Build-along — start your Lab 5 API

~45 minutes, in your own repo. Everything you type today is the start of Lab 5, and you'll save it the way you'll save all of Lab 5: **one commit each time something works.** If you fall behind, that's fine: the file we end with is posted after class, and you can pick up from there.

You need the codespace for the repo you made from the course template. Open it now if it isn't open.

Start here: the class notebook

Class starts with a notebook, before the build-along. The notebook needs a few Python packages, so first you'll give it a project of its own, with **uv**. In the codespace terminal, from the top of your repo:

```
mkdir -p lecture-activities/lecture-10
cd lecture-activities/lecture-10
uv init --bare
uv add sqlmodel jupysql pandas jupyterlab
F=eng190bo-fa26-lecture-10-activity-skeleton.ipynb
curl -O https://course.aiprototypes.org/resources/files/$F
uv run jupyter lab
```

What those three `uv` commands did:

- `uv init --bare` made a `pyproject.toml`: a file that says *this folder is a Python project.*
- `uv add` wrote the packages into `pyproject.toml`, recorded the exact versions it chose in `uv.lock`, and installed them into `.venv/`, an environment that belongs to this folder alone.
- `uv run` runs a command inside that environment, so JupyterLab and the notebook see exactly those packages.

`pyproject.toml` and `uv.lock` are worth committing: they let anyone rebuild the same environment with `uv sync`. `.venv/` never is: it's big, it's rebuilt from those two files, and uv puts a `.gitignore` inside it so git leaves it out.

1. Ctrl-click (Cmd-click on a Mac) the `http://127.0.0.1:8888/lab?token=…` link that JupyterLab prints.
2. Double-click the notebook in JupyterLab's file list.

When you're done with the notebook, press **Ctrl-C** in the terminal JupyterLab is running in, answer `y`, and `cd ../..` back to the top of your repo. The build-along needs two terminals of its own.

---

1. Get the lab files (5 min)

In the codespace terminal, from the top of your repo (`cd` back up if you're still in `lecture-activities/`). Everything for Lab 5 goes in a `src/` folder, apart from the template's own files:

```
mkdir src
cd src
curl -LO https://course.aiprototypes.org/resources/files/eng190bo-fa26-lab-5-files.zip
python -m zipfile -e eng190bo-fa26-lab-5-files.zip .
rm eng190bo-fa26-lab-5-files.zip
git add CONTRACT.md check.py && git commit -m "Add Lab 5 contract and check script"
```

That was your first commit of the lab. Look at it, then send it to GitHub:

```
git log --oneline          # newest first: yours is at the top
git push
```

A commit lives in your codespace until you `push`. Now it's on GitHub too, which is your backup if the codespace goes away.

From here on, every command runs in `src/`. If you open a new terminal, `cd src` first.

You now have `CONTRACT.md` — the four shapes your application programming interface (API) promises — and `check.py`, which checks them. Skim §1 of the contract while we wait for everyone.

2. The smallest thing that runs (7 min)

Make `main.py` in `src/`:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/ideas")
def list_ideas():
    return [{"id": 1, "title": "Automatic plant waterer"}]
```

Start it:

```
uvicorn main:app --reload
```

The terminal doesn't give you a prompt back. It isn't frozen — it's running your server, which waits for requests forever. Open a **second terminal** with the `+` in the terminal panel, and in that one:

```
curl localhost:8000/ideas
```

Then two more ways in:

- The **PORTS** tab, next to the terminal: port 8000 is listed. Click the globe to open it in a browser, and add `/ideas` to the web address (URL).
- Change the URL to **`/docs`**. Click `GET /ideas` → *Try it out* → *Execute*. You'll use this page constantly.

Change the title in `main.py` and save. Watch the first terminal: `--reload` restarted the server. Refresh — new title.

It works, so it's a commit. In the second terminal:

```
git status                  # main.py is new ("untracked")
git add main.py
git commit -m "Serve a hardcoded list of ideas"
```

3. What's left to do (2 min)

In the second terminal:

```
python check.py
```

Most of it fails, and that's the point: the failures are your to-do list, and each message says what's wrong. Read the first one.

4. A real database (10 min)

Replace `main.py` with a table, an engine, and a `GET` that reads from them:

```python
from fastapi import FastAPI
from sqlmodel import SQLModel, Field, Session, create_engine, select

class Idea(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    title: str
    body: str = ""

engine = create_engine("sqlite:///app.db", echo=True)
SQLModel.metadata.create_all(engine)

app = FastAPI()

@app.get("/ideas")
def list_ideas():
    with Session(engine) as s:
        return s.exec(select(Idea)).all()
```

Save, and `curl localhost:8000/ideas` again. You get `[]` — an empty board is an empty array. Now look at the **first** terminal: that `SELECT` is the SQL (Structured Query Language) your object-relational mapper (ORM) just sent, exactly like the notebook.

`ls` — there's an `app.db` now. It's a real database file, and you can open it by hand:

```
sqlite3 app.db ".schema"
```

Commit this step, but look first:

```
git status                  # main.py changed... and where's app.db?
git diff                    # every line you changed, - old and + new
git add main.py
git commit -m "Read ideas from a SQLite database"
```

`app.db` isn't in `git status` at all. The template's `.gitignore` lists `*.db`, because a database is data, not code: your code creates it, and it stays on your machine.

5. `POST /ideas` — the obvious way first (10 min)

Add this to the bottom of `main.py`:

```python
@app.post("/ideas", status_code=201)
def create_idea(idea: Idea):
    with Session(engine) as s:
        s.add(idea)
        s.commit()
        s.refresh(idea)
        return idea
```

In `/docs`, `POST /ideas` → *Try it out* → send `{"title": "Sticker vending machine"}`. 201, with an `id` the database picked. `GET /ideas` shows it. In the first terminal, find the `INSERT` and the `COMMIT`.

Now send `{}`. No title. What should happen? What does?

Run the checks and find the one that agrees with you:

```
python check.py
```

Then find the real error: not in the response — in the **first terminal**, the one running the server. Read the last line of the traceback.

> We'll stop here and work out together why it happened and what the fix is. Fix it, then run `python check.py` again.

Once the 422 checks pass, commit it, with a message that says what changed and why:

```
git diff
git add main.py
git commit -m "Add POST /ideas with a separate request model, so bad input is a 422"
```

6. One idea, by its id, and a 404 (8 min)

Two of Lab 5's endpoints take an id from the URL, `GET /ideas/{id}` and `POST /ideas/{id}/vote`, and both have to say 404 when there's no such idea. Build the first one now.

At the top of `main.py`, add `HTTPException` to the FastAPI import:

```
from fastapi import FastAPI, HTTPException
```

Then add this route at the bottom:

```
@app.get("/ideas/{idea_id}")
def get_idea(idea_id: int):
    with Session(engine) as s:
        idea = s.get(Idea, idea_id)
        if idea is None:
            raise HTTPException(404, "no such idea")
        return idea
```

`{idea_id}` in the path becomes the `idea_id` argument, and `idea_id: int` makes FastAPI turn the text in the URL into a number. `s.get` looks a row up by its primary key, and gives you `None` if there isn't one.

In `/docs`, try three ids:

- one that exists: **200**, and the idea;
- `99999`: **404**, from your `raise`;
- `abc`: **422**, and your function never ran. FastAPI rejected it because `abc` isn't an `int`.

Commit it:

```
git add main.py
git commit -m "Add GET /ideas/{id}, with a 404 for an id that isn't there"
```

7. Read your history, and practise submitting (3 min)

```
git log --oneline          # your five, on top of the template's "Initial commit"
git show HEAD~2            # what the SQLite step changed
git push                   # all of it, to GitHub
```

Refresh your repo on github.com and click **commits**: the same history, the same messages.

Now a dry run of how you'll submit Lab 5, so it isn't new when you do it for real. You'll hand in a zip of your code and your history as a file:

```
git archive --format=zip --prefix=lab-05/ -o ../practice.zip HEAD
python -m zipfile -l ../practice.zip    # what's inside
git log --stat > ../practice-log.txt
rm ../practice.zip ../practice-log.txt
```

The zip is your last commit: `main.py`, `CONTRACT.md` and `check.py`, and no `app.db`. Anything you haven't committed isn't in it, which is one more reason to commit as you go.

Where you are

In `python check.py`, every Contract check should pass except the ones about votes: `vote_count`, the `votes` list, and voting. Your repo has five commits of your own, pushed to GitHub. What's left for Lab 5 is in `CONTRACT.md` and the Lab 5 handout:

- a `vote` table
- `vote_count` on every idea in `GET /ideas` — the notebook's B3 join
- a `votes` list on `GET /ideas/{id}`
- `POST /ideas/{id}/vote`
- the 404s and 422s in the rest of `check.py`

> One Quality check, voting on a missing idea answers 404, already passes, but only because the vote route doesn't exist yet. It'll mean something once you've written it.