A Page That Talks to Your API

JavaScript, the DOM, and fetch

E190BO · AI-Amplified Prototyping

Five big ideas

1. **Your server serves the page too,** so the page and the API share one origin, and nothing needs permission.
2. **JavaScript changes the tree:** it runs once when the page loads, then again only when something happens.
3. **The page is one more client** of your API, like `curl` and `check.py`.
4. **A 500 is still a response.** `fetch` only fails when there's no response at all, so checking `response.ok` is your job.
5. **Sending is a form, some JSON, and `textContent`** for whatever comes back.

1 · Your server serves the page too

One more line in `main.py`

js-1-served/main.py

```python
from fastapi import FastAPI
from fastapi.staticfiles import StaticFiles
from sqlmodel import SQLModel

app = FastAPI()
ideas = [
    {"id": 1, "title": "Automatic plant waterer", "body": "", "vote_count": 2},
    {"id": 2, "title": "Sticker vending machine", "body": "", "vote_count": 3},
    {"id": 3, "title": "Bike lock that texts you", "body": "", "vote_count": 0},
]

class IdeaCreate(SQLModel):          # what a client may send: no id
    title: str
    body: str = ""

@app.get("/ideas")
def list_ideas():
    return ideas

@app.post("/ideas", status_code=201)
def create_idea(payload: IdeaCreate):
    idea = {"id": len(ideas) + 1, **payload.model_dump(), "vote_count": 0}
    ideas.append(idea)
    return idea

# Everything in web/, from this same server. Last: "/" matches every path.
app.mount("/", StaticFiles(directory="web", html=True), name="web")
```

Same server, same origin

`http://` `localhost` `:8000` `/ideas`

- An **origin** is scheme + host + port. The page and `/ideas` come from the same one.
- So the page asks with a **relative URL**: `fetch("/ideas")`, never `fetch("http://localhost:8000/ideas")`.
- The browser sends a relative URL back to wherever the page came from: your codespace in Lab 6, the internet in Lab 7. `localhost` in a browser means *the laptop the browser is on*.

▶ LIVE Run it

1. Terminal 1, in the folder: `uvicorn main:app --reload`.
2. Browser: `localhost:8000/`: last video's page. `/ideas`: the JSON. `/docs`: still there.
3. Terminal 1 logs every file the page needed: `GET /`, `GET /style.css`.

The mount goes last. FastAPI tries routes in order, and a mount at `/` matches every path. Above your routes, it answers first: `GET /ideas` is **404** `{"detail":"Not Found"}`, and `POST /ideas` is **405**.

If the page came from another origin

- Page from `http://localhost:8080`, asking `http://localhost:8000/ideas`: **two origins.**
- The server still answers **200**. The **browser** won't let the page read it, unless the response names the page's origin:

```
Access-Control-Allow-Origin: http://localhost:8080
```

- That opt-in is **CORS** (cross-origin resource sharing). Only browsers enforce it, so `curl`, `check.py` and `pytest` never see it.
- **In Lab 6 the page and the API share a server, so there's nothing to configure.** If the Console says *blocked by CORS policy*, your page is coming from somewhere other than your API.

2 · JavaScript changes the tree

Find, make, fill, put

js-2-dom/web/app.js

```
const ideas = [
  { id: 1, title: "Automatic plant waterer", vote_count: 2 },
  { id: 2, title: "Sticker vending machine", vote_count: 3 },
];

const list = document.querySelector("#ideas");      // find an element

for (const idea of ideas) {
  const li = document.createElement("li");          // make a new one
  li.textContent = `${idea.title} (${idea.vote_count} votes)`;
  list.append(li);                                  // put it in the page
}
```

- `querySelector` takes a CSS selector and gives you the element; `textContent` sets its text.
- `` `...${x}...` `` is a template string, like Python's f-string.

When your code runs

```
<script src="app.js" defer></script>   <!-- in <head> -->
<ul id="ideas"></ul>                   <!-- in <body>, and empty -->
```

1. **Once, when the page loads.** The browser builds the DOM from the HTML; then, because of `defer`, it runs `app.js` from top to bottom.
2. **Then only when something happens:** a click, a form being submitted, or an answer coming back from your API.

- Without `defer`, a script in `<head>` runs before `<body>` exists, so `querySelector("#ideas")` returns `null`. **Console**: `Uncaught TypeError: Cannot read properties of null (reading 'append')`.
- The page never redraws itself on its own. It shows what your code last put there.

3 · The page is one more client

`fetch`, and `await`

js-3-fetch/web/app.js

```
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();
```

- `await` pauses **this function**, which must be `async`, until the answer arrives; the rest of the page stays live.
- It's the same `GET /ideas` that `curl` and `check.py` send: your API can't tell them apart.

▶ LIVE Watch the request

1. Terminal 1, in the folder: `uvicorn main:app --reload`; open `localhost:8000`: three ideas.
2. DevTools → **Network**: tick **Disable cache**, and reload. Click the `ideas` row: status **200**; **Response** is the same JSON `curl` gets.
3. Terminal 1: `"GET /ideas HTTP/1.1" 200 OK`, after the page, `style.css` and `app.js`.

4 · A 500 is still a response

`fetch` only fails when nothing answers

What happened	`fetch`	`response.ok`	Your function
200, 201	resolves	`true`	carries on
404, 422, 500	**resolves**	`false`	**carries on, unless you check**
server down, network gone	**rejects**: `TypeError: Failed to fetch`	—	jumps to `catch`

- A 404 or a 500 *is* an answer, so to `fetch` it's a success.
- `response.ok` is `true` for any 2xx status. Checking it is your job.

▶ LIVE Break the API

1. Still `js-3-fetch`. In `main.py`, import `HTTPException`, and make the first line of `list_ideas`: `raise HTTPException(500)`
2. Save; `--reload` restarts the server. Reload the page: **an empty page, and nothing on it says why.**
3. **Console**: `Uncaught (in promise) TypeError: ideas is not iterable`.
4. **Network**: `ideas` is a red **500**; its Response is `{"detail":"Internal Server Error"}`.

Check ok, and say so on the page

js-4-ok/web/app.js

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

async function loadIdeas() {
  listStatus.textContent = "Loading ideas…";
  try {
    const response = await fetch("/ideas");
    if (!response.ok) {                          // a 404 or a 500 lands here
      throw new Error(`The server said ${response.status}.`);
    }
    const ideas = await response.json();
    list.replaceChildren();
    for (const idea of ideas) {
      const li = document.createElement("li");
      li.textContent = `${idea.title} (${idea.vote_count} votes)`;
      list.append(li);
    }
    listStatus.textContent = ideas.length ? "" : "No ideas yet.";
  } catch (err) {                                // ...and so does no response at all
    listStatus.textContent = `Couldn't load ideas. ${err.message} `;
    const retry = document.createElement("button");
    retry.textContent = "Try again";
    retry.addEventListener("click", loadIdeas);
    listStatus.append(retry);
  }
}

loadIdeas();
```

▶ LIVE Break it three ways

`index.html` adds `<p id="list-status" role="status"></p>` above the list.

1. In `js-4-ok`, add the same `raise HTTPException(500)`, and reload the page: *Couldn’t load ideas. The server said 500.* [**Try again**]
2. Terminal 1: **Ctrl-C**. Click **Try again**: *Couldn’t load ideas. Failed to fetch* [**Try again**].
3. Delete the `raise`, start the server again, and click **Try again**: the ideas are back, no page reload.
4. In `list_ideas`, add `time.sleep(2)` (and `import time`): *Loading ideas…* for two seconds.

5 · Sending: a form, JSON, and `textContent`

Handle the form's `submit`

```
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}.`;
  }
});
```

input → `JSON.stringify` → `POST` → `IdeaCreate` → 201 → `loadIdeas` → `GET` → `textContent`

demos/js-5-form/web/app.js, below js-4-ok's `loadIdeas`; the form is the first video's, plus `id="add-form"`

`textContent`, never `innerHTML`, for data

```
li.textContent = idea.title;
```

The title is shown as **text**, exactly as typed.

```
li.innerHTML = idea.title;
```

The title is read as **HTML**: if it contains tags, the browser builds them.

Someone adds an idea titled:

```
<img src=x onerror=alert(1)>
```

- With `innerHTML`, every visitor's browser builds that image, the image fails to load, and `onerror` **runs their code**.
- That's **cross-site scripting** (XSS). Anything that came from your API goes in with `textContent`.

▶ LIVE Add an idea, and a hostile title

1. Terminal 1, in `js-5-form`: `uvicorn main:app --reload`; open the page with **Network** open.
2. Add `Desk fan that follows you`: *Added.*, and it's fourth. `POST ideas` → **Payload** `{"title":"Desk fan that follows you"}`; **Response 201** `{"id":4,…,"vote_count":0}`.
3. Add `<img src=x onerror=alert(1)>`: it shows up as those characters.
4. Change `textContent` to `innerHTML` in the loop, reload: **an alert pops up.** Change it back.

What’s yours, and Lab 6

Lab 6 asks for	Chapter
The mount line, last; `fetch("/ideas")`, never a host	1
List every idea with its `vote_count`; add an idea from a form	2, 3, 5
Honest about the network: loading, error and empty states, a way to retry, and a failed add that keeps what was typed	4, 5
`textContent` for anything from your API	5
New: voting, with a `voter`	all of it, on the `POST /ideas/{id}/vote` from your contract

Check-in topics: **Trace** (chapters 3 and 5), **DOM** (2), **Live change** (4).

Plain JavaScript, on purpose: a framework like React would hide the DOM the check-in asks about.