The Hypermedia Stack

Build one real app start to finish on Astro + Datastar + Directus — a hypermedia-first stack with a CMS backend, SSR endpoints that stream, and no SPA framework in sight.

Roger Stringer · rogerstringer.com

June 16, 2026

Contents

Why Hypermedia, Why Now

Somewhere along the way, "build a web app" came to mean "stand up a JSON API, then build a second application in JavaScript to consume it." Two codebases, two mental models, a build step that rivals the app itself, and a client that re-implements routing, caching, and state the browser already does for free. For a Figma or a Google Sheets, that complexity earns its keep. For the other 90% of what we build — dashboards, forms, content sites, internal tools — it's a tax you pay every day and rarely needed to.

Hypermedia is the older idea the industry is rediscovering. The server sends HTML. The browser renders it. When something changes, the server sends *more* HTML, and you swap it in. No client-side data layer, no serialization boundary, no "now re-derive the UI from this JSON." The network speaks the same language as the page. HTMX made this respectable again; Datastar takes it further, adding fine-grained reactivity and real-time streaming without making you leave the model.

The objection is always the same: *but you can't build rich, real-time interfaces this way.* You can. That's the entire point of this guide. We're going to build a genuinely interactive app — live updates, streaming, the works — and there won't be a single-page-app framework anywhere in it.

What you get back for dropping the SPA is substantial: one codebase and one mental model, HTML you can read in View Source, no hydration bugs, no client/server state drift, and a stack one developer can hold in their head. The page is the source of truth, the way the web was designed.

This guide is the capstone for three I've written separately — Datastar (https://rogerstringer.com/guides/guide-to-datastar-with-astro), Directus (https://rogerstringer.com/guides/mastering-directus), and Astro running underneath them both. Each is useful on its own. Together they're a complete way to build software. By the end you'll have shipped a real app on all three, and you'll understand *why* each piece is there. Let's meet them.

Meet the Stack

Three pieces, three jobs, one clean seam between each. Here's the division of labour before we write a line of code.

Directus — the data and the API. Directus wraps your database (we'll use Postgres) in an instant REST and GraphQL API, an admin UI for modeling and editing content, an auth and permissions system, and — the part that matters most for us — a realtime WebSocket layer. Model your collections once and you get the API, the admin, the auth, and live subscriptions for free. We're not building a backend; we're configuring one. (The deep version is Mastering Directus (https://rogerstringer.com/guides/mastering-directus).)

Astro — the server and the rendering. Astro is our web framework, running in SSR mode. It owns routing, renders HTML on the server, and hosts the handful of server endpoints our app needs — the ones that talk to Directus and stream updates back to the browser. Astro is where *our* code lives. It's the conductor between the browser and Directus.

Datastar — the reactivity. Datastar is a small script you drop in with a `<script>` tag. It gives plain HTML reactive superpowers through `data-*` attributes: bind inputs to signals, show and hide things, fire requests, and — crucially — receive streamed HTML from the server and patch it into the page over Server-Sent Events. No build step, no virtual DOM, no components. (Deep version: Guide to Datastar with Astro (https://rogerstringer.com/guides/guide-to-datastar-with-astro).)

Now trace one interaction through all three. The browser asks Astro for a page. Astro calls Directus with the SDK, gets the data, renders HTML, sends it down. A user clicks "upvote." Datastar fires a request to an Astro endpoint. Astro tells Directus to increment the vote. Directus's realtime layer notices the change and pushes it to Astro's open SSE stream, which forwards an HTML patch to *every* connected browser. Everyone's screen updates. No JSON ever crossed into client code; no client state needed reconciling.

That round trip — browser !Astro !Directus !back out to every browser — is the whole architecture. Everything left in this guide is building it.

What We're Building

We're building **Soapbox** — a live Q&A board for talks, AMAs, and all-hands. You've used something like it: the audience posts questions, everyone upvotes the ones they care about, the best float to the top in real time, and the host marks them answered as they go. It's small enough to finish and real enough to be worth finishing — and it exercises every part of the stack.

Here's the spec, in the spirit of the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide — behaviour first, no tech:

```
# Soapbox — live Q&A board
- A session has a title and a list of questions.
- Anyone can post a question (with an optional name) — no login.
- Anyone can upvote a question; the list sorts by votes, highest first.
- New questions and new votes appear live for everyone, no refresh.
- The host (logged in) can mark a question answered or delete it.
- A closed session is read-only.
- Out of scope: threaded replies, multiple rooms per host, moderation queues.
```

That one page touches all of it. **Reading** the list is Directus + the SDK + Astro rendering (chapters 5–7). **Posting and upvoting** are forms and mutations writing back through Astro (chapter 10). **The live updates** — the marquee feature — are Datastar reactivity over an SSE stream fed by Directus realtime (chapters 8–9). **The host's powers** are auth and permissions (chapter 11). Then we ship it (chapter 12).

A word on scope, because it's the most important engineering decision in any build-along: Soapbox is deliberately small. Every feature above maps to a stack capability worth teaching, and nothing's there just to look impressive. When we hit a spot where the simple version has a real-world flaw — and we will, around vote counting — I'll show you the flaw, ship the simple version anyway, and tell you exactly what you'd do differently at scale in the final chapter. That's the honest way to learn a stack: build the small thing correctly, and know where its edges are.

Next, we get all three running.

Project Setup

Let's get all three running locally. I'll use pnpm; npm works the same.

Astro, in SSR mode. Scaffold a minimal project and add a server adapter so Astro renders on each request and can host endpoints:

```
pnpm create astro@latest soapbox   # choose: Empty, TypeScript (strict)
cd soapbox
pnpm astro add node                 # the SSR adapter
pnpm add @directus/sdk
```

Tell Astro to render on the server, in `astro.config.mjs`:

```
import { defineConfig } from "astro/config";
import node from "@astrojs/node";

export default defineConfig({
  output: "server",
  adapter: node({ mode: "standalone" }),
});
```

Directus, in Docker. Drop a `docker-compose.yml` beside the project — Postgres plus Directus, the same shape as the Mastering Directus (https://rogerstringer.com/guides/mastering-directus) guide:

```
services:
  database:
    image: postgres:16
    environment:
      POSTGRES_USER: directus
      POSTGRES_PASSWORD: directus
      POSTGRES_DB: directus
    volumes: ["./data/db:/var/lib/postgresql/data"]

  directus:
    image: directus/directus:11
    ports: ["8055:8055"]
    depends_on: [database]
    environment:
      KEY: replace-me
      SECRET: replace-me
      DB_CLIENT: pg
      DB_HOST: database
      DB_USER: directus
      DB_PASSWORD: directus
      DB_DATABASE: directus
      ADMIN_EMAIL: admin@example.com
      ADMIN_PASSWORD: admin
      WEBSOCKETS_ENABLED: "true"
```

`docker compose up -d`, and Directus is at `http://localhost:8055`. The one line that's easy to skip and we'll need badly: `WEBSOCKETS_ENABLED: "true"` — that's the realtime layer powering our live updates. (Pin the image tags to current releases; these are illustrative.)

Wire them together with an `.env` in the Astro project:

```
PUBLIC_DIRECTUS_URL=http://localhost:8055
DIRECTUS_TOKEN=          # we'll fill this in the auth chapter
```

Datastar is just a script. We'll add it to the base layout in chapter 7, but here's the tag:

```
<script type="module"
```

```
  src="https://cdn.jsdelivr.net/gh/starfederation/datastar@v1.0.0/bundles/datastar.js"></
script>
```

That's the entire toolchain: `pnpm dev` serves Astro on :4321, Docker serves Directus on :8055, a CDN serves Datastar. No bundler config, no client framework, no API gateway — three processes, one of which is a script tag. Now let's give Directus something to serve.

Modeling the Data in Directus

Open `http://localhost:8055`, log in with the admin credentials from the compose file, and let's model Soapbox. Two collections.

`sessions` — a Q&A room. Create the collection with a UUID primary key and add:

- `title` — string, required
- `slug` — string (we'll route on this)
- `status` — dropdown: `open` / `closed`, default `open`

`questions` — the heart of it. UUID primary key, plus:

- `body` — text, required
- `author_name` — string, nullable (questions can be anonymous)
- `votes` — integer, default `0`
- `answered` — boolean, default `false`
- `session` — a Many-to-One relation to `sessions`
- `date_created` — the built-in "Created On" field (add it from the collection's optional system fields)

The relation is the only fiddly part, and Directus makes it a menu choice: on `questions`, add a field of type *Many to One* and point it at `sessions`. Directus creates the foreign key, and on the `sessions` side you can add the matching *One to Many* (`questions`) so a session shows its questions. The full tour of relations is in Mastering Directus (https://rogerstringer.com/guides/mastering-directus); for now the menu defaults are exactly right.

Model it once and notice what you now have, for free: a REST endpoint at `/items/questions`, a GraphQL one, a filterable and sortable query layer, an admin UI where you can hand-edit questions while testing, and — because we flipped `WEBSOCKETS_ENABLED` on — a realtime subscription for the `questions` collection. We haven't written a line of backend code and the backend is done.

Create one `sessions` row by hand — title "Launch AMA", slug `launch`, status `open` — and a couple of questions under it, so there's something to render in the next chapter.

A quick word on that `votes` integer. Storing a counter directly is the simple choice, and the one we'll build on — it keeps the whole upvote flow to a single field. It also can't tell you *who* voted, and two votes landing in the same millisecond can race. That's a real limitation, not an oversight; we'll name it precisely and show the grown-up version in the final chapter. For a live Q&A where approximate counts are fine, the integer is the right amount of engineering.

Fetching with the Directus SDK

All our data access goes through one small module. Create `src/lib/directus.ts`:

```
import {
  createDirectus, rest, readItems, createItem, updateItem, deleteItem,
} from "@directus/sdk";

// A typed schema keeps the SDK honest about our collections.
export type Question = {
  id: string;
  body: string;
  author_name: string | null;
  votes: number;
  answered: boolean;
  session: string;
  date_created: string;
};
export type Session = {
  id: string; title: string; slug: string; status: "open" | "closed";
};
type Schema = { questions: Question[]; sessions: Session[] };

export const directus = createDirectus<Schema>(
  import.meta.env.PUBLIC_DIRECTUS_URL
).with(rest());

export { readItems, createItem, updateItem, deleteItem };
```

That generic `<Schema>` does quiet work: `readItems("questions", …)` now returns typed `Questions`, and a typo in a field name is a compile error rather than a 2am surprise. It's the [context-engineering](https://rogerstringer.com/guides/context-engineering-the-craft-of-agents-md) principle applied to data — encode the shape once, get caught early forever.

Now a helper to load a session and its questions, sorted the way Soapbox wants — unanswered first, then by votes. Create `src/lib/board.ts`:

```
import { directus, readItems } from "./directus";

export async function getBoard(slug: string) {
  const [session] = await directus.request(
    readItems("sessions", { filter: { slug: { _eq: slug } }, limit: 1 })
  );
  if (!session) return null;

  const questions = await directus.request(
    readItems("questions", {
      filter: { session: { _eq: session.id } },
      sort: ["answered", "-votes", "-date_created"],
      limit: 200,
    })
  );
  return { session, questions };
}
```

Everything here runs on the server, inside Astro — the Directus URL and (soon) token never reach the browser. The SDK's `filter`/`sort` query language is the same one you'd use over raw REST; if you've read the Directus guide it'll look familiar. One module, fully typed, and the rest of the app just calls `getBoard()`. Next we turn that data into a page.

Rendering with Astro

Time to see Soapbox in a browser. Astro routes by file, so the board lives at `src/pages/[slug].astro` — the `[slug]` becomes our session slug.

Start with a layout, `src/layouts/Base.astro`, which is where the Datastar script lives so every page gets it:

```
---
const { title } = Astro.props;
---
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>{title}</title>
    <script type="module"
      src="https://cdn.jsdelivr.net/gh/starfederation/datastar@v1.0.0/bundles/
datastar.js"></script>
  </head>
  <body><slot /></body>
</html>
```

Then the board page. It loads data on the server and renders it — plain HTML, no client JS yet:

```
---
import Base from "../layouts/Base.astro";
import Question from "../components/Question.astro";
import { getBoard } from "../lib/board";

const board = await getBoard(Astro.params.slug!);
if (!board) return Astro.redirect("/404");
const { session, questions } = board;
---
<Base title={session.title}>
  <main>
    <h1>{session.title}</h1>
    <ul id="questions">
      {questions.map((q) => <Question q={q} />)}
    </ul>
  </main>
</Base>
```

And the component for a single question, `src/components/Question.astro`:

```
---
import type { Question } from "../lib/directus";
const { q } = Astro.props as { q: Question };
---
<li id={`q-${q.id}`} class={q.answered ? "answered" : ""}>
  <button>%{q.votes}</button>
  <p>{q.body}</p>
  <small>{q.author_name ?? "Anonymous"}</small>
</li>
```

Load `http://localhost:4321/launch` and there's your board — server-rendered, fully readable in View Source, working even with JavaScript disabled. That `id="questions"` on the `<ul>` and the `id="q-…"` on each item aren't decoration: they're the targets Datastar will patch when updates stream in. Hold onto that idea — **stable IDs are the seam** between server-rendered HTML and live updates. Everything reactive we add from here hangs off those handles. Speaking of which: let's make it move.

Adding Reactivity with Datastar

Datastar adds behaviour to the HTML we already have, through `data-*` attributes. No components to mount, no state to initialize — you annotate the markup and it comes alive. Let's start with the ask box.

Signals are Datastar's reactive variables. Declare them with `data-signals` and bind an input to one with `data-bind`:

```
<form data-signals="{ body: '', name: '' }">
  <textarea data-bind="body" placeholder="Ask a question…"></textarea>
  <input data-bind="name" placeholder="Your name (optional)" />
  <button
    data-on-click="@post('/api/ask', { contentType: 'form' })"
    data-attr-disabled="$body.length === 0">
    Ask
  </button>
</form>
```

Three attributes, and you already have more than vanilla forms give you. `data-bind` keeps `$body` and `$name` in sync with the fields. `data-attr-disabled="$body.length === 0"` disables the button reactively until there's something to ask — a live expression, re-evaluated as you type, with no event handler written. `data-on-click="@post('/api/ask')"` fires a request to an Astro endpoint, sending the signals along.

The upvote button is the same idea, one attribute on the markup from the last chapter:

```
<button data-on-click={`@post('/api/vote/${q.id}')`}>%{q.votes}</button>
```

Here's the part that makes Datastar more than "HTMX with extra steps." When that `@post` returns, it doesn't hand you JSON to deal with — it returns *Server-Sent Events*, and Datastar applies them for you. The endpoint can reply with new HTML to patch into the page (a `datastar-patch-elements` event) or new signal values (`datastar-patch-signals`), and Datastar finds the target by `id` and swaps it. Your click handler and your server response speak HTML the whole way through.

Which means the interesting half of "reactivity" actually lives on the server: the endpoints that receive these requests and stream back patches. And because those patches travel over a *stream*, the very same mechanism that answers your click can push an update that someone *else's* click caused. That's the next two chapters — first the stream that makes the board live, then the endpoints that write to it.

SSE Endpoints That Stream

This is the chapter that earns the whole stack. We're going to make the board update live — someone in row 12 upvotes a question and your screen reorders itself — with no client-side state management anywhere.

The shape: the browser opens one long-lived connection to an Astro endpoint. That endpoint listens to Directus's realtime layer, and every time a question changes, it streams a fresh copy of the list back as a Datastar patch. Datastar swaps it into `#questions`. That's the entire mechanism.

First, a function that renders the list to an HTML string — we'll call it from the stream:

```
// src/lib/render.ts
import type { Question } from "./directus";

const esc = (s: string) =>
  s.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;");

const row = (q: Question) => `
  <li id="q-${q.id}" class="${q.answered ? "answered" : ""}">
    <button data-on-click="@post('/api/vote/${q.id}')">%${q.votes}</button>
    <p>${esc(q.body)}</p>
    <small>${esc(q.author_name ?? "Anonymous")}</small>
  </li>`;

export const renderQuestions = (questions: Question[]) =>
  `<ul id="questions">${questions.map(row).join("")}</ul>`;
```

That `esc` is not optional — rendering user text into an HTML string without escaping is the XSS hole we'll underline in the final chapter.

Now the stream, at `src/pages/api/stream/[slug].ts`:

```
import type { APIRoute } from "astro";
import { createDirectus, realtime, staticToken } from "@directus/sdk";
import { getBoard } from "../../../lib/board";
import { renderQuestions } from "../../../lib/render";

// Datastar's wire format: one SSE event that patches elements by id.
const patch = (html: string) =>
  `event: datastar-patch-elements\ndata: elements ${html.replace(/\n/g, "")}\n\n`;

export const GET: APIRoute = async ({ params }) => {
  const slug = params.slug!;
  const rt = createDirectus(import.meta.env.PUBLIC_DIRECTUS_URL)
    .with(staticToken(import.meta.env.DIRECTUS_TOKEN))
    .with(realtime());

  const stream = new ReadableStream({
    async start(controller) {
      const enc = new TextEncoder();
      const send = (html: string) => controller.enqueue(enc.encode(patch(html)));

      // 1. Push the current board immediately.
      const board = await getBoard(slug);
      if (board) send(renderQuestions(board.questions));

      // 2. Re-push whenever any question changes.
      await rt.connect();
      const { subscription } = await rt.subscribe("questions", {
        query: { fields: ["id"] }, // fires on create, update (votes!), and delete
      });
      for await (const _ of subscription) {
        const next = await getBoard(slug);
        if (next) send(renderQuestions(next.questions));
      }
    },
  });

  return new Response(stream, {
    headers: {
      "Content-Type": "text/event-stream",
      "Cache-Control": "no-cache",
      Connection: "keep-alive",
```

```
    },
  });
};
```

The browser opens the stream with one attribute on the board — `data-on-load` fires the GET and Datastar keeps it open:

```
<main data-on-load="@get('/api/stream/launch')">
```

That's the live board. Trace it once, concretely: the page loads, `data-on-load` opens the stream, the endpoint sends the current list, then parks on the Directus subscription. Anyone, anywhere, creates or upvotes a question — Directus fires a realtime event — our loop re-queries and streams a new `#questions` — Datastar patches every connected board. The clients hold no state; the server-rendered HTML *is* the state, re-sent whenever it changes.

Two honest notes, banked for later chapters: re-querying and re-sending the *whole* list on every change is delightfully simple and totally fine for a Q&A board — at large scale you'd patch the single row that changed (chapter 13). And long-lived SSE connections have real deployment quirks — buffering, idle timeouts — that we handle in chapter 12. For now: it's live.

Forms & Mutations

Now the writes. Two endpoints, both tiny, both ending the same way: change Directus, then let the stream do the talking.

Asking a question — `src/pages/api/ask.ts`. Datastar sent our `body` and `name` signals as form data:

```
import type { APIRoute } from "astro";
import { directus, createItem, readItems } from "../../lib/directus";
import { staticToken } from "@directus/sdk";

const client = () => directus.with(staticToken(import.meta.env.DIRECTUS_TOKEN));

export const POST: APIRoute = async ({ request }) => {
  const form = await request.formData();
  const body = String(form.get("body") ?? "").trim();
  if (!body) return new Response("empty", { status: 422 });

  const [session] = await client().request(
    readItems("sessions", { filter: { slug: { _eq: "launch" } }, limit: 1 })
  );
  await client().request(
    createItem("questions", {
      body,
      author_name: String(form.get("name") ?? "") || null,
      session: session.id,
      votes: 0,
    })
  );

  // Clear the form via a signal patch. The new question arrives on the stream.
  return new Response(
    `event: datastar-patch-signals\ndata: signals { body: '', name: '' }\n\n`,
    { headers: { "Content-Type": "text/event-stream" } }
  );
};
```

Notice what the response does *not* do: it doesn't return the new question. It doesn't need to. Creating the row trips Directus realtime, which trips the stream from the last chapter, which patches the list on every screen — including the asker's. The endpoint's only job for *this* client is to reset the form, which it does with a `datastar-patch-signals` event. One source of truth, one code path, no "optimistically add it here and also re-fetch it there."

Upvoting — `src/pages/api/vote/[id].ts`:

```
import type { APIRoute } from "astro";
import { directus, readItems, updateItem } from "../../../lib/directus";
import { staticToken } from "@directus/sdk";

export const POST: APIRoute = async ({ params }) => {
  const client = directus.with(staticToken(import.meta.env.DIRECTUS_TOKEN));
  const [q] = await client.request(
    readItems("questions", { filter: { id: { _eq: params.id } }, limit: 1 })
  );
  if (q) await client.request(updateItem("questions", q.id, { votes: q.votes + 1 }));
  return new Response(null, { status: 204 }); // the stream handles the UI
};
```

That read-then-write is the race condition I keep promising to point at: two upvotes landing together can both read `votes: 4` and both write `5`, losing one. For a Q&A board, occasionally undercounting by one is invisible — but it's exactly the kind of thing the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) verify step exists to catch, and chapter 13 shows the atomic version. We ship the simple one knowingly.

Step back and feel the shape: every mutation is *write to Directus, return (almost) nothing, let the stream broadcast the truth.* No optimistic UI to reconcile, no cache to invalidate, no JSON round-tripping into client state. The hard problem SPAs pour enormous effort into — keeping client state in sync with the

server — we deleted by never having client state in the first place.

Auth & Access Control

So far one omnipotent admin token does everything, which is fine on localhost and unacceptable in public. Let's split Soapbox into the three trust levels it actually has, using Directus permissions — no auth code of our own.

The public (anonymous visitors). In Directus, the built-in **Public** policy controls what an unauthenticated request can do. Grant it exactly:

- `sessions`: **read**
- `questions`: **read**, and **create** (field-limited — only `body`, `author_name`, `session`; never `answered` or `votes`)

That's the whole public surface — and notice we did *not* grant `update`. So how do anonymous upvotes work? They don't go direct. They go through our Astro endpoint, which holds a scoped token. The browser can never write `votes` itself; it can only ask our server to, and our server decides the rules. The permission boundary lives on the server, exactly where it belongs.

The server (our Astro endpoints). Create a dedicated Directus user — call it `soapbox-server` — in a role that can create questions and update `votes` and `answered`, and nothing else. Generate a static token for it and put it in `.env`:

```
DIRECTUS_TOKEN=the-soapbox-server-token
```

That's the token every endpoint we've written already reaches for. It's scoped: if the server is compromised it can mess with questions, not nuke your database. Least privilege — the same principle the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) and Fleet guides keep hammering: a token per job, scoped to the job.

The host (you, logged in). Marking answered and deleting are host powers. Directus has its own auth — log the host in against `/auth/login`, keep the session, and have Astro check it before calling Directus with host privileges:

```
// src/pages/api/answer/[id].ts
export const POST: APIRoute = async ({ params, locals }) => {
  if (!locals.isHost) return new Response("forbidden", { status: 403 });
  await hostClient().request(updateItem("questions", params.id!, { answered: true }));
  return new Response(null, { status: 204 }); // stream repaints, answered sinks down
};
```

And the host's buttons only render for the host — a tiny bit of middleware sets `Astro.locals.isHost` from the session cookie, so the markup simply isn't sent to anonymous visitors:

```
{Astro.locals.isHost && (
  <button data-on-click={`@post('/api/answer/${q.id}')`}>Mark answered</button>
)}
```

Three tiers, all enforced by Directus policies plus one cookie check, zero bespoke auth framework. Public reads and asks; the server token does the controlled writes; the host, behind a login, gets the sharp tools. And when the host marks a question answered, that's just another `questions` change — so it rides the same stream and sinks down everyone's board in real time, for free.

Deploying the Whole Thing

Three processes locally; the same three in production, plus a few realities that long-lived streams force you to take seriously.

Directus goes up the way the Mastering Directus (https://rogerstringer.com/guides/mastering-directus) guide covers in depth — the official image against managed Postgres, on Railway, Fly, or your own box. The one Soapbox-specific must: keep `WEBSOCKETS_ENABLED=true`, and make sure your host (and any proxy in front of Directus) allows WebSocket upgrades — or realtime dies silently and the board just stops updating.

Astro builds to a Node server (we chose the standalone adapter back in setup):

```
pnpm build && node ./dist/server/entry.mjs
```

Wrap that in a small Dockerfile or point Railway/Render at the repo. Set the same env vars — `PUBLIC_DIRECTUS_URL` now your production Directus, `DIRECTUS_TOKEN` the scoped server token from the last chapter (set it as a secret; never commit it).

The SSE reality check. A streaming endpoint is a connection that's *supposed* never to end, and a lot of infrastructure assumes connections are short. The three things that will bite you, and the fixes:

- **Proxy buffering.** Nginx and friends buffer responses by default, which holds your events hostage until the buffer fills — so updates arrive in clumps, or not at all. Disable it for the stream route; `X-Accel-Buffering: no` as a response header is the portable nudge.
- **Idle timeouts.** Proxies and platforms kill "idle" connections after 30–60 seconds, and a quiet Q&A board looks idle. Send a heartbeat comment (`: keep-alive\n\n`) every 20 seconds to keep the pipe warm; Datastar ignores it.
- **Connection ceilings.** Each viewer holds one open connection and one Directus subscription. A single Node process handles plenty for a Q&A board, but it's a real ceiling — size for your expected concurrent viewers, and note that serverless platforms which cap request duration are the wrong home for this. A long-running Node process is the right one.

Fold the heartbeat and the buffering header into the stream endpoint from chapter 9 and it's production-shaped:

```
return new Response(stream, {
  headers: {
    "Content-Type": "text/event-stream",
    "Cache-Control": "no-cache",
    "X-Accel-Buffering": "no",
    Connection: "keep-alive",
  },
});
// inside start(), keep the connection warm:
// const hb = setInterval(() => controller.enqueue(enc.encode(": keep-alive\n\n")),
20000);
// clear it when the stream closes.
```

None of this is exotic — it's the standard tax on real-time, and it's a fixed, one-time cost. Pay it once and Soapbox runs the same in production as on your laptop: Directus holding the data, Astro rendering and streaming, Datastar patching the page.

Gotchas & What's Next

We built Soapbox the honest way — simple first, with the edges flagged as we passed them. Here they are in one place, each with its grown-up version, plus where to take the stack next.

The vote race. Our upvote reads `votes`, adds one, writes it back — two simultaneous votes can clobber each other. The fix is to make the increment atomic so the database does the addition: a small Directus Flow (or a custom endpoint running `UPDATE questions SET votes = votes + 1`) removes the race entirely. For real per-person voting — one vote each, no double-dipping — promote votes to their own collection (one row per person per question) and count rows, which kills the race as a side effect.

Escaping, always. We render user text into HTML strings in the stream, which is an XSS hole the instant `esc` slips. Anything user-supplied that becomes HTML must be escaped — no exceptions — and it's worth a line in your project's `AGENTS.md` so an agent never quietly drops it. (That's the context-engineering (https://rogerstringer.com/guides/context-engineering-the-craft-of-agents-md) habit in action.)

Re-rendering the whole list. Every change re-sends every question. Beautifully simple, and fine into the hundreds. Past that, send a patch for the *one* row that changed — Datastar happily patches a single `#q-123` — and re-sort only when needed. Reach for it when the list gets long, not before.

Reconnection. Networks drop. SSE auto-reconnects in the browser, and because our stream sends the full current list on connect, a reconnect heals itself — the board is correct again the moment the pipe reopens. That falls out of the "server-rendered HTML is the state" design for free; it's worth appreciating how much misery that one decision saved.

When *not* to reach for this stack. If your UI is genuinely application-like — a canvas editor, a spreadsheet, an offline-first app with heavy local state — the SPA complexity you'd be avoiding is complexity you actually need; reach for the heavier client. Hypermedia wins for the vast middle: content, dashboards, forms, feeds, anything where the server can stay the source of truth. Soapbox sits squarely in that middle, and so does most of what most of us build.

Where to take it next. Multiple sessions per host (you modeled `sessions` for exactly this). Question search via Directus's filter layer. A host dashboard. Rate-limiting the ask endpoint — the rate-limiting feature from the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide drops straight in. Each is an afternoon, because the foundation — model in Directus, render in Astro, stream with Datastar — doesn't change.

That's the whole stack, on one real app. You modeled data without writing a backend, rendered it without a client framework, and made it live without a single line of client-side state management. Three pieces, one mental model, HTML from end to end.

If you want each layer on its own: Mastering Directus (https://rogerstringer.com/guides/mastering-directus) for the data, Guide to Datastar with Astro (https://rogerstringer.com/guides/guide-to-datastar-with-astro) for the reactivity, and the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) for the discipline that keeps a build like this honest. Now go build something that didn't need a SPA.

About Roger

I'm Roger Stringer — I build things, break them, and write up what I learned so you don't have to learn it the hard way. These Field Guides come straight out of that work.

Working on something bigger? I take on a handful of fractional CTO (https://rogerstringer.com/guides/the-fractional-cto-field-guide) engagements — helping founders and teams set technical direction, build AI-powered workflows, and actually ship the hard parts. If you're wrestling with the kind of problem this guide covers and want someone in your corner who's done it before, that's exactly what I help with. Drop me a line. (mailto:roger.stringer@hey.com)

And if a guide helped, got something wrong, or you just want to compare notes, I'd love to hear from you:

- **Email** — roger.stringer@hey.com (mailto:roger.stringer@hey.com)
- **X** — @freekrai (https://x.com/freekrai)
- **GitHub** — github.com/freekrai (https://github.com/freekrai)
- **LinkedIn** — linkedin.com/in/rogerstringer (https://www.linkedin.com/in/rogerstringer)

New guides go up as I hit problems worth documenting — follow along wherever suits you.