Build-along — ship the idea board in a box

~55 minutes, in your codespace. You'll package the idea board as a Docker image, run it as a container, and give it a real database that outlives it. Then you'll break it in the ways it breaks in production, and find out what went wrong from its logs alone. Everything today runs in your codespace: no account anywhere, and nothing goes on the internet. If you fall behind, that's fine: the files we end with are posted after class.

Use the codespace you're working on your frontend lab in. Today's work goes in a folder of its own, at the top of the repo, and nothing in `src/` changes. Don't commit it: it isn't part of the lab.

Start here (5 min)

If your server is running, stop it with **Ctrl-C**: today's containers need port 8000.

In the codespace terminal, from the top of your repo:

```
mkdir -p lecture-activities/lecture-12
cd lecture-activities/lecture-12
curl -LO https://course.aiprototypes.org/resources/files/eng190bo-fa26-lecture-12-activity.zip
python -m zipfile -e eng190bo-fa26-lecture-12-activity.zip .
rm eng190bo-fa26-lecture-12-activity.zip
```

Two images take a minute to download, so start that now, in a **second terminal**:

```
docker pull python:3.13-slim && docker pull postgres:17
```

You now have a `src/` folder, laid out the way your app will be when you deploy it:

- **`src/main.py`**, the idea board's application programming interface (API), ready to ship. Read the comment at the top: it takes its database from the `DATABASE_URL` environment variable, says which one in its first line of log, and has a `GET /health`.
- **`src/web/`**, the page, served by the same app.
- **`src/requirements.txt`**, just the packages the running app needs.

Run it the way you always have, to see it work before it goes in a box:

```
cd src
uvicorn main:app
```

The first line of the log is `Database: sqlite:///app.db`. Open port 8000 from the **PORTS** tab, add an idea, then **Ctrl-C** and `cd ..` back to `lecture-activities/lecture-12`.

Every command from here on runs from `lecture-activities/lecture-12`, the folder that holds `src/`. If you open a new terminal, `cd` there first.

1. A `Dockerfile` (12 min)

You've used one of these since your first codespace: the course template has a `Dockerfile` in its `.devcontainer/` folder, and every codespace is a container built from it. That's why everyone's has the same Python and the same tools. Now you'll write one for the idea board.

Make a file called `Dockerfile`, with no extension, in `lecture-activities/lecture-12`, next to `src/`, not in it:

```
FROM python:3.13-slim
WORKDIR /app
COPY src/ .
RUN pip install --no-cache-dir -r requirements.txt
CMD ["uvicorn", "main:app"]
```

Build an image called `idea-board` from this folder, and start a container from it:

```
docker build -t idea-board .
docker run --rm -p 8000:8000 idea-board
```

`--rm` throws the container away when it stops. `-p 8000:8000` connects port 8000 of your codespace to port 8000 of the container.

Read the log, then open port 8000 from the **PORTS** tab.

- Does the log say anything is wrong?
- In your second terminal, what does `curl localhost:8000/ideas` say?
- Find the container's name with `docker ps`, and ask from inside it:

```
docker exec <name> python -c "import urllib.request as u; \
  print(u.urlopen('http://127.0.0.1:8000/ideas').read())"
```

> We'll stop here and work out together what happened, and why.

Then fix the `CMD` so the server listens on every network interface, not just the container's own: add `"--host", "0.0.0.0"`. Stop the container (**Ctrl-C**), build again, run again, and open the page.

The platform picks the port. A platform tells your app which port to use in an environment variable called `PORT`. Change the end of your `CMD` to `"--port", "$PORT"`, build, and run. What does the log say?

That form of `CMD` runs uvicorn directly, and nothing fills in `$PORT`: that's a shell's job. So ask for a shell, and give it the whole command as one string, with a default for when `PORT` isn't set:

```
ENV PORT=8000
CMD ["sh", "-c", "exec uvicorn main:app --host 0.0.0.0 --port $PORT"]
```

Build it, then play the platform, and tell it to use port 5000:

```
docker run --rm -e PORT=5000 -p 8000:5000 idea-board
```

Port 8000 in the **PORTS** tab still opens the page. Why?

2. What's in your image (8 min)

Make a file `src/.env` holding a pretend secret:

```
echo "DATABASE_URL=postgresql://postgres:hunter2@db.example.com:5432/app" > src/.env
```

Build again, and list what's in the image, by running `ls` in a container instead of the server:

```
docker build -t idea-board .
docker run --rm idea-board ls -a /app
docker run --rm idea-board cat /app/.env
```

Your database from **Start here** and your secret are both in the image, and anyone who can get the image can read them.

Make a `.dockerignore` next to the `Dockerfile`:

```
.env
*.db
```

Build, and list again. Are they gone?

In a `.dockerignore`, a pattern without a folder only matches at the top of the folder you build from, and your files are one folder down, in `src/`. Change it to:

```
**/.env
**/*.db
**/__pycache__
```

Build and list again: only what the app needs.

Speed. Add a comment anywhere in `src/main.py`, build, and watch the steps: `pip` installs everything again. Docker reuses each step until something it depends on changes, and then redoes every step after it. Change the middle of your `Dockerfile` so it copies the list of packages first, installs them, and copies the rest last:

```
COPY src/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY src/ .
```

Build once, change your comment, and build again. Which steps say `CACHED`?

3. A database that outlives the container (12 min)

Run the image, add an idea on the page, then **Ctrl-C** and run the same `docker run` command again. Where's your idea?

The app made `app.db` inside its container, and every container starts again from the image. A platform starts a new container every time you deploy, so production needs a database that lives somewhere else.

Docker Compose runs several containers together, described in one file. Make a file called `compose.yaml` next to your `Dockerfile`:

```yaml
services:
  app:
    build: .
    ports:
      - "8000:8000"
    environment:
      DATABASE_URL: postgresql://e190bo:localdev@db:5432/app
    depends_on:
      db:
        condition: service_healthy

  db:
    image: postgres:17
    environment:
      POSTGRES_USER: e190bo
      POSTGRES_PASSWORD: localdev
      POSTGRES_DB: app
    volumes:
      - pgdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U e190bo -d app"]
      interval: 2s
      retries: 15

volumes:
  pgdata:
```

Two services: `app`, built from your `Dockerfile`, and `db`, the official Postgres image, keeping its data in a **volume** called `pgdata`. `localdev` is a local-only password: this file gets committed, so a real one must never go in it. Build and start both, in the background, then read the app's log:

```
docker compose up --build -d
docker compose logs app
```

- What's the first line of the log now? Why is the database's host `db`, and not `localhost`?
- Add an idea on the page. Then remove both containers, and start them again:

  ```
  docker compose down
  docker compose up -d
  ```

  Is your idea still there? Where does it live?
- `curl localhost:8000/health`. Then `docker compose stop db`, and ask again. Then `docker compose start db`.

4. Break it, and read the log (12 min)

In pairs. One of you looks away. The other makes **one** change to the `app` service in `compose.yaml`, chosen from this list, runs `docker compose up -d`, closes the file, and doesn't say which:

Break	How
A wrong password	change `localdev` in `DATABASE_URL`

Break	How
A host that doesn't exist	change `@db:` to `@nowhere:`
`localhost`	change `@db:` to `@localhost:`
A misspelled variable	change `DATABASE_URL:` to `DATABSE_URL:`
The wrong port	add `PORT: "5000"` under `environment:`

The other person turns back and works out what's wrong **without opening `compose.yaml`**, from only these:

```
docker compose ps -a
docker compose logs app
curl localhost:8000/health
```

Say which line told you. Then put the file back, `docker compose up -d`, and swap, until you've each found three.

Two of these don't look like errors at all. Which two, and what would they do to a real app?

When you're done, tidy up. `-v` deletes the volume too, and every idea in it:

```
docker compose down -v
```

Where you are

Your app is an image that runs the same way anywhere. It listens on `0.0.0.0` and whatever port it's given, it ships only what it needs, it keeps its data in a database that outlives it, and you can read what it says when it doesn't start. And `compose.yaml` is the whole app on one page: two services, and the variable that connects them.

What deploying it adds:

- A platform such as Railway runs the same two services: your image, from your GitHub repo, and a Postgres database, with `PORT` set and a public web address (URL).
- `DATABASE_URL` comes from the platform's variables, and the password in it never goes in a file in git.
- **Migrations:** the tables are made and changed by files you write, because a production database can't be deleted and started again.

Your frontend lab ends with the first of those: its deploy kit is the `Dockerfile` you wrote today, with four blanks. This folder was practice: keep it for reference, or delete it.