Lab 6 — Frontends and Deployment

Design web interfaces and deploy them

A face for the idea board application programming interface (API) from Lab 5. Lab 6 gets its own repo and codespace, and it starts from a working Lab 5 API that we give you. At the end, you put it on the internet: the API and the page together, in one container, on Railway.

In Lab 5, the only thing that ever called your API was you, or `check.py`, and both of you knew what to expect back. This time the caller is a web page that a person is looking at. The network can be slow, your server can be down, and a request can fail halfway through. A page that assumes every request works will sit there looking broken, and the person using it has no idea why. Most of this lab is making the page tell them what's going on.

What to build

One page for the idea board, in plain HTML (HyperText Markup Language), CSS (Cascading Style Sheets), and JavaScript.

1. **It reads and writes through your API.** It lists every idea with its `vote_count`, adds a new idea from a form, and lets someone vote. A vote needs a `voter`, and where the name comes from is up to you. Showing who voted (`GET /ideas/{id}`) is optional.
2. **Semantic, accessible HTML.** A screen reader should be able to make sense of the page: one `<h1>`, headings in order, a label on every form control, real `<button>`s for things you click.
3. **Honest about the network.** Show loading, error, and empty states that a person can understand. An error says what went wrong and gives them a way out, such as a retry. A spinner that never stops doesn't count. A failed add shouldn't throw away what they typed.
4. **Works at phone width.** At about 375 pixels wide, nothing overflows sideways and nothing needs zooming.
5. **Three design tasks:** search and sort, a spotlight on the leading idea, and a light and dark theme. Each one takes some HTML, some CSS and some JavaScript; they're described below.
6. **One design decision you can defend.** Pick one thing you deliberately cut, simplified, or did differently, and be ready to say why.
7. **On the internet.** Your app in a Docker container, deployed on Railway from your repo, at a live `https://` web address (URL) that works on a phone. We give you the `Dockerfile` with four blanks to fill in.

Your page talks to the Lab 5 API that comes in the starter (below). It meets Lab 5's contract, and everyone gets the same one, so how your own Lab 5 went doesn't matter here. Apart from one line in `main.py` and four blanks in a `Dockerfile`, everything you build in this lab is the page.

Get started

Lab 6 lives in a new repo, with a new codespace, made the same way as your Lab 5 one. Your Lab 5 repo stays as it is.

1. Make the repo from the template

1. Signed in to GitHub, open the course template: https://github.com/joshbrake/e190bo-devcontainer-template.
2. Click **Use this template → Create a new repository**, and fill it in:
   - **Owner: your own account**, not an organization.
   - **Repository name:** `e190bo-lab-06`.
   - **Visibility: Private.**
3. Click **Create repository**.

Do not click Fork. A fork of the template is stuck public, and its history isn't yours.

2. Create its codespace

In **your new repo**, click **Code → Codespaces → Create codespace on main**. It's a new codespace, so the first build takes a few minutes, like your first one did. You're ready when the terminal prints `E190BO environment ready.`

Every codespace uses up your storage allowance, even when it's stopped. If you run low, delete your Lab 5 codespace once your Lab 5 check-in is done: on https://github.com/codespaces, open the three-dot menu (**…**) next to it and choose **Delete**. Push first: deleting a codespace throws away anything in it that isn't on GitHub, but your Lab 5 repo itself is untouched.

3. Download the starter

In the new codespace's terminal, from the repo root:

```
mkdir src && cd src
curl -LO https://course.aiprototypes.org/resources/files/eng190bo-fa26-lab-6-starter.zip
python -m zipfile -e eng190bo-fa26-lab-6-starter.zip .
rm eng190bo-fa26-lab-6-starter.zip
git add . && git commit -m "Add Lab 6 starter"
git push
```

The starter holds:

- **`main.py`**, a working Lab 5 API. It's the same code for everyone, so nobody builds on a shaky Lab 5. Read it before you start: your page is its client.
- **`CONTRACT.md`**, Lab 5's contract: the exact shapes your page sends and gets back.
- **`requirements-app.txt`**, the three packages your app needs to run, for the Docker image.
- **`check.py`**, Lab 5's self-check, with two new parts: checks on your page, and on your `Dockerfile`. Leave it as it came. If a check looks wrong, tell us instead of editing it.

There's no `web/` folder yet: you make it. Until you build the page, `check.py` has one FAIL (`GET / serves your page`), and it skips the other page checks because there's no page to check. It skips the `Dockerfile` checks too, until you put the app on the internet at the end. That's expected. The API checks already pass.

Everything in this lab runs from `src/`: the server, `check.py`, and the git commands. A new terminal starts at the repo root, so `cd src` first. This handout uses the commands themselves rather than the template's `make` shortcuts, so you know what each one does. From here on, paths are relative to `src/`: `web/index.html` means `src/web/index.html`.

Where the page lives, and how it's served

Your page is a few static files in a `web/` folder next to `main.py`:

```
e190bo-lab-06/
  src/
    main.py
    CONTRACT.md
    web/
      index.html
      app.js
      style.css
    check.py
```

Your API server serves those files too. Add this as the **last line** of `main.py`, with the import at the top:

```
from fastapi.staticfiles import StaticFiles

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

Start the server the same way as in Lab 5, from `src/`:

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

Then open the forwarded port 8000 from the **PORTS** tab. `/` is now your page (`html=True` makes it serve `web/index.html`), and `/ideas` and `/docs` still work alongside it.

A few things follow from that one line:

- **It has to be last.** FastAPI checks your routes in the order you declared them, and a mount at `"/"` matches every path. If it comes before your routes, `GET /ideas` looks for a file called `web/ideas` and returns a 404, and `POST /ideas` returns a 405. The page loads but the API is gone. The tests check for this and tell you if it happens.
- **`web` is relative to where you start the server.** Run `uvicorn` from `src/`, the folder that holds both `main.py` and `web/`. If the server can't find the folder, it won't start at all: `RuntimeError: Directory 'web' does not exist`.
- **Use relative URLs (web addresses) in your JavaScript.** Write `fetch("/ideas")`, never `fetch("http://localhost:8000/ideas")`. The browser sends a relative URL back to wherever the page came from, which is your server. `localhost` in the browser means *your laptop*, not the codespace, and once it's deployed, the server moves again. A relative URL keeps working in both places without changes. The tests check this too.
- **Changes to `web/` need no restart, but they do need an uncached reload.** The server reads the files fresh on every request, so save and reload the browser. It doesn't tell the browser not to cache them, though, and after an ordinary reload Chrome can keep running your *old* `app.js`. So while you work on the page, keep the developer tools open with **Network → Disable cache** ticked (below). Without them open, a hard reload (`Ctrl-Shift-R`, or `Cmd-Shift-R` on a Mac) does the same job once.

If your copy of the template has a `make web`, don't use it: it serves a `web/` folder on a second port, 8080. That's exactly the two-server setup this lab avoids.

Why there's no CORS to deal with

The browser treats each scheme + host + port as a separate **origin**. A page can freely read responses from its own origin. It can only read a response from a *different* origin if that server opts in with cross-origin resource sharing (CORS) headers (`Access-Control-Allow-Origin`). Your page and your API come from the same server, so they share one origin and there's nothing to configure. You'll still run into CORS eventually, so learn the name now. The Console error says *"blocked by CORS policy"*, and it means a page on one origin called an API on another that didn't say yes. The browser enforces this, not the server, which is why `curl` and `check.py` never see it.

The five things you need to know

You've met HTML, `curl`, and the idea board API (Lab 5). These five are new.

1. The DOM, and when your code runs

The browser turns your HTML into a tree of objects called the **DOM** (Document Object Model). That tree, not the file, is what's on screen. JavaScript changes the page by changing the tree: find an element, set its text, build new elements and put them in.

Your script runs **once**, top to bottom, when the page loads. Load it with `defer`:

```
<script src="app.js" defer></script>
```

`defer` means "run this after the whole page has been parsed." Without it, a script in `<head>` runs before `<body>` exists, `document.querySelector` returns `null`, and the Console shows `Cannot read properties of null (reading 'addEventListener')`.

After that first run, your code only runs again when something *happens*: a click, a form submit, or a response arriving for a request you were `await`ing. The page never refreshes itself when the database changes. It only knows what you last fetched.

2. `fetch`, `await`, and `response.ok`

```
const response = await fetch("/ideas");
const ideas = await response.json();
```

`await` pauses *this function* until the response arrives. The rest of the page stays live in the meantime, and that gap is where your loading state belongs.

> ⚠ **One trap, on purpose.** `fetch` only fails (rejects) when there's **no response at all**, because the server is down or the network is gone. A `404` or a `500` is still a response, so `fetch` succeeds and your code carries on. Checking whether the request actually worked is your job, and that's what `response.ok` is for. Try it: add `raise HTTPException(500)` as the first line of your `GET /ideas` function and reload the page. Then undo that and stop the server instead. Those are two different failures, and your page needs to handle both. You will be asked about this.

To send JavaScript Object Notation (JSON), say that it's JSON:

```
await fetch("/ideas", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ title: "Plant waterer" }),
});
```

If you leave out the header, the browser labels the body as plain text, and FastAPI answers `422` with `Input should be a valid dictionary`.

3. `textContent`, never `innerHTML`, for data

Anything that came from your API, such as a title, a body, or a voter name, goes into the page with `textContent`. `innerHTML` reads its input as HTML, so an idea titled `<img src=x onerror=alert(1)>` would *run code* in the browser of everyone who opens the board. That's **cross-site scripting (XSS)**, and it is one of the most common security bugs on the web. With `textContent`, the same title just shows up as those characters.

4. Forms, and `event.preventDefault()`

Put your add-an-idea inputs in a real `<form>`. That gets you Enter-to-submit, and it tells assistive technology that the inputs belong together. Then handle the form's `submit` event, not a click on the button.

By default, submitting a form makes the browser *leave the page*. It loads `/?title=...` and throws away everything your script was doing. The first line of your handler stops that:

```
form.addEventListener("submit", async (event) => {
  event.preventDefault();
  // read the inputs, send them yourself
});
```

One more thing. A `<button>` inside a form submits the form unless you say `type="button"`.

5. The browser's developer tools

Open them with `F12`, or `Cmd-Option-I` on a Mac, or right-click → **Inspect**. For this lab they're as important as `/docs` was for Lab 5.

- **Console:** JavaScript errors, with a file and line number you can click. Check it first when nothing happens. A red 404 for `favicon.ico` is harmless.

- **Network:** every request the page makes. Click one to see its status, the body you sent (**Payload**), and what came back (**Response**). "Is it my page or my API?" gets answered here. Tick **Disable cache** and leave it ticked: it only works while the developer tools are open. The throttling menu has **Offline**, which is a quick way to see what your page does when the network is gone.
- **Elements → Styles:** click an element to see every CSS rule that applies to it. A rule that lost is shown ~~crossed out~~.
- **Device toolbar** (`Ctrl-Shift-M`, or `Cmd-Shift-M` on a Mac): shows the page at phone width.

The smallest thing that runs

Start here, with the mount line already in `main.py`. `web/index.html`:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Idea board</title>
  <script src="app.js" defer></script>
</head>
<body>
  <h1>Idea board</h1>
  <ul id="ideas"></ul>
</body>
</html>
```

`web/app.js`:

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

async function loadIdeas() {
  const response = await fetch("/ideas");
  const ideas = await response.json();
  for (const idea of ideas) {
    const li = document.createElement("li");
    li.textContent = `${idea.title} (${idea.vote_count} votes)`;
    list.append(li);
  }
}

loadIdeas();
```

Reload the page and your ideas are there. That's a working frontend. It also assumes every request succeeds and it has no way to add or vote. The rest of the lab is fixing both.

You'll need roughly this much JavaScript:

For	Write
Find an element	`document.querySelector("#ideas")`
Put data in it	`el.textContent = idea.title`
Make a new one	`document.createElement("li")`, then `list.append(li)`

For	Write
Replace a list's contents	`list.replaceChildren(...items)`
React to a click	`button.addEventListener("click", () => vote(idea))`
React to a form	`form.addEventListener("submit", (event) => { event.preventDefault(); … })`
Read an input	`document.querySelector("#title").value`
Did it work?	`response.ok` (true for any 2xx), `response.status`
Catch a failure	`try { … } catch (err) { … }` around the `await`
Stop a double click	`button.disabled = true`
Remember something in this browser	`localStorage.setItem("voter", name)` / `localStorage.getItem("voter")`
React to every keystroke in a box	`search.addEventListener("input", render)`
React to a new choice in a menu	`select.addEventListener("change", render)`, then `select.value`
Keep some of a list, or reorder it	`ideas.filter((idea) => …)`, `[...ideas].sort((a, b) => …)`
Turn a class on or off	`item.classList.toggle("top", isTop)`
Hide or show an element	`el.hidden = true`
Set an attribute	`el.setAttribute("aria-pressed", "true")`, or `el.dataset.theme = "dark"` for `data-theme`
Ask about the person's system	`matchMedia("(prefers-color-scheme: dark)").matches`

And roughly this much HTML and CSS:

For	Write
A labelled input	`<label for="title">Title</label> <input id="title">`
Text a screen reader announces when it changes	`<p role="status"></p>`
"This part is still loading"	`aria-busy="true"` on the element, `"false"` when done
Ready for phones	`<meta name="viewport" content="width=device-width, initial-scale=1">`
A column that stops growing	`max-width: 42rem; margin: 0 auto;`
A row that wraps instead of overflowing	`display: flex; flex-wrap: wrap;`
A drop-down menu, labelled	`<label for="sort">Sort by</label> <select id="sort"><option value="votes">Most votes</option>…</select>`
A colour you can change in one place	`:root { --ink: #1f2937; }`, then `color: var(--ink);`
Different values when an attribute is set	`:root[data-theme="dark"] { --ink: #e5e7eb; }`

When two CSS rules disagree about the same element, one wins and the Styles pane shows the other crossed out. Work out *why* that one won. You will be asked.

Three design tasks

Once the page lists, adds and votes, and is honest about the network, add these three. They all work from the ideas your page already fetched, so none of them needs a change to the API. Each needs some HTML, some CSS and some JavaScript, and how it looks is up to you.

1. Search and sort

Someone looking for an idea about bikes shouldn't have to scroll past every other one.

- **HTML:** a search box and a sort menu above the list, each with a `<label>`. The menu offers at least *most votes*, *newest* and *title, A to Z*. (Ids only go up, so a bigger `id` is a newer idea.)
- **CSS:** the two controls sit side by side on a wide screen and stack at phone width.
- **JavaScript:** typing in the box shows only the ideas whose title or details contain what's typed, and changing the menu reorders them. Neither makes a new request: redraw from the list you already have. Say how many are showing ("Showing 2 of 8 ideas"), and when nothing matches, say that, in words that can't be mistaken for an empty board.

2. A spotlight on the leader

- **HTML:** room on each idea for a "Top idea" label.
- **CSS:** the leading idea stands out from the rest, and not by colour alone: the label is text, so it means something to someone who can't see the colour.
- **JavaScript:** after every redraw, the idea with the most votes gets the label and the styling. Decide what happens on a tie, and when every idea has 0 votes. It has to update after someone votes.

3. A light and a dark theme

- **HTML:** a `<button>` that switches between them. It keeps the same name either way and uses `aria-pressed="true"` or `"false"` to say whether the dark theme is on.
- **CSS:** put your colours in custom properties (`--ink`, `--bg`, and so on), use them everywhere, and give the dark theme its own values under `:root[data-theme="dark"]`. Text has to stay readable in both.
- **JavaScript:** the button sets `data-theme` on `<html>`, updates `aria-pressed`, and remembers the choice in Local Storage, so a reload keeps it. Until someone chooses, follow the system's own setting.

`check.py` can't see any of this: it only reads your HTML before your script runs. Try each one yourself, at phone width as well, and expect the check-in to ask about them.

Running `check.py`

Same as Lab 5: the server runs in one terminal and `check.py` in the other, both from `src/`.

```
python check.py
```

It prints a PASS or FAIL line for each check, in three parts, and every FAIL says what to look at:

- **The API.** Lab 5's checks, because the API still has to keep its contract, and you edit `main.py` in this lab. Your page is one more thing that uses it.
- **The page.** These read your HTML exactly as the server sends it, *before any JavaScript runs*. They check its structure: it's served, its scripts load, no host is hardcoded, controls are labelled, headings are in order. They can't see anything your script draws, and they can't tell whether your error states work. That part is the check-in.
- **The `Dockerfile`**, skipped until you add one at the end of the lab.

`check.py` writes ideas like `Test idea 4f2a9c` into your database, so you'll see them on your page. For a clean board, stop the server, delete `app.db`, and start it again.

Git in this lab: branches, a merge, and a conflict

Lab 5 was a straight line of commits on `main`. In this lab you'll work on a **branch**: a line of commits off to the side, which only joins `main` when it works. It's how you try something without breaking what you have — and how every team you'll ever join works. Run the commands below from `src/`, like the rest of the lab.

Practice first: Learn Git Branching (~25 min)

Same game as Lab 5, next levels, all on the **Main** tab. Each level name links straight to it.

Menu location	Level	Why it matters here
Introduction Sequence — Level 2	Branching in Git	what a branch is
Introduction Sequence — Level 3	Merging in Git	what a merge makes
Ramping Up — Level 4	Reversing Changes in Git	undoing a commit that's already there
Introduction Sequence — Level 4 *(optional)*	Rebase Introduction	the other way to combine work. Read it, but merge in this lab

Build the page on a branch

```
git switch -c page              # make a branch called "page", and move to it
# ...build, and commit as you go, exactly like Lab 5...
git push -u origin page         # the first push of a new branch says where it goes
```

When the page works and `check.py` passes, bring it into `main`:

```
git switch main
git merge page
git push
```

A merge conflict, on purpose (~10 min)

A **conflict** is git asking you a question: two branches changed the same line, and it won't guess which one you meant. After `page` is merged:

```
git switch -c heading-short
# change the text of your <h1> in web/index.html to something short, then:
git commit -am "Shorten the heading"

git switch main
git switch -c heading-long
# change the SAME <h1> to something longer, then:
git commit -am "Spell out the heading"

git switch main
git merge heading-short         # clean: main hadn't changed, so git just moves forward
git merge heading-long          # CONFLICT
```

Git's message names the file `src/web/index.html`, its path from the repo root. From `src/`, that's `web/index.html`. Open it. Git has written both versions into the file:

```
<<<<<<< HEAD
    <h1>Ideas</h1>
=======
    <h1>Course project idea board</h1>
>>>>>>> heading-long
```

Above the `=======` is what `main` has now; below is what the branch wanted. Decide what the line should be (one, the other, or a new one), then delete all three marker lines. Then finish the merge:

```
git add web/index.html
git commit --no-edit            # keep the message git wrote: "Merge branch 'heading-long'"
git branch -d heading-short heading-long
git log --graph --oneline       # see the fork, and the merge that joined it
```

Leave the markers in by mistake and `check.py` will tell you: the page now has two `<h1>`s.

Put it on the internet

Your page and your API come from one server, so they ship as one thing: a Docker **container**, started from an **image** that holds Python, your packages, and everything in `src/`. The video *An Image That Runs Anywhere* explains every line of the `Dockerfile`, and *From Compose to Railway* walks through Railway. Do this on `main`, once your page is merged.

1. Fill in the `Dockerfile` (~10 min)

From the **repo root** (`cd ..` if you're in `src/`), download the deploy kit:

```
git switch main
curl -LO https://course.aiprototypes.org/resources/files/eng190bo-fa26-lab-6-deploy.zip
python -m zipfile -e eng190bo-fa26-lab-6-deploy.zip .
rm eng190bo-fa26-lab-6-deploy.zip
```

You now have a `Dockerfile` and a `.dockerignore` at the top of your repo, next to `README.md`, not in `src/`. The `.dockerignore` is complete: it keeps your `app.db` and any `.env` out of the image. The `Dockerfile` has four blanks, `____`, and the comment above each one says what goes there. Fill them in, then check them:

```
cd src
python check.py             # with your server running, as usual
cd ..
```

2. Build it and run it in your codespace (~10 min)

Stop your dev server first, because both want port 8000. From the repo root:

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

The first build downloads Python and your packages, so give it a minute. Then open port 8000 from the **PORTS** tab: it's your page, served from inside the container. Read the last line of the log. It should say `Uvicorn running on http://0.0.0.0:8000`; if it says `127.0.0.1`, nothing outside the container can reach it, so look at blank 3 again. Stop it with **Ctrl-C**.

Commit and push:

```
git add Dockerfile .dockerignore
git commit -m "Put the idea board in a container"
git push
```

3. Deploy it on Railway (~15 min)
Railway builds your image from your GitHub repo and runs it on its own machines. If you don't have a Railway account yet, set one up first, with the *Set up your Railway account* sheet on the Course Materials page (about 10 minutes): it also gives Railway access to this repo, and makes sure it can't cost you anything.

1. Sign in at https://railway.com with your GitHub account.
2. **New Project → GitHub repo → `e190bo-lab-06` → Deploy Now**. If your repo isn't listed, let Railway's GitHub app see it first: it's private. Railway finds the `Dockerfile` at the top of your repo, says `Using detected Dockerfile!` in the build log, builds the image, and starts it with `PORT` set to the port it chose.
3. On the new service: **Settings → Networking → Public Networking → Generate Domain**.
4. Open your `https://…up.railway.app` address on your phone, on cell data rather than wifi.

From now on, every push to `main` builds and deploys again. If a deploy fails, open that deploy's logs and read them from the bottom.

Check it from your codespace, in `src/`:

```
API_URL=https://your-app.up.railway.app python check.py
```

Two things to know:

- **Your ideas reset whenever Railway deploys again.** The app keeps its ideas in `app.db`, a file inside its container, and every deploy starts a new container from the image. That's fine here; a database that outlives the container is a bigger step.
- **Railway runs on your own account and credits.** Check what your plan includes, and keep the app up until your check-in is done.

What to submit on Canvas

The same three parts as Lab 5, from your `e190bo-lab-06` repo.

1. **A one-page executive summary.** Put your **live URL** at the top. Then: what you built, the design decision you'd defend, and how you debugged the first request that failed.
2. **Your code, as a zip.** The zip is built from your last commit on `main`, so anything you haven't committed isn't in it. This time, run it from the **repo root** (`cd ..` if you're in `src/`), so the zip has your `Dockerfile` and `.dockerignore` as well as `src/`:

   ```
   git switch main
   git status                  # nothing left uncommitted? commit it first
   git archive --format=zip --prefix=lab-06/ -o lab-06.zip HEAD src Dockerfile .dockerignore
   python -m zipfile -l lab-06.zip    # check what's inside
   ```

   Your database, `app.db`, isn't in it, because `.gitignore` keeps it out of your commits.
3. **Your git history**, as a file, also from the repo root:

   ```
   git log --graph --stat > lab-06-git-log.txt
   ```

   `--graph` draws your branches and merges down the left side; `--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 mainly from a **10-minute one-on-one check-in**, not from the submission. You demo your work and answer questions about it, without AI.

While you build, the only AI you may use is the Lab 6 tutor in Glassroom AI. No other AI: not a chatbot such as ChatGPT or Claude, not AI code completion in your editor (such as Copilot), and not an agent working in your codespace. At the check-in, no AI at all. The tutor won't write the lab for you, so build in a way that leaves you able to explain everything you shipped.

Come with your live URL open on your phone, and in your codespace your server running, the page open, and the developer tools open.

Check-in topics

You start by opening your live URL on your phone, then adding an idea and voting on it in your codespace. Then four of these five, and **Git is always one of them**:

- **Trace:** Walk a value from a form field to an API call and back into the DOM.
- **DOM:** Show me where you change the page, and explain when that code runs.
- **CSS:** Two rules are fighting over this element. Which wins, and why?
- **Live change:** Your API is about to return an error every time. Make the page handle it gracefully.
- **Git:** Show me your history as a graph. Where did you branch, where did you merge, and how did you resolve your conflict?

Getting help

- The Lab 6 tutor in Glassroom AI. It will not give you answers.
- Office hours
- Zulip, for concepts. Not for each other's code.