The Backend: APIs & Services

2026-05-10 · alert currant Gans

API First. Anyone who doesn't do this, will be fired.

The Why

Block 4 left us with `app/classifier.py`, a function that turns a 28-by-28 array of pixels into a prediction dictionary. The function works. `predict.py` proves it on real MNIST samples. What it cannot do is be reached from anywhere except the same Python process that imported it.

This Block. is the first that exposes the application to the outside world. Every client that follows, whether a browser frontend, an automated test runner, or a monitoring probe, talks to the service through the contract you design here.

HTTP is the protocol that every browser, every CLI tool, and every web service in this course speaks. Figure 1 shows the path a single request takes through the stack you will build in this Block.

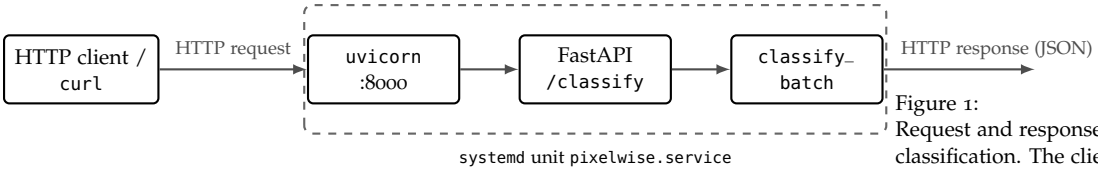

Figure 1: Request and response flow for a single classification. The client sends an HTTP request to `uvicorn`; FastAPI dispatches it to the `/classify` handler, which calls `classify_batch` from Block 4. The dictionary returned by `classify_batch` is serialised to JSON and travels back along the same path as the HTTP response. The whole stack runs under a `systemd` unit that restarts it if it crashes.

A service is what `predict.py` becomes when you wrap it in HTTP and run it under a supervisor. `uvicorn` is the HTTP server that hosts the FastAPI application; `systemd` is the supervisor that keeps `uvicorn` running, restarts it when it crashes, and captures its logs, turning it into someting that survives.

Hands On Experience

THE CONTRACT BECOMES THE PRODUCT. Once the service [1] is running, the request and response shapes are what every other component in the system depends on. Change them silently and the frontend breaks; document them clearly and new clients can integrate without ever reading your Python. Ones an API is public, it will be digested, users will build on it, and it will be hard to change [2].

THIS FIFTH BLOCK introduces the backend service layer for PixelWise. By the end you will have

- understood HTTP and REST well enough to design and probe an endpoint,
- written request and response contracts as Pydantic models,
- wrapped `classify_batch` in a FastAPI application,
- added API key authentication and basic rate limiting,
- run the service under `systemd`, observed crashes, and read its logs with `journalctl`.

[1] T. Winters, T. Manshreck, and H. Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, Sebastopol, CA, 2020. URL https://scholar.google.com/scholar?q=Software+Engineering+at+Google+Winters+Manshreck+Wright

[2] F. P. B. Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley, Reading, MA, 2 edition, 1995. URL https://scholar.google.com/scholar?q=The+Mythical+Man-Month+Brooks

THE SCRIPT-TO-SERVICE JUMP. A script runs once, prints, exits. A service runs continuously, accepts requests, returns responses, survives crashes. The work in this block is what turns the first into the second; the classifier itself does not change.

Optional Exercise: Catch Up to `v0.3`

CATCHING UP VIA TAG. This catch-up extends the one in Block 4. If you have not done that one, do it first: It lands you at `v0.2`, the end-of-Block 3 state. The exercise here picks up from there and rewinds to `v0.3`, the end-of-Block 4 state, with the classifier module in place, the model artefact pulled into `models/`, and `.env` pinned to the model release.

TAG MAP. `v0.3` marks the end of Block 4, `v0.4` will mark the end of this block. Browse the full set at `https://github.com/schutera/pixelwise/tags`.

DEV ONLY. Block 4 deliberately left `prod` behind: The classifier had no HTTP entry point, so pulling the code there would have added an unused `app/classifier.py` to the production checkout. This catch-up does the same. Bring `dev` to `v0.3` and leave `prod` where Block 4 left it; the `systemd` exercise later in this block is what finally promotes the service to `prod`.

THREE LAYERS TO RESTORE ON `dev`. Block 4 produced Git-tracked changes, the new `app/classifier.py` and `predict.py`, an updated `setup-server.sh`, and a refreshed `.env.example`. It also produced unversioned state, the model artefact `digit_classifier_v1.pkl` in `models/`, plus three new `.env` entries, `MODEL_REPO`, `MODEL_VERSION`, and `MODEL_PATH`. `git reset --hard v0.3` restores the first layer, `cp .env.example .env` seeds the third, and the extended `setup-server.sh` pulls the artefact once `.env` is in place.

RUN THE CATCH-UP ON `dev`. Rewind the repository to `v0.3`, refresh `.env` from the updated example, run `setup-server.sh` to pull system packages and the model artefact, reinstall pinned dependencies if your virtual environment was wiped, and run `predict.py` to confirm the integration works end to end. The exact commands are in the margin.

SEQUENCE ON `dev`.

```
cd ~/pixelwise
git fetch origin --tags
git reset --hard v0.3
cp .env.example .env
bash setup-server.sh
source .venv/bin/activate
pip install -r requirements.txt
python predict.py
```

If `.venv/` is missing, run `python3 -m venv .venv` before the `source` step.

YOU ARE NOW at the state where Block 4 ended: `dev` is at `v0.3` and `prod` still trails behind, ready to start Block 5.

ALREADY HAVE A FORK? If your own fork carries the `v0.3` tag, swap `origin` on `dev` with `git remote set-url origin git@github.com:<you>/pixelwise.git` so push and pull continue to target your account. The HTTPS clone on `prod` stays read-only.

HTTP and REST

EVERY WEB INTERACTION IS A REQUEST AND A RESPONSE. The client sends a request that names a method, a path, and an optional body; the server sends back a status code, headers, and an optional body. The protocol carries no state of its own: each request is interpretable on its own, without knowledge of any previous request from the same client.

HTTP is the request and response protocol that sits underneath every browser call, every `curl` invocation, and every API in this course. A reference overview of the methods, status codes, and headers lives here.

METHODS AND QUERIES YOU WILL USE.

- `GET` retrieves,
- `POST` submits,
- `PUT` replaces,
- `DELETE` removes.

PixelWise uses `POST` for `/classify` when the client submits a drawing, and `GET` for `/results` and `/health` when the client reads state.

STATUS CODES YOU WILL SEE MOST. Three-digit numbers grouped by leading digit. `2xx` means success, `4xx` means the client made a mistake, `5xx` means the server made a mistake.

- `200` OK,
- `201` Created,
- `400` Bad Request,
- `401` Unauthorized,
- `404` Not Found,
- `422` Unprocessable Entity,
- `429` Too Many Requests,
- `500` Internal Server Error.

REST PRINCIPLES layer convention on top of the protocol. Every distinct thing the API exposes is a resource with a stable URL: `/classify`, `/results`, `/health`. Methods do what their names suggest. Responses are JSON. Two clients written by two teams, six months apart, agree on what the API does without ever speaking, because the conventions remove ambiguity.

STATELESS is the load-bearing property of HTTP. The server keeps no per-client memory between requests; everything the server needs to act is in the current request, including any authentication token. This is what lets you scale.

REST is the architectural style that maps the HTTP verbs onto application resources. Each resource has a stable URL; each verb has a clear meaning. We use REST because it is the convention every frontend, every tool, and every framework in this stack expects, not because it is the only valid choice. A reference introduction to the style and its conventions lives here.
HTTP status codes reference: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`
REST overview: `https://restfulapi.net/`

THE INTERACTIVE DEFINITION OF AN API is the documentation that ships with the running service. FastAPI generates an OpenAPI specification from your code, exposes the raw JSON at `/openapi.json`, and renders it through two built-in interfaces: Swagger UI at `/docs` and ReDoc at `/redoc`. Exposing them changes the social contract: A teammate does not need to read your source to integrate, it's your responsibility to fulfill the stated contract. Block 3's `DEBUG=true` flag controls whether the documentation endpoints are exposed, so the same service is self-documenting in development and silent in production.

ONE SPEC, MANY RENDERERS. `/openapi.json` is the contract; Swagger UI and ReDoc are two ways to look at it. The same spec format is what Flask, Django, Express, and Spring all emit, and the same renderers and SDK generators work against any of them.
OpenAPI: `https://www.openapis.org/`
Swagger Editor: `https://editor.swagger.io/`

Exercise: HTTP, REST, and `curl`

PROBE A PUBLIC API before writing your own. `httpbin.org` is a free request-and-response echo service: It returns a JSON document describing exactly what it just received, so you can read status codes, headers, and the body of your own request side by side with the response. On `dev`, with the virtual environment active:

```
curl -i https://httpbin.org/get
curl -i -X POST https://httpbin.org/post \
    -H "Content-Type: application/json" \
    -d '{"hello": "world"}'
```

READ EACH SECTION OF THE RESPONSE. `HTTP/2 200` is the status line. The lines that follow are headers; the blank line marks the boundary; the JSON below it is the body. Every HTTP response in this course has the same shape.

THE BODY IS JSON. JSON is the universal serialisation format for web APIs: A text-shaped tree of objects, arrays, strings, numbers, booleans, and `null`. Your `GET` response body is short enough to read in full:

```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.81.0"
  },
  "origin": "203.0.113.42",
  "url": "https://httpbin.org/get"
}
```

JSON SYNTAX Objects `{...}`, arrays `[...]`, strings in double quotes, numbers and booleans and `null` unquoted, commas between items but never trailing. Spec: `https://www.json.org/`

Curly braces enclose an object, each key is a quoted string, and each value is a primitive, an array, or another object. PixelWise's `/classify` response will use the same shape, with keys `prediction`, `confidence`, and `scores`, parsed identically by any client in any language.

PICK THE VERB for each PixelWise endpoint and say what success looks like:

- `/classify`: Submit a drawing for prediction.
- `/results`: Read the last 10 predictions.
- `/health`: Check whether the service is alive.

ANSWERS. `POST /classify`: Image data does not fit in a query string; `POST` carries it in the body and is not cached. `200 OK` with the prediction.

`GET /results`: Safely cacheable, no side effect. `200 OK`.

`GET /health`: Reads liveness, no side effect. `200 OK`, or `503 Service Unavailable` when unhealthy.

FastAPI in Practice

FastAPI is a modern Python web framework built on top of Pydantic and the ASGI standard. It generates interactive API documentation automatically, validates inputs and outputs against the models you declared, and runs on `uvicorn`, an asynchronous HTTP server fast enough that your bottleneck will be the model, not the framework.

Pydantic. Python data-validation library. A class inherits from `BaseModel` and declares typed fields; Pydantic validates incoming data against the schema and raises a structured error on mismatch. FastAPI runs the same check at the HTTP boundary and returns `422 Unprocessable Entity` on failure.
https://docs.pydantic.dev/

FastAPI: https://fastapi.tiangolo.com/
Pydantic: https://docs.pydantic.dev/
uvicorn: https://www.uvicorn.org/
ASGI: https://asgi.readthedocs.io/

A path operation is the unit of routing in FastAPI: A Python function whose decorator binds it to one HTTP method and one URL path. When the application starts, FastAPI walks the decorated functions and builds a routing table that `uvicorn` consults on every incoming request. The type hints on the parameters are not just documentation: They are read at registration time and turned into a validator that runs before your function body executes. A request that fails the validator never reaches your code. It is rejected at the boundary with a `422` response whose body lists exactly which field was wrong and why, so the caller can fix the request without guessing at your implementation.

The decorator is the contract. `@app.post("/classify", response_model=ClassifyResponse)` encodes three things at once: The method clients must use, the path they must hit, and the schema the response must satisfy. The first two shape the routing table. The third shapes what leaves the server. FastAPI runs the value your function returns through `ClassifyResponse` on the way out, drops fields that are not declared on the model so internal state cannot leak by accident, and fills the OpenAPI schema served at `/docs` from the same class. A handler that returns the wrong shape fails inside the framework on the first request, long before a downstream client notices that the documented contract has drifted from the implementation.

`curl` is a command-line HTTP client. It writes the bytes of a request directly to a socket and prints the bytes of the response,

with no browser, no JavaScript, and no implicit defaults beyond what you pass on the command line. That is what makes it useful for debugging: The request you send is exactly the request you typed, so the only place a bug can hide is in your invocation or in the server. The interactive `/docs` page, by contrast, does a lot on your behalf. It JSON-encodes the body for you, sets the `Content-Type: application/json` header on every `POST`, and, once you click `Authorize`, attaches the API key header to subsequent calls. A request that succeeds from `/docs` and fails from `curl` therefore points at something the form was adding silently, almost always a header you forgot on the command line. A request that fails from both rules out the client and points back at the handler. The exercise below gives the exact `curl` invocations against the running service so you can compare the two side by side.

curl: `https://curl.se/`
`-X` sets the HTTP method.
`-H` sets a header.
`-d` sets the request body.
`-i` prints the response headers alongside the body, useful when a status code surprises you.

THREE ENDPOINTS cover the surface this block exposes: `POST /classify` performs inference, `GET /results` is a stub that a later persistence layer will fill in, and `GET /health` reports liveness so a monitor can tell whether the service is up.

Exercise: Building `app/main.py`

INSTALL THE RUNTIME DEPENDENCIES into the dev virtual environment and pin them.

```
cd ~/pixelwise
source .venv/bin/activate
pip install fastapi uvicorn[standard] \
    slowapi pydantic
pip freeze > requirements.txt
```

THE `[standard]` EXTRA on `uvicorn` pulls in optional but recommended packages that ship alongside the core server: `httptools` for a faster HTTP parser, `uvloop` for a faster event loop on Linux, `websockets` for the WebSocket protocol, and `watchfiles` so `--reload` can detect code changes during development. Without the extra you still get a working server, just slower and without auto-reload.

WIRE THE PATH OPERATIONS FROM THE THEORY ABOVE INTO A WORKING SERVICE. From the repository root on `dev`, create the file:

```
cd ~/pixelwise
nano app/main.py
```

CTRL + R IN THE TERMINAL starts a reverse search through your shell history. Hit it, type a few characters of a command you ran earlier, such as `pip freeze`, and the most recent match appears. Press Ctrl + R again to step further back, Enter to run the match, or the right arrow to edit it before running. This is much faster than retyping or looking up commands again or scrolling with the up arrow.

```
from fastapi import FastAPI
from pydantic import BaseModel
import numpy as np
from app.classifier import classify_batch

class ClassifyRequest(BaseModel):
    pixels: list[list[int]]

class ClassifyResponse(BaseModel):
    prediction: str
    confidence: float
    scores: dict[str, float]

app = FastAPI()

@app.get("/health")
def health():
    return {"status": "ok", "model_version": "v1"}

@app.get("/results")
def results():
    return {"results": [], "note": "persistence not yet implemented"}

@app.post("/classify", response_model=ClassifyResponse)
def classify(req: ClassifyRequest):
    arr = np.array(req.pixels, dtype=np.uint8)[np.newaxis]
    return classify_batch(arr)[0]
```

RUN IT LOCALLY from the repository root, so that `uvicorn` can resolve `app.main:app` as a Python import against the `app/` package on disk:

```
cd ~/pixelwise
uvicorn app.main:app --reload --port 8000
```

The `--reload` flag restarts on every code change, which is what you want in development and never want in production.

TEST IT FROM THE BROWSER. Open `http://localhost:8000/docs` to see the auto-generated OpenAPI page.

TRY `GET /health`. Click "Try it out" and "Execute". A `200` with `{"status": "ok", "model_version": "v1"}` is the first end-to-end loop. The first real `POST /classify` round-trip lands in Block 7, when the frontend sends an actual digit drawn on a canvas.

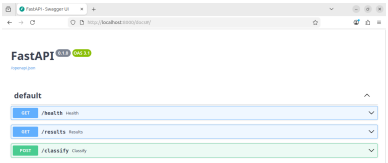

Figure 2: The Swagger UI rendered at `/docs`. FastAPI generates this page from the type hints and Pydantic models in `app/main.py`. Each endpoint expands to show its request schema, an example payload, and an interactive button that issues a live request against the running service.

OPTIONAL: TRY `POST /classify` ANYWAY. A Python list comprehension is not valid JSON, which is why the default `[[0]]` returns `422`. Print a real 28-by-28 body in a Python REPL with

```
import json
print(json.dumps(
  {"pixels":
  [[0]*28 for _ in range(28)]}))
```

and paste the printed line into Swagger's request body field.

OPTIONAL: DESIGN A FOURTH ENDPOINT. The three endpoints above cover inference, retrieval, and liveness. What is missing? Before reading on, sketch one more endpoint on paper. Pick the HTTP method and path, write the Pydantic request and response models, and decide what status code each error case returns. Plausible candidates include `GET /version` so a deployment can report the running build and model hash, `POST /feedback` so a frontend can flag wrong predictions for later retraining, or `POST /classify/batch` so a client can submit many images in a single round trip. Additional endpoints can help where the current contract has gaps, and to feel how naming, shape, and status codes are decisions you make before any code runs.

Authentication and Rate Limiting

Who is calling, and how often?

API KEY AUTHENTICATION is the simplest form that fits this course. The client sends a secret in a header; the server compares it to the value in `.env` and rejects requests that do not match with `401 Unauthorized`. FastAPI exposes the header through a small dependency function that runs before every protected handler, which the exercise wires up.

WHY A HEADER? Headers travel separately from the body and the URL, so the secret never ends up in the path component of an access log. Putting the key in a query parameter would write it into every log line the request touched on the way to the server.

RATE LIMITING prevents both abuse and accident. `slowapi` attaches a per-route decorator that counts requests per client IP and rejects them with HTTP `429` when the limit is exceeded. A budget like "30 requests per minute per IP" is enough to stop a runaway client without inconveniencing a normal one.

WHERE DOES RATE LIMITING BELONG? Application-level limits are easy to write and travel with the code. Reverse-proxy limits, a topic we will revisit later in the course, run in front of the application and protect it even when it is overwhelmed. Both layers are useful; the latter is harder to bypass. slowapi: `https://github.com/laurentS/slowapi`

PYDANTIC, THE API KEY MIDDLEWARE, AND THE RATE LIMITER together form a three-stage filter at the boundary of the application. A request that arrives must parse as JSON, match the schema, present a valid key, and stay within the rate budget before any inference code runs. Each stage rejects with a different status code, so a client failure tells you exactly which contract was violated.

(Optional) Exercise: API Key Auth and Rate Limiting

ADD THE DEFENSIVE LAYER FROM THE THEORY ABOVE TO THE SERVICE. Reopen `app/main.py` from `~/pixelwise` on `dev`:

```
cd ~/pixelwise
nano app/main.py
```

Add the API key dependency alongside the existing imports:

```
from fastapi import Header, HTTPException, Depends
import os

def verify_api_key(x_api_key: str = Header(...)):
    if x_api_key != os.getenv("SECRET_API_KEY"):
        raise HTTPException(
            status_code=401, detail="Invalid API key")
```

Then attach `verify_api_key` to the `/classify` route by extending its existing decorator with `dependencies=[...]`; the handler body itself does not change:

```
@app.post("/classify",
          response_model=ClassifyResponse,
          dependencies=[Depends(verify_api_key)])
def classify(req: ClassifyRequest):
    arr = np.array(req.pixels,
                   dtype=np.uint8)[np.newaxis]
    return classify_batch(arr)[0]
```

THE HEADER TRAVELS IN EVERY REQUEST. `X-API-Key` is a custom header name; FastAPI's `Header(...)` maps it to the function parameter automatically. A missing header returns 422 from Pydantic, a wrong value returns 401 from your handler.

ADD RATE LIMITING with `slowapi`. Add the imports and the limiter setup near the top of the file:

```
from fastapi import Request
from slowapi import Limiter
from slowapi.util import get_remote_address
from slowapi.middleware import SlowAPIMiddleware

limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_middleware(SlowAPIMiddleware)
```

Then stack the limiter decorator above the route decorator and add a `request: Request` parameter so `slowapi` can identify the caller:

```
@app.post("/classify",
          response_model=ClassifyResponse,
          dependencies=[Depends(verify_api_key)])
@limiter.limit("30/minute")
def classify(request: Request,
             req: ClassifyRequest):
    arr = np.array(req.pixels,
                   dtype=np.uint8)[np.newaxis]
    return classify_batch(arr)[0]
```

TEST `POST /classify` FROM SWAGGER. Reload `/docs`, expand the endpoint, click "Try it out", and fill the key from your `.env` into the `x_api_key` header field that now sits alongside the request body. Press "Execute". Without a key the classify request returns `401`; with the right key the same request goes through.

OPTIONAL: ADD A GLOBAL AUTHORIZE BUTTON. Swap `Header(...)` for `APIKeyHeader` from `fastapi.security` to register the API key as an OpenAPI security scheme. Swagger then renders a global "Authorize" button at the top of `/docs`; after clicking it once, every Try-it-out includes the header automatically.

OPTIONAL: WATCH A 429 HAPPEN. A small Bash loop firing 40 `curl` requests at the endpoint inside the same minute should see the last ten return `429 Too Many Requests`. The remaining requests stay rejected until the window rolls over.

From Dev to Production: `systemd`

`uvicorn` RUNS FINE, but one Ctrl+C and it is gone. A closed terminal kills it. A reboot kills it. A panic in a worker kills it without restart. We got the interface right, but none of the above is the behaviour of a robust service.

`systemd` is the OS-level supervisor that comes with Ubuntu. It starts services on boot, restarts them when they crash, captures their output to a structured log, and exposes a uniform interface for operators. The contract between you and `systemd` is a unit file, an INI-style description with three sections: An `[Unit]` block for metadata and dependencies, a `[Service]` block naming the user, working directory, environment file, command to run, and restart policy, and an `[Install]` block declaring when the unit should be started. The exercise below has the full file PixelWise needs.

WE USE A FRACTION of what `systemd` can do. It also manages timers (cron with dependency awareness), socket activation (start on connection), resource limits (cap CPU/memory per service), and dependency ordering (start the database before the app). It is the init system of virtually every modern Linux server.
systemd: `https://www.freedesktop.org/wiki/Software/systemd/`

`systemctl` is the operator's interface to `systemd`. `start`, `stop`, and `restart` are the verbs you reach for during deployment; `status` reports the current state and the last few lines of output; `enable` marks the service as one that should start on boot; `daemon-reload` tells `systemd` to re-read its unit files after you edit one. The exercise runs the full sequence on `prod`.

`journalctl` is where you go first when something breaks. `journalctl -u pixelwise` prints every log line from your service since boot; `-f` follows new lines as they arrive, the way `tail -f` follows a file; `--since "5 min ago"` narrows the window when the bug is recent.

`gunicorn` is what production Python deployments put in front of `uvicorn` to spawn multiple worker processes, giving concurrency and crash isolation. For our classroom setup, a single `uvicorn` worker behind `systemd` is sufficient; students who deploy beyond this course will encounter `gunicorn` within the first week.

Exercise: Running as a `systemd` Service

PROMOTE THE SERVICE FROM A FOREGROUND `uvicorn` TO A SUPERVISED `systemd` UNIT. Author the unit file on `dev` alongside the app code first, then refer to the marginnote for exact commands.

AUTHOR THE UNIT FILE on `dev` at `deploy/pixelwise.service` in the repository. Keeping it under version control means anyone can rebuild `prod` from a clean VM by cloning, pulling, and copying, instead of remembering what was typed into `nano` six months ago.

```
# deploy/pixelwise.service
[Unit]
Description=PixelWise API
After=network.target

[Service]
User=produser
WorkingDirectory=/opt/pixelwise
EnvironmentFile=/opt/pixelwise/.env
ExecStart=/opt/pixelwise/.venv/bin/uvicorn \
    app.main:app --host 0.0.0.0 --port 8000
Restart=always
RestartSec=3

[Install]
WantedBy=multi-user.target
```

PRODUCTION HAS SOME CATCHING UP TO DO. FIRST DEV PUSH THEN PROD PULL.

```
git add deploy/pixelwise.service
app/main.py requirements.txt
.env.example
```

```
git commit -m "Add systemd service
file and FastAPI service with auth
and limits"
```

```
git push
```

```
ssh produser@192.168.56.11
cd /opt/pixelwise
git pull
source .venv/bin/activate
pip install -r requirements.txt
```

TEACH `setup-server.sh` TO INSTALL THE UNIT. Open the file in nano on `dev` and append a small block at the bottom so any future catch-up or fresh provision picks it up:

```
nano setup-server.sh
```

```
# Install, start, and report the systemd unit on prod
if [ -f deploy/pixelwise.service ] && \
   command -v systemctl >/dev/null 2>&1 && \
   id produser >/dev/null 2>&1; then
    sudo cp deploy/pixelwise.service /etc/systemd/system/pixelwise.service
    sudo systemctl daemon-reload
    sudo systemctl enable pixelwise
    sudo systemctl restart pixelwise
    sudo systemctl status pixelwise
fi
```

WHY SHIP THE UNIT FILE IN THE REPO? The unit file is part of how the app runs, so it belongs next to the code. Diffs are reviewable, deploys are reproducible, and there are no secrets inside, since `EnvironmentFile` only points at `/opt/pixelwise/.env`, which stays on `prod`.

PROD-ONLY BY DESIGN. The block runs only when `produser` exists, which marks the prod VM. On `dev` the user is missing, so `setup-server.sh` skips the whole step instead of copying a unit that references a user who does not exist there. On `prod`, the script installs, enables, restarts, and reports the service status on every run, picking up any change to `deploy/pixelwise.service` from the latest `git pull`. The `--no-pager` flag keeps the output inline so the script does not stop.

INSTALL THE UNIT ON PROD after the pull. Run `setup-server.sh`; it copies the unit, reloads `systemd`, enables the service, restarts it, and prints the status block in one shot:

```
bash setup-server.sh
```

KILL IT ON PURPOSE. A service that has never crashed is a service whose recovery has never been tested.

```
sudo systemctl stop pixelwise
sudo systemctl status pixelwise
sudo systemctl start pixelwise
journalctl -u pixelwise --since "5 min ago"
```

READ THE STATUS OUTPUT. `Active: active (running)` is what you want. `Active: failed` means the unit started and exited; the last few log lines below the status block usually say why. `Active: inactive (dead)` means the unit was never started.

FIRST END-TO-END LOOP ON THE NETWORK. From `dev`, hit the unprotected health endpoint on prod:

```
curl http://192.168.56.11:8000/health
```

A response of `{"status":"ok","model_version":"v1"}` comes back from the `systemd`-supervised `uvicorn` on prod. No body, no key, just a cross-machine round-trip.

THE SYSTEM IS REACHABLE ACROSS MACHINES ON A SHARED NETWORK FOR THE FIRST TIME.

Self-Reflection and Recap

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

- What is the difference between running `uvicorn` manually and running it under `systemd`?
- Why does Pydantic validation matter for security as well as correctness?
- What happens when the `systemd` service crashes, and how does `Restart=always` help?
- Why is `GET /results` a stub right now, and what would have to arrive to fill it in?
- How would you debug a failing endpoint using only `journalctl`?
- Which of the three rejection points (Pydantic, API key, rate limiter) is hardest to bypass, and why?

Recap of key concepts:

- HTTP is a stateless request-response protocol; REST layers conventions on top so APIs are predictable across teams.
- Pydantic models are the API contract; FastAPI enforces them at the boundary and renders them as live documentation.
- `POST /classify`, `GET /results`, `GET /health` cover the surface this block exposes; `/results` is a deliberate stub awaiting a persistence layer.
- API keys answer "who is calling"; rate limits answer "how often".
- `systemd` turns a script into a service: starts on boot, restarts on crash, logs to `journalctl`.

Milestone. PixelWise runs as a managed `systemd` service on the `prod` VM, listening on `http://192.168.56.11:8000`. The endpoints answer requests from `dev` and from your host machine over the host-only adapter; the machines outside the network cannot reach the service yet. `POST /classify` returns live digit predictions, `GET /results` is stubbed pending persistence, and `GET /health` reports liveness. You have killed and restarted the backend at least once and read the logs with `journalctl`. The contract is now the load-bearing surface of your backend: Every component that follows will speak through it.

Security thread. The API key lives in `.env`, never in code. Pydantic models are the input validation boundary; rate limits cap any single caller; the `X-API-Key` header keeps the secret out of access logs. Transport-layer encryption is a thread we will revisit later in the course.

Teaser. Where do all those predictions go once the response has been sent?

WHAT IS MISSING IS MEMORY. The service answers each request and forgets it the moment the response goes out. That is fine for a smoke test and broken for everything else: A user looking back at their last ten drawings has nothing to show, an operator investigating a wrong prediction cannot see what was sent, and a retraining loop cannot use real-world inputs because none of them survived. Block 6 puts PostgreSQL behind the service so every classification becomes a row, `GET /results` starts returning real history, and the service grows the audit trail the rest of the course depends on.