

PROJECTS

Food Tracker (React + Express + Prisma)

A fifth companion to the Food Tracker quartet: the same React + Express app as its raw-SQL sibling, rebuilt with Prisma as the data layer throughout — schema modeling, migrations, barcode and recipe caching, expiry alerts, search, and deployment, closing with a full, honest scorecard of everywhere Prisma genuinely won, genuinely cost something, or changed nothing at all against the raw-SQL sibling course.

12 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: PROJECT OVERVIEW & PRISMA SETUP

Food Tracker (React + Express + Prisma)

Chapter 1 · Project Overview & Prisma Setup

This course is a direct companion to **Food Tracker (React + Express)** — the same app, the same React-plus-Express architecture, built a second time with one deliberate change: **Prisma in place of raw SQL via `better-sqlite3`**. That sibling course made "deliberately no ORM" its own honest choice; this one exists specifically to show what an ORM changes, and what it costs, for someone who already knows SQL well and wants real hands-on experience with the other side of that tradeoff.

What the App Actually Does

The spec is identical to the sibling course's own — this course is not re-deciding what the app does, only how its data layer is written:

- **Scan a barcode** with a phone or webcam camera, looking it up against **Open Food Facts** (free, open, no API key) to pull in 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 none — searchable in real time as you type.
- **Look up recipes** via **TheMealDB** (also free, no key) using ingredients close to expiring.

A weekly meal planner stays explicitly **out of scope**, matching every other Food Tracker course on this site.

Why This Course Exists: SQL You Already Know vs. an ORM You Don't

Food Tracker (React + Express) already makes the "same language, both ends" case for a Node/Express backend — that argument doesn't need repeating here, since this course shares it unchanged. What this course adds is a second, narrower comparison sitting entirely inside that same architecture: raw, hand-written SQL against Prisma's schema-first, generated-client approach to the exact same database.

ASPECT	FOOD TRACKER (REACT + EXPRESS)	THIS COURSE
Schema definition	Hand-written <code>CREATE TABLE</code> SQL in <code>schema.sql</code>	A <code>schema.prisma</code> model, one field per line, no SQL DDL written by hand
Applying schema changes	Manual <code>ALTER TABLE</code> statements — no migrations system	Prisma Migrate — every schema change becomes a real, versioned migration file, applied with <code>npx prisma migrate dev</code>
Querying	Raw SQL strings with <code>?</code> placeholders, run directly through <code>better-sqlite3</code>	Prisma Client's generated query methods — <code>create()</code> , <code>findMany()</code> , <code>update()</code> — built from the schema itself
Sync vs. async	Deliberately synchronous — <code>better-sqlite3</code> is unusual among Node DB libraries in having no <code>async</code> / <code>await</code>	Fully <code>async</code> / <code>await</code> , matching how most Node database tools actually work
Dev tooling	None beyond the SQL itself	<code>npx prisma studio</code> — a real, local GUI for browsing and editing the database

THE ONE-SENTENCE VERSION OF THIS WHOLE COURSE

Every route this course writes will look and behave differently from its sibling's own version of the same route — not because the app changed, but because **Prisma changes how you talk to the database**, from hand-written, synchronous SQL strings to generated, asynchronous, schema-derived method calls.

The Real Cost of an ORM, Named Honestly

Food Tracker (React + Express) names its own honest cost plainly: skipping an ORM means giving up migrations and a query builder, in exchange for total transparency about what SQL is actually running. Prisma's own honest cost runs the other way. Prisma Client's generated methods hide the actual SQL being executed — genuinely convenient day to day, but it's exactly the kind of abstraction that can let an **N+1 query problem** creep in unnoticed: a loop that looks like ordinary JavaScript can quietly issue one database round-trip per iteration instead of one query total, with nothing in the code itself flagging it. This course will call that out directly wherever it's a real risk, rather than presenting Prisma as free convenience with no downside.

Scaffolding the Project

The client side is identical to the sibling course — a Vite-powered React app, unaffected by what the backend does with its database:

```
# the React client — same as Food Tracker (React + Express)
npm create vite@latest client -- --template react
cd client && npm install
```

The server side starts the same way too, then adds Prisma on top:

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

# Prisma itself — a dev-time CLI, plus the client your route code actually imports
npm install prisma --save-dev
npm install @prisma/client

# scaffolds prisma/schema.prisma and a .env file, wired for SQLite
npx prisma init --datasource-provider sqlite
```

`prisma init` generates a near-empty starting schema — no `Item` model yet, since Chapter 2 is where the real data modeling happens:

```
// prisma/schema.prisma
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}
```

And the accompanying `.env` file, pointing Prisma at a local SQLite file the same way `better-sqlite3` pointed at `foodtracker.db` in the sibling course:

```
# .env
DATABASE_URL="file:./dev.db"
```

A minimal server, confirming everything works end to end before any real feature — or any actual database query — exists yet:

```
// 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 — UNCHANGED FROM THE SIBLING COURSE

Vite's dev server and Express run on two different ports (typically 5173 and 3001), which the browser treats as two different origins. Without `cors()`, every request from the React app to the Express API is blocked outright. Prisma doesn't touch this at all — it's purely a client/server-port issue, included here for completeness since this chapter may be read on its own without the sibling course alongside it.

TWO DEV-TIME TOOLS WORTH RUNNING EARLY

`node --watch server/index.js` restarts Express automatically on save, the same convenience Vite already gives the React side. And once Chapter 2 adds a real model, `npx prisma studio` opens a local, in-browser GUI for the actual SQLite database — genuinely useful for checking that a route did what it was supposed to, without writing a single SQL query to look.

Where This Course Is Headed

Data modeling with a real `schema.prisma` model and Prisma Migrate, barcode lookup with a Prisma-backed cache table, the camera-scanning React component (reused essentially unchanged — it's frontend-only, with no database layer involved), the add-item flow, expiry alerts built on Prisma's typed `DateTime` queries, item history with live search via Prisma's `contains` filter, marking items used, recipe lookup, cross-cutting state management, deployment (with Prisma's own build-time generation step added), and a capstone tying every chapter into one complete, working app — closing with an honest comparison back to the sibling course's own raw-SQL version.

Hands-On Exercises

EXERCISE 1

In one sentence, state this course's own core comparison claim against Food Tracker (React + Express). Then, using this chapter's own "Real Cost of an ORM" section, name the one concrete risk Prisma introduces that raw SQL doesn't have.

 [View solution](#)

EXERCISE 2

Explain the sync-vs-async difference between `better-sqlite3` and Prisma Client, and why it means a Prisma-backed route handler has to be written differently from its sibling's own version — even though both routes do the same thing.

 [View solution](#)

EXERCISE 3

Using this chapter's own compare-table, explain what problem Prisma Migrate actually solves that the sibling course's own "manual `ALTER TABLE` statements" approach doesn't.

 [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 — identical spec to Food Tracker (React + Express)
- **This course's own throughline:** the same app, the same React + Express architecture, with Prisma replacing raw SQL via `better-sqlite3`
- **Real payoff:** generated, type-shaped queries; Prisma Migrate's real, versioned schema changes; a GUI database browser via `npx prisma studio`
- **Honest cost:** the actual SQL is hidden behind Prisma Client, which is exactly what makes an N+1 query problem easy to introduce without noticing
- **Setup:** the same Vite React client as the sibling course; an Express server with `prisma` and `@prisma/client` installed, `prisma/schema.prisma` scaffolded via `npx prisma init --datasource-provider sqlite`
- **Next chapter:** Data Modeling with Prisma: schema.prisma & Prisma Migrate

CHAPTER 2: DATA MODELING WITH PRISMA

Food Tracker (React + Express + Prisma)

Chapter 2 · Data Modeling with Prisma: schema.prisma & Prisma Migrate

Every Food Tracker course stores the same shape of data. Food Tracker (React + Express) reaches for the plainest possible way to store it — raw SQL, no ORM at all. This chapter is where that choice actually diverges: the same schema, expressed instead as a Prisma model, migrated with Prisma Migrate, and queried through Prisma Client's generated methods.

The Shared Pantry Item Schema, in Prisma

The same fields the sibling course wrote as a raw `CREATE TABLE` statement, expressed here as a Prisma model:

```
// prisma/schema.prisma
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

model Item {
  id          Int           @id @default(autoincrement())
  name        String
  barcode     String?
  category    String?
  expiryDate  DateTime?
  status      String        @default("active")
  addedAt     DateTime      @default(now())
  usedAt      DateTime?
}
```

Every design decision from the sibling course's own Chapter 2 still applies exactly as reasoned there: `expiryDate` is nullable — `null` once an item is marked used, not deleted — and the row itself is never removed, so the combined history list (Chapter 7) always has something to show. `addedAt` defaults to the current time at insert via `@default(now())`, the direct Prisma equivalent of the sibling's own `DEFAULT (datetime('now'))` — the database layer stamps it either way, so the client is never trusted to supply an honest timestamp.

RAW SQL COLUMN (SIBLING COURSE)	PRISMA FIELD (THIS COURSE)	WHAT ACTUALLY CHANGED
<code>id INTEGER PRIMARY KEY AUTOINCREMENT</code>	<code>id Int @id @default(autoincrement())</code>	Same behavior — a type plus attributes, not a full DDL clause
<code>name TEXT NOT NULL</code>	<code>name String</code>	Non-nullable is the default for a plain type in Prisma; nullability is opt-in, not opt-out
<code>barcode TEXT (nullable)</code>	<code>barcode String?</code>	Nullability is the <code>?</code> on the type itself
<code>expiry_date TEXT (nullable)</code>	<code>expiryDate DateTime?</code>	A genuine type change — a real <code>DateTime</code> , not a formatted text string. Chapter 6 revisits exactly this distinction.
<code>status TEXT NOT NULL DEFAULT 'active'</code>	<code>status String @default("active")</code>	Same behavior
<code>added_at TEXT NOT NULL DEFAULT (datetime('now'))</code>	<code>addedAt DateTime @default(now())</code>	Same DB-stamped-timestamp idea; again, a real <code>DateTime</code> instead of a formatted string
<code>used_at TEXT (nullable)</code>	<code>usedAt DateTime?</code>	Same typed-vs-text distinction as <code>expiryDate</code>

A QUIETER CHANGE WORTH NOTICING: FIELD NAMING

Prisma's own convention is camelCase field names (`expiryDate`, `addedAt`, `usedAt`), not the sibling course's snake_case column names (`expiry_date`, `added_at`, `used_at`). That's not a cosmetic detail — it means every JSON response this course's API sends back uses different field names than the sibling's, and the React frontend has to match whichever backend it's actually talking to. A small, easy-to-miss consequence of adopting Prisma's own idiomatic style rather than mirroring the sibling's naming exactly.

Why Prisma Here

The sibling course's own reasoning still holds: Node has no single dominant ORM the way Django has its own built-in one — Prisma, Sequelize, Drizzle, and Knex all have real adoption, with no one of them as close to "the obvious choice" as Django's ORM is for Django. Prisma is used here specifically because it's the most widely adopted schema-first option in the current Node/TypeScript ecosystem, and because it's the same tool Website Rebuild with Next.js already uses elsewhere on this site — a genuine consistency benefit, not just a preference.

```
// db.js
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

export default prisma;
```

Applying the schema above means running an actual migration, not hand-writing an `ALTER TABLE` statement:

```
npx prisma migrate dev --name init
```

This creates a real, timestamped folder under `prisma/migrations/` containing the generated SQL for that specific change, applies it to the local `dev.db` file, records that it's been applied, and regenerates Prisma Client so its query methods match the schema exactly. Every future schema change — adding a field, changing a type — goes through the same command again, each one producing its own new migration file.

THE REAL COST OF PRISMA'S OWN SCHEMA-FIRST WORKFLOW

Editing `schema.prisma` alone changes nothing — the migration has to actually run before Prisma Client's generated methods reflect the new shape, and forgetting that step is a genuinely common source of confusing "field doesn't exist" errors that have nothing to do with the code being wrong. The sibling course has no equivalent extra step; a hand-written SQL query is correct or it isn't, with no separate generation stage in between.

Express Route Structure

The same route shape as the sibling course, rewritten against Prisma Client's generated, asynchronous methods:

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

const router = Router();

router.get("/", async (req, res) => {
  const items = await prisma.item.findMany({
    orderBy: { addedAt: "desc" }
  });
  res.json(items);
});

router.post("/", async (req, res) => {
  const { name, barcode, category, expiryDate } = req.body;
  const item = await prisma.item.create({
    data: { name, barcode, category, expiryDate }
  });
  res.status(201).json(item);
});

export default router;

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

Two concrete differences worth naming against the sibling's own version of the exact same route. First, both handlers are declared `async`, with `await` in front of every Prisma call — the sync-vs-async split Chapter 1 already covered. Second, the `POST` handler here responds with the *entire created item*, since `create()` returns the full row by default; the sibling's own `better-sqlite3` version could only return `{ id: result.lastInsertRowid }`, since a raw `INSERT` doesn't hand back the row it just created.

On SQL injection: the sibling course's protection comes from remembering to always use `?` placeholders instead of string-concatenating values into a query — a real safeguard, but one the developer has to keep applying correctly every time. Prisma Client's query methods take structured objects, not raw query strings, so there's no string-concatenation path available through the normal API at all. The same protection exists, but here it's structural rather than a discipline to maintain.

THIS CHAPTER'S OWN HONEST TRADEOFF, STATED PLAINLY

Prisma trades the sibling course's total transparency — every query is exactly the SQL string sitting right there in the route handler — for real migrations, a generated query API, and SQL-injection protection that's structural rather than a habit to maintain. The cost, named honestly in the warn-box above: a schema change now requires an explicit migration step before it takes effect, where the sibling course's raw SQL takes effect the moment the query runs.

Where This Course Is Headed

Barcode lookup next, with a `BarcodeCache` model reusing this chapter's own schema pattern, then the camera-scanning React component — reused essentially unchanged from the sibling course, since it's frontend-only and never touches the database layer.

Hands-On Exercises

EXERCISE 1

Using this chapter's own compare-table, explain what actually changes about the `expiryDate` field going from the sibling course's `TEXT` column to this course's `DateTime?` field — and why that change matters for later chapters, not just this one.

 [View solution](#)

EXERCISE 2

Explain the field-naming difference between this course (`expiryDate` , `addedAt`) and the sibling course (`expiry_date` , `added_at`), and what concrete part of the app that difference actually affects.

[View solution](#)

EXERCISE 3

Explain why the POST route in this chapter can respond with the full created item, while the sibling course's own POST route can only respond with an id — and name the real, concrete step (not present in the sibling course) that has to happen before a `schema.prisma` change actually takes effect.

[View solution](#)

CHAPTER 2 QUICK REFERENCE

- **Schema:** the same fields as every sibling course, defined as an `Item` model in `schema.prisma` — `id, name, barcode, category, expiryDate, status, addedAt, usedAt`
- **Real type change:** `expiryDate` / `usedAt` / `addedAt` are genuine `DateTime` fields here, not formatted text strings — Chapter 6 builds directly on this
- **Naming:** camelCase fields (Prisma's own convention) instead of the sibling's snake_case columns — a real difference in every JSON response
- **Migrations:** `npx prisma migrate dev --name init` — versioned, tracked, real SQL generated automatically
- **Real cost:** `schema.prisma` changes do nothing on their own until a migration actually runs
- **Routes:** the same Router-per-resource pattern as the sibling, now `async` / `await` throughout
- **Security:** SQL injection protection is structural here — Prisma Client's methods take structured data, not raw query strings
- **Next chapter:** Barcode Lookup: Integrating Open Food Facts

CHAPTER 3: BARCODE LOOKUP: INTEGRATING OPEN FOOD FACTS

Food Tracker (React + Express + Prisma)

Chapter 3 · Barcode Lookup: Integrating Open Food Facts

The same integration as the sibling course, for the same reasons — a server-side proxy to Open Food Facts, not a direct call from React. What changes here is the cache itself: a Prisma model instead of a raw SQL table, and a genuinely safer write pattern than the sibling's own check-then-insert.

Why Proxy Through the Server At All

Open Food Facts needs no API key, so this isn't about hiding a secret. The real reasons are unchanged from the sibling course:

- **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 already knows about.
- **Consistency.** Every client gets identical lookup behavior, defined in one place.
- **Future-proofing.** If Open Food Facts' own API shape changes, or a second data source gets added later, only the server needs to change.

Extending the Schema: A Barcode Cache

A second model in `schema.prisma`, reusing Chapter 2's own field-naming convention:

```
// prisma/schema.prisma (appended)
model BarcodeCache {
  barcode String @id
  name     String?
  category String?
  cachedAt DateTime @default(now())
}
```

```
npx prisma migrate dev --name add_barcode_cache
```

`barcode` is the primary key here — deliberately different from the `Item` model, 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 prisma from "../db.js";

const router = Router();

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

  const cached = await prisma.barcodeCache.findUnique({ where: { 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;

  const saved = await prisma.barcodeCache.upsert({
    where: { barcode },
    update: { name, category },
    create: { barcode, name, category }
  });
});
```

```
    res.json(saved);
  });

export default router;

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

The check-first-then-fetch shape is identical to the sibling course's own route. The write at the end is where this course diverges: `upsert()` instead of a plain insert.

	SIBLING COURSE	THIS COURSE
Cache check	<code>db.prepare(...).get(barcode)</code>	<code>prisma.barcodeCache.findUnique(...)</code>
Cache write	A plain <code>INSERT</code> , run only after the check above found nothing	<code>upsert()</code> — insert if the barcode doesn't exist, update it if it does
Two requests for the same brand-new barcode, near-simultaneously	Both pass the initial check (neither sees a cache hit yet). The first <code>INSERT</code> succeeds; the second violates the <code>barcode</code> primary key and fails.	Both pass the initial check the same way. The first <code>upsert()</code> creates the row; the second <code>upsert()</code> for the same barcode simply updates it instead of failing.

A PLACE PRISMA'S OWN ABSTRACTION GENUINELY BUYS SOMETHING

This is a real edge case in the sibling course's own design, not a hypothetical one: its check-then-insert pattern has an honest gap the moment two requests for a never-before-cached barcode land close enough together, since `barcode` is declared `PRIMARY KEY` there too. Prisma's `upsert()` is built specifically for exactly this "write it if it's missing, otherwise that's fine" situation — a case where reaching for the ORM's own generated method is a genuine improvement over the raw SQL equivalent, not just a different way of writing the same thing.

NO EXTRA PACKAGE NEEDED FOR THE HTTP CALL — UNCHANGED FROM THE SIBLING

`fetch` is a genuine Node.js global as of Node 18, working identically here whether or not Prisma is involved. This has nothing to do with the database layer at all — it's the same fact, reused.

OPEN FOOD FACTS' DATA IS GENUINELY INCONSISTENT — UNCHANGED FROM THE SIBLING

Because Open Food Facts is crowdsourced, 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 (Prisma's `String?` fields accept that directly) rather than throwing — Chapter 5's own add-item form has to be built expecting that, exactly as in the sibling course.

Where This Course Is Headed

The camera-scanning React component next — reused essentially unchanged from the sibling course, since decoding a barcode client-side has nothing to do with which backend, or which database layer, 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

Using this chapter's own compare-table, explain the exact race condition the sibling course's check-then-insert pattern can hit, and why `upsert()` doesn't have the same problem.

[View solution](#)

EXERCISE 3

Explain why barcode is the `@id` (primary key) on `BarcodeCache` but not on `Item`, even though both models have a barcode field.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- **Why proxy:** caching, consistency across clients, future-proofing — not secrecy (no API key needed)
- **New model:** `BarcodeCache`, keyed by `barcode` — one lookup per barcode, unlike `Item`
- **Route:** `GET /api/lookup/:barcode` — checks the cache first via `findUnique()`, else calls Open Food Facts and saves the result
- **Real Prisma win:** `upsert()` closes a genuine race condition the sibling's own check-then-insert pattern has for a brand-new barcode requested twice at once
- **Node's built-in fetch:** no axios/node-fetch dependency needed since Node 18 — unrelated to the database layer
- **Real gotcha, unchanged:** 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: CAMERA-BASED BARCODE SCANNING IN REACT

Food Tracker (React + Express + Prisma)

Chapter 4 · Camera-Based Barcode Scanning in React

This is the one chapter in this course with no Express, no Prisma, and no database involvement of any kind — everything here runs entirely in the browser. It's also the third time this exact component has been built across the Food Tracker family: Food Tracker (React + Firebase)'s own Chapter 4 built it first, Food Tracker (React + Express)'s own Chapter 4 reused it unchanged, and what follows here is that same code again, unmodified a second time.

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.

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 ORM, no database client of any kind, 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, for a third time in a row.

Wiring It to This Course's Own Backend

Everything above this point is identical, character for character, to both sibling courses. Here's the parent component that wires it up:

```
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 instead of a plain `fetch` — the entire difference between those two courses' use of this component came down to that one line. Compare it instead to Food Tracker (React + Express)'s own `handleDetected`, and there's **no difference at all**: the exact same `fetch("/api/lookup/${barcode}")` call, because Chapter 3's route path never changed between the two courses — only what happens *inside* that route, behind the same URL, changed from raw SQL to Prisma.

THE POINT OF BUILDING THE IDENTICAL COMPONENT A THIRD TIME

Three Food Tracker courses now share this exact hook: one calling a Cloud Function, two calling the identical REST endpoint over two completely different data-access technologies underneath it. The

lesson from the sibling course still holds and gets sharper here — a well-designed frontend component doesn't need to know or care what's on the other end of its one callback prop, and it turns out it doesn't even need to know whether that server is talking to the database in raw SQL or through an ORM. The backend's own internal implementation is a genuinely swappable detail this component was never built around in the first place.

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) 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 Prisma-backed `POST` route, taking this scanned barcode's lookup result the rest of the way into the database.

Hands-On Exercises

EXERCISE 1

Explain why this chapter's ScanScreen needs zero changes — not even the one line that differed between the Firebase and Express siblings — despite this course's own backend switching from raw SQL to Prisma between Chapters 2 and 3.

 [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 a third time, not rebuilt** — identical to both Food Tracker (React + Firebase) and Food Tracker (React + Express) 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
- **What differs from the raw-SQL sibling here:** nothing — same URL, same route path, same hook, since Prisma changed what happens inside Chapter 3's route, not its shape
- **Next chapter:** Building the Add-Item Flow

CHAPTER 5: BUILDING THE ADD-ITEM FLOW

Food Tracker (React + Express + Prisma)

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 via Chapter 2's `Item` model. The form itself is identical in spirit to the sibling course's own; what genuinely changes is what happens once a submission reaches the server.

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:

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

function AddItemForm({ initialData = {}, onSave }) {
  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, expiryDate: expiryDate || null })
    });
  };
}
```

```

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

    onSave(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>
  );
}

```

A SMALL, CONCRETE PAYOFF OF CHAPTER 2'S OWN NAMING CHOICE

Look closely at the request body: `{ name, barcode, category, expiryDate: expiryDate || null }` — the exact same field name, `expiryDate`, all the way from the React state variable to the JSON key to the Prisma field it lands in. The sibling course's own version of this same line has to write `expiry_date: expiryDate || null` — translating its camelCase React state into the snake_case key its raw-SQL backend expects. That translation step is small, but it's a real extra thing to get right on every field, every time, and this course simply doesn't have it, purely as a side effect of Prisma's own camelCase convention matching JavaScript's own.

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("/", async (req, res) => {
  const { name, barcode, category, expiryDate } = req.body;

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

  const item = await prisma.item.create({
    data: {
      name: name.trim(),
      barcode: barcode || null,
      category: category || null,
      expiryDate: expiryDate ? new Date(expiryDate) : null
    }
  });

  res.status(201).json(item);
});
```

Two differences from the sibling's own version of this route worth naming directly. First, the explicit `Date.parse` check still runs here — Prisma's own type system does add a real backstop underneath it (passing a genuinely malformed value into a `DateTime` field raises a real validation error from Prisma Client itself), but that error has no useful HTTP status or message attached to it by default, so the explicit check stays because it produces a clean `400` response the raw Prisma error wouldn't. Second, the response is just `res.status(201).json(item)` — the whole created row, returned automatically by `create()`, rather than a response object assembled field by field.

WHAT THE SIBLING'S OWN MANUALLY-ASSEMBLED RESPONSE ACTUALLY MISSES

The sibling course's `POST` handler responds with `{ id: result.lastInsertRowid, name, barcode, category, expiry_date }` — built by hand from the variables already in scope. Look at what's missing: `status` and `added_at` are both real columns on every item, and neither one makes it into that response, simply because nobody added them to the object literal. This course's `res.status(201).json(item)` can't have that specific bug — the whole row Prisma just created is what gets sent back, automatically, whether or not every field was explicitly remembered.

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. Neither this course nor its raw-SQL sibling ever had that gap: every write already passes through an Express route, because that's simply how this architecture works, regardless of what's on the other side of that route's own database call.

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 — a Prisma query using `gte` / `lte` against the real `DateTime` field this course has had since Chapter 2, 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

Using this chapter's own finding-box, explain exactly what's missing from the sibling course's manually-assembled POST response, and why this course's `res.status(201).json(item)` can't have the same bug.

 [View solution](#)

EXERCISE 3

Explain why the explicit `isNaN(Date.parse(expiryDate))` check still belongs in this route even though Prisma's own type system would reject a genuinely malformed date value on its own.

 [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)
- **No field-name translation:** `expiryDate` is the same name in React state, the JSON body, and the Prisma field — unlike the sibling's camelCase-to-snake_case step
- **Client-side validation:** real UX value (instant feedback), zero security value (trivially bypassable)
- **Server-side validation:** the actual gate — an explicit check still needed for a clean 400, even with Prisma's own type system underneath it
- **Real correctness win:** `create()`'s full-row response can't accidentally omit a field the way the sibling's hand-assembled response does (it's missing status and added_at)
- **This course's own architectural advantage, shared with the sibling:** every write already passes through Express by construction — no separate Security Rules layer needed, unlike the Firebase sibling

- **Next chapter:** Expiry Alerts

CHAPTER 6: EXPIRY ALERTS WITH PRISMA'S TYPED DATE QUERIES

Food Tracker (React + Express + Prisma)

Chapter 6 · Expiry Alerts with Prisma's Typed Date Queries

Every item added in Chapter 5 now has a real row, with a real (or `null`) `expiryDate`. This chapter turns that stored date into something useful — and it's where Chapter 2's own promise about a real `DateTime` field, instead of a formatted text string, actually pays off.

The Alerts Route

```
// routes/items.js (appended)
router.get("/alerts", async (req, res) => {
  const threeDaysFromNow = new Date(Date.now() + 3 * 24 * 60 * 60 * 1000);

  const items = await prisma.item.findMany({
    where: {
      status: "active",
      expiryDate: {
        not: null,
        lte: threeDaysFromNow
      }
    },
    orderBy: { expiryDate: "asc" }
  });

  res.json(items);
});
```

`status: "active"` excludes anything already marked used (Chapter 8's own territory); `not: null` excludes items with no expiry date at all — both conditions matter for the same reasons as the sibling course. `lte: threeDaysFromNow` is where this route genuinely diverges: instead of SQLite's own `date('now', '+3 days')` computed inside the query string, the three-day window is computed as a real JS `Date` in ordinary JavaScript, then handed to Prisma as a query parameter.

CHAPTER 2'S OWN PROMISE, PAID OFF

The sibling course's real gotcha: its `expiry_date <= date('now', '+3 days')` comparison only gives the correct chronological answer because every date happens to be stored as `YYYY-MM-DD` text, consistently — a format lexicographic comparison agrees with, purely by discipline. Nothing in that schema stops a future bug from inserting a date in a different format and silently breaking every comparison, with no error anywhere. This course's `expiryDate` field is typed as `DateTime` in `schema.prisma`, and every write to it goes through Prisma's own serialization — there's no code path in this app that can write an `expiryDate` value in some other, inconsistent shape. Under the hood, on SQLite, the comparison still ultimately relies on consistently-formatted stored values, exactly like the sibling's own comparison does — the real difference is that here, *consistency is structurally guaranteed by the type system*, not maintained by every developer remembering to format dates the same way every time.

NO COMPOSITE INDEX NEEDED HERE EITHER — UNRELATED TO PRISMA, STILL TRUE

Food Tracker (React + Firebase)'s own equivalent query needs a composite index, because Firestore requires one whenever a query filters on one field and orders by another simultaneously. Neither this course nor its raw-SQL sibling has that requirement — SQLite runs a query like this correctly with no index declared at all, and at this app's own realistic scale (a single household's pantry), a full table scan is genuinely fast enough not to matter. This has nothing to do with Prisma specifically; it's the same underlying SQLite engine either way, just reached through a different query API.

A Custom Hook for the Dashboard

Unchanged from the sibling course — this hook has no idea what database, or what query engine, sits behind the route it calls:

```
// 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());
  }, []);

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

```

    setLoading(false);
  }, []);

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

  return { alerts, loading, refresh };
}

```

`refresh` is exposed deliberately, not just called once internally — Chapter 8's own "mark used" action needs a way to trigger a fresh alerts fetch immediately afterward.

The Dashboard Component

```

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.expiryDate.slice(0, 10)}
        </li>
      ))}
    </ul>
  );
}

```

A REAL, HONEST COST OF THE DATETIME TYPE: DISPLAY FORMATTING

The sibling course's `item.expiry_date` is a plain `"2026-09-01"` string, ready to display as-is. This course's `item.expiryDate` is a full ISO-8601 timestamp once it's traveled through Prisma and JSON — something like `"2026-09-01T00:00:00.000Z"` — because a JS `Date` object always serializes to a complete timestamp, not just a date. Displaying it directly would show that whole string, time component and all; `.slice(0, 10)` above is the plainest fix, grabbing just the date portion. The sibling course never has to do this. This is the honest flip side of Chapter 2's own typed-field payoff: a

real type buys correctness at the storage and query layer, and adds a small formatting step at the display layer that a plain text field never needed.

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

`3 * 24 * 60 * 60 * 1000` is written directly in the route above for clarity in this chapter, but a real app should pull that number from one shared constant rather than repeating the literal calculation anywhere the query might be duplicated later.

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 backed by Prisma's own `contains` filter.

Hands-On Exercises

EXERCISE 1

Using this chapter's own finding-box, explain the real difference between "this comparison happens to work because every date is formatted consistently" (the sibling course) and "this comparison is guaranteed to work because of the field's own type" (this course) — even though both ultimately store the date as text in SQLite.

 [View solution](#)

EXERCISE 2

Explain why this course's dashboard needs `item.expiryDate.slice(0, 10)` while the sibling course's dashboard can display `item.expiry_date` directly — what's actually different about the value each one receives?

 [View solution](#)

EXERCISE 3

Explain why "three days from now" is computed in plain JavaScript in this course's route, rather than inside the query the way the sibling's `date('now', '+3 days')` does it — and name one real consequence of computing it that way instead.

[View solution](#)

CHAPTER 6 QUICK REFERENCE

- **Route:** GET `/api/items/alerts` — status: "active", expiryDate: { not: null, lte: threeDaysFromNow }, ordered ascending
- **Chapter 2's payoff:** a typed `DateTime` field structurally guarantees consistent date formatting — the sibling's identical-looking comparison only works by developer discipline
- **Real new cost:** a `DateTime` value serializes to a full ISO timestamp in JSON, so the dashboard needs `.slice(0, 10)` (or real formatting) the sibling never needed
- **Where the date math happens:** in JavaScript here, inside the SQL string in the sibling — unrelated to Prisma vs. raw SQL specifically, just a different query API's own capabilities
- **No composite index needed:** same as the sibling — a SQLite fact, not a Prisma one, still a real contrast against the Firebase sibling
- **useExpiryAlerts:** unchanged from the sibling course — a custom hook exposing alerts, loading, and a callable refresh
- **Next chapter:** Item History & Live Search-as-You-Type

CHAPTER 7: ITEM HISTORY & LIVE SEARCH-AS-YOU-TYPE

Food Tracker (React + Express + Prisma)

Chapter 7 · Item History & Live Search-as-You-Type with Prisma's contains Filter

Every item ever added stays in the database forever — Chapter 2's own design decision, active or used. This chapter surfaces that whole history, searchable in real time — and it's the one chapter in this course where a naive Prisma port genuinely regresses a feature that already worked correctly in the sibling course, verified directly against Prisma's own current documentation rather than assumed.

The Search Route — First Pass

```
// routes/items.js (appended)
router.get("/search", async (req, res) => {
  const q = req.query.q || "";
  const items = await prisma.item.findMany({
    where: { name: { contains: q } },
    orderBy: { addedAt: "desc" },
    take: 50
  });
  res.json(items);
});
```

Deliberately no `status` filter, matching the sibling course exactly — 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.

A Real, Verified Regression

The sibling course's own `LIKE '%' || ? || '%'` is case-insensitive for ASCII automatically — a genuine, well-known SQLite behavior. It's tempting to assume Prisma's `contains` just carries that same behavior forward on top of the identical SQLite database. Checked directly against Prisma's own current documentation, it doesn't:

VERIFIED AGAINST PRISMA'S OWN DOCS, NOT ASSUMED

Prisma's documentation states it plainly for the SQLite provider: *"By default, text fields created by Prisma Client in SQLite databases do not support case-insensitive filtering."* The `mode: "insensitive"` option that fixes this cleanly on PostgreSQL or MongoDB **is not supported on SQLite at all**. So the route above, exactly as written, is *case-sensitive* — searching `"milk"` would **not** find an item named `"Milk"`. This is a genuine functional regression against the sibling course's own working search, not a hypothetical edge case.

WHY THIS IS WORTH STOPPING ON

Every other chapter in this course has framed Prisma's own costs as things that are inconvenient, or that need an extra step, but never things that silently break a feature the sibling already got right for free. This one does exactly that, unless it's caught and fixed. It's a genuine example of why porting an app to an ORM isn't purely mechanical — a query that looks like a faithful translation of the original SQL can behave differently in a way that only shows up when someone actually searches for something.

The Real Fix: COLLATE NOCASE

Prisma's own documented fix is to add SQLite's `COLLATE NOCASE` to the column itself, restoring ASCII case-insensitive comparison for every filter type — `contains`, `startsWith`, `endsWith`, and `equals` alike. Because `name` already exists as a plain column since Chapter 2, and SQLite doesn't support altering a column's collation in place, this means generating a migration without applying it, editing the SQL by hand, then applying the edited version:

```
# generates the migration file without running it
npx prisma migrate dev --name add_name_collate_nocase --create-only
```

Then editing the generated `migration.sql` to recreate the table with `name` declared `COLLATE NOCASE` — SQLite's own standard pattern for changing an existing column, since columns can't be altered directly (simplified here to the essential steps; a production migration would also need to reapply any indexes):

```
-- migration.sql (hand-edited)
PRAGMA foreign_keys=OFF;
```

```
CREATE TABLE "new_Item" (
  "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
  "name" TEXT NOT NULL COLLATE NOCASE,
  "barcode" TEXT,
  "category" TEXT,
  "expiryDate" DATETIME,
  "status" TEXT NOT NULL DEFAULT 'active',
  "addedAt" DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  "usedAt" DATETIME
);

INSERT INTO "new_Item" SELECT * FROM "Item";
DROP TABLE "Item";
ALTER TABLE "new_Item" RENAME TO "Item";

PRAGMA foreign_keys=ON;
```

```
# applies the hand-edited migration
npx prisma migrate dev
```

With that migration applied, the exact same route code from the first pass — nothing about the query itself changes — now matches case-insensitively, because the comparison behavior lives in the column's own collation, not in the query.

--CREATE-ONLY EXISTS SPECIFICALLY FOR THIS

Prisma normally applies a migration immediately after generating it. `--create-only` generates the SQL file and stops there, deliberately leaving room to hand-edit it before anything touches the real database — exactly the workflow this fix needs, and a real, documented part of the Prisma CLI, not a workaround.

A Reusable Debounce Hook

Unchanged from the sibling course — firing a search request on every keystroke would flood the server, so this 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}`}
        </li>
      ))}
    </ul>
  </div>
);
}

```

THE HONEST LIMIT OF CONTAINS, UNRELATED TO PRISMA SPECIFICALLY

A leading wildcard — what `contains` compiles down to — can never use a standard database index efficiently, even with one declared on `name`, since a match could start anywhere in the string. The same real limitation the sibling course names for its own raw `LIKE '%...%'` applies here identically; it's a fact about this query shape on SQLite, not something Prisma changes either way. At this app's own realistic scale, that cost is genuinely invisible.

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. 200–400ms is a common, comfortable range — 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 a Prisma-backed `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 the real regression this chapter found: why does the naive contains-based search route fail to match "Milk" when a user searches "milk," and why wouldn't simply adding mode: "insensitive" to the query fix it?

 [View solution](#)

EXERCISE 2

Explain why fixing this requires hand-editing a generated migration file with COLLATE NOCASE, rather than a change to schema.prisma or to the search route's own query code.

 [View solution](#)

EXERCISE 3

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](#)

CHAPTER 7 QUICK REFERENCE

- **Route:** GET /api/items/search?q=... — prisma.item.findMany with contains, no status filter, the full history
- **Real, verified regression:** Prisma's contains defaults to case-sensitive on SQLite; mode: "insensitive" isn't supported on SQLite at all — unlike the sibling's automatically case-insensitive raw LIKE
- **The real fix:** COLLATE NOCASE added to the column via a hand-edited migration (prisma migrate dev --create-only, edit the SQL, then apply it)
- **Honest limit, unchanged from the sibling:** a leading wildcard can't use an index — a SQLite fact, not a Prisma one
- **useDebouncedSearch:** unchanged from the sibling course — cleanup cancels the previous pending timer on every keystroke

- **Next chapter:** Marking Items Used

CHAPTER 8: MARKING ITEMS USED

Food Tracker (React + Express + Prisma)

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 sibling course does this with one atomic `UPDATE ... WHERE id = ? AND status = 'active'`. Replicating that single-query guard through Prisma turns out to need a specific, real feature — verified directly against Prisma's own documentation and issue history rather than assumed.

The Route

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

  try {
    const item = await prisma.item.update({
      where: {
        id: Number(id),
        status: "active"
      },
      data: {
        status: "used",
        usedAt: new Date(),
        expiryDate: null
      }
    });
    res.json(item);
  } catch (error) {
    if (error.code === "P2025") {
      return res.status(404).json({ error: "Item not found, or already used" });
    }
    throw error;
  }
});
```

```
}  
});
```

`where: { id: Number(id), status: "active" }` is doing the same job as the sibling's own `WHERE id = ? AND status = 'active'` — a single, atomic query that only matches a row currently active *and* with the requested id, run as one statement rather than a check followed by a separate write.

VERIFIED AGAINST PRISMA'S OWN DOCS AND ISSUE HISTORY, NOT ASSUMED

Prisma's `update()` normally only accepts unique fields (like `id`) in its `where` clause — a plain field like `status` can't be added to it on its own. Combining a non-unique field alongside the actual unique identifier, exactly as done here, is a real, specific Prisma capability: introduced as a preview feature (`extendedWhereUnique`) in Prisma 4.5.0, and made stable — no preview flag needed — as of **Prisma 5.0.0**. Without it, this guard would need `updateMany()` instead, which can filter on any field but doesn't return the updated record directly.

	SIBLING COURSE	THIS COURSE
The guard itself	<code>WHERE id = ? AND status = 'active'</code> in one <code>UPDATE</code>	<code>where: { id, status: "active" }</code> in one <code>update()</code> — same atomic single-query shape
Signal when the guard fails	<code>result.changes === 0</code> — a value the route checks with an <code>if</code>	A thrown <code>PrismaClientKnownRequestError</code> with code <code>P2025</code> — an error the route catches with <code>try / catch</code>
The id itself	<code>id</code> passed straight through as a string — SQLite's own flexible typing handles the comparison	<code>Number(id)</code> — Prisma's typed <code>Int</code> field expects a real number, not a numeric string

That last row matters in practice, not just in theory: forgetting `Number(id)` here is a genuinely easy mistake, since `req.params.id` is always a string regardless of what the underlying field's own type is.

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` 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.expiryDate.slice(0, 10)}
          <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 `status: "active"` filter — it disappears from the dashboard immediately, with 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 specific Prisma feature makes `where: { id, status: "active" }` possible on `update()`, and which Prisma version made it stable without a preview flag.

[!\[\]\(34543dd4ff7f078317aba2ea094681a5_img.jpg\) View solution](#)**EXERCISE 2**

Using this chapter's own `compare-table`, explain why this route needs a `try/catch` rather than an `if (result.changes === 0)` check, and what `P2025` actually means.

[!\[\]\(9cbf1bb4a206d9681b6d411f6f46a945_img.jpg\) View solution](#)**EXERCISE 3**

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.

[!\[\]\(4b405757c230db049e58f47a9672bc64_img.jpg\) View solution](#)

CHAPTER 8 QUICK REFERENCE

- **Route:** PATCH /api/items/:id/use — update() with where: { id, status: "active" }, clearing expiryDate to null
- **The real feature:** non-unique fields alongside a unique id in update()'s where clause — stable since Prisma 5.0.0
- **Different failure signal:** a thrown P2025 error here, a returned changes: 0 in the sibling — same atomic guard, different way of reporting "no match"
- **Easy-to-forget detail:** Number(id) — req.params.id is always a string; the sibling's SQLite comparison tolerates that, Prisma's typed Int field doesn't
- **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
- **Next chapter:** Recipe Lookup with TheMealDB

CHAPTER 9: RECIPE LOOKUP WITH THEMEALDB

Food Tracker (React + Express + Prisma)

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 — reusing Chapter 3's own caching pattern, and Chapter 1's own concurrency story, largely unchanged.

Extending the Schema: A Recipe Cache

The same shape as Chapter 3's own `BarcodeCache`, keyed by ingredient this time:

```
// prisma/schema.prisma (appended)
model RecipeCache {
  ingredient String @id
  meals      String // JSON array, stored as text – see note below
  cachedAt   DateTime @default(now())
}
```

WHY MEALS IS A PLAIN STRING, NOT PRISMA'S OWN JSON TYPE

Prisma does have a dedicated `Json` field type — but checked directly against real, current Prisma issue history, its support for the SQLite provider specifically has been genuinely inconsistent across versions, with open bug reports even where partial support exists. Rather than depend on a still-uneven feature, this model stores the meal list the same way the sibling course does — as a plain string, serialized and parsed by hand — for portability and to avoid a dependency this course can't fully vouch for.

The Suggestion Route

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

```

const router = Router();

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

  const cached = await prisma.recipeCache.findUnique({ where: { ingredient: 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 || [];

  await prisma.recipeCache.upsert({
    where: { ingredient: key },
    update: { meals: JSON.stringify(meals) },
    create: { ingredient: key, meals: JSON.stringify(meals) }
  });

  return meals;
}

router.get("/suggest", async (req, res) => {
  const threeDaysFromNow = new Date(Date.now() + 3 * 24 * 60 * 60 * 1000);

  const expiring = await prisma.item.findMany({
    where: {
      status: "active",
      expiryDate: { not: null, lte: threeDaysFromNow }
    },
    select: { name: true }
  });

  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;

```

`upsert()` for the cache write is the exact same choice Chapter 3 made for `BarcodeCache`, and for the same reason — it safely handles two concurrent lookups of the same never-before-cached ingredient, where a plain create could collide on the `ingredient` primary key. `select: { name: true }` narrows the alerts-style query to just the one field this route actually needs, matching the sibling's own `SELECT name FROM items` — both courses already avoid pulling the whole row here.

A REAL ADVANTAGE THAT HAS NOTHING TO DO WITH PRISMA

Food Tracker (Django) named its own equivalent fan-out honestly as sequential — one ingredient's TheMealDB call waiting for the previous one to finish. Here, exactly as in the sibling course, `Promise.all` fires every ingredient's lookup **concurrently**, because Node's own async model makes that the natural way to write it. This win comes from Node and JavaScript, not from Prisma — it's identical in this course and its raw-SQL sibling, since both are Express apps running on the same runtime.

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

Unchanged from the sibling course — nothing here touches a date field or a naming convention that differs between the two courses:

```
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 >
        </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 `BarcodeCache`.

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 `RecipeCache` stores meals as a plain `String` rather than Prisma's own `Json` field type, given what this chapter found checking Prisma's real issue history.

 [View solution](#)

EXERCISE 3

Explain why the cache write here uses `upsert()`, tracing the reasoning back to the specific chapter that first introduced this pattern and the specific problem it solves.

 [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 model:** `RecipeCache`, keyed by ingredient, meals stored as a JSON-encoded `String` (not Prisma's `Json` type — inconsistent SQLite support, checked directly)
- **Reused pattern:** `upsert()` for the cache write, same as Chapter 3's `BarcodeCache`, same reason
- **Real advantage, unrelated to Prisma:** `Promise.all` fires every ingredient lookup concurrently — a Node/JavaScript fact, identical in both this course and its raw-SQL sibling
- **Real gotcha, unchanged:** `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: STATE MANAGEMENT ACROSS THE APP

Food Tracker (React + Express + Prisma)

Chapter 10 · State Management Across the App

The second chapter in this course reused essentially unchanged from the sibling, alongside Chapter 4's camera scanner. Nothing here touches a route, a schema, or a query — this is React state coordination, several steps removed from whichever database technology actually handles a mutation.

The Problem, Concretely

Every feature so far has managed its own state: `useExpiryAlerts`, `useDebounceSearch`, `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. 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:

```
// 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:

```
// 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:

```
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 $O(\text{features} \times \text{mutations})$ 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 — and entirely independently of whether that action's own route ran a raw SQL statement or a Prisma query underneath it.

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 — 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 against SQLite on each relevant change is genuinely fast enough not to need it — true whether that re-fetch runs through Prisma or through raw SQL.

Where This Course Is Headed

Deployment next — serving the built React app from the same Express process, with Prisma's own build-time generation step added into that pipeline.

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.

[!\[\]\(5ac3086483c0adb44d83d000c5b59c1b_img.jpg\) 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.

[!\[\]\(64e79d60bec7204428db2687d5804419_img.jpg\) 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.

[!\[\]\(bfcdde234195490afd1bc42918901886_img.jpg\) 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, or how the route it called stores data underneath
- **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: DEPLOYMENT

Food Tracker (React + Express + Prisma)

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. Most of that story is identical to the sibling course; Prisma adds two real, extra steps to the pipeline that better-sqlite3 never needed.

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 — UNCHANGED FROM THE SIBLING COURSE

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. The API routes must always be registered first — nothing about Prisma changes this; it's an Express-level fact.

React Router handles navigation entirely in the browser — a URL like `/history` never actually exists as a real file on the server. `app.get("*", ...)` always returns `index.html` instead of a 404, 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 — UNCHANGED FROM THE SIBLING

Food Tracker (React + Firebase)'s own deployment chapter got an SPA rewrite rule and automatic HTTPS from Firebase Hosting configuration alone. This course has to hand-write both, exactly like its raw-SQL sibling — the catch-all route above *is* this course's own SPA rewrite rule, and HTTPS termination has to be arranged separately too.

Prisma's Own Real Deployment Steps

Neither of these exists for the sibling course, and skipping either one produces a real, working-locally-but-broken-on-the-server failure:

```
# on the deploy server, after pulling the latest code
npm install
npx prisma generate      # regenerate Prisma Client for this environment
npx prisma migrate deploy # apply any migrations not yet run on this database
npm run build            # builds the React client, as above
```

WHY PRISMA GENERATE RUNS AGAIN HERE, EVEN THOUGH SCHEMA.PRISMA HASN'T CHANGED

Prisma Client is generated code, written into `node_modules` — and `node_modules` is never committed to version control. A fresh `npm install` on the deploy server rebuilds `node_modules`

from scratch, which means the generated client has to be rebuilt there too. The `@prisma/client` package does include a `postinstall` hook that usually runs `prisma generate` automatically after `npm install` — but environments that skip install scripts (a Docker multi-stage build, certain CI caching setups run with `--ignore-scripts`) can silently skip it. Running `prisma generate` explicitly, rather than trusting the hook alone, is the defensive, honest version of this step.

MIGRATE DEPLOY IS NOT MIGRATE DEV — AND THE TWO MUST NOT BE CONFUSED

`prisma migrate dev`, used throughout this course's own local development, can create new migrations and, in some drift scenarios, prompt to reset the development database — genuinely useful behavior while building, genuinely dangerous in production. `prisma migrate deploy` only applies migrations that already exist as committed files in `prisma/migrations/`; it never generates a new one, and it never prompts. It's the one Prisma command actually meant to run against a real, populated database.

DEPLOYMENT STEP	SIBLING COURSE	THIS COURSE
Build the client	<code>npm run build</code>	Same
Prepare the backend	Nothing extra — <code>better-sqlite3</code> needs no separate step	<code>npx prisma generate</code>
Apply the schema	Already applied — <code>db.exec(schema.sql)</code> runs at server startup	<code>npx prisma migrate deploy</code>
Start the process	<code>pm2 start server/index.js</code>	Same

Environment Configuration and Process Management

```
# .env (production)
NODE_ENV=production
PORT=3001
DATABASE_URL="file:./prod.db"
```

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:

```
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) sitting in front of the Node process, terminating HTTPS and forwarding plain HTTP internally. This has nothing to do with Prisma either way.

THE SIBLING'S OWN PERSISTENT-VOLUME GOTCHA, NOW EXPRESSED THROUGH DATABASE_URL

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, exactly as in the sibling course. The difference here is just where that file path lives: not a hardcoded string inside `db.js`, but the `DATABASE_URL` value in `.env`. Confirming the deployment target keeps whatever path `DATABASE_URL` points at across restarts is the same real, easy-to-overlook step either course needs — it just needs checking in a different file here.

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 the real difference between `prisma migrate dev` and `prisma migrate deploy`, and why only the second one belongs in a deployment script.

 [View solution](#)

EXERCISE 3

Explain why this chapter runs `npx prisma generate` explicitly on the deploy server rather than relying solely on `@prisma/client`'s own `postinstall` hook.

 [View solution](#)

CHAPTER 11 QUICK REFERENCE

- **Build:** `npm run build` produces `client/dist/`, served via `express.static` — identical to the sibling
- **Order matters:** API routes must be registered before the catch-all — an Express fact, unrelated to Prisma
- **Two real extra steps:** `npx prisma generate` (rebuild the generated client on this environment) and `npx prisma migrate deploy` (apply committed migrations, production-safe)
- **Don't confuse the two migrate commands:** `migrate dev` can create migrations and reset the dev database; `migrate deploy` only applies what's already committed, never prompts
- **Process management & TLS:** `pm2` and an `nginx` reverse proxy, identical to the sibling
- **Real gotcha, relocated:** the SQLite file still needs a persistent volume — now tracked via `DATABASE_URL` in `.env` rather than a hardcoded path
- **Next chapter:** Capstone

CHAPTER 12: CAPSTONE — A COMPLETE, WORKING FOOD TRACKER (PRISMA EDITION)

Food Tracker (React + Express + Prisma)

Chapter 12 · Capstone: A Complete, Working Food Tracker (Prisma Edition)

Marcus keeps his own household pantry tracked with the app this course just spent eleven chapters building — the same app, the same React-plus-Express architecture as its sibling, with Prisma running the data layer throughout. What follows is the same ordinary session the sibling course's own capstone walked through, this time through this course's own real code.

STEP 1 — SCANNING A NEW ITEM

Marcus scans a carton of milk. Chapter 4's `useBarcodeScanner` hook decodes the barcode entirely client-side — genuinely unchanged, character for character, from both the Firebase and raw-SQL siblings — then `handleDetected` calls `fetch("/api/lookup/...")`. Chapter 3's route checks `BarcodeCache` first via `findUnique()` and falls back to a live Open Food Facts call, writing the result with `upsert()` — the fix for a real race condition the sibling's own plain `INSERT` doesn't have.

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 request body is `{ name, barcode, category, expiryDate }`, the same field name end to end with no snake_case translation needed. Chapter 2's `prisma.item.create()` is the real validation gate, and it responds with the full created row — `status` and `addedAt` included, a real correctness detail the sibling's own hand-assembled response actually misses.

STEP 3 — A FEW DAYS LATER, AN ALERT

Chapter 6's alerts route surfaces the milk once it's within three days of its expiry date — `expiryDate: { not: null, lte: threeDaysFromNow }`, comparing a real typed `DateTime` rather than leaning on every date being formatted consistently by hand. The dashboard

shows `item.expiryDate.slice(0, 10)` — the one small display-formatting cost of that same type safety.

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 `useDebounceSearch` waits for him to stop typing, then hits `prisma.item.findMany({ where: { name: { contains: q } } })` — matching case-insensitively *only* because Chapter 7's own migration added `COLLATE NOCASE` to the `name` column by hand. Without that fix, searching "yog" would never have matched an item actually named "Yogurt" — a real regression the sibling course never had to fix at all.

STEP 5 — MARKING THE MILK USED

The milk gets finished. Chapter 8's `PATCH /api/items/:id/use` — never a plain link — calls `prisma.item.update({ where: { id, status: "active" }, data: { ... } })`, the same atomic single-query guard as the sibling's own `WHERE id = ? AND status = 'active'`, wrapped in `try / catch` to turn a thrown `P2025` into a clean 404 if Marcus somehow clicks twice.

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, genuinely unchanged from the sibling course. 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, entirely independently of whether the route they each call runs a raw SQL query or a Prisma one underneath.

STEP 7 — A RECIPE SUGGESTION

With chicken and spinach both nearing expiry, Chapter 9's `Promise.all` fan-out queries TheMealDB for both concurrently — a real Node/JavaScript fact, identical in this course

and its raw-SQL sibling, owing nothing to Prisma either way. `RecipeCache` reuses Chapter 3's own `upsert()` pattern for the same race-safety reason, storing each meal list as a plain JSON-encoded string rather than depending on Prisma's own inconsistent `Json`-type support for SQLite.

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, pm2 keeping the process alive, nginx terminating HTTPS in front of it — all identical to the sibling. What isn't identical: `npx prisma generate` ran on the deploy server to rebuild the generated client, and `npx prisma migrate deploy` applied every migration this course's own eleven chapters accumulated, including Chapter 7's own hand-edited `COLLATE NOCASE` fix — with the underlying SQLite file's own persistence tracked through `DATABASE_URL` rather than a hardcoded path.

Chapter Attribution

STEP	CHAPTER(S) APPLIED
1 — Scanning	Chapter 4 (camera scanning, unchanged), Chapter 3 (lookup route + BarcodeCache upsert)
2 — Adding the item	Chapter 5 (AddItemForm, no field translation), Chapter 2 (schema.prisma, create())
3 — Expiry alert	Chapter 6 (typed DateTime query, display formatting)
4 — Search	Chapter 7 (contains filter, the COLLATE NOCASE fix)
5 — Marking used	Chapter 8 (update() with extended where, P2025 handling)
6 — Automatic updates	Chapter 10 (ItemsContext, notifyChange, version — unchanged)
7 — Recipe suggestion	Chapter 9 (Promise.all, RecipeCache upsert, plain-String meals)
8 — Real deployment	Chapter 11 (prisma generate, migrate deploy, DATABASE_URL)

The Full Scorecard: SQL vs. Prisma, Chapter by Chapter

Chapter 1 opened this course with a specific promise: not "Prisma is better," but an honest account of what it actually costs and buys, chapter by chapter, against a sibling course doing the identical job in raw SQL. Eleven chapters later, here's the complete tally:

WHERE PRISMA GENUINELY WON

- **Chapter 3 & 9** — `upsert()` closes a real race condition the sibling's own check-then-insert pattern has, for both the barcode and recipe caches.
- **Chapter 5** — `create()`'s full-row response structurally can't omit a field the way the sibling's hand-assembled response actually does (missing `status`, `added_at`).
- **Chapter 5** — matching field names end to end (no snake_case translation) is a small but real, permanent reduction in places a bug could hide.
- **Chapter 6** — a typed `DateTime` field makes date-format consistency structural, not a matter of every developer remembering to format dates the same way.

WHERE PRISMA GENUINELY COST SOMETHING

- **Chapter 2** — every schema change needs an explicit migration step; the sibling's raw SQL takes effect the moment the query runs.
- **Chapter 6** — a `DateTime` value serializes to a full ISO timestamp in JSON, needing `.slice(0, 10)` the sibling never needed.
- **Chapter 7** — the one real regression in this whole course — `contains` defaults to case-sensitive on SQLite, unlike the sibling's automatically case-insensitive `LIKE` — fixed only by a hand-edited migration.
- **Chapter 8** — replicating the sibling's own atomic guard needed a specific, real Prisma feature (extended `where`, stable since 5.0.0) and a different failure-handling shape (a thrown `P2025`, not a returned count).
- **Chapter 11** — two real extra deployment steps, `prisma generate` and `prisma migrate deploy`, that the sibling's deployment never needed.

WHERE NOTHING CHANGED AT ALL

Chapter 1's own architecture, Chapter 4's camera scanner, Chapter 9's `Promise.all` concurrency, and Chapter 10's `ItemsContext` — four genuinely large pieces of this app that Prisma never touches,

because they either predate the database layer entirely or live in a part of the stack (the browser, Node's own async model) Prisma has no reason to reach into.

WHAT THIS WHOLE COURSE WAS REALLY ABOUT

Someone who already knows SQL well, deciding whether an ORM is worth learning, doesn't need to be told Prisma is better or worse — they need the specific, honest list above. Real wins (Chapter 3's race-condition fix, Chapter 6's structural type safety), real costs (Chapter 7's actual regression, Chapter 11's extra deploy steps), and a solid stretch of the app where the choice simply didn't matter. That specific, itemized answer — not a verdict — is this course's own real closing point.

HONEST SCOPE NOTE

This capstone deliberately stops short of several things, matching the sibling course's own honest limits: the weekly meal planner named as future work 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; 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

Using this chapter's own scorecard, name one place Prisma genuinely improved on the sibling course and one place it genuinely cost something real — explaining, for each, what specifically made it a win or a cost rather than just a difference.

[View solution](#)**CHAPTER 12 QUICK REFERENCE — COURSE COMPLETE**

- **8 steps, 11 prior chapters** — one continuous, realistic session with the finished, deployed app
- **4 real Prisma wins:** upsert() race-safety (Ch.3, Ch.9), full-row create() responses (Ch.5), no field-name translation (Ch.5), structural DateTime consistency (Ch.6)
- **5 real Prisma costs:** the migration step (Ch.2), display formatting (Ch.6), the case-sensitivity regression (Ch.7), extended-where research and different error handling (Ch.8), two extra deploy steps (Ch.11)
- **4 chapters unaffected either way:** architecture (Ch.1), the camera scanner (Ch.4), Promise.all concurrency (Ch.9), Context-based state (Ch.10)
- **This course's own closing point:** a specific, itemized answer to "what does Prisma actually cost and buy," not a verdict
- **Food Tracker (React + Express + Prisma) is now complete** — 12/12 chapters