56.11/api/health
```

TRAILING SLASH MATTERS. proxy_pass http://127.0.0.1:8000/ strips the matched /api/ prefix before forwarding. Without the trailing slash, /api/classify would reach uvicorn as /api/classify and FastAPI would answer 404 because the route is registered as /classify. One character, opposite outcomes.

try_files IS THE STATIC FALLBACK. For a single-page frontend like ours, the directive tries the requested path, then a directory listing, then returns 404. A more elaborate frontend with client-side routing would fall back to index.html instead so deep links survive a reload. Our app has one page, so the 404 fallback is correct.

RELOAD, NOT RESTART. systemctl reload nginx tells the master process to re-read the configuration and gracefully replace its workers without dropping any in-flight connection. restart would kill the master and start it again, briefly closing the listening sockets. Reload is the default for configuration changes; restart is for the rare case where reload is not enough.

The first line should print `PixelWise frontend stub`. The second should print the same JSON `uvicorn` would have given you on port 8000, except now Nginx is the one that took the request. Two different programs answered two paths, and from the outside they look like one service on one port.

DID IT TAKE ON `prod`? `curl -s http://192.168.56.11/` should print the placeholder text. `curl -s http://192.168.56.11/api/health` should print `{"status":"ok","model_version":"v1"}`. The first answer came from Nginx reading a file on disk; the second traveled from Nginx to `uvicorn` on port 8000 and back. The browser sees one origin, `http://192.168.56.11`, even though two very different processes produced the two responses.

KEEP THE BACKEND ROUTE BARE. That `/api/health` answer came out of a handler registered as `/health`: the `/api/` prefix is a public label Nginx peels off before the request reaches FastAPI. Do not rename the handler to `/api/health` so it matches the browser. Nginx would still strip `/api/` and forward `/health`, which then matches no route, and every proxied call answers `404`. Test `uvicorn` on port 8000 at `/health`, never `/api/health`.

The Static Frontend

HTML FOR STRUCTURE, CSS FOR LAYOUT, JAVASCRIPT FOR BEHAVIOUR. Those are the three pieces a browser needs to render an interactive page, and the three files PixelWise ships.

- HyperText Markup Language, HTML, describes the document tree.
- Cascading Style Sheets, CSS, describe how it looks.
- JavaScript, JS, describes what happens when the user clicks, draws, or types.

The browser is the runtime, the network is the transport, and the server is a file system that happens to speak HTTP.

WHY NO FRAMEWORK? React, Vue, and Svelte solve problems that show up at scale: Component reuse across dozens of views, state management across complex interactions, routing across many pages. PixelWise has one page, one canvas, one results panel. A framework would add a Node.js toolchain, a bundler, a learning curve orthogonal to this course, and an extra layer hiding the bytes that actually travel between browser and server. Vanilla JS keeps the HTTP layer visible.

THE PAGE IS A SINGLE CANVAS, TWO BUTTONS, AND A RESULTS PANEL. The drawing area is a `<canvas>` element shown at 280 by 280 pixels, a tenfold magnification of the 28 by 28 grid the model actually reads. The JavaScript paints straight into that grid, so the user draws at the model's own resolution and watches the strokes pixelate as they land. A `Classify` button extracts the pixel array and posts it. A results panel renders the predicted digit, the confidence, and a scrolling history pulled from `GET /api/results`. That is the whole user surface.

THE `fetch` CALL is the bridge between the browser and the backend. `fetch` is the browser's modern HTTP client, a Promise-based function that takes a URL and an options object and returns a response the JavaScript can read as JSON, text, or bytes. A `POST` to `/api/classify` carries the pixel array in the body and the API key in a header, and waits for the JSON answer. The exercise below builds this into `app.js`, where the `classify` function issues exactly this request.

THE RELATIVE PATH IS THE HEADLINE. `fetch("/api/classify", ...)` names no hostname or port. It resolves against the URL the page was loaded from, so the same JavaScript runs unchanged in the classroom on `http://192.168.56.11` and in production on `https://pixelwise.yourdomain.com`. The frontend has no knowledge of where the backend lives, because the backend lives wherever the frontend was served from.

SAME-ORIGIN BY DESIGN. The page is served from `http://192.168.56.11/` and posts to `/api/classify` on the same host. The browser treats both as the same origin because the scheme, host, and port all match.

OPEN BY DESIGN. The `X-API-Key` header sits on `POST /api/classify` because that endpoint is gated. `GET /api/results` and `GET /api/health` carry no header because they are unprotected: The browser renders the results panel and probes the service without juggling a secret in client-side JavaScript. That is acceptable for the demo here; in a real product `/api/results` would move behind a session cookie or a short-lived token from a proper login flow.

Exercise: Build and Deploy the Frontend

CREATE THE FRONTEND DIRECTORY ON `dev`. The three files live in the repository under `frontend/` so they travel with the code, get reviewed in pull requests, and reach `prod` through the same Git path as the Python:

```
cd ~/pixelwise
mkdir -p frontend
nano frontend/index.html
```

The HTML is small. One canvas, two buttons, a result line, a recent-predictions list, one stylesheet link, and one script tag at the bottom so the document tree is built before the JavaScript looks for elements in it:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport"
        content="width=device-width, initial-scale=1">
    <title>PixelWise</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>PixelWise</h1>
    <p>Draw a digit, then click Classify.</p>
    <canvas id="pad" width="280" height="280"></canvas>
    <div class="controls">
        <button id="classify">Classify</button>
        <button id="clear">Clear</button>
    </div>
    <div id="result"></div>
    <h2>Recent predictions</h2>
    <ul id="history"></ul>
    <script src="app.js"></script>
</body>
</html>
```

AUTHOR THE STYLESHEET next to the HTML. The styling is minimal: A centred column, a visible canvas border, a button row, a result line, and a monospace history list. The point is readability, not visual design:

WHY A 280-PIXEL CANVAS? The model expects 28 by 28 grayscale pixels. Drawing at that size on a modern display would feel like trying to write on a postage stamp. So the visible canvas is that 28 by 28 grid magnified ten times: The user gets a comfortable pad, the model gets the input shape it was trained on, and nothing is hidden, because the chunky blocks on screen are exactly the pixels that travel to the API.

```
nano frontend/style.css
```

```
body {
    font-family: system-ui, sans-serif;
    max-width: 480px;
    margin: 2rem auto;
    padding: 0 1rem;
}
canvas {
    border: 1px solid #333;
    background: #fff;
    touch-action: none;
    display: block;
}
.controls { margin: 0.5rem 0 1rem 0; }
button {
    padding: 0.5rem 1rem;
    margin-right: 0.5rem;
    font-size: 1rem;
}
#result {
    font-size: 1.25rem;
    min-height: 1.5rem;
    margin-bottom: 1rem;
}
#history li { font-family: monospace; }
```

AUTHOR THE SCRIPT. `frontend/app.js` wires the canvas to the mouse, builds the pixel array, posts it, renders the prediction, and refreshes the history list. Open the file:

```
nano frontend/app.js
```

```
// frontend/app.js
const API_KEY = "REPLACE_ME";  // overwritten on deploy

const N = 28;       // the model's input grid
const SCALE = 10;   // on-screen pixels per grid cell (280 / 28)

const pad = document.getElementById("pad");
const view = pad.getContext("2d");
view.imageSmoothingEnabled = false;  // draw crisp blocks

// The real drawing happens on a hidden 28x28 grid. The
// visible canvas is that grid magnified ten times, so
```

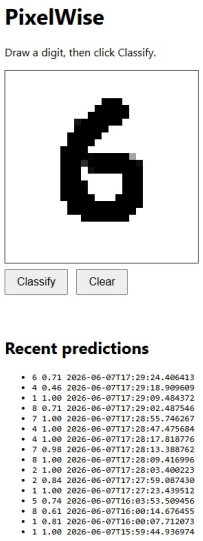

Figure 2: The finished frontend in the browser: The canvas, the `Classify` and `Clear` buttons, the result line, and the recent-predictions list. Nginx serves these files and forwards the `/api/` calls to FastAPI on `localhost:8000`.

```
// the user paints at the model's own resolution.
const grid = document.createElement("canvas");
grid.width = N; grid.height = N;
const gctx = grid.getContext("2d");
gctx.lineWidth = 2.5;
gctx.lineCap = "round"; gctx.lineJoin = "round";

let drawing = false;

function render() {
    // Magnify the grid onto the canvas, smoothing off.
    view.drawImage(grid, 0, 0, pad.width, pad.height);
}

function clearPad() {
    gctx.fillStyle = "#fff";
    gctx.fillRect(0, 0, N, N);
    render();
}
clearPad();

// Mouse positions map onto the 28x28 grid via SCALE.
pad.onmousedown = e => {
    drawing = true; gctx.beginPath();
    gctx.moveTo(e.offsetX / SCALE, e.offsetY / SCALE);
};
pad.onmousemove = e => {
    if (!drawing) return;
    gctx.lineTo(e.offsetX / SCALE, e.offsetY / SCALE);
    gctx.stroke(); render();
};
pad.onmouseup = pad.onmouseleave = () => { drawing = false; };

function getPixels() {
    // The grid is already 28x28: read it and invert.
    const data = gctx.getImageData(0, 0, N, N).data;
    const pixels = [];
    for (let y = 0; y < N; y++) {
        const row = [];
        for (let x = 0; x < N; x++)
            row.push(255 - data[(y * N + x) * 4]);
        pixels.push(row);
    }
    return pixels;
```

```
}

async function classify() {
    const r = await fetch("/api/classify", {
        method: "POST",
        headers: {
            "Content-Type": "application/json",
            "X-API-Key": API_KEY
        },
        body: JSON.stringify({ pixels: getPixels() })
    });
    const out = document.getElementById("result");
    if (!r.ok) { out.textContent = "Error " + r.status; return; }
    const d = await r.json();
    out.textContent = `Prediction: ${d.prediction} ` +
        `(${(d.confidence * 100).toFixed(1)}%)`;
    refresh();
}

async function refresh() {
    const r = await fetch("/api/results");
    if (!r.ok) return;
    const ul = document.getElementById("history");
    ul.innerHTML = "";
    for (const row of (await r.json()).results) {
        const li = document.createElement("li");
        li.textContent = `${row.prediction}  ` +
            `${row.confidence.toFixed(2)}  ${row.created_at}`;
        ul.appendChild(li);
    }
}

document.getElementById("classify").onclick = classify;
document.getElementById("clear").onclick = () => {
    clearPad();
    document.getElementById("result").textContent = "";
};
refresh();
```

Commit, push, pull. On dev, stage the three files and push:

```
cd ~/pixelwise
git add frontend/
git commit -m "Add static drawing frontend"
```

Why invert the channel? The MNIST training set stores digits as light strokes on a dark background, with zero meaning empty and high values meaning ink. The canvas draws black on white, the opposite convention. `255 - data[i]` flips the polarity so the model sees the input distribution it was trained on. Skip the inversion and the classifier will report 8 for a blank canvas because the white background looks like a dense glyph.

The API key in client code. `const API_KEY` sits in plaintext JavaScript that any visitor can view, so it is not a real secret. That is fine for the classroom:

```
git push
```

On `prod`, pull and copy the frontend into the document root. The repository is a Git clone under `/opt/pixelwise/`, and the `location /` block in our Nginx config serves `/var/www/pixelwise/`, so copy the files across:

```
ssh produser@192.168.56.11
cd /opt/pixelwise
git pull
sudo cp -r frontend/* /var/www/pixelwise/
```

EDIT THE API KEY IN PLACE. The script ships with `const API_KEY = "REPLACE_ME";` the value that actually works lives in `/opt/pixelwise/.env` as `SECRET_API_KEY`. On `prod`, replace the placeholder so the deployed script carries a real key. A small `sed` command keeps it auditable:

```
KEY=$(grep ^SECRET_API_KEY /opt/pixelwise/.env \
    | cut -d= -f2)
sudo sed -i \
    "s/REPLACE_ME/$KEY/" \
    /var/www/pixelwise/app.js
```

The same pattern moves into `setup-server.sh` in the next section, so a fresh provision picks up the right key without any manual editing.

DID THE FRONTEND TAKE? Open `http://192.168.56.11/` in your dev's browser. You should see the heading, the canvas, two buttons, and an empty recent-predictions list. Draw a clear digit, click `Classify`, and read the result line: The predicted digit and its confidence should appear, and the history list should grow by one entry that survives a page reload because it now lives in PostgreSQL.

DID IT TAKE? If the page loads but `Classify` produces `Error 401`, the API key in `app.js` does not match the one in `.env`. If the page loads but `Classify` produces `Error 404`, the `proxy_pass` trailing slash got lost in a save. If the page does not load at all, run `sudo nginx -t` on `prod`: A syntax error in the config block keeps the daemon from applying the new configuration.

Teach `setup-server.sh` to Provision the Edge

THE COMMANDS YOU JUST TYPED BELONG IN THE SCRIPT. A fresh `prod` VM should reach the same `v0.6` state with no manual steps. On `dev`, add `nginx` to the `apt install` line and append one prod-gated, idempotent stanza after the systemd unit from Block 5:

```
# Install Nginx site and deploy the frontend on prod
if [ -f deploy/pixelwise.nginx ] && \
   command -v nginx >/dev/null 2>&1 && \
   id produser >/dev/null 2>&1; then

    sudo mkdir -p /var/www/pixelwise
    sudo cp -r frontend/* /var/www/pixelwise/

    KEY=$(grep ^SECRET_API_KEY /opt/pixelwise/.env \
        | cut -d= -f2)
    sudo sed -i "s/REPLACE_ME/$KEY/" \
        /var/www/pixelwise/app.js

    sudo cp deploy/pixelwise.nginx \
        /etc/nginx/sites-available/pixelwise
    sudo ln -sf /etc/nginx/sites-available/pixelwise \
        /etc/nginx/sites-enabled/pixelwise
    sudo rm -f /etc/nginx/sites-enabled/default
    sudo nginx -t && sudo systemctl reload nginx
fi
```

PROD-ONLY AND IDEMPOTENT. The `id produser` guard keeps the edge off `dev`, where there is no public traffic to serve. Every step is safe to rerun: `ln -sf` refreshes the symlink, `cp` overwrites in place, and `sed` reads the live `SECRET_API_KEY` so the repository never holds the real key.

COMMIT, PUSH, RUN, TAG. Push the script, rerun it on `prod` to confirm it converges to the same state, then tag the block's end state:

```
# on dev
git add setup-server.sh
git commit -m "Provision Nginx and frontend in setup-server.sh"
git push

# on prod
ssh produser@192.168.56.11
cd /opt/pixelwise
git pull
bash setup-server.sh
```

TAG `v0.6`.

```
git tag -a v0.6 -m "Nginx proxy and
frontend"
git push origin v0.6
```

The end state is the static frontend behind an Nginx reverse proxy, provisioned by `setup-server.sh`. An annotated tag pushed by name travels reliably; `--follow-tags` skips lightweight tags.

Optional TLS and HTTPS

THE CORE STACK ANSWERS A BROWSER OVER PLAIN HTTP. A request to `http://192.168.56.11/api/classify` carries the pixel array, the path, and every header, including `X-API-Key`, as readable bytes on the wire. On a host-only network with two VMs that is a theoretical risk; on a public network it is a credential leak waiting for the first packet capture. The fix is Transport Layer Security, TLS, which encrypts everything between browser and server with a key the two parties agree on at handshake time.

HTTPS IS HTTP OVER TLS. Same protocol, same methods, same headers, same status codes: A TLS handshake opens before the first HTTP byte, the rest of the connection is encrypted, and the browser shows a lock icon. Nginx holds the certificate and the private key, so the TLS layer lives in one place at the edge of the system.

TLS IN THREE SENTENCES. The server presents a certificate that binds a hostname to a public key. The browser checks it against a trust store of known authorities. If it validates, the two sides agree on a session key with public-key cryptography, then encrypt the rest of the conversation symmetrically.
TLS overview: `https://datatracker.ietf.org/doc/html/rfc8446`

SELF-SIGNED CERTIFICATES ARE TRAINING WHEELS. A self-signed certificate is one the server signed itself. It encrypts the channel just as well as one from a public authority, but the browser warns because it does not recognise the signer. For the classroom that warning is the lesson: The chain of trust between your browser and a random IP address is exactly zero. The classroom can run on it; a production deployment with users cannot, and swaps in a certificate from a real authority, leaving the rest of the configuration unchanged.

THE PRODUCTION REPLACEMENT IS LET'S ENCRYPT. Let's Encrypt is a free certificate authority. Its client `certbot` proves you control the domain by answering a challenge on port 80, installs a certificate every browser trusts, and renews it automatically through a `certbot.timer` systemd unit. With a real domain pointing at a public IP, provisioning is three commands:

Let's Encrypt: `https://letsencrypt.org/`
certbot: `https://certbot.eff.org/`

```
sudo apt install -y certbot python3-certbot-nginx
sudo certbot --nginx \
    -d pixelwise.example.com \
    --redirect --agree-tos -m you@example.com
sudo systemctl status certbot.timer
```

`certbot --nginx` edits the site configuration in place, filling in the `ssl_certificate` paths and rewriting the port 80 redirect, so nothing in your `deploy/pixelwise.nginx` needs to know it ever ran self-signed first.

OUT OF SCOPE FOR THE VM LAB. Let's Encrypt needs a public, internet-routable hostname to issue a certificate, which the host-only network does not provide. The commands here map where the real path goes once you leave the classroom; the mechanics are identical, only the signer changes.

Self-Reflection and Recap

Self-Reflection questions to guide your thoughts during and after the exercises:

- Why does `uvicorn` stay on port 8000 once Nginx is in place, and what does Nginx gain by owning port 80 alone?
- The frontend calls the API at the relative path `/api/` and ships the key in plain JavaScript any visitor can read. What does the relative path buy, and why is the visible key acceptable here?
- A browser shows `502 Bad Gateway`. Do you read `/var/log/nginx/access.log` or `journalctl -u pixelwise` first, and why?
- `setup-server.sh` automates site configuration. What does the next deploy undo if you edit `/etc/nginx/` on `prod` by hand?
- If you ran the optional TLS section: A self-signed certificate makes the browser warn instead of connecting quietly. What does that warning still teach about the chain of trust, even on a network you fully control?

Recap of key concepts:

- A reverse proxy is the public face of an internal service: Nginx faces the network, serves the static frontend, and forwards `/api/` requests to `uvicorn` on localhost.
- A static frontend of plain HTML, CSS, and JavaScript runs in any browser without a build step, and a same-origin relative `fetch` keeps the API contract simple.
- The site configuration lives in the repository and `setup-server.sh` installs it in one idempotent run, so a fresh VM reaches the same state with no manual edits.
- TLS terminates at the edge, where a self-signed certificate teaches the handshake and the role of Nginx as terminator before a real deployment swaps in a certificate from a public authority.

Bridge. The stack works, but the workflow does not. Every change reached `prod` by hand, a `git push` on `dev`, an `ssh` into `prod`, a `git pull`, a `bash setup-server.sh`, and a hope that nothing breaks. Yes, this is Handwerk - but even so it is okay to hand-of what's possible. The application stays exactly as it is; next we automate that path, so a push checks and ships itself.

Cross-origin is the advanced case. The relative `fetch` keeps the page and the API on one origin, so the browser's same-origin policy never intervenes. Serve them from different domains and the browser blocks the read until the server opts in with Cross-Origin Resource Sharing, CORS, headers. That is the first deployment puzzle beyond this block.
CORS reference: `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`

Teaser. We ship by hand on every push, the server runs commands typed by a tired student at 11pm, one wrong `rm` away from a broken site. What could possibly go wrong?