Food Tracker (React + Express)

A Complete 12-Chapter Full-Stack JavaScript Course

Topics covered:

One language, both ends · raw SQL via better-sqlite3 · barcode lookup & caching

Camera scanning · server-side validation · expiry alerts · native search

Marking items used · concurrent recipe lookup · shared Context state · deployment

Capstone: one complete, working app, chapter by chapter

Exercises: 36 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

One of four Food Tracker courses — see also FastAPI, Django,

and React + Firebase editions

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 12

Project Overview & Full-Stack JS Setup

Food Tracker (React + Express)

Chapter 1 · Project Overview & Full-Stack JS Setup

This is one of four courses building the exact same app in four genuinely different architectures — Food Tracker (FastAPI), Food Tracker (Django), and Food Tracker (React + Firebase) are its siblings. Every one of them scans a barcode, tracks a use-by date, and alerts you before something goes to waste. This course's own angle is the most straightforward of the four to describe, and one of the most common in real professional practice: **the same language, front and back.**

What the App Actually Does

Before any architecture talk, the shared spec every course in the quartet is building toward:

- **Scan a barcode** with a phone or webcam camera, look it up against **Open Food Facts** (free, open, no API key) to fetch the product's name and details automatically.
- **Record a use-by date** for the item, and see it flagged once it's **expiring soon**.
- **Keep a full history** of every item ever added — some still active with a real expiry date, some already marked used with no expiry date at all — searchable in real time as you type, so re-adding something you've bought before is fast.
- **Look up recipes** via **TheMealDB** (also free, no key) that use ingredients close to expiring.

A weekly meal planner is explicitly **out of scope** for all four courses — named future work, not something any of them will build.

Why "One Language, Both Ends" Is a Genuine Case, Not Just Convenience

Food Tracker (FastAPI) and Food Tracker (Django) both write their backend in Python and their frontend in JavaScript — a real, unavoidable context switch every time work crosses that boundary. This course removes that switch entirely: React on the client, Node running an Express server, both written in JavaScript. The practical payoffs are concrete, not just aesthetic:

- **One shared data format, natively.** JSON is JavaScript's own native object literal syntax on both sides — no serializing a Python dict into JSON on the way out and parsing it back into a dict on the way in. A JS object built on the server and a JS object consumed on the client are structurally the same kind of thing.

- **One package ecosystem.** `npm` serves both halves of the app — no separate `pip` / `requirements.txt` world to keep in sync with a separate JS toolchain.
- **One team, fewer context switches.** A developer fixing a bug that spans "what the API returns" and "how the UI renders it" can stay in one language the entire time.

The honest caveat, named here so it doesn't need repeating later: sharing a *language* doesn't automatically mean sharing *types* across the network boundary. Without extra tooling (shared TypeScript interfaces, a schema-validation library), the Express server and the React client can still silently drift out of sync about what shape a response actually has — this course builds without that extra layer, matching its own realistic scope, and names the tradeoff honestly rather than pretending "same language" solves it for free.

The Architecture Contrast, Precisely

COURSE	BACKEND	WHERE BUSINESS LOGIC LIVES
Food Tracker (FastAPI)	A FastAPI process you write, run, and deploy	Python code, running on a server you manage
Food Tracker (Django)	A Django process you write, run, and deploy	Python code, running on a server you manage
Food Tracker (React + Express)	A Node/Express process you write, run, and deploy	**JavaScript code, the same language as the frontend**, running on a server you manage
Food Tracker (React + Firebase)	No server process you write or deploy	Mostly Security Rules (configuration) plus a few Cloud Functions

> **THE ONE-SENTENCE VERSION OF THIS WHOLE COURSE**
>
> In the other three Food Tracker courses, the frontend and backend are written in two different languages (Python/JS) or the backend barely exists as code at all (Firebase). In this one, **the exact same language runs on both sides of the network boundary** — the genuine "full-stack JavaScript" case, with its real benefits (one data format, one package ecosystem) and its one honest limitation (no automatic type-sharing across that boundary without extra tooling this course doesn't add).

Scaffolding the Project

Two separate processes, kept in one repository: a Vite-powered React client, and a plain Express server.

```
# the React client
npm create vite@latest client -- --template react
cd client && npm install

# the Express server, in a sibling folder
cd ..
mkdir server && cd server
npm init -y
npm install express cors dotenv
```

A minimal server, confirming the setup works end to end before any real feature exists:

```
// server/index.js
import express from "express";
import cors from "cors";

const app = express();
app.use(cors());
app.use(express.json());

app.get("/api/health", (req, res) => {
  res.json({ status: "ok" });
});

const PORT = process.env.PORT || 3001;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

CORS IS NOT OPTIONAL DURING DEVELOPMENT

Vite's dev server runs on its own port (typically 5173); Express runs on its own (3001 here). From the browser's perspective, those are two different origins, and without `cors()` the browser blocks every request from the React app to the Express API outright. This is a genuinely common first-hour stumbling block for anyone new to a two-process full-stack setup — worth understanding now rather than debugging blind later.

AUTO-RESTART THE SERVER DURING DEVELOPMENT

`node --watch server/index.js` (built into modern Node, no extra dependency needed) restarts the Express process automatically on every file save — the same convenience Vite already gives the React side for free.

Where This Course Is Headed

Data modeling and Express API routes, barcode lookup, the camera-scanning React component (shared almost verbatim with Food Tracker (React + Firebase)), the add-item flow, expiry alerts, item history with live search, marking items used, recipe lookup, cross-cutting state management once every feature needs to talk to every other feature, deployment, and a capstone tying every chapter into one complete, working app.

Hands-On Exercises

EXERCISE 1

In one sentence, state this course's own core architectural claim. Then explain what specifically breaks (or doesn't break) that claim once network requests are involved, using this chapter's own honest caveat about type-sharing.

View solution

EXERCISE 2

Explain why the React dev server and the Express server being on different ports causes a real problem in the browser, and what cors() actually does about it.

View solution

EXERCISE 3

Using this chapter's own comparison table, explain how this course's "where business logic lives" column differs from both Python siblings' own column and from the Firebase sibling's own column.

View solution

CHAPTER 1 QUICK REFERENCE

- **The shared app** — barcode scan (Open Food Facts) → expiry tracking → alerts → searchable history → recipe lookup (TheMealDB); no meal planner
- **This course's own throughline:** the same language, JavaScript, runs on both the client and the server
- **Real payoff:** one native data format (JSON), one package ecosystem (npm)
- **Honest limit:** same language does not mean shared types across the network without extra tooling this course doesn't add
- **Setup:** Vite scaffolds the React client; a plain Express server with cors() and express.json() is the backend
- **Next chapter:** Data Modeling & Express API Routes

CHAPTER 2 OF 12

Data Modeling & Express API Routes

Food Tracker (React + Express)

Chapter 2 · Data Modeling & Express API Routes

Every course in this quartet stores the same shape of data. What genuinely differs here is deliberate: this course reaches for the *plainest* possible way to store it — no ORM at all, just SQL, written directly.

The Shared Pantry Item Schema

The same fields Food Tracker (Django) modeled with Django's ORM and Food Tracker (FastAPI) models with SQLAlchemy, expressed here as a plain SQL table:

```sql
-- schema.sql
CREATE TABLE IF NOT EXISTS items (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  barcode TEXT,
  category TEXT,
  expiry_date TEXT,              -- nullable: NULL once used, or never set
  status TEXT NOT NULL DEFAULT 'active',
  added_at TEXT NOT NULL DEFAULT (datetime('now')),
  used_at TEXT
);
```

Every design decision from the Django course's own Chapter 2 still applies exactly as reasoned there: `expiry_date` is nullable — `NULL` once an item is marked used, not deleted — and the row itself never gets removed, so the combined history list (Chapter 8) always has something to show. `added_at` defaults to the current time at insert, mirroring Django's own `auto_now_add` — the database itself stamps it, so the client never has to be trusted to supply an honest timestamp.

Why No ORM Here — A Deliberate, Honest Choice

Python has one dominant default in each of this quartet's other two backend courses — Django's own built-in ORM, and SQLAlchemy for FastAPI. Node's own ecosystem has no single equivalent default; several exist (Prisma, Sequelize, Drizzle, Knex), each with real adoption but none as close to "the obvious choice" as Django's own ORM is for Django. Rather than pick one somewhat arbitrarily, this course uses `better-sqlite3` directly — a genuine SQL library, not an ORM at all — and writes real SQL by hand throughout.

```
npm install better-sqlite3

// db.js
import Database from "better-sqlite3";
import fs from "fs";

const db = new Database("foodtracker.db");
db.exec(fs.readFileSync("./schema.sql", "utf8"));

export default db;
```

THE REAL COST OF SKIPPING AN ORM

Django's own migration system (`makemigrations` / `migrate`) tracks every model change as a versioned, reversible file. This course has none of that — changing the schema later means hand-writing an `ALTER TABLE` statement and running it manually, with no built-in history of what changed or when. For a single-table app at this course's own realistic scale, that's a reasonable tradeoff; it stops being one the moment the schema grows large or the team grows past one person.

BETTER-SQLITE3 IS DELIBERATELY SYNCHRONOUS

Unlike most Node database drivers, `better-sqlite3` has no `async` / `await` at all — `db.prepare(sql).get()` returns its result immediately, blocking the event loop for that one query. This is a real, intentional departure from Node's usual async-everything convention, justified by SQLite's own local-file nature (no network round-trip to wait on) and by the library's own measured performance advantage over async alternatives for this exact use case.

Express Route Structure

Routes live in their own module, mounted under a common prefix — the pattern every later chapter's own routes build on:

```
// routes/items.js
import { Router } from "express";
import db from "../db.js";

const router = Router();

router.get("/", (req, res) => {
  const items = db.prepare("SELECT * FROM items ORDER BY added_at DESC").all();
  res.json(items);
});

router.post("/", (req, res) => {
  const { name, barcode, category, expiry_date } = req.body;
  const result = db.prepare(
    "INSERT INTO items (name, barcode, category, expiry_date) VALUES (?, ?, ?, ?)"
  ).run(name, barcode, category, expiry_date);
  res.status(201).json({ id: result.lastInsertRowid });
});

export default router;

// server/index.js (mounting the router)
import itemsRouter from "./routes/items.js";
app.use("/api/items", itemsRouter);
```

Note the parameterized `?` placeholders in the `INSERT` — `better-sqlite3` handles escaping automatically, the same real protection against SQL injection that an ORM would otherwise provide implicitly. Skipping an ORM does not mean skipping this protection; it just means it's the developer's own responsibility to always use placeholders rather than string-concatenating values into a query.

> **THIS CHAPTER'S OWN HONEST TRADEOFF, STATED PLAINLY**
>
> Writing raw SQL directly is more transparent — nothing is generated or hidden behind an abstraction layer — and avoids picking among several competing, less-dominant Node ORMs. The real cost is everything an ORM would otherwise give for free: schema migrations, a query builder, and object-relational mapping convenience. This course accepts that cost deliberately, at a scale small enough that it stays a reasonable one.

Where This Course Is Headed

Barcode lookup next, then the camera-scanning React component (shared almost verbatim with Food Tracker (React + Firebase)), building on this chapter's own routing pattern for every remaining feature.

Hands-On Exercises

EXERCISE 1

Explain why this course uses better-sqlite3 directly instead of an ORM, and name the one concrete capability this choice gives up compared to Django's own migration system.

View solution

EXERCISE 2

Explain what makes better-sqlite3 unusual among Node database libraries, and why that unusual choice is still justified for this specific app.

View solution

EXERCISE 3

Explain why the ? placeholders in the INSERT statement matter for security, and what would happen if a value were string-concatenated into the SQL directly instead.

View solution

CHAPTER 2 QUICK REFERENCE

- **Schema:** id, name, barcode, category, expiry_date (nullable), status, added_at, used_at — same shape as every sibling course
- **No ORM:** better-sqlite3, real SQL written by hand, deliberately (Node has no single dominant ORM the way Django does)
- **Real cost:** no migrations system — schema changes are manual ALTER TABLE statements
- **Deliberately synchronous:** better-sqlite3 has no async/await, unlike most Node DB drivers
- **Routes:** Router-per-resource, mounted under /api/items — the pattern every later chapter reuses
- **Security:** parameterized ? placeholders prevent SQL injection, the same as an ORM would
- **Next chapter:** Barcode Lookup: Integrating Open Food Facts

CHAPTER 3 OF 12

Barcode Lookup: Integrating Open Food Facts

Food Tracker (React + Express)

Chapter 3 · Barcode Lookup: Integrating Open Food Facts

Every course in this quartet integrates Open Food Facts the same way in spirit: through a server the client trusts, not directly from the browser. Here, that server is the same Express process built in Chapter 2 — one more route, reusing the exact same `db` connection.

Why Proxy Through the Server At All

Open Food Facts needs no API key — there's no secret to hide, so this isn't about security the way an authenticated third-party API would demand. The real reasons to route the lookup through Express rather than calling Open Food Facts directly from React are the same ones the Django and Firebase courses reasoned through in their own Chapter 3/4:

- **Caching.** The same barcode gets scanned repeatedly over time — caching the result server-side avoids re-querying Open Food Facts for a product this app has already looked up.
- **Consistency.** Every client — this React app today, a future mobile app tomorrow — gets identical lookup behavior, defined in one place.
- **Future-proofing.** If Open Food Facts' own API shape ever changes, or a second data source gets added later, only the server needs to change.

Extending the Schema: A Barcode Cache

Reusing Chapter 2's own `schema.sql` file and `db.js` connection, one more table:

```
-- schema.sql (appended)
CREATE TABLE IF NOT EXISTS barcode_cache (
  barcode TEXT PRIMARY KEY,
  name TEXT,
  category TEXT,
  cached_at TEXT NOT NULL DEFAULT (datetime('now'))
);
```

`barcode` is the primary key here — deliberately different from Chapter 2's own `items` table, where the same barcode can legitimately appear on many separate rows (many separate purchases of the same product over time). One barcode maps to exactly one cached product lookup, but potentially many pantry items.

The Lookup Route

```
// routes/lookup.js
import { Router } from "express";
import db from "../db.js";

const router = Router();

router.get("/:barcode", async (req, res) => {
  const { barcode } = req.params;

  const cached = db.prepare("SELECT * FROM barcode_cache WHERE barcode = ?").get(barcode);
  if (cached) return res.json(cached);

  const response = await fetch(
    `https://world.openfoodfacts.org/api/v2/product/${barcode}.json`
  );
  const data = await response.json();

  if (data.status !== 1) {
    return res.status(404).json({ error: "Product not found" });
  }

  const name = data.product.product_name || null;
  const category = data.product.categories_tags?.[0] || null;

  db.prepare(
    "INSERT INTO barcode_cache (barcode, name, category) VALUES (?, ?, ?)"
  ).run(barcode, name, category);

  res.json({ barcode, name, category });
});

export default router;

// server/index.js
import lookupRouter from "./routes/lookup.js";
app.use("/api/lookup", lookupRouter);
```

NO EXTRA PACKAGE NEEDED FOR THE HTTP CALL

`fetch` is a genuine Node.js global as of Node 18 — no `axios` or `node-fetch` dependency required to call an external API from the server. The exact same `fetch` API a browser already knows now works identically on the server side, one more small piece of "the same language, both ends" this course keeps returning to.

OPEN FOOD FACTS' DATA IS GENUINELY INCONSISTENT

Because Open Food Facts is a crowdsourced, community-maintained database, a valid barcode can still return a product with a missing name, no category, or sparse data generally — `data.status === 1` only means "a product exists for this barcode," not "this product has complete data." The route above already returns `null` for missing fields rather than throwing — Chapter 5's own add-item form has to be built expecting that, with a manual-entry fallback for whatever the lookup didn't provide.

SAME LESSON, REUSED ROUTE PATTERN

Django's own view called `requests`; Firebase's Cloud Function called `fetch` in an isolated serverless context. Here, the exact same integration lives as one more route in the same Express app already serving `/api/items` — no new deployment target, no new runtime, just another file mounted the same way Chapter 2 established.

Where This Course Is Headed

The camera-scanning React component next — shared almost verbatim with Food Tracker (React + Firebase), since decoding a barcode client-side has nothing to do with which backend receives it afterward.

Hands-On Exercises

EXERCISE 1

Explain the three reasons this chapter gives for proxying the Open Food Facts lookup through Express rather than calling it directly from React, given that no API key is involved.

View solution

EXERCISE 2

Explain why barcode is the primary key in barcode_cache but not in the items table, even though both tables have a barcode column.

View solution

EXERCISE 3

Explain why data.status === 1 is not the same guarantee as "this product has complete data," and what the lookup route does about that in practice.

View solution

CHAPTER 3 QUICK REFERENCE

- **Why proxy:** caching, consistency across clients, future-proofing — not secrecy (no API key needed)
- **New table:** barcode_cache, keyed by barcode (one lookup per barcode, unlike items)
- **Route:** GET /api/lookup/:barcode — checks the cache first, else calls Open Food Facts and caches the result
- **Node's built-in fetch:** no axios/node-fetch dependency needed since Node 18
- **Real gotcha:** Open Food Facts data is crowdsourced and often incomplete — null fields are expected, not an error
- **Next chapter:** Camera-Based Barcode Scanning in React

CHAPTER 4 OF 12

Camera-Based Barcode Scanning in React

Food Tracker (React + Express)

Chapter 4 · Camera-Based Barcode Scanning in React

This chapter builds the piece of the app that feeds Chapter 3's Express route a barcode in the first place — and it's the one chapter in this entire course with no Express, no Node, and no server involvement at all. Everything here runs in the browser, which is exactly why this exact component was already built once before: Food Tracker (React + Firebase)'s own Chapter 4 built it first, and what follows is that same code, unchanged.

Requesting Camera Access

```
const stream = await navigator.mediaDevices.getUserMedia({
  video: { facingMode: "environment" },
});
```

`facingMode: "environment"` requests the **rear** camera on a phone — the one actually useful for scanning a product. `getUserMedia` also only works over **HTTPS** (or `localhost` during development) — worth knowing now, and something to check specifically once Chapter 11 deploys this app for real, since an Express deployment doesn't get HTTPS by default the automatic way Firebase Hosting does.

Decoding Barcodes From Video Frames

Two genuinely different approaches exist. The native browser `BarcodeDetector` API is fast and built in — but as of general browser support, it's available in Chrome/Edge on Android and desktop, and **not** in Safari on iOS. A JS library like **ZXing** (`@zxing/browser`) works everywhere, at some added CPU cost, since it decodes frames in pure JavaScript rather than using a native implementation. The practical pattern: try `BarcodeDetector` where it exists, fall back to ZXing where it doesn't.

```
import { useEffect, useRef } from "react";

function useBarcodeScanner(onDetected) {
  const videoRef = useRef(null);

  useEffect(() => {
    let stream;
    let stopped = false;

    async function start() {
      stream = await navigator.mediaDevices.getUserMedia({
        video: { facingMode: "environment" },
      });
      videoRef.current.srcObject = stream;
      await videoRef.current.play();

      if ("BarcodeDetector" in window) {
        const detector = new BarcodeDetector({ formats: ["ean_13", "upc_a"] });
        const scan = async () => {
          if (stopped) return;
          const barcodes = await detector.detect(videoRef.current);
          if (barcodes.length > 0) {
            onDetected(barcodes[0].rawValue);
            return;
          }
          requestAnimationFrame(scan);
        };
        scan();
      } else {
        // fall back to ZXing's BrowserMultiFormatReader here
      }
    }

    start();
    return () => {
      stopped = true;
      stream?.getTracks().forEach((track) => track.stop());
    };
  }, [onDetected]);

  return videoRef;
}
```

Note what this hook does *not* import, call, or reference anywhere: no `fetch` to a specific URL, no backend SDK, nothing at all. It only calls one thing — `onDetected(barcode)` — a plain callback prop the parent supplies. That single design decision is exactly what makes reusing it unchanged possible.

Wiring It to This Course's Own Backend

The only place this course's own identity shows up is in the parent component's callback — everything above this point is identical, character for character, to what Food Tracker (React + Firebase)'s own Chapter 4 already built:

```
function ScanScreen() {
  const handleDetected = async (barcode) => {
    const response = await fetch(`/api/lookup/${barcode}`); // Chapter 3's Express route
    const result = await response.json();
    // ...hand result to the add-item flow, Chapter 5
  };
  const videoRef = useBarcodeScanner(handleDetected);









  return
```

Compare this to Food Tracker (React + Firebase)'s own `handleDetected`, which called a Cloud Function (`lookupBarcode({ barcode })`) instead of a plain `fetch`. That one line is the **entire** difference between the two courses' use of this component — a callable Cloud Function on one side, a REST `fetch` call on the other, both feeding an identical scanning hook.

THE POINT OF BUILDING THE IDENTICAL COMPONENT TWICE

Seeing the exact same camera-scanning code plug into two completely different backends — a Cloud Function in the Firebase course, a plain REST endpoint here — is itself the lesson: a well-designed frontend component doesn't need to know or care what's on the other end of its one callback prop. The backend is a genuinely swappable detail, not something the component is built around. This is the payoff of the "one language, both ends" throughline from Chapter 1, made concrete: the same component works unmodified against a hand-written Express route just as easily as it worked against Firebase's own managed infrastructure.

DESKTOP TESTING DOESN'T TELL THE WHOLE STORY

A laptop webcam is a poor stand-in for the real experience — no autofocus hunting, no holding a curved product steady, none of the resolution constraints a phone camera actually has. Test on a real phone against an HTTPS URL (a tunneling tool like ngrok during local development, since this course's own Express deployment doesn't get HTTPS for free the way Firebase Hosting does) before trusting that scanning "works."

TESTING ONLY IN CHROME HIDES A REAL BUG

If the ZXing fallback branch is left as a stub (as it is above, for brevity) rather than actually implemented, the scanner will work perfectly in Chrome-based browser testing and then **silently fail** for every iPhone Safari user — a large share

of any real phone-camera app's actual audience. `"BarcodeDetector" in window` being false doesn't throw an error; it just quietly does nothing unless the fallback path is genuinely built and tested, not just sketched in a comment.

Where This Course Is Headed

The add-item flow next — a React form component plus an Express POST endpoint, taking this scanned barcode's lookup result the rest of the way into the database, with client- and server-side validation both covered honestly.

Hands-On Exercises

EXERCISE 1

Explain exactly what changed and what stayed identical between this chapter's ScanScreen and Food Tracker (React + Firebase)'s own version of it. Why does so little actually need to change?

View solution

EXERCISE 2

Explain what happens if the useEffect cleanup function omits stream?.getTracks().forEach(track => track.stop()), and why this matters specifically for a scan screen a user might navigate to and away from repeatedly.

View solution

EXERCISE 3

Explain why leaving the ZXing fallback as an unimplemented stub could pass all of a developer's own testing and still be a real bug in production. Which users specifically would be affected, and why wouldn't Chrome-based testing ever catch it?

View solution

CHAPTER 4 QUICK REFERENCE

- **This component is reused, not rebuilt** — identical to Food Tracker (React + Firebase) Chapter 4's own useBarcodeScanner hook
- `facingMode: "environment"` — the rear camera, not the front-facing one
- `BarcodeDetector` — native, fast, Chrome/Edge/Android; not on Safari/iOS — needs a ZXing fallback

- **Cleanup** — always stop every track from the stream in the effect's cleanup function, or the camera stays on after unmount
- **The one line that differs:** handleDetected calls fetch("/api/lookup/...") here, vs. a Cloud Function in the Firebase course
- **Next chapter:** Building the Add-Item Flow

CHAPTER 5 OF 12

Building the Add-Item Flow

Food Tracker (React + Express)

Chapter 5 · Building the Add-Item Flow

Chapter 4's scanner hands off a barcode lookup result — or nothing at all, if the scan misses or the product isn't in Open Food Facts. Either way, this chapter is where that result actually becomes a row in the `items` table Chapter 2 defined.

The Form Component

Pre-filled where Chapter 3's lookup provided data, editable everywhere, and fully usable even with nothing pre-filled at all — a genuine manual-entry fallback, not an afterthought:

```jsx
import { useState } from "react";

function AddItemForm({ initialData = {}, onSaved }) {
  const [name, setName] = useState(initialData.name || "");
  const [category, setCategory] = useState(initialData.category || "");
  const [expiryDate, setExpiryDate] = useState("");
  const [barcode] = useState(initialData.barcode || null);
  const [error, setError] = useState(null);

  const handleSubmit = async (e) => {
    e.preventDefault();
    setError(null);

    if (!name.trim()) {
      setError("Item name is required.");
      return;
    }

    const response = await fetch("/api/items", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ name, barcode, category, expiry_date: expiryDate || null }),
    });

    if (!response.ok) {
      const data = await response.json();
      setError(data.error || "Something went wrong.");
      return;
    }

    onSaved(await response.json());
  };

  return (
    <form onSubmit={handleSubmit}>
      <input value={name} onChange={(e) => setName(e.target.value)} placeholder="Item name" />
      <input value={category} onChange={(e) => setCategory(e.target.value)} placeholder="Category" />
      <input type="date" value={expiryDate} onChange={(e) => setExpiryDate(e.target.value)} />
      {error && <p className="form-error">{error}</p>}
      <button type="submit">Add Item</button>
    </form>
  );
}
```

`expiryDate || null` matters here specifically: an empty string is not the same value as a missing date, and Chapter 2's schema expects `NULL`, not an empty string, whenever no expiry date applies.

The Client-Side Check, and Why It's Not Enough on Its Own

`if (!name.trim())` above catches an empty name instantly, before any network request — real, useful UX, giving immediate feedback with no round-trip delay. But it's running entirely inside code the browser executes, which means it's also code a user (or a malicious script, or a stray `curl` command) can simply never run at all. Nothing about the client-side check stops a POST request built by hand from reaching the server with no `name` field whatsoever.

The Real Gate: Server-Side Validation

Chapter 3's POST route, updated to actually check what it receives before touching the database:

```
// routes/items.js
router.post("/", (req, res) => {
  const { name, barcode, category, expiry_date } = req.body;

  if (!name || typeof name !== "string" || !name.trim()) {
    return res.status(400).json({ error: "name is required" });
  }
  if (expiry_date && isNaN(Date.parse(expiry_date))) {
    return res.status(400).json({ error: "expiry_date is not a valid date" });
  }

  const result = db.prepare(
    "INSERT INTO items (name, barcode, category, expiry_date) VALUES (?, ?, ?, ?)"
  ).run(name.trim(), barcode || null, category || null, expiry_date || null);

  res.status(201).json({ id: result.lastInsertRowid, name, barcode, category, expiry_date });
});
```

WHY THIS APP ALREADY HAS A REAL GATE, BY CONSTRUCTION

Food Tracker (React + Firebase) had to introduce Security Rules specifically because its React client writes directly to Firestore — with no server code sitting in between by default, nothing would validate a write at all unless Security Rules were deliberately added later to fill that gap. This course never had that gap in the first place: every single write already passes through Chapter 2's own Express route, because that's simply how this architecture works. Server-side validation here isn't a bolted-on addition; it's the natural, unavoidable consequence of choosing "the client always talks to my own server" back in Chapter 1.

NEVER TRUST REQ.BODY

Anything arriving in `req.body` came from outside this process, regardless of which client sent it or how carefully that client's own form was built. A missing field, a wrong type, or a deliberately malformed request are all real possibilities the server must check for itself — the client-side check earlier in this chapter exists purely to make the honest, well-behaved case pleasant; it does no security work whatsoever.

THE LOOKUP-MISS CASE IS NOT AN EDGE CASE

A meaningful share of real barcodes won't resolve to a name at all (Chapter 3's own honest note about Open Food Facts' inconsistent data) — `initialData` being `{}` the whole way through, with the user typing every field by hand, needs to be a genuinely first-class path through this form, not something only handled if there happens to be time for it.

Where This Course Is Headed

Expiry alerts next — an Express endpoint querying for items nearing their expiry date, paired with a React dashboard component.

Hands-On Exercises

EXERCISE 1

Explain why the client-side name check in AddItemForm provides no real security, even though it correctly prevents an empty name from being submitted through the form's own UI.

View solution

EXERCISE 2

Explain the finding-box's own claim: why did Food Tracker (React + Firebase) need to add Security Rules specifically to get real write validation, while this course's Express route already provides it "by construction"?

View solution

EXERCISE 3

Explain why expiryDate || null matters in the form's submit handler — what would go wrong if an empty string were sent to the server instead of null when no date is entered?

View solution

CHAPTER 5 QUICK REFERENCE

- **AddItemForm:** pre-filled from Chapter 4's scan result, fully usable with nothing pre-filled (the manual-entry fallback)
- **Client-side validation:** real UX value (instant feedback), zero security value (trivially bypassable)

- **Server-side validation:** the actual gate — checks name presence/type and expiry_date validity before touching the database
- **This course's own architectural advantage:** every write already passes through Express by construction — no separate Security Rules layer needed, unlike the Firebase sibling
- **Gotcha:** an empty string and a missing value are not the same thing — always normalize to null before hitting the database
- **Next chapter:** Expiry Alerts

CHAPTER 6 OF 12

Expiry Alerts

Food Tracker (React + Express)

Chapter 6 · Expiry Alerts

Every item added in Chapter 5 now has a real row, with a real (or null) `expiry_date`. This chapter turns that stored date into something actually useful: a list of what needs to be used soon.

The Alerts Route

```
// routes/items.js (appended)
router.get("/alerts", (req, res) => {
  const items = db.prepare(`
    SELECT * FROM items
    WHERE status = 'active'
      AND expiry_date IS NOT NULL
      AND expiry_date <= date('now', '+3 days')
    ORDER BY expiry_date ASC
  `).all();
  res.json(items);
});
```

`status = 'active'` excludes anything already marked used (Chapter 9's own territory); `expiry_date IS NOT NULL` excludes items with no expiry date at all — both conditions matter, since a used item retains its historical row but no longer has a live expiry date to alert on. `date('now', '+3 days')` is SQLite's own built-in date arithmetic, computing "three days from today" directly inside the query.

THIS COMPARISON ONLY WORKS BECAUSE DATES ARE STORED AS ISO 8601 TEXT

SQLite has no dedicated date type — `expiry_date` is a `TEXT` column, and `<=` between two text values compares them **lexicographically** (character by character), not chronologically. This happens to give the correct chronological result here *only* because every date in this schema is consistently stored as `YYYY-MM-DD` — a format where lexicographic order and chronological order agree. Storing even one date in a different format (`MM/DD/YYYY`, for instance) would silently break every comparison in this route, with no error at all — just wrong results.

NO COMPOSITE INDEX NEEDED HERE — A REAL, HONEST CONTRAST

Food Tracker (React + Firebase)'s own equivalent query needed a composite index, because Firestore requires one whenever a query filters on one field and orders by another simultaneously — exactly this route's own shape (`WHERE status = ...`, `ORDER BY expiry_date`). SQLite has no such requirement: a query like this one runs correctly with no index declared at all, and at this app's own realistic scale (a single household's pantry, at most a few hundred rows), a

> full table scan on every request is genuinely fast enough that adding an index wouldn't even be noticeable. This isn't SQL being "better" in general — Firestore's index requirement exists for good reasons at Firestore's own intended scale — but at this specific app's own size, it's a real, fair point in this course's favor, named honestly rather than glossed over.

A Custom Hook for the Dashboard

```
// hooks/useExpiryAlerts.js
import { useState, useEffect, useCallback } from "react";

function useExpiryAlerts() {
  const [alerts, setAlerts] = useState([]);
  const [loading, setLoading] = useState(true);

  const refresh = useCallback(async () => {
    setLoading(true);
    const response = await fetch("/api/items/alerts");
    setAlerts(await response.json());
    setLoading(false);
  }, []);

  useEffect(() => { refresh(); }, [refresh]);

  return { alerts, loading, refresh };
}
```

`refresh` is exposed deliberately, not just called once internally — Chapter 9's own "mark used" action needs a way to trigger a fresh alerts fetch immediately afterward, since an item marked used should disappear from this list right away, not just on the next full page reload.

The Dashboard Component

```jsx
function ExpiryDashboard() {
  const { alerts, loading } = useExpiryAlerts();

  if (loading) return <p>Loading...</p>;
  if (alerts.length === 0) return <p>Nothing expiring soon.</p>;

  return (
    <ul>
      {alerts.map((item) => (
        <li key={item.id}>
          {item.name} — expires {item.expiry_date}
        </li>
      ))}
    </ul>
  );
}
```

KEEP THE "HOW MANY DAYS IS SOON" THRESHOLD IN ONE PLACE

`'+3 days'` is hardcoded directly in the SQL string above for clarity in this chapter, but a real app should pull that number from one shared constant (an environment variable, or a config file) rather than repeating the literal string anywhere the query might be duplicated later — the same "define it once" discipline this course already applied to the accent color and schema definitions.

Where This Course Is Headed

Item history and live search-as-you-type next — a debounced search built as a custom React hook, hitting a new Express search endpoint.

Hands-On Exercises

EXERCISE 1

Explain why the alerts query's date comparison only works correctly because every date in this schema is stored in YYYY-MM-DD format, and what would happen if one date were stored in a different format.

View solution

EXERCISE 2

Explain why this course's alerts query needs no composite index while the equivalent Firestore query in Food Tracker (React + Firebase) does, and why this isn't simply "SQL is better than Firestore" in general.

View solution

EXERCISE 3

Explain why useExpiryAlerts exposes its own refresh function rather than only fetching once internally on mount.

View solution

CHAPTER 6 QUICK REFERENCE

- **Route:** GET /api/items/alerts — status='active' AND expiry_date IS NOT NULL AND expiry_date <= date('now', '+3 days')
- **Real gotcha:** SQLite text-date comparison only works because every date uses YYYY-MM-DD consistently
- **Fair SQL advantage:** no composite index needed at this app's realistic scale, unlike the Firebase sibling's equivalent query
- **useExpiryAlerts:** a custom hook exposing alerts, loading, and a callable refresh (needed by Chapter 9's own mark-used action)
- **Next chapter:** Item History & Live Search-as-You-Type

CHAPTER 7 OF 12

Item History & Live Search-as-You-Type

Food Tracker (React + Express)

Chapter 7 · Item History & Live Search-as-You-Type

Every item ever added stays in the `items` table forever — Chapter 2's own design decision, active or used. This chapter surfaces that whole history, searchable in real time as the user types.

The Search Route

```
// routes/items.js (appended)
router.get("/search", (req, res) => {
  const q = req.query.q || "";
  const items = db.prepare(`
    SELECT * FROM items
    WHERE name LIKE '%' || ? || '%'
    ORDER BY added_at DESC
    LIMIT 50
  `).all(q);
  res.json(items);
});
```

Deliberately no `status` filter — unlike Chapter 6's own alerts query, this route searches the **entire** history, active and used items both, since re-adding something bought before is exactly the case this search exists for. `'%' || ? || '%'` wraps the parameter in SQL wildcards while still keeping it fully parameterized — the same SQL-injection protection from Chapter 2 applies here without any extra effort.

THE SAME REAL SQL ADVANTAGE THE DJANGO SIBLING ALREADY NAMED

Food Tracker (React + Firebase) had to add a lowercase-copy field, `nameLower`, purely because Firestore has no native case-insensitive substring search at all — every item's name has to be duplicated into a second, search-friendly field just to make matching work. SQLite's own `LIKE` operator is case-insensitive for ASCII text *by default*, with no shadow field, no duplicated data, and no extra write-time bookkeeping required. This is the exact same honest advantage Food Tracker (Django) already claimed for its own `icontains` lookup — a genuine, real SQL win, not specific to any one framework built on top of SQL.

THE HONEST LIMIT OF THIS CONVENIENCE

A leading wildcard (`'%' || ? || '%'`) can never use a standard database index efficiently, even if one existed on `name` — the database has no choice but to check every row, since a match could start anywhere in the string. At this app's own realistic scale (Chapter 6's own point, repeated here), that cost is genuinely invisible. It would become a

real, different problem at a scale large enough to need dedicated full-text search — a tool like SQLite's own FTS5 extension, or an external search service, neither of which this course builds.

A Reusable Debounce Hook

Firing a search request on every single keystroke would flood the server with requests for a query the user hasn't finished typing yet. Debouncing delays the actual fetch until typing pauses:

```
// hooks/useDebouncedSearch.js
import { useState, useEffect } from "react";

function useDebouncedSearch(query, delay = 300) {
  const [results, setResults] = useState([]);

  useEffect(() => {
    if (!query) {
      setResults([]);
      return;
    }

    const timeoutId = setTimeout(async () => {
      const response = await fetch(`/api/items/search?q=${encodeURIComponent(query)}`);
      setResults(await response.json());
    }, delay);

    return () => clearTimeout(timeoutId);
  }, [query, delay]);

  return results;
}
```

THE CLEANUP LINE IS NOT OPTIONAL — THE SAME LESSON AS CHAPTER 4'S CAMERA STREAM

Without `return () => clearTimeout(timeoutId)`, every keystroke would still schedule its own timer, and every one of those timers would eventually fire — the delay would only push the flood of requests later, not prevent it. Because `useEffect` re-runs this whole function on every `query` change, each run's cleanup cancels the *previous* run's still-pending timer before scheduling a new one — the same "always clean up what the last effect run started" discipline Chapter 4's camera-stream cleanup already taught, applied here to a timer instead of a media stream.

The Search Component

```
function SearchBox() {
  const [query, setQuery] = useState("");
  const results = useDebouncedSearch(query);

  return (
    <div>
      <input
        value={query}
        onChange={(e) => setQuery(e.target.value)}
        placeholder="Search item history..."
      />
      <ul>
        {results.map((item) => (
          <li key={item.id}>
            {item.name} — {item.status === "used" ? "used" : `active, expires ${item.expiry_date}`}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

300MS IS A REASONABLE DEFAULT, NOT A UNIVERSAL CONSTANT

Too short a delay defeats the purpose of debouncing at all; too long makes the search feel sluggish and unresponsive. 200–400ms is a common, comfortable range for this kind of type-ahead search — worth tuning against real typing speed rather than treating as a fixed rule.

Where This Course Is Headed

Marking items used next — a React action and an Express PATCH endpoint, tying directly back into both this chapter's own search results and Chapter 6's own alerts dashboard.

Hands-On Exercises

EXERCISE 1

Explain why this course's search route needs no shadow field the way Food Tracker (React + Firebase)'s nameLower field does, and name the SQLite feature responsible.

View solution

EXERCISE 2

Explain what would happen if useDebouncedSearch's useEffect omitted its cleanup function, and why the fix is described as "the same lesson" as Chapter 4's camera-stream cleanup.

View solution

EXERCISE 3

Explain why the search route deliberately has no status filter, unlike Chapter 6's own alerts route.

View solution

CHAPTER 7 QUICK REFERENCE

- **Route:** GET /api/items/search?q=... — SQL LIKE '%'||?||'%', no status filter, the full history
- **Real SQL advantage:** SQLite's LIKE is case-insensitive for ASCII by default — no nameLower shadow field needed, unlike the Firebase sibling
- **Honest limit:** a leading wildcard can't use an index — invisible at this app's scale, a real cost at a larger one
- **useDebouncedSearch:** a reusable hook, delay tunable, cleanup cancels the previous pending timer on every keystroke
- **Same lesson as Chapter 4:** always clean up what the previous effect run started
- **Next chapter:** Marking Items Used

CHAPTER 8 OF 12

Marking Items Used

Food Tracker (React + Express)

Chapter 8 · Marking Items Used

Every earlier chapter built toward this exact moment: an item is finally used up, and the row created back in Chapter 5 needs to reflect that — without ever disappearing from the history Chapter 7 searches.

The Route

```
// routes/items.js (appended)
router.patch("/:id/use", (req, res) => {
  const { id } = req.params;

  const result = db.prepare(`
    UPDATE items
    SET status = 'used', used_at = datetime('now'), expiry_date = NULL
    WHERE id = ? AND status = 'active'
  `).run(id);

  if (result.changes === 0) {
    return res.status(404).json({ error: "Item not found, or already used" });
  }

  res.json({ id, status: "used" });
});
```

Two things worth noticing in that single query. First, `expiry_date = NULL` — the same nullable-not-deleted design Chapter 2 committed to from the start, now paying off: the row stays in the table forever, but its expiry date genuinely goes away, exactly the way Chapter 6's own alerts query (`expiry_date IS NOT NULL`) already expects. Second, `AND status = 'active'` in the `WHERE` clause — the update only actually touches a row that's currently active, which means marking an already-used item "used" again is a no-op rather than silently re-stamping `used_at` with a new, incorrect timestamp.

THIS MUST BE A PATCH, NEVER A PLAIN LINK

A state-changing action like this must never be reachable via a plain `GET` — a browser's own link-prefetching, a crawler following every link on a page, or simply a user middle-clicking to open in a new tab could all trigger a `GET` request without the user ever intending to mark anything used. `PATCH` (or `POST`) requires the request to come from

an explicit action — a button's `onClick` firing a real `fetch` call — never something a browser might do on its own while merely loading or navigating a page.

The React Action

```
function ExpiryDashboard() {
  const { alerts, loading, refresh } = useExpiryAlerts();

  const markUsed = async (id) => {
    await fetch(`/api/items/${id}/use`, { method: "PATCH" });
    refresh();
  };

  if (loading) return <p>Loading...</p>;
  if (alerts.length === 0) return <p>Nothing expiring soon.</p>;

  return (
    <ul>
      {alerts.map((item) => (
        <li key={item.id}>
          {item.name} — expires {item.expiry_date}
          <button onClick={() => markUsed(item.id)}>Mark Used</button>
        </li>
      ))}
    </ul>
  );
}
```

CHAPTER 6'S REFRESH() FINALLY EARNS ITS KEEP

Chapter 6 exposed `refresh` from `useExpiryAlerts` specifically for this moment, rather than only fetching once internally. Calling `refresh()` right after the `PATCH` succeeds re-runs the alerts query, and since the item's `status` is now `'used'`, it no longer matches that query's own `WHERE status = 'active'` condition — it disappears from the dashboard immediately, with no full page reload and no manual list-filtering logic on the frontend at all. The backend query is the single source of truth for "what's currently expiring soon"; the frontend just asks it again.

WAIT-THEN-REFRESH, NOT OPTIMISTIC UPDATES

A more polished app might remove the item from the UI immediately, before the server even responds ("optimistic" updating), then roll back if the request fails. This course deliberately keeps the simpler approach — wait for the `PATCH` to actually succeed, then re-fetch — accepting a small, honest delay in exchange for never showing the user a state the server hasn't actually confirmed.

Where This Course Is Headed

Recipe lookup with TheMealDB next — an Express route and a React results component, matching items nearing expiry against real recipes.

Hands-On Exercises

EXERCISE 1

Explain what the AND status = 'active' clause in the UPDATE statement actually prevents, and what would go wrong without it if a user managed to click "Mark Used" twice on the same item.

View solution

EXERCISE 2

Explain why marking an item used must never be implemented as a plain GET request, with a concrete example of how a GET-based version could be triggered unintentionally.

View solution

EXERCISE 3

Explain how calling refresh() after a successful markUsed request causes the item to disappear from the dashboard, tracing exactly which earlier chapter's own query condition makes that work.

View solution

CHAPTER 8 QUICK REFERENCE

- **Route:** PATCH /api/items/:id/use — sets status='used', used_at, and clears expiry_date to NULL
- **Guard:** AND status = 'active' in the WHERE clause makes re-marking an already-used item a safe no-op
- **Never a GET:** a state-changing action must require an explicit request, not something a browser could trigger while merely loading a page
- **The payoff:** refresh() from Chapter 6 re-runs the alerts query, which naturally excludes the now-used item — no manual frontend filtering needed
- **Deliberate simplicity:** wait-then-refresh, not optimistic UI updates
- **Next chapter:** Recipe Lookup with TheMealDB

CHAPTER 9 OF 12

Recipe Lookup with TheMealDB

Food Tracker (React + Express)

Chapter 9 · Recipe Lookup with TheMealDB

Chapter 6's alerts query already knows what's expiring soon. This chapter takes that same list and asks a second free API, TheMealDB, what could actually be cooked with it.

Extending the Schema: A Recipe Cache

The same caching pattern Chapter 3 established for barcode lookups, applied to per-ingredient recipe results:

```
-- schema.sql (appended)
CREATE TABLE IF NOT EXISTS recipe_cache (
  ingredient TEXT PRIMARY KEY,
  meals TEXT NOT NULL,          -- JSON array, stored as text
  cached_at TEXT NOT NULL DEFAULT (datetime('now'))
);
```

The Suggestion Route

```
// routes/recipes.js
import { Router } from "express";
import db from "../db.js";

const router = Router();

async function lookupIngredient(ingredient) {
  const key = ingredient.toLowerCase().trim().replace(/\s+/g, "_");

  const cached = db.prepare("SELECT meals FROM recipe_cache WHERE ingredient = ?").get(key);
  if (cached) return JSON.parse(cached.meals);

  const response = await fetch(
    `https://www.themealdb.com/api/json/v1/1/filter.php?i=${key}`
  );
  const data = await response.json();
  const meals = data.meals || [];

  db.prepare(
    "INSERT OR REPLACE INTO recipe_cache (ingredient, meals, cached_at) VALUES (?, ?, datetime('now'))"
  ).run(key, JSON.stringify(meals));

  return meals;
}

router.get("/suggest", async (req, res) => {
  const expiring = db.prepare(`
    SELECT name FROM items
    WHERE status = 'active' AND expiry_date IS NOT NULL
      AND expiry_date <= date('now', '+3 days')
  `).all();

  const perIngredient = await Promise.all(
    expiring.map((item) => lookupIngredient(item.name))
  );

  const matchCounts = {};
  perIngredient.flat().forEach((meal) => {
    if (!matchCounts[meal.idMeal]) {
      matchCounts[meal.idMeal] = { ...meal, matchCount: 0 };
    }
    matchCounts[meal.idMeal].matchCount++;
  });

  const sorted = Object.values(matchCounts).sort((a, b) => b.matchCount - a.matchCount);
  res.json(sorted.slice(0, 10));
});
```

```
export default router;
```

A meal matching three expiring ingredients ranks above one matching only one — the same relevance-by-match-count sorting every course in this quartet uses for its own recipe feature.

A REAL, NAMED ADVANTAGE OF THIS COURSE'S OWN ARCHITECTURE

Food Tracker (Django) named its own equivalent fan-out honestly as sequential — one ingredient's TheMealDB call waiting for the previous one to finish, a real, admitted performance cost. Here, `Promise.all` fires every ingredient's lookup **concurrently** instead of one after another, because Node's own async model makes that the natural way to write it, not a special optimization bolted on afterward. Five expiring ingredients means five requests in flight at once here, rather than five requests run one after the other — a genuine payoff of Chapter 1's own "one language, both ends" framing, where JavaScript's async-first design turns out to matter for more than just tooling convenience.

THEMEALDB'S INGREDIENT NAMES ARE EXACT-MATCH, AND PANTRY NAMES RARELY ARE

`filter.php?i=` expects TheMealDB's own specific ingredient vocabulary (`chicken_breast`, not `chicken` or `chicken breasts`) — a real, generic pantry item name like "Trader Joe's Organic Chicken Thighs" won't match cleanly no matter how it's normalized. The `.toLowerCase().replace(/\s+/g, "_")` normalization above handles simple cases; it does not solve the deeper problem of a free-text product name not lining up with a curated recipe database's own fixed vocabulary. This app's own honest scope stops at "best-effort matching," not guaranteed matches for every real product name.

The React Results Component

```
function RecipeSuggestions() {
  const [recipes, setRecipes] = useState([]);

  useEffect(() => {
    fetch("/api/recipes/suggest")
      .then((r) => r.json())
      .then(setRecipes);
  }, []);

  return (
    <ul>
      {recipes.map((meal) => (
        <li key={meal.idMeal}>
          <img src={meal.strMealThumb} alt={meal.strMeal} width="60" />
          {meal.strMeal} — uses {meal.matchCount} expiring ingredient{meal.matchCount > 1 ? "s" : ""}
        </li>
      ))}
    </ul>
  );
}
```

THE CACHE IS PER-INGREDIENT, NOT PER-SUGGESTION

Caching keyed by `ingredient` rather than by the whole combination of expiring items means a cached "chicken" lookup gets reused the next time chicken appears in the alerts list, regardless of what else happened to be expiring alongside it that day — the same granular-caching principle as Chapter 3's own `barcode_cache`.

Where This Course Is Headed

State management across the whole app next — where component-local state ends and a shared approach begins, now that scanning, alerts, history, and recipes all need to talk to each other.

Hands-On Exercises

EXERCISE 1

Explain the concrete difference in behavior between Promise.all(expiring.map(...)) and a for loop that awaits each lookupIngredient call one at a time, for five expiring ingredients.

View solution

EXERCISE 2

Explain why a generic pantry item name like "Trader Joe's Organic Chicken Thighs" might fail to match anything in TheMealDB even after normalization, and why this is described as an honest scope limit rather than a bug to fix.

View solution

EXERCISE 3

Explain why recipe_cache is keyed by ingredient rather than by the full combination of expiring items on any given day, and what benefit that specific choice provides.

View solution

CHAPTER 9 QUICK REFERENCE

- **Route:** GET /api/recipes/suggest — fans out to TheMealDB per expiring ingredient, merges and sorts by match count
- **New table:** recipe_cache, keyed by ingredient, storing the JSON meal list as text

- **Real advantage:** Promise.all fires every ingredient lookup concurrently — a genuine payoff of Node's async model, vs. Food Tracker (Django)'s own honestly-named sequential fan-out
- **Real gotcha:** TheMealDB expects its own exact ingredient vocabulary — generic product names often won't match cleanly even after normalization
- **Next chapter:** State Management Across the App

CHAPTER 10 OF 12

State Management Across the App

Food Tracker (React + Express)

Chapter 10 · State Management Across the App

Every feature so far has managed its own state: `useExpiryAlerts`, `useDebouncedSearch`, `RecipeSuggestions`' own `useState`. That worked fine when each feature only needed to know about itself. It stops being enough the moment one action needs to update several of them at once.

The Problem, Concretely

Chapter 8's `markUsed` called `refresh()` directly on the one hook it happened to have a reference to — `useExpiryAlerts`. But marking an item used should really affect three separate features at once:

- **Alerts** (Chapter 6) — the item should disappear, since it's no longer active.
- **Search results** (Chapter 7) — if the user's current search happens to include this item, its status should update to reflect "used."
- **Recipe suggestions** (Chapter 9) — an ingredient that's no longer expiring shouldn't keep influencing recipe matches.

Wiring `markUsed` to call three separate refresh functions directly works today, but it doesn't scale: every new feature that cares about item changes means going back and adding one more manual call at every single place items get mutated (add, mark used, and anything added later). That's a real, growing coordination cost, not a hypothetical one.

A Shared Change-Notification Context

Rather than wiring components directly to each other, one shared signal any component can both trigger and listen to:

```jsx
// context/ItemsContext.jsx
import { createContext, useContext, useState, useCallback } from "react";

const ItemsContext = createContext(null);

function ItemsProvider({ children }) {
  const [version, setVersion] = useState(0);
  const notifyChange = useCallback(() => setVersion((v) => v + 1), []);

  return (
    <ItemsContext.Provider value={{ version, notifyChange }}>
      {children}
    </ItemsContext.Provider>
  );
}

function useItemsContext() {
  return useContext(ItemsContext);
}
```

`version` is deliberately just a number, not the actual item data — this Context coordinates *when to refetch*, it doesn't try to own or duplicate every feature's own data-fetching logic.

Wiring Every Consumer Through the Same Signal

Chapter 6's hook, updated to refetch whenever `version` changes, instead of only on mount:

```jsx
// hooks/useExpiryAlerts.js (updated)
function useExpiryAlerts() {
  const { version } = useItemsContext();
  const [alerts, setAlerts] = useState([]);

  useEffect(() => {
    fetch("/api/items/alerts").then((r) => r.json()).then(setAlerts);
  }, [version]);

  return { alerts };
}
```

And Chapter 8's `markUsed`, now calling the shared signal instead of one specific hook's own refresh function:

```jsx
const { notifyChange } = useItemsContext();

const markUsed = async (id) => {
  await fetch(`/api/items/${id}/use`, { method: "PATCH" });
  notifyChange();
};
```

`RecipeSuggestions` and the search hook get the same one-line change: depend on `version` in their own `useEffect`, instead of being individually wired to whatever action happened to trigger the update.

WHAT ACTUALLY CHANGED HERE

Before this chapter, every place items get mutated needed to know about every feature that cares — an `O(features × mutations)` wiring problem that only grows as the app grows. After this chapter, a mutation only needs to know about one thing: call `notifyChange()`. Every feature that cares about item changes subscribes to `version` on its own, entirely independently of whatever action happened to trigger the change. Adding a tenth feature later means that feature subscribes to `version` itself — it does not mean going back to add one more call inside `markUsed`, `addItem`, and everywhere else a mutation happens.

THIS IS A REAL LIMIT, NAMED HONESTLY

Bumping `version` causes every component consuming `ItemsContext` to re-render, even ones whose own displayed data didn't actually change as a result. At this app's own scale — a handful of features, infrequent mutations (a user adding or using an item, not hundreds of updates per second) — that's genuinely invisible. It would become a real performance concern in an app with many more Context consumers and much more frequent updates, where a more granular state library (Zustand, Jotai, or splitting into several smaller contexts) would be the more honest choice instead of one shared version counter.

CONTEXT SOLVES COORDINATION, NOT CACHING

This pattern intentionally does not try to hold a single shared copy of every item, avoiding re-fetching, in one Context-level cache — each hook still fetches its own view from its own Express route (alerts, search, or recipes). A library like React Query or SWR exists specifically to add that caching layer on top of this same coordination idea; this course keeps the simpler version, since re-fetching a small SQLite query on each relevant change is genuinely fast enough not to need it.

Where This Course Is Headed

Deployment next — serving the built React app from the same Express process, and the environment configuration that goes with a real deployment.

Hands-On Exercises

EXERCISE 1

Explain the concrete coordination problem with Chapter 8's original approach (markUsed calling refresh() directly) once a third feature, recipe suggestions, also needs to react to the same mutation.

View solution

EXERCISE 2

Explain why version is stored as a plain number rather than the actual items array, and what this Context is and isn't responsible for as a result.

View solution

EXERCISE 3

Explain the honest limit named in this chapter's own warn-box, and describe a scenario (in terms of app size or update frequency) where that limit would actually start to matter in practice.

View solution

CHAPTER 10 QUICK REFERENCE

- **The problem:** Chapter 8's direct refresh() call only reaches one hook — doesn't scale as more features need to react to the same mutation
- **ItemsContext:** holds one plain number, version, plus notifyChange() to bump it
- **Every relevant hook:** depends on version in its own useEffect, refetching its own data independently
- **The real change:** a mutation only calls notifyChange() once — it no longer needs to know which features are listening
- **Honest limit:** every Context consumer re-renders on every version bump — fine at this scale, a real cost at a much larger one
- **Not a caching layer:** each hook still fetches its own data from its own route; a library like React Query would add caching on top of this same idea
- **Next chapter:** Deployment

CHAPTER 11 OF 12

Deployment

Food Tracker (React + Express)

Chapter 11 · Deployment

Every prior chapter ran two separate processes — Vite's dev server and Express — talking across the CORS boundary Chapter 1 set up. A real deployment collapses that back down to one.

Building the Client

```
cd client
npm run build
# produces client/dist/ — a static index.html, JS, and CSS bundle
```

Serving the Build From Express

```
// server/index.js
import path from "path";
import { fileURLToPath } from "url";

const __dirname = path.dirname(fileURLToPath(import.meta.url));

// API routes registered first
app.use("/api/items", itemsRouter);
app.use("/api/lookup", lookupRouter);
app.use("/api/recipes", recipesRouter);

// static build + SPA fallback registered last
if (process.env.NODE_ENV === "production") {
  const clientDist = path.join(__dirname, "../client/dist");
  app.use(express.static(clientDist));
  app.get("*", (req, res) => {
    res.sendFile(path.join(clientDist, "index.html"));
  });
}
```

REGISTRATION ORDER IS NOT COSMETIC — IT DETERMINES BEHAVIOR

Express matches routes in the order they're registered, and `app.get("*", ...)` matches **every** path, including `/api/items`. If the catch-all were registered *before* the API routers, every single API request would be swallowed by

it, silently returning `index.html` instead of JSON — a real, subtle bug with no error message anywhere, just API calls that mysteriously "stop working" the moment static serving is added. The API routes must always be registered first.

Why the Catch-All Route Exists At All

React Router (or any client-side router) handles navigation entirely in the browser — a URL like `/history` never actually exists as a real file on the server. Without the catch-all, refreshing the page on `/history` would hit Express directly, find no matching route, and return a 404. `app.get("*", ...)` always returns `index.html` instead, letting React Router take over and render the correct view client-side once the page loads.

THE HONEST COST OF "NO BACKEND YOU WRITE" NOT APPLYING HERE

Food Tracker (React + Firebase)'s own deployment chapter got an SPA rewrite rule and automatic HTTPS from Firebase Hosting configuration alone — a few lines of JSON, no server code. This course has to hand-write both: the catch-all route above *is* this course's own SPA rewrite rule, and HTTPS termination (below) has to be arranged separately too, rather than coming free with the platform. Neither is difficult, but both are real, additional work this course's own architecture — "a server you write and run yourself," established back in Chapter 1 — genuinely requires that the Firebase sibling's architecture doesn't.

Environment Configuration and Process Management

```
# .env (production)
NODE_ENV=production
PORT=3001
```

A plain `node server/index.js` process exits the moment it crashes, and doesn't restart on its own. A process manager like **pm2** keeps the server running, restarting it automatically on a crash or a server reboot — the Node equivalent of what a platform's own process supervisor would otherwise provide:

```
npm install -g pm2
pm2 start server/index.js --name foodtracker
pm2 startup   # configures pm2 to launch on system boot
```

TLS Termination

Chapter 4's own `getUserMedia` requires HTTPS in production — no exception. Express itself doesn't handle TLS certificates; the standard approach is a reverse proxy (nginx, matching the pattern already covered on this site in Nginx In Depth) sitting in front of the Node process, terminating HTTPS and forwarding plain HTTP internally — the same general shape Food Tracker (Django)'s own deployment chapter used with gunicorn sitting behind a real web server, just with Node in place of gunicorn.

BETTER-SQLITE3'S FILE NEEDS A PERSISTENT VOLUME

Because the entire database is one file on disk, deploying to a platform with an **ephemeral filesystem** — one that wipes local storage on every redeploy or restart — would silently lose every item ever tracked. This is a different SQLite gotcha than Food Tracker (Django)'s own production concern (that course named SQLite's concurrency limits under multi-user load); here, at this app's own realistic single-household scale, concurrency was never really the risk — storage persistence is. Confirming the deployment target keeps the database file across restarts is a genuine, easy-to-overlook step.

Where This Course Is Headed

One chapter left: a capstone tying every chapter into one complete, working app.

Hands-On Exercises

EXERCISE 1

Explain exactly what would happen to a request for /api/items if the catch-all route were registered before the API routers instead of after, and why no error would appear anywhere to signal the problem.

View solution

EXERCISE 2

Explain why the catch-all route needs to exist at all, given that React Router already handles client-side navigation.

View solution

EXERCISE 3

Explain the difference between this chapter's own SQLite production concern (ephemeral storage) and Food Tracker (Django)'s own SQLite production concern (concurrency), and why each course names a different risk as the more relevant one.

View solution

CHAPTER 11 QUICK REFERENCE

- **Build:** npm run build produces client/dist/, served via express.static
- **Order matters:** API routes must be registered before the catch-all app.get("*", ...), or it swallows every API request

- **The catch-all's job:** returns index.html for any unmatched path, letting React Router handle client-side navigation on refresh
- **Honest cost:** this course hand-configures what Firebase Hosting provided automatically (SPA rewrite, HTTPS) for the Firebase sibling
- **Process management:** pm2 restarts the server automatically on crash or reboot
- **TLS:** a reverse proxy (nginx) in front of Node, the same general shape as the Django sibling's gunicorn-behind-a-web-server model
- **Real gotcha:** better-sqlite3's single database file needs a persistent volume — an ephemeral filesystem silently loses all data
- **Next chapter:** Capstone

CHAPTER 12 OF 12

Capstone: A Complete, Working Food Tracker

Food Tracker (React + Express)

Chapter 12 · Capstone: A Complete, Working Food Tracker

Marcus keeps his own household pantry tracked with the app this course just spent eleven chapters building — one Express process, one SQLite file, one React app, all deployed and running for real. What follows is one ordinary session with it.

STEP 1 — SCANNING A NEW ITEM

Marcus scans a carton of milk. Chapter 4's `useBarcodeScanner` hook decodes the barcode entirely client-side, then `handleDetected` calls `fetch("/api/lookup/...")` — Chapter 3's route, which checks `barcode_cache` first and falls back to a live Open Food Facts call, caching whatever it finds.

STEP 2 — ADDING IT

Chapter 5's `AddItemForm` pre-fills the name and category from that lookup. Marcus sets the expiry date and submits; the client-side check catches an empty name instantly, but it's Chapter 3's own `POST /api/items` route — the real gate, by construction — that actually validates and inserts the row, exactly as Chapter 5's finding-box promised it would.

STEP 3 — A FEW DAYS LATER, AN ALERT

Chapter 6's alerts route surfaces the milk once it's within three days of its (consistently `YYYY-MM-DD`-formatted) expiry date — a plain SQL range query needing no index at all, at this app's own realistic scale.

STEP 4 — SEARCHING FOR SOMETHING BOUGHT BEFORE

Wanting to check if he's bought a particular brand of yogurt before, Marcus types "yog" into the search box. Chapter 7's `useDebouncedSearch` waits for him to stop typing, then hits the search route's native, case-insensitive `LIKE` match — no shadow field required, unlike the Firebase sibling's own `nameLower`.

STEP 5 — MARKING THE MILK USED

The milk gets finished. Chapter 8's `PATCH /api/items/:id/use` — never a plain link — sets `status='used'`, stamps `used_at`, and clears `expiry_date` to `NULL`, guarded by `AND status = 'active'` so accidentally clicking twice changes nothing the second time.

STEP 6 — EVERYTHING UPDATES, WITHOUT BEING TOLD TO INDIVIDUALLY

Marking the milk used calls Chapter 10's `notifyChange()` — one call, with no knowledge of who's listening. The alerts dashboard disappears the milk from its own list; the recipe suggestions (Chapter 9) stop counting it as an expiring ingredient — both react on their own, independently, exactly as Chapter 10's whole redesign was built to make possible.

STEP 7 — A RECIPE SUGGESTION

With chicken and spinach both nearing expiry, Chapter 9's `Promise.all` fan-out queries TheMealDB for both concurrently — not sequentially, a real, honest advantage over the Django sibling's own admitted sequential cost — merging and sorting the results by how many expiring ingredients each recipe actually uses.

STEP 8 — ALL OF THIS, ACTUALLY DEPLOYED

Marcus's session happens against a real deployment: Express serving the built React app via Chapter 11's own catch-all SPA route (registered *after* the API routes, never before), pm2 keeping the process alive, nginx terminating HTTPS in front of it, and the SQLite file sitting on a volume that survives a redeploy.

Chapter Attribution

STEP	CHAPTER(S) APPLIED
1 — Scanning	Chapter 4 (camera scanning), Chapter 3 (lookup route + barcode_cache)
2 — Adding the item	Chapter 5 (AddItemForm, server-side validation), Chapter 2 (schema, INSERT)
3 — Expiry alert	Chapter 6 (date range query, YYYY-MM-DD discipline)
4 — Search	Chapter 7 (debounce hook, native LIKE)
5 — Marking used	Chapter 8 (PATCH route, active-only guard, expiry_date=NULL)

STEP	CHAPTER(S) APPLIED
6 — Automatic updates	Chapter 10 (ItemsContext, notifyChange, version)
7 — Recipe suggestion	Chapter 9 (Promise.all fan-out, recipe_cache, relevance sort)
8 — Real deployment	Chapter 11 (static serving, SPA fallback, pm2, TLS, persistent volume)

WHAT THIS WHOLE COURSE WAS REALLY ABOUT

Chapter 1 opened with a single claim: the same language runs on both sides of this app. Every step above is that claim paying off concretely — JSON needing no translation between client and server, Chapter 4's scanning hook plugging into this course's own Express route with a one-line change from its Firebase sibling, Chapter 9's `Promise.all` turning JavaScript's async-first design into a genuine performance advantage, and Chapter 10's Context-based coordination solving a real cross-feature problem with nothing more than React's own built-in tools. None of these payoffs required abandoning the honest limits named along the way — no ORM, no shadow-field-free search that scales infinitely, no free HTTPS — this course's own throughline was never "JavaScript solves everything," just "the same language, both ends, is a real and specific advantage where it actually applies."

HONEST SCOPE NOTE

This capstone deliberately stops short of several things: the weekly meal planner named as future work all the way back in Chapter 1 was never built; there's no offline/PWA support; no multi-user ownership model exists anywhere in this course — every item belongs to whoever can reach the server, matching the same honest gap named in Food Tracker (Django)'s own capstone, unlike the Firebase sibling course's own Chapter 11, which did implement real per-user Firebase Authentication; Chapter 9's own honest limit on matching generic branded product names against TheMealDB's fixed vocabulary was never solved, only named; and no automated test suite or CI pipeline was covered anywhere in this course.

Hands-On Exercises

EXERCISE 1

Trace Step 6 in detail: explain exactly what markUsed calls, and how that one call results in both the alerts dashboard and the recipe suggestions updating, without either being called directly.

View solution

EXERCISE 2

Pick any two steps from Marcus's session and explain how each one depends on at least two earlier chapters working together, not just one chapter in isolation.

View solution

EXERCISE 3

Explain why this course's honest scope note names the same missing multi-user ownership model as Food Tracker (Django)'s own capstone, while explicitly contrasting that against Food Tracker (React + Firebase), which did build real per-user authentication.

View solution

CHAPTER 12 QUICK REFERENCE — COURSE COMPLETE

- **8 steps, 11 prior chapters** — one continuous, realistic session with the finished, deployed app
- **This course's own throughline, closed out:** the same language, both ends, is a real, specific advantage — not a universal solution
- **Honest scope note:** no meal planner, no offline/PWA, no multi-user ownership model (same gap as the Django sibling), TheMealDB matching remains best-effort, no automated tests/CI
- **Food Tracker (React + Express) is now complete** — 12/12 chapters