Build-along — a page for the idea board

~55 minutes, in your codespace. You'll build a web page for the idea board, served by an application programming interface (API) like your Lab 5 one, and then break the API on purpose to see what your page does. Every step ends in a commit, the way Lab 5 did. If you fall behind, that's fine: the files we end with are posted after class.

You need the codespace for the repo you made from the course template: the one with your Lab 5 work in `src/`. Open it now if it isn't open. Today's work goes in a folder of its own, next to `src/`, and nothing in `src/` changes.

Start here: the files (5 min)

If your Lab 5 server is running, stop it with **Ctrl-C**: today's server needs port 8000.

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

```
mkdir -p lecture-activities/lecture-11
cd lecture-activities/lecture-11
curl -LO https://course.aiprototypes.org/resources/files/eng190bo-fa26-lecture-11-activity.zip
python -m zipfile -e eng190bo-fa26-lecture-11-activity.zip .
rm eng190bo-fa26-lecture-11-activity.zip
```

You now have:

- **`main.py`**, a stand-in for your Lab 5 API. It answers `GET /ideas` and `POST /ideas` in exactly the shapes `CONTRACT.md` promises, each idea with a `vote_count`, from a Python list instead of a database. Your page can't tell the difference, because all it ever sees is HTTP (Hypertext Transfer Protocol) and JSON (JavaScript Object Notation).
- **`web/index.html`**, the page: a heading, a form for adding an idea, and an empty list.
- **`web/style.css`**, a few styles, so you can ignore CSS (Cascading Style Sheets) today.
- **`web/app.js`**, empty apart from a comment: you'll write it.

Read `main.py` while everyone catches up. Nothing in it should be new.

Every command today runs from `lecture-activities/lecture-11`. If you open a new terminal, `cd` there first.

1. Serve the page (5 min)

Start the API, as in Lab 5:

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

Open port 8000 from the **PORTS** tab. `/ideas` is the JSON, but `/` is `{"detail":"Not Found"}`: nothing serves the page yet.

Add this import at the top of `main.py`:

```
from fastapi.staticfiles import StaticFiles
```

and this as the **last line**:

```
app.mount("/", StaticFiles(directory="web", html=True), name="web")
```

Save, and reload `/`: there's the page. `/ideas` and `/docs` still answer beside it.

Try it: move the `app.mount(...)` line up, just under `app = FastAPI()`, save, and open `/ideas` again. What do you get, and why? Move it back to the bottom.

Commit. In a **second terminal** (`cd lecture-activities/lecture-11` first):

```
git status                  # only files in this folder
git add .
git commit -m "Serve the idea board page from the API"
```

2. List the ideas (8 min)

First, set up the browser's developer tools, and keep them open from here on. Press **F12** (or **Cmd-Option-I** on a Mac), open the **Network** tab, and tick **Disable cache**. Without it, the browser can keep running your *old* `app.js` after you've saved a new one, because the server doesn't tell it not to.

Now put this in `web/app.js`, below the comment:

```
const list = document.querySelector("#ideas");

async function loadIdeas() {
  const response = await fetch("/ideas");     // GET /ideas, from this server
  const ideas = await response.json();        // the body, parsed from JSON
  list.replaceChildren();                     // empty the list
  for (const idea of ideas) {
    const li = document.createElement("li");
    li.textContent = `${idea.title} (${idea.vote_count} votes)`;
    list.append(li);
  }
}

loadIdeas();
```

Save, and reload the page: three ideas. Changes to `web/` need no restart, just a reload.

Look at it two ways:

- **Network** tab, then reload: click the `ideas` row. Its status, and its **Response**, the same JSON as `curl`.
- **Elements** tab: three `<li>`s in the list. Then **View page source**: the list in the file is empty. The script put them in the DOM (Document Object Model), the page as a tree, and the DOM is what you see.

Commit:

```
git add .
git commit -m "List the ideas from GET /ideas"
```

3. Add an idea (10 min)

The form is already in `index.html`. Put this at the bottom of `web/app.js`:

```
const form = document.querySelector("#add-form");
const addStatus = document.querySelector("#add-status");

form.addEventListener("submit", async (event) => {
  event.preventDefault();                    // stay on this page
  const title = document.querySelector("#title").value;
  const response = await fetch("/ideas", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ title: title }),
  });
  if (response.ok) {
    form.reset();                            // a failed add keeps the text
    addStatus.textContent = "Added.";
    loadIdeas();
  } else {
    addStatus.textContent = `Couldn't add that. The server said ${response.status}.`;
  }
});
```

Add an idea. In the **Network** tab, click the `POST`: its **Payload** is the JSON your code built, and its **Response** is what your API sent back. Which status, and where did the new `id` come from?

Trace it, with a neighbour. One of you points at each line in turn, and the other says what happens, all the way from the text in the box to the new idea on the screen.

Commit:

```
git add .
git commit -m "Add ideas from a form"
```

4. Break the API (20 min)

In `main.py`, change the first import to:

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

and make this the first line inside `list_ideas`:

```
    raise HTTPException(500)
```

Save: `--reload` restarts the server. (The ideas you added are gone now. This stand-in keeps them in a Python list, and a restart is a new process with a new list: exactly why Lab 5 has a database.) Reload the page.

- What does someone using the page see?
- What does the **Console** say? What does the **Network** tab say?

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

Then make `loadIdeas` honest about the network. Replace everything from `const list` to the end of `loadIdeas` with this, and fill in the two `TODO`s:

```
const list = document.querySelector("#ideas");
const listStatus = document.querySelector("#list-status");

async function loadIdeas() {
  listStatus.textContent = "Loading ideas…";
  try {
    const response = await fetch("/ideas");
    // TODO 1: if the response isn't OK, throw an Error that says its status.

    const ideas = await response.json();
    list.replaceChildren();
    for (const idea of ideas) {
      const li = document.createElement("li");
      li.textContent = `${idea.title} (${idea.vote_count} votes)`;
      list.append(li);
    }
    listStatus.textContent = ideas.length ? "" : "No ideas yet.";
  } catch (err) {
    // TODO 2: put "Couldn't load ideas." and err.message in listStatus,
    // then add a "Try again" button to it that calls loadIdeas when clicked.
  }
}
```

For `TODO 2` you need `document.createElement("button")`, `addEventListener("click", loadIdeas)` and `listStatus.append(...)`.

When your page says something useful about the 500:

1. Stop the server with **Ctrl-C**, then click **Try again**. Is the message different? Why? (Don't reload the page here: the page itself comes from that server, so the browser would show its own error page instead of yours.)
2. Delete the `raise`, start the server again, and click **Try again**. The ideas come back, with no page reload.

Commit:

```
git add .
git commit -m "Show loading, error and empty states"
```

If you finish early

- **A hostile title.** Add an idea titled exactly `<img src=x onerror=alert(1)>`. It should show up as those characters. If an alert pops up instead, your page put data in with `innerHTML`, so the browser built a real image tag and ran its `onerror`. That's cross-site scripting (XSS), and the fix is `textContent`.
- **A slow server.** Put `time.sleep(2)` as the first line of `list_ideas`, and `import time` at the top, then reload. What do you see for those two seconds? Take it out again.
- **A missing header.** Delete the `headers:` line from the form's `fetch`, save, reload, and add an idea. What does the page say, and what's in the response? Put the line back.
- **A form that copes.** Stop the server and add an idea. Nothing appears on the page, because nothing catches a `fetch` that fails outright. Make the form cope, without losing what was typed.

Where you are

Your page lists the ideas, adds new ones, and tells the truth when a request fails. That's most of what Lab 6 asks of a page, and the part its check-in tests live, by breaking your API.

What Lab 6 adds:

- **Voting**, with a `voter`, through `POST /ideas/{id}/vote` from your contract.
- **Semantic, accessible HTML** that its tests check: headings in order, a label on every control.
- **Phone width:** nothing overflows sideways at 375 pixels.
- **One design decision** you can defend.
- **Git branches**, a merge, and a merge conflict, on purpose.

Lab 6 starts in a new repo, from a starter with a complete, working Lab 5 API. This folder was practice: keep it, for reference.