Lab 5 — Backends

Servers, APIs, and databases

The goal of this lab is to give you an introduction to databases, servers, and Application Programming Interfaces (APIs). In Labs 6 and 7, we will build on top of this to build a user interface for it (Lab 6) and deploy it on the internet (Lab 7). Each of those labs starts in a fresh repo from a working version of the previous lab that we give you, so you're never building on shaky ground. The deliverables for the labs moving forward will be a little different than what was submitted for your first four labs. In those labs, we worked primarily out of a Jupyter notebook. In the remaining labs, you'll submit your code along with its Git history, which shows not only the final code that you wrote, but a log of all of the changes that you made along the way.

What to build

The main project for the next couple of labs will be to build an idea board, an app where we can post ideas and vote on them. In this lab we will build the database and an API to interact with it. It will be composed of two tables and four endpoints:

- `GET /ideas` — list all ideas, each with its `vote_count`
- `POST /ideas` — add an idea
- `GET /ideas/{id}` — return one idea, with its votes
- `POST /ideas/{id}/vote` — vote for one

The exact shapes are in **`CONTRACT.md`**, which comes with the lab files (below). Everything it does *not* mention, like your file layout, table names, extra fields, extra endpoints, whether you use an object-relational mapper (ORM) at all, is yours to decide.

Get the lab files into your repo

Lab 5 lives in the repo you made from the Codespaces template. Labs 6 and 7 will each get a new repo of their own. To download the files, you can use the code below. In your Codespace terminal, download the files as shown:

```
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
```

This will give you a `CONTRACT.md` document and a `check.py` script. Commit them and push that commit to GitHub:

```
git add CONTRACT.md check.py && git commit -m "Add Lab 5 contract and check script"
git push
```

Getting started on Lab 5

Here are a few additional topics that will help you get started on Lab 5.

1. A server doesn't exit

When you start your server, the terminal **stops giving you a prompt back**. It runs your program, which is an infinite loop waiting for requests.

This means you will need **two terminals**: one running the server, one for everything else. In VS Code, the `+` in the terminal panel. `Ctrl-C` stops a server.

2. `uvicorn`

`uvicorn` is the program that actually listens on the port and hands requests to your code. You can run the app using the following command.

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

This will run uvicorn pointing at **"the variable `app`, inside `main.py`"**. If you rename your file or your variable that argument will change. `--reload` restarts the server whenever you save, so leave it on while building.

3. Ports, and how you reach them

Your server listens on port **8000** *inside* your codespace. VS Code notices and forwards it: look for the **PORTS** tab next to the terminal, which gives you a web address (URL) you can open in a browser.

From inside the codespace, `http://localhost:8000` works in a terminal. `curl` and `check.py` use this URL.

4. `/docs` is very helpful!

FastAPI reads your code and generates a full API browser at **`http://localhost:8000/docs`**. You can send a `POST` with a JavaScript Object Notation (JSON) body by filling in a form, instead of typing a long `curl`.

5. `check.py`

`check.py` checks your API against `CONTRACT.md`. It sends real requests to your running server, the way a web page would, and prints PASS or FAIL for each promise. With your server running in the other terminal, run it from `src/`:

```
python check.py
```

Every FAIL says what to look at. Fix the first FAIL first: later checks often fail because of an earlier one. The **Contract** checks are what Lab 6's page relies on, so they have to pass. The **Quality** checks are good API behavior: aim to pass all of them. It's a short Python script, so open it and see how it works.

6. `git`: save your work in small, working steps

Git is how you document your work and an important part of the learning objectives for this lab. It shows a history of your work in small steps.

Here's the loop you should use with git.

```
git status                  # what changed?
git diff                    # exactly what changed — read it
python check.py             # does it still work?
git add main.py             # choose what goes in this step
git commit -m "Add POST /ideas with a separate request model"
git push                    # send it to GitHub: your backup
```

What makes a good commit:

- **One idea.** "Add the votes table" is a commit. "Lab 5" is not.
- **It works.** Run `python check.py` first. A history of working steps is one you can go back to.
- **The message says what, and why if it isn't obvious.** Write it for your future self: `Count votes with a LEFT JOIN so unvoted ideas show 0`.

Two commands are helpful for looking back over your commit history:

```
git log --oneline           # your history, one line per commit
git show <hash>             # what one commit changed
```

VS Code or the GitHub web app also have convenient ways to view the git history in a graphical user interface (GUI).

The smallest thing that runs

Put this in `main.py` (in `src/`, next to `CONTRACT.md`), run it, and see it work *before* you add a database:

```python
from fastapi import FastAPI

app = FastAPI()

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

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

Then open `/docs`, click the endpoint, and hit Execute.

As you build out additional routes, you'll need these pieces from FastAPI. The **FastAPI Reading** sheet posted with this handout walks through each of these in the FastAPI docs, in about 30 minutes.

For	Write
A route	`@app.get("/ideas")` / `@app.post("/ideas")`
Part of the URL	`@app.get("/ideas/{idea_id}")` → `def f(idea_id: int)`
A JSON request body	a class, taken as a parameter: `def f(payload: IdeaCreate)`
"I made something"	`@app.post("/ideas", status_code=201)`
"That doesn't exist"	`raise HTTPException(404, "no such idea")`
A response with an extra field	`return {**idea.model_dump(), "vote_count": n}`

We'll also use the `sqlmodel` Python library for interacting with our SQLite database. This is what you will need from that module.

For	Write
A table	`class Idea(SQLModel, table=True):`
Connecting	`engine = create_engine("sqlite:///app.db")`
Creating the tables	`SQLModel.metadata.create_all(engine)`
Reading	`with Session(engine) as s: s.exec(select(Idea)).all()`
One row, by its id	`s.get(Idea, idea_id)`, which is `None` if there's no such row
A column that points at another table	`idea_id: int = Field(foreign_key="idea.id")`
Writing	`s.add(obj); s.commit(); s.refresh(obj)`
Seeing the SQL (Structured Query Language) it sends	`create_engine("sqlite:///app.db", echo=True)`

The id column is always the same line: `id: int | None = Field(default=None, primary_key=True)`. It's `None` until you save the row, and then the database fills it in.

For `vote_count` you need one more thing: a join through the ORM. Leave `echo=True` on while you build it and read what comes out.

> ⚠ **One trap to watch out for** A `SQLModel` class with `table=True` **skips validation**. If you use your table class as the request body, a junk `POST` sails past it and blows up at the database — you'll get a `500` where `check.py` expects a `422`. The fix is a *separate, non-table* class for the body.

What to submit on Canvas

1. **A one-page executive summary** — what you built, one design decision you made that the contract left open, and the bug that took you longest.
2. **Your code, as a zip.** The zip is built from your last commit, so anything you haven't committed isn't in it. From `src/`:

   ```
   git status                  # nothing left uncommitted? commit it first
   git archive --format=zip --prefix=lab-05/ -o ../lab-05.zip HEAD
   python -m zipfile -l ../lab-05.zip    # check what's inside
   ```

   `git archive` packs up `src/` exactly as it is in your last commit. Your database, `app.db`, isn't in it, because `.gitignore` keeps it out of your commits.
3. **Your git history**, as a file:

   ```
   git log --stat > ../lab-05-git-log.txt
   ```

   `--stat` lists the files each commit touched.

Both files land at the top of your repo, next to `src/`. Right-click each one in the VS Code explorer ⟶ **Download**, and upload them to Canvas. Then delete them: they don't belong in the repo.

Made a fix after submitting? Commit it, make both files again, and resubmit.

How this is graded

Your lab grade comes primarily from a **10-minute one-on-one check-in**, not from the submission. You demo your work and answer questions about it, without AI.

Check-in topics

Here are a few topics you should be prepared to discuss at your check-in.

- **Trace** — Walk one request end to end through your own code, URL to response body.
- **Design** — Why did you split the data into these two tables?
- **SQL** — Show me the SQL your ORM sends for `GET /ideas`. Read it to me.
- **Live change** — Add a field right now: model and endpoint.
- **Git** — Walk me through your history. Pick a commit and show me what it changed.

Getting help

- The Lab 5 tutor in Glassroom AI.
- Office hours
- Zulip, for concepts. Not for each other's code.