

Homework — Modern Web Stack

Estimated time: ~45 min · No code required · Just your browser and its dev tools

Three short tool-tour exercises. The goal: take the abstract picture from the theory page (browser ↔ API ↔ database, static vs. dynamic, frontend / backend / database as three separable jobs) and see *it yourself* in real sites you already use. Open your browser's developer tools, click around, and notice the three tiers in the wild. Take a screenshot at each step.

Before you start

All three tasks work in any modern browser (Chrome, Edge, Firefox, Safari). Open developer tools with **F12** (Windows/Linux) or **Cmd + Option + I** (macOS), or right-click any element and choose *Inspect*. You're only looking — you won't change anything on these sites.

Task 1 — See the 3-tier model in action

A real production app is constantly talking to its backend. We'll watch that traffic happen live and notice that the calls look nothing like the URL in the address bar.

1. Open notion.so (sign in, or use any public Notion page somebody has shared with you).
2. Right-click anywhere on the page → **Inspect**. In the dev-tools panel that opens, click the **Network** tab.
3. Reload the page (**Ctrl/Cmd + R**). Watch requests stream in. Wait a few seconds for the traffic to settle.
4. Now click anywhere on the page — open a sub-page, expand a toggle, anything interactive. *New* rows will appear in the Network tab as the frontend

asks the backend for things.

- Find a request whose path starts with `/api/`. Click it. In the side panel, open the **Headers** tab and note the *Request URL*. Then open the **Response** tab and note that the body is **JSON**, not HTML.
- Take a screenshot of the Network tab with the `/api/` row selected and the JSON response visible.

✓ **Self-check:** you can answer (a) is the URL of an `/api/` request the same as the URL in your browser's address bar? (*no — the page URL is something like `notion.so/...`, the API URL starts with `/api/` and goes to a different backend path*); (b) what **format** is the response in? (**JSON**); (c) approximately **how many** `/api/` calls did one click generate? — you now have a number, even if it's "lots". You've just watched the frontend tier (the React app in your browser) talk to the backend tier (Notion's servers) over HTTP, exchanging JSON instead of HTML.

Task 2 — Tell a static site from a dynamic one

Not every site does what Notion does. Some ship a fully-rendered HTML page and call it done; others ship a near-empty shell and let JavaScript fill it in. View Source is the cheapest way to tell them apart.

- Open three sites in three tabs:
 - news.ycombinator.com
 - github.com (your own profile if you have an account, otherwise github.com/torvalds)
 - astro.build
- In each tab, right-click on the page background → **View Page Source** (or **Ctrl/Cmd + U**). This shows you the raw HTML the server sent *before* any JavaScript ran.
- For each site, scan the source and decide: is the visible page content (story titles, profile name, marketing copy) **already in the HTML**, or is the `<body>`

mostly empty with one big `<div id="root"></div>` and a stack of `<script>` tags?

4. Write a one-line verdict per site: "*server-rendered / static*" vs. "*client-rendered SPA*".

✓ **Self-check:** Hacker News and the Astro marketing site are largely **already rendered** — the story titles and marketing copy are right there in the HTML the server sent. Many SPAs (and parts of GitHub) ship a near-empty `<body>` and let JavaScript populate it after load. You can now classify any site you visit by this same trick: *view source → is the content there, or not?* If "not", you're looking at a single-page app that's about to make a bunch of `/api/` calls just like Notion did in Task 1.

Task 3 — Read a job posting through this lens

The 3-tier picture isn't just an academic model — it's literally how teams are organised and how job postings are written. Once you see it, you can't un-see it.

1. Open [linkedin.com/jobs](https://www.linkedin.com/jobs) or [indeed.com](https://www.indeed.com). Search for "**Full-stack Engineer**".
2. Open three different listings from three different companies.
3. In each posting, scroll to the *Requirements*, *Tech stack*, or *You'll work with* section.
4. For each posting, sort the named technologies into three columns:
 - **Frontend:** React, Vue, Svelte, Angular, HTML/CSS, TypeScript on the browser side, Tailwind, Next.js, ...
 - **Backend:** Python, Django, FastAPI, Node.js, Express, Go, Java, Spring, Ruby on Rails, .NET, ...
 - **Database:** PostgreSQL, MySQL, MongoDB, Redis, DynamoDB, ...
5. Note which tier each company seems **most heavy on** — some postings are 80% frontend, others 80% backend, "full-stack" sometimes leans one way more than the other.

6. Park anything that doesn't fit those three columns (e.g., *Docker*, *Kubernetes*, *AWS*, *Terraform*, *GitHub Actions*). Lessons 06 (architecture) and the cloud lesson you've already done cover those.

✓ **Self-check:** for each of your three postings you have a three-column breakdown; you can name which tier each posting leans on most heavily; the words "full-stack" no longer feel mysterious — you can read a job ad and immediately see *which slice of the 3-tier stack the role actually lives in*.

What "done" looks like

You finish this homework when you have, in your notes:

- A screenshot of the Notion **Network** tab with at least one `/api/` request open and its JSON response visible — plus your own answers to the three Task 1 questions (URL difference, response format, approximate call count).
- A one-line verdict per site (Hacker News / GitHub / Astro) on whether it's server-rendered or a client-rendered SPA — based on what you saw in *View Source*.
- Three Frontend / Backend / Database breakdowns — one per job posting — plus a one-line note on which tier each company leans on hardest.

You've now seen the picture from the theory page in three separate places: live in the dev tools (Task 1), in the source of static vs. dynamic sites (Task 2), and in how the industry hires for it (Task 3). Next lesson we move down one level — what the act of *writing* code actually looks like in 2026.