Food Tracker (React + Firebase)

A Complete 13-Chapter Backend-as-a-Service Course

Topics covered:

Why a BaaS architecture · Firestore data modeling · Cloud Functions
Camera barcode scanning · direct client writes · Security Rules
Expiry alerts · live search · recipe lookup · Firebase Authentication
Deployment

Capstone: one complete, working, per-user app, chapter by chapter

Exercises: 39 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

One of four Food Tracker courses — see also FastAPI, Django,
and React + Express editions

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 13

Project Overview & Why a Backend-as-a-Service

Food Tracker (React + Firebase)

Chapter 1 · Project Overview & Why a Backend-as-a-Service

This is one of four courses building the exact same app in four genuinely different architectures — Food Tracker (FastAPI), Food Tracker (Django), and Food Tracker (React + Express) are its siblings. Every one of them scans a barcode, tracks a use-by date, and alerts you before something goes to waste. What differs, chapter by chapter, is *where the backend actually lives* — and this course's own answer is the most unusual of the four: nowhere you wrote yourself.

What the App Actually Does

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

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

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

What Firebase Actually Provides

"Firebase" isn't one thing — it's a bundle of managed services, and this course leans on four of them directly:

- **Firestore** — a NoSQL, document-based database. Chapter 2 goes deep on what that means for this app's own data model, in direct contrast to the SQL schemas the other three courses use.
- **Cloud Functions** — small, serverless backend functions, triggered by an HTTP call or a database event. Chapters 3 and 10 use these for exactly the two places this app still needs a genuine server: proxying the Open Food Facts and TheMealDB lookups.

- **Firestore Security Rules** — declarative rules controlling who can read or write what, enforced by Firebase itself rather than by code you run. Chapter 6 covers these in depth, once the app starts writing directly to the database from the browser.
- **Firebase Authentication** — managed sign-in and user accounts, covered in Chapter 11 as a natural extra this course can afford that its siblings reasonably skip.
- **Firebase Hosting** — static hosting for the finished React build, covered in Chapter 12.

The Architecture Contrast, Precisely

COURSE	BACKEND	WHERE BUSINESS LOGIC LIVES
Food Tracker (FastAPI)	A FastAPI process you write, run, and deploy	Python code you author, running on a server you manage
Food Tracker (Django)	A Django process you write, run, and deploy	Python code you author, running on a server you manage
Food Tracker (React + Express)	A Node/Express process you write, run, and deploy	JavaScript code you author, running on a server you manage
Food Tracker (React + Firebase)	No server process you write or deploy	Mostly Security Rules (configuration, not code) plus a small number of Cloud Functions for the few things that genuinely need one

> **THE ONE-SENTENCE VERSION OF THIS WHOLE COURSE**
>
> In the other three Food Tracker courses, **all** business logic and validation lives in code you write and run yourself. In this one, most of it lives in **configuration** — Security Rules — and managed infrastructure, with your own code shrunk down to the handful of places that genuinely need a trusted secret or heavier compute than the client should do alone.

Setting Up a Firebase Project

Create a project at the Firebase console, enable Firestore in **production mode** (not test mode — test mode allows unrestricted reads and writes, which Chapter 6 will replace with real rules), then bring the Firebase SDK into the React app:

```
npm install firebase

// firebase.js
import { initializeApp } from "firebase/app";
import { getFirestore } from "firebase/firestore";

const firebaseConfig = {
  apiKey: "AIzaSy...",
  authDomain: "food-tracker-xxxx.firebaseapp.com",
  projectId: "food-tracker-xxxx",
};

const app = initializeApp(firebaseConfig);
export const db = getFirestore(app);
```

THAT CONFIG OBJECT IS NOT A SECRET — REALLY

Coming from a "never expose your API key" mindset, pasting `apiKey` directly into frontend source code looks alarming. It isn't: Firebase's client-side config values identify *which* Firebase project a request is talking to — they don't grant access to anything by themselves. Actual access control is enforced entirely by Security Rules (Chapter 6), evaluated on Firebase's own servers for every single read and write, regardless of what config values the client presents. A genuine secret — an API key for a third-party service that must never reach the browser — is exactly why Cloud Functions exist at all, and exactly what Chapters 3 and 10 use them for.

DEVELOP AGAINST EMULATORS, NOT THE LIVE PROJECT

`npm install -g firebase-tools`, then `firebase login` and `firebase init emulators` sets up local Firestore and Cloud Functions emulators. Building against these instead of the real cloud project from day one avoids racking up usage, avoids polluting real data with test writes, and makes Chapter 6's security-rules work far faster to iterate on.

Where This Course Is Headed

Data modeling in Firestore, barcode lookup via a Cloud Function, the camera-scanning React component (shared almost verbatim with Food Tracker (React + Express)), direct client writes to Firestore, Security Rules as the real gatekeeper, expiry alerts, item history with the honest limits of Firestore's own search capability, marking items used, recipe lookup, Firebase Authentication, deployment, and a capstone tying every chapter into one complete, working app.

Hands-On Exercises

EXERCISE 1

In one sentence, state the fundamental architectural difference between this course and its three siblings. Then explain, using this chapter's own comparison table, what "where business logic lives" actually means for each of the four.

View solution

EXERCISE 2

Explain why Firebase's client-side config object (apiKey, projectId, etc.) is safe to include directly in frontend source code, when a traditional server API key is not. What actually enforces access control instead?

View solution

EXERCISE 3

Name the four Firebase products this chapter previews, and match each one to the later chapter that covers it in depth.

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
- **Firestore** — NoSQL document database (Ch.2)
- **Cloud Functions** — serverless functions for the two places a real secret is needed (Ch.3, Ch.10)
- **Security Rules** — the actual gatekeeper once the client writes directly to the database (Ch.6)
- **Firebase Authentication** — real user accounts, a near-free extra (Ch.11)
- **This course's own throughline:** business logic lives mostly in configuration and managed infrastructure, not in a server process you write and run
- **Next chapter:** Data Modeling in Firestore

CHAPTER 2 OF 13

Data Modeling in Firestore

Food Tracker (React + Firebase)

Chapter 2 · Data Modeling in Firestore

Chapter 1 named Firestore as this course's own database, in contrast to the SQL every sibling course uses. This chapter designs the Pantry Item around it properly — and the honest answer up front is that Firestore's document model doesn't just store the same data differently, it changes what "the schema" even means.

The Relational Shape (What the Other Three Courses Use)

Food Tracker (FastAPI), Food Tracker (Django), and Food Tracker (React + Express) all model a Pantry Item as one row in a single SQL table — `id`, `name`, `barcode`, `category`, a nullable `expiry_date`, a `status` column, `added_at`, and a nullable `used_at`. The database enforces column types and nullability up front, at write time — a row that violates the schema is rejected before it's ever stored.

Firestore's Document Model

Firestore stores data as **collections** of **documents** — each document a JSON-like object whose fields can be strings, numbers, booleans, timestamps, arrays, or nested maps. Critically, Firestore itself enforces **no schema at all**: nothing stops one document in a collection from having fields another document in the same collection lacks entirely. This is **schema-on-read** rather than SQL's **schema-on-write** — the database doesn't validate shape at all; whatever validates it is the application code, or nothing.

For this app: one top-level `items` collection, one document per Pantry Item, with fields `name`, `barcode`, `category`, `expiryDate`, `status`, `addedAt`, and `usedAt`.

A Real Firestore Document

```
// items/8xJ2kLpQmN4vR7wZ (an active item)
{
  name: "Greek Yogurt, 500g",
  barcode: "5901234123457",
  category: "dairy",
  status: "active",
  expiryDate: Timestamp(2026-08-14),
  addedAt: Timestamp(2026-08-02),
}

// items/qT9nB3fXcW1yH6dP (a used item — expiryDate omitted entirely)
{
  name: "Sourdough Loaf",
  barcode: "5901234654321",
  category: "bakery",
  status: "used",
  addedAt: Timestamp(2026-07-28),
  usedAt: Timestamp(2026-07-30),
}
```

Notice the used item doesn't set `expiryDate` to `null` — it omits the field entirely. This is a deliberate modeling choice, not an oversight: a Firestore range query (`where("expiryDate", "<", someDate)`) only ever matches documents where that field actually exists and is a comparable type. A document missing the field is automatically excluded from the results — quietly and correctly — which is exactly the behavior the expiry-alerts query in Chapter 7 depends on. SQL's `NULL` behaves differently in comparisons (a `NULL` value in a `WHERE expiry_date < X` clause is neither true nor false, and is excluded for a different underlying reason) — the end result looks similar here, but it's worth knowing the two databases arrive at it through genuinely different mechanics.

Document IDs: Not the Barcode

A tempting shortcut is using the barcode itself as the document ID, since it's already a natural unique identifier for the product. Resist it: the same product (the same barcode) can be bought — and tracked — more than once, each purchase with its own expiry date and its own lifecycle. The document ID needs to identify *one purchased instance*, not one product. Let Firestore auto-generate the document ID, and store the barcode as an ordinary field instead, exactly as shown above.

Use Firestore's Timestamp Type, Not Date Strings

Firestore has a native `Timestamp` type specifically for dates — use it for `expiryDate`, `addedAt`, and `usedAt`, rather than storing an ISO date string. Range queries, sorting, and Chapter 7's own "expiring within

N days" logic all rely on genuine timestamp comparison, not string comparison that happens to work for well-formatted ISO strings but isn't a real date comparison at all.

Firestore vs. SQL, Side by Side

SQL (THE OTHER THREE COURSES)	FIRESTORE (THIS COURSE)	WHAT'S GENUINELY DIFFERENT
Table + column schema	Collection of documents, no enforced schema	Schema-on-write vs. schema-on-read
Primary key (often auto-increment)	Auto-generated document ID	Similar in spirit; Firestore IDs are opaque strings, not sequential integers
Nullable column (`NULL`)	Field simply absent from the document	"No value" is modeled as absence, not a special null marker
`WHERE expiry_date < X`	`where("expiryDate", "<", X)`	Documents missing the field are excluded automatically, same practical outcome via a different mechanism

Querying Firestore: A First Look

```
import { collection, query, where, getDocs } from "firebase/firestore";
import { db } from "./firebase";

const activeItemsQuery = query(
  collection(db, "items"),
  where("status", "==", "active")
);
const snapshot = await getDocs(activeItemsQuery);
const activeItems = snapshot.docs.map(doc => ({ id: doc.id, ...doc.data() }));
```

THE RELATIONAL-VS-DOCUMENT MOTIF, ONE MORE TIME

Comparative Linux Distributions, `cp1`, and MongoDB Fundamentals 5's own embedding-vs-referencing material have all touched some version of this same underlying question on this site: relational structure vs. flexible document structure, and what each trades away. This app's own data is honestly a fairly easy case for a document database — flat, no real nested relationships yet — so the interesting document-modeling decisions are still ahead, not here: Chapter 10's recipe results are exactly the kind of data (an ingredient list nested inside a recipe) where embedding-vs-referencing actually becomes a real design question again.

GET SOME OF SQL'S SAFETY BACK, VOLUNTARILY

Firestore won't stop a typo'd field name or an inconsistent shape — so define a single TypeScript interface (or at minimum, a shared constants file) for what an Item document actually looks like, and use it at every single write site in the app. It's optional and unenforced by the database, which is exactly why skipping it is easy to regret later.

A SILENT TYPO CAN CREATE A SECOND, INVISIBLE FIELD

Writing `expiryDate` in one part of the app and `expiry_date` in another doesn't raise an error — Firestore happily stores both as two separate fields on the same document, and neither Chapter 7's alerts query nor anything else will notice until items mysteriously stop showing up as expiring. This class of bug has no SQL equivalent, since a misspelled column name there fails immediately and loudly. The shared-interface habit from the tip box above is the real defense against it.

Where This Course Is Headed

Barcode lookup via a Cloud Function, the camera-scanning React component, direct client writes to Firestore, Security Rules as the real gatekeeper, expiry alerts built on exactly the query shown above, item history and its search limitations, marking items used, recipe lookup, Firebase Authentication, deployment, and a capstone tying every chapter together.

Hands-On Exercises

EXERCISE 1

Explain the difference between a field being "absent" in Firestore and a column being NULL in SQL. Why does a used item's document simply omit expiryDate rather than setting it to null, and how does that choice affect Chapter 7's own expiry-alerts query?

View solution

EXERCISE 2

Explain why this chapter argues against using the barcode itself as a Firestore document ID, and what should be used instead.

View solution

EXERCISE 3

This chapter says Firestore's schema-on-read flexibility is a genuine trade-off, not a free win. What cost does it push onto the application that SQL's schema-on-write avoids, and how does the tip box's shared-interface suggestion partially address it?

View solution

CHAPTER 2 QUICK REFERENCE

- **Collection/document model** — one `items` collection, one document per Pantry Item, no enforced schema
- **Field absence, not NULL** — a used item simply omits `expiryDate`; range queries exclude documents missing the field automatically
- **Document ID** — auto-generated, never the barcode (which identifies a product, not a purchased instance)
- **Firestore** `Timestamp` — always for dates, never a plain string
- **This chapter's own throughline:** schema-on-read trades database-enforced consistency for flexibility — a cost the app's own code has to pick up instead
- **Next chapter:** Barcode Lookup via a Cloud Function

CHAPTER 3 OF 13

Barcode Lookup via a Cloud Function

Food Tracker (React + Firebase)

Chapter 3 · Barcode Lookup via a Cloud Function

This is the first piece of this course's own backend code — and the honest first question worth asking is whether it needs to exist at all.

Does This Actually Need a Cloud Function?

Open Food Facts is free, public, and requires no API key — so, strictly speaking, the React app could call it directly with `fetch()` from the browser, no Cloud Function involved. Chapter 1's own warn-box justified Cloud Functions by "a genuine secret that must never reach the browser" — and there's no secret here at all. So why route it through one anyway? Three reasons that have nothing to do with secrecy:

- **Caching.** Many different users of this app will scan the same common products — the same barcode gets looked up repeatedly. A server-side cache means Open Food Facts only gets queried once per barcode, ever, no matter how many people scan it.
- **Consistent error handling.** A third-party API's own error shapes are the third-party API's own business — normalizing "not found," "malformed barcode," and "service unavailable" into one consistent response shape is easier to do in one place than in every component that might trigger a lookup.
- **Future-proofing.** If a paid, key-requiring nutrition database ever replaces or supplements Open Food Facts, the swap happens entirely inside this one function — zero client-side changes, and the key never has to touch the browser at all.

Writing a Callable Cloud Function

Firebase's **callable functions** (`onCall`) handle the client-server plumbing — CORS, request/response serialization, and (from Chapter 11 onward) authentication context — automatically, which is why this course uses them rather than a raw HTTP endpoint.

```
// functions/index.js
const { onCall, HttpsError } = require("firebase-functions/v2/https");
const { getFirestore } = require("firebase-admin/firestore");

exports.lookupBarcode = onCall(async (request) => {
  const barcode = request.data.barcode;
  if (!barcode) throw new HttpsError("invalid-argument", "barcode is required");

  const db = getFirestore();
  const cacheRef = db.collection("barcodeCache").doc(barcode);
  const cached = await cacheRef.get();
  if (cached.exists) {
    return cached.data();
  }

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

  if (data.status !== 1) {
    throw new HttpsError("not-found", "No product found for this barcode");
  }

  const product = {
    name: data.product.product_name || "Unknown item",
    category: data.product.categories_tags?.[0]?.replace("en:", "") || "uncategorized",
  };

  await cacheRef.set(product);
  return product;
});
```

Calling It From React

```
import { getFunctions, httpsCallable } from "firebase/functions";

const functions = getFunctions();
const lookupBarcode = httpsCallable(functions, "lookupBarcode");

async function handleScan(barcode) {
  try {
    const result = await lookupBarcode({ barcode });
    console.log(result.data); // { name, category }
  } catch (err) {
    if (err.code === "functions/not-found") {
      // fall back to manual entry — see Chapter 5
    }
  }
}
```

The Cache: a Second Collection, With a Different ID Rule

`barcodeCache` is a second Firestore collection, entirely separate from `items` — and here, unlike Chapter 2's own `items` collection, using the barcode itself as the document ID is exactly correct. Chapter 2 rejected barcode-as-ID for `items` because that collection tracks one document per *purchased instance*, and the same product can be bought more than once. `barcodeCache` tracks one document per *product*, full stop — there's only ever one canonical name/category for a given barcode, so keying the cache by barcode directly is the right modeling decision this time, not a repeat of Chapter 2's mistake.

NOT EVERY CLOUD FUNCTION EXISTS TO HIDE A SECRET

Chapter 1 introduced Cloud Functions as the answer to "a genuine secret that must never reach the browser." This chapter's own Cloud Function protects no secret at all — Open Food Facts needs no key. It exists for caching, consistency, and future flexibility instead. Both are legitimate reasons to reach for server-side code in an otherwise serverless architecture; secrecy is only one of them.

WHAT THE EMULATOR DOES AND DOESN'T FAKE

Running this function against the Firebase emulator (set up in Chapter 1) executes your actual function code locally and reads/writes the local Firestore emulator — but the `fetch()` call to Open Food Facts still goes out over the real network to the real service. Emulating Firebase doesn't mean faking every external HTTP call your function makes.

A CLOUD FUNCTION IS NOT AUTOMATICALLY RATE-LIMITED OR SECURE

Since Chapter 1 established that Firebase's client config isn't secret, nothing inherently stops someone outside this app from calling `lookupBarcode` directly, as often as they like, using nothing but that public config. Wrapping code in a Cloud Function controls *where the code runs*, not *who's allowed to run it* — that's a separate concern, handled by

App Check or, more relevantly for this app, the Firebase Authentication covered in Chapter 11. Don't mistake "it's a Cloud Function" for "it's protected."

Where This Course Is Headed

The camera-scanning React component that feeds a barcode into this function, direct client writes to Firestore for the add-item flow, Security Rules as the real gatekeeper, expiry alerts, item history and search, marking items used, recipe lookup (a second Cloud Function, reusing everything from this chapter), Firebase Authentication, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain why routing the Open Food Facts lookup through a Cloud Function isn't strictly required, unlike Chapter 1's own justification for Cloud Functions. Name the three reasons this chapter gives for using one anyway.

View solution

EXERCISE 2

Chapter 2 said never use the barcode as the document ID for the items collection. This chapter uses the barcode as the document ID for barcodeCache. Explain what's different about the two collections that makes both of these the correct decision.

View solution

EXERCISE 3

Explain the warn-box's distinction between "where code runs" and "who's allowed to run it." Why doesn't putting the barcode lookup in a Cloud Function automatically prevent abuse, and what will actually address that later in the course?

View solution

CHAPTER 3 QUICK REFERENCE

- **Cloud Function reasons, this time** — caching, consistent error handling, future-proofing (not secrecy, unlike Chapter 1's own justification)

- `onCall` — Firebase's callable-function pattern; handles CORS and (later) auth context automatically
- `barcodeCache` — a second collection, correctly keyed by barcode (one document per product, not per purchase)
- **Emulators fake Firebase, not the outside world** — a local function still makes real HTTP calls to real third-party APIs
- **A Cloud Function ≠ automatically secure** — it controls where code runs, not who can call it; that's Chapter 11's job
- **Next chapter:** Camera-Based Barcode Scanning in React

CHAPTER 4 OF 13

Camera-Based Barcode Scanning in React

Food Tracker (React + Firebase)

Chapter 4 · Camera-Based Barcode Scanning in React

This chapter builds the piece of the app that feeds Chapter 3's Cloud Function a barcode in the first place — and it's the one chapter in this entire course with no Firebase involvement at all. Everything here runs in the browser, which is exactly why the component built in this chapter will reappear, unchanged, in Food Tracker (React + Express) when that sibling course reaches its own Chapter 4.

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. `"user"` would request the front-facing camera instead, which is the wrong default for this app entirely. `getUserMedia` also only works over **HTTPS** (or `localhost` during development) — a real deployment gotcha worth knowing now, though not one this course has to worry much about, since Chapter 12's Firebase Hosting serves everything over HTTPS by default.

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;
}
```

Wiring It to This Course's Own Backend

The hook above knows nothing about Firebase — it just calls `onDetected(barcode)`. The parent component supplies that callback, and that's the only place this course's own identity shows up at all:

```
function ScanScreen() {
  const handleDetected = async (barcode) => {
    const result = await lookupBarcode({ barcode }); // Chapter 3's Cloud Function
    // ...hand result.data to the add-item flow, Chapter 5
  };
  const videoRef = useBarcodeScanner(handleDetected);









  return
```

Food Tracker (React + Express)'s own `ScanScreen` will look identical except for one line — `handleDetected` will call its Express server's own POST endpoint instead of a Cloud Function. Everything above this line is shared, unmodified, code.

THE POINT OF BUILDING THE IDENTICAL COMPONENT TWICE

Seeing the exact same camera-scanning code plug into two completely different backends — a Cloud Function here, a plain REST endpoint in the Express course — is itself the lesson: a well-designed frontend component doesn't need to know or care what's on the other end of its one callback prop. The backend is a genuinely swappable detail, not something the component is built around.

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 Firebase Hosting preview channel, or 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

Direct client writes to Firestore for the add-item flow (this scanned barcode's own destination), Security Rules as the real gatekeeper, expiry alerts, item history and search, marking items used, recipe lookup, Firebase Authentication, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain why this chapter's camera-scanning component will reappear unchanged in Food Tracker (React + Express). What is the one thing that actually differs between the two courses' use of it?

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

- `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
- **This component is backend-agnostic** — it only calls one `onDetected(barcode)` callback; the parent decides what happens next
- **This chapter's own throughline:** the identical component will power Food Tracker (React + Express) too — only the parent's callback differs

- **Next chapter:** Building the Add-Item Flow with Direct Firestore Writes

CHAPTER 5 OF 13

Building the Add-Item Flow with Direct Firestore Writes

Food Tracker (React + Firebase)

Chapter 5 · Building the Add-Item Flow with Direct Firestore Writes

Chapter 4 got a barcode; Chapter 3 turned it into a product name and category. This chapter finally saves the item — and along the way, hits a genuine fork this course's three siblings never face at all: **should this write go straight from the browser to the database, or through a Cloud Function?**

Two Ways to Write to Firestore

In FastAPI, Django, and Express, there's only ever one path: the client sends data to your server, and your server writes to the database. There's no other option, because there's no other code that's allowed to touch the database at all. In this course, the client can write to Firestore **directly**, with no server code involved — or it can call a Cloud Function that performs the write on the client's behalf. Both are real, valid options, and this app will end up using both, in different chapters, for different reasons.

Direct Client Writes: The Common Path

```
import { collection, addDoc, serverTimestamp } from "firebase/firestore";
import { db } from "./firebase";

async function addItem({ name, barcode, category, expiryDate }) {
  await addDoc(collection(db, "items"), {
    name,
    barcode,
    category,
    status: "active",
    expiryDate, // a Firestore Timestamp, built from the form's date input
    addedAt: serverTimestamp(),
  });
}
```

This is the Firestore-idiomatic default, not a shortcut taken to save effort: it skips a network round-trip through a Function, it's less code to maintain, and — most importantly — Chapter 6's Security Rules are what actually

make this safe, not the absence of a server. A BaaS app doesn't need a server function standing between the client and the database just to "protect" an ordinary write.

When a Cloud Function Write Makes More Sense Instead

Writing through a Function is the better choice when:

- The write needs **elevated privilege** the client itself should never be granted (bypassing Security Rules deliberately, from trusted server code).
- A value must be **computed or verified server-side**, because it can't be trusted coming from the client at all.
- The write needs to happen **atomically alongside another side effect** — sending a notification, writing an audit log entry — guaranteed to succeed or fail together.

None of that applies to adding a Pantry Item. There's no privileged operation, nothing here needs server-side verification, and there's no coupled side effect at write time. The direct-client-write path is the right choice for this specific flow — a conclusion worth stating plainly rather than leaving as an exercise for later.

The Add-Item Component

A scanned-and-looked-up item pre-fills name and category; a lookup failure (Chapter 3's `not-found` error) instead surfaces a plain manual-entry field for the name. Either path converges on the same form: confirm or edit the details, pick an expiry date, and save.

```
function AddItemForm({ scanned }) {
  const [name, setName] = useState(scanned?.name ?? "");
  const [category, setCategory] = useState(scanned?.category ?? "");
  const [expiryDate, setExpiryDate] = useState(null);

  const handleSubmit = async (e) => {
    e.preventDefault();
    if (!name || !expiryDate) return; // basic client-side check, see below
    await addItem({ name, barcode: scanned?.barcode, category, expiryDate });
  };
  // ...form markup
}
```

A live, type-ahead search across previously-added item names — so re-adding something bought before doesn't mean retyping it — is deliberately **not** built here. That's Chapter 7's own feature, once the item history exists to search against.

WHERE "THE BACKEND" ACTUALLY IS, FOR THIS FEATURE

In every sibling course, adding an item means writing server-side code — a route handler, a view function — that owns the create-item logic. Here, for the common case, there is no backend code for it at all. The write happens directly from the browser to the database, and the only real gatekeeper is a set of declarative rules covered in the very next chapter. This is the most concrete, hands-on demonstration yet of what "no server you write and deploy yourself" (Chapter 1) actually means in practice.

USE SERVERTIMESTAMP(), NOT NEW DATE()

`addedAt: serverTimestamp()` asks Firestore's own server to fill in the timestamp at the moment it processes the write, rather than trusting whatever the client's local clock happens to say. A client's clock can be wrong, in a different timezone than expected, or — worst case — deliberately manipulated; a server-assigned timestamp removes that whole category of problem for data as important as "when was this actually added."

CLIENT-SIDE VALIDATION IS A UX NICETY HERE, NOT A SECURITY BOUNDARY

The `if (!name || !expiryDate) return;` check above only stops an honest user from submitting an incomplete form by accident — it does nothing to stop a modified client, a browser console, or a direct API call from writing whatever it wants straight to Firestore. Exactly like Chapter 2's schema-on-read and Chapter 3's "a Cloud Function isn't automatically secure," nothing in this chapter is actually enforced yet. Chapter 6's Security Rules are the one place that is.

THIS CODE WILL FAIL WITH A PERMISSION ERROR UNTIL CHAPTER 6

Chapter 1 set up Firestore in **production mode**, which denies every read and write by default until Security Rules explicitly allow them. Running this chapter's own `addDoc` call right now, before Chapter 6 is written, will fail with a permission-denied error — expected behavior, not a sign this chapter's code is broken.

Where This Course Is Headed

Security Rules as the real gatekeeper for everything built in this chapter, expiry alerts, item history and search, marking items used, recipe lookup, Firebase Authentication, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain the two ways to write to Firestore this chapter presents, and why it concludes the direct-client-write path is the right choice specifically for the add-item flow, rather than routing it through a Cloud Function.

View solution

EXERCISE 2

Explain why the client-side check in AddItemForm isn't a security boundary, and name what this chapter says actually is.

View solution

EXERCISE 3

Explain why running this chapter's own addDoc code before Chapter 6 exists will fail with a permission error, and why that's expected rather than a bug in this chapter's own code.

View solution

CHAPTER 5 QUICK REFERENCE

- **Two write paths** — direct client write, or through a Cloud Function — a fork that only exists in a BaaS architecture
- **This app's add-item flow** — direct client write; no elevated privilege, server-side verification, or coupled side effect needed
- `serverTimestamp()` — always for `addedAt`, never a client-generated `new Date()`
- **Client-side validation ≠ security** — it's a UX nicety; Chapter 6's Security Rules are the real enforcement
- **Expect a permission error right now** — production-mode Firestore denies everything until Chapter 6's rules allow it
- **Next chapter:** Firestore Security Rules

CHAPTER 6 OF 13

Firestore Security Rules

Food Tracker (React + Firebase)

Chapter 6 · Firestore Security Rules

Three separate chapters have now said some version of "this isn't actually enforced yet." Chapter 2's schema is voluntary. Chapter 3's Cloud Function protects no one by simply existing. Chapter 5's client-side validation can be skipped by anyone willing to bypass the app's own JavaScript. This chapter is where that thread finally resolves — Security Rules are the one thing in this entire architecture that's genuinely, unavoidably enforced.

Rules Are Code, But Not the App's Code

Security Rules live in their own file, `firestore.rules`, written in a declarative rules language and deployed independently of the React app itself (`firebase deploy --only firestore:rules`). They're evaluated entirely on Firebase's own servers, for every single read and write, regardless of what the client's own code does or doesn't check first — precisely the gap Chapter 5's own warn-box left open.

The Default-Deny Starting Point

```
// firestore.rules — the production-mode default from Chapter 1
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /items/{itemId} {
      allow read, write: if false; // this is exactly what caused Chapter 5's permission error
    }
  }
}
```

Writing Real Rules for `items`

```
match /items/{itemId} {
  allow read: if true; // no auth yet — Chapter 11 tightens this

  allow create: if request.resource.data.name is string
                && request.resource.data.name.size() > 0
                && request.resource.data.status in ['active', 'used']
                && request.resource.data.addedAt == request.time;

  allow update, delete: if true; // tightened once auth exists — Chapter 11
}
```

`allow read: if true` is appropriately permissive for now — there's no user system yet, so there's no one to distinguish between. It gets revisited directly once Chapter 11 adds Firebase Authentication. The interesting line is `create`: it requires `name` to actually be a non-empty string, requires `status` to be one of exactly two valid values, and — the detail worth pausing on — requires `addedAt` to equal `request.time`, the server's own clock at the moment the write is evaluated. A client using `serverTimestamp()` (Chapter 5's own recommendation) satisfies this automatically; a client that tries to fake `addedAt` with its own local `Date` value gets rejected outright. Chapter 5 recommended `serverTimestamp()` as good practice; this rule makes it the only option that works at all.

A Different Set of Rules for `barcodeCache`

```
match /barcodeCache/{barcode} {
  allow read: if true;
  allow write: if false; // clients never write this directly — see below
}
```

This isn't a typo — `write: if false` genuinely blocks every client write, and the app still works, because the only writer to this collection is Chapter 3's Cloud Function, which uses the **Firebase Admin SDK** rather than the regular client SDK. Admin SDK access, by design, **bypasses Security Rules entirely** — it runs inside Firebase's own trusted server environment, the same place a genuine secret would live if this app had one. Rules govern what the client is allowed to do; they say nothing at all about code running with Admin privileges.

> **THE PAYOFF CHAPTER FOR A THREAD THAT'S RUN SINCE CHAPTER 2**
>
> Nothing earlier in this course was actually enforced on its own — not the data shape (Chapter 2), not the mere existence of a Cloud Function (Chapter 3), not client-side validation (Chapter 5). Security Rules are the first, and only, place in this entire architecture where something genuinely cannot be bypassed by a client, no matter how the client is modified. Everything else in this course has been building toward the moment this sentence stops being a warning and starts being true.

TEST RULES BEFORE THEY'RE LIVE FOR EVERYONE

The Firebase emulator set up in Chapter 1 runs Security Rules locally, letting you try a rule change against realistic reads and writes before deploying it to the actual project. The Firebase console's own Rules Playground offers a similar quick sanity check directly in the browser. Either is faster, and far safer, than editing rules and finding out what broke only after deploying to production.

THE MOST COMMON REAL FIRESTORE MISTAKE

Writing `allow read, write: if true;` everywhere "just to get it working," and forgetting to tighten it later, is the single most common real-world Firestore security failure. Unlike a bug in the app's own code, which affects however many users hit that one code path, a bad security rule potentially exposes or corrupts every document in the affected collection, for every user, the instant it's deployed. Always test a rules change in the emulator first.

Where This Course Is Headed

Expiry alerts (querying the data these rules now actually protect), item history and search, marking items used, recipe lookup, Firebase Authentication (which will directly tighten the `read` / `update` / `delete` rules left permissive above), deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain why `allow write: if false` was the correct starting rule from Chapter 1's own production-mode setup, and connect it directly to the permission-denied error Chapter 5 warned would happen.

View solution

EXERCISE 2

Explain what `request.resource.data.addedAt == request.time` actually verifies, and which earlier chapter's own recommendation this rule now turns into an actual requirement.

View solution

EXERCISE 3

Explain why Chapter 3's Cloud Function can still write to barcodeCache despite this chapter's own `allow write: if false` rule for that collection. What mechanism explains this?

View solution

CHAPTER 6 QUICK REFERENCE

- `firestore.rules` — evaluated on Firebase's servers for every read/write, deployed separately from the app
- **Default-deny** — `if false` is what caused Chapter 5's own permission error; expected, not a bug
- `addedAt == request.time` — enforces that `serverTimestamp()` was actually used, turning Chapter 5's recommendation into a real requirement
- **Admin SDK bypasses rules entirely** — exactly why Chapter 3's Cloud Function can still write to `barcodeCache` despite `write: if false`
- **This chapter's own throughline:** Security Rules are the first genuinely unavoidable enforcement point in this whole course
- **Next chapter:** Expiry Alerts

CHAPTER 7 OF 13

Expiry Alerts

Food Tracker (React + Firebase)

Chapter 7 · Expiry Alerts

Chapter 6 finally protected real data with real rules. This chapter is the first payoff of having that data at all: surfacing which items are about to go to waste.

The On-Demand Query

```
import { collection, query, where, Timestamp, getDocs } from "firebase/firestore";

async function getExpiringSoonItems(daysAhead = 3) {
  const now = Timestamp.now();
  const threshold = Timestamp.fromMillis(now.toMillis() + daysAhead * 24 * 60 * 60 * 1000);

  const q = query(
    collection(db, "items"),
    where("status", "==", "active"),
    where("expiryDate", "<=", threshold)
  );
  const snapshot = await getDocs(q);
  return snapshot.docs.map(doc => ({ id: doc.id, ...doc.data() }));
}
```

Filtering on `status == "active"` is technically redundant with Chapter 2's own "used items omit `expiryDate` entirely" design — a range comparison already excludes documents missing the field. It's included anyway as explicit, readable intent rather than relying on an implicit side effect of the data model to do the filtering silently.

The Composite Index Requirement

Run the query above for the first time, and Firestore throws an error rather than executing it — something like `"The query requires an index."` This isn't a mistake in the query; it's Firestore telling you the truth about how it works. A single-field query gets an automatic index for free. A query combining an equality filter on one field (`status`) with a range filter on a different field (`expiryDate`) needs a **composite index**, declared ahead of time — either by clicking the link Firestore's own error message provides (which pre-fills the exact

index needed), or by committing a `firestore.indexes.json` file to source control so the same index gets created automatically in every environment.

SQL handles the equivalent query differently: an unindexed multi-column `WHERE` clause still runs — just slower, via a full table scan, with no error at all. Firestore refuses outright rather than running a query it can't serve efficiently at scale. Neither behavior is objectively better; they reflect two different philosophies about whether "it'll just be slow" is an acceptable default.

Rendering the Dashboard

```
function ExpiryDashboard() {
  const [expiring, setExpiring] = useState([]);

  useEffect(() => {
    getExpiringSoonItems(3).then(setExpiring);
  }, []);

  return (


        {expiring.map(item =>
• {item.name} — expires {item.expiryDate.toDate().toLocaleDateString()}
  )}



  );
}
```

Should Alerts Be Proactive? An Optional Scheduled Function

The dashboard above only tells anyone anything if they actually open the app. A genuinely proactive alert — a notification that arrives even when nobody's looking — needs code running on a schedule, independent of any user visiting the page. Firebase's `onSchedule` lets a Cloud Function run on a cron-style schedule with no server kept alive between runs:

```
const { onSchedule } = require("firebase-functions/v2/scheduler");

exports.dailyExpiryCheck = onSchedule("every day 08:00", async (event) => {
  const expiringItems = await getExpiringSoonItemsAdmin(3); // same query, Admin SDK
  // write a notification doc, or send via Firebase Cloud Messaging
});
```

This is explicitly optional — a stretch feature, not the MVP. The on-demand dashboard query above is what this course actually requires; a scheduled proactive notification is a genuine enhancement worth attempting once the core app works, not a prerequisite for it.

What the Other Three Courses Would Need Instead

COURSE	HOW IT WOULD RUN A DAILY PROACTIVE CHECK
Food Tracker (FastAPI)	A scheduler library (e.g. APScheduler) running inside a continuously-alive server process
Food Tracker (Django)	Celery Beat, django-crontab, or an OS-level cron job, all requiring something to stay running
Food Tracker (React + Express)	`node-cron` or a system crontab entry, again requiring a persistently-running process
Food Tracker (React + Firebase)	`onSchedule` — no process kept running between invocations at all

THE "NO SERVER YOU DEPLOY YOURSELF" IDEA, APPLIED TO TIME ITSELF

Every sibling course needs something alive around the clock just to notice a date has passed — a whole server process exists partly to support one function that fires once a day. This course's scheduled function exists, compute-wise, only for the seconds it actually executes. It's the same architectural theme from Chapter 1, now showing up in the dimension of *when* code runs, not just where.

TESTING A SCHEDULED FUNCTION WITHOUT WAITING A DAY

The Firebase emulator doesn't fire scheduled functions automatically on a real clock — invoke the underlying handler directly during development instead of waiting for 8:00 AM to roll around, and only rely on the actual schedule once deployed to production.

"THE QUERY REQUIRES AN INDEX" IS NOT A SYNTAX ERROR

It's easy to misread that message as a mistake in how the query was written and start second-guessing the code. Read the error itself — Firestore includes a direct console link to auto-create the exact index the query needs. Following that link is almost always faster than debugging code that was never actually broken.

Where This Course Is Headed

Item history with live search (and Firestore's own honest limits on text search), marking items used, recipe lookup, Firebase Authentication, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain what a Firestore composite index actually is, why this chapter's own query needs one, and how SQL handles the equivalent multi-column query differently.

View solution

EXERCISE 2

Explain the difference between the on-demand dashboard query and the optional scheduled Cloud Function. What real problem does the scheduled function solve that the on-demand query, by itself, never can?

View solution

EXERCISE 3

Explain the finding-box's claim that this course applies its "no server you deploy yourself" idea to time itself. What would each of the three sibling courses need to keep running just to check expiry dates once a day, and why doesn't this course need the equivalent?

View solution

CHAPTER 7 QUICK REFERENCE

- **On-demand query** — `where("status","==","active")` + `where("expiryDate","<=",threshold)`, required for this course
- **Composite index** — required for multi-field filter combinations; Firestore refuses to run the query until one exists, unlike SQL's slower-but-runs approach
- `onSchedule` — a scheduled Cloud Function, optional/stretch, for proactive daily checks
- **No persistent process needed** — unlike the cron/Celery/node-cron setup every sibling course would require
- **This chapter's own throughline:** "no server you deploy yourself" applies to scheduled/background work too, not just request-driven code
- **Next chapter:** Item History & Live Search-as-You-Type

CHAPTER 8 OF 13

Item History & Live Search-as-You-Type

Food Tracker (React + Firebase)

Chapter 8 · Item History & Live Search-as-You-Type

Every item ever added lives in one list — active items with a real expiry date, used items without one, exactly as Chapter 2 modeled them. This chapter makes that combined history searchable in real time, and runs straight into the most honest limitation Firestore has.

Firestore Has No Full-Text Search

Firestore's query operators are genuinely limited: `==` for exact match, range operators for ordering, `array-contains` / `in` for membership — and nothing resembling SQL's `LIKE '%term%'`, and no built-in full-text search at all. This isn't a missing feature to work around quietly; it's a real, well-known consequence of the same document-database tradeoff Chapter 2 introduced.

The Prefix-Query Trick — and Its Real Limits

```
const q = query(
  collection(db, "items"),
  where("name", ">=", searchTerm),
  where("name", "<", searchTerm + "")
);
```

This genuinely works — for an exact, case-sensitive **prefix** match. The trailing value appended after `searchTerm` in the second `where()` call is the Unicode private-use character U+F8FF, which sorts after virtually every normal character — appending it is what makes the range capture every string starting with `searchTerm` and nothing past it. Typing `"Gr"` matches `"Greek Yogurt"`. It has two real limits worth being honest about: it is **case-sensitive** (`"gr"` will not match `"Greek Yogurt"` at all), and it only matches from the **start** of the string — searching `"yogurt"` will not find `"Greek Yogurt"` using this technique alone.

Fixing Case-Sensitivity: a `nameLower` Shadow Field

Firestore has no query-time equivalent of SQL's `LOWER()` — the only fix is storing a second, already-lowercased field alongside the original, and querying against that instead. Chapter 5's add-item flow needs

one more field:

```
await addDoc(collection(db, "items"), {
  name,
  nameLower: name.toLowerCase(), // new — used for case-insensitive search
  // ...the rest, unchanged from Chapter 5
});
```

Chapter 6's Security Rules need one more clause too — requiring `nameLower` to actually equal `name.lower()` at write time, the same "voluntary schema, enforced by rules" pattern that chapter already established for every other field.

When Prefix Search Genuinely Isn't Enough

Neither trick above solves mid-word or typo-tolerant search. Two real, honest options exist once that's actually needed:

- **Filter client-side.** Pull the (realistically small, single-user) item history once, and filter it in plain JavaScript on every keystroke — no per-keystroke network call, no Firestore query limitation to work around at all.
- **Mirror into a dedicated search service.** Algolia, Typesense, or Meilisearch, kept in sync with Firestore via a Cloud Function trigger on every document write — the genuinely scalable answer, and genuinely more infrastructure than this app needs.

For a personal pantry tracker realistically holding a few hundred items at most, client-side filtering is the right call — named explicitly rather than left ambiguous. A dedicated search service is the honest answer *if* this ever needed to scale to thousands of items per user, not something this course builds.

The Actual Implementation

```
function useItemHistory() {
  const [allItems, setAllItems] = useState([]);
  useEffect(() => {
    const q = query(collection(db, "items"), orderBy("addedAt", "desc"));
    getDocs(q).then(snap => setAllItems(snap.docs.map(d => ({ id: d.id, ...d.data() }))));
  }, []);
  return allItems;
}

function ItemHistorySearch() {
  const allItems = useItemHistory();
  const [term, setTerm] = useState("");

  const filtered = term
    ? allItems.filter(item => item.nameLower.includes(term.toLowerCase()))
    : allItems;

  return (
    <>
      {term} setTerm(e.target.value)} placeholder="Search items..." />

    {filtered.map(item =>
  • {item.name}{item.status === "active" ? ` — expires ${item.expiryDate.toDate().toLocaleDateString()
    )}



  );
}
```

Notice there's no debounce anywhere in this component. Debouncing exists to avoid firing a network request on every keystroke — but this approach loads the small dataset once and filters it entirely in memory afterward, so there's no per-keystroke network call to debounce in the first place.

A REAL TRADEOFF, NOT A WORKAROUND FAILURE

Firestore's lack of native full-text search is a genuine consequence of the same document-database design Chapter 2 introduced, not a bug or an oversight. The right fix scales with the actual size of the problem — client-side filtering for a small personal dataset, a dedicated search service for a large or shared one. Knowing which one an app actually needs, rather than reaching for the more complex answer by default, is the real skill this chapter is teaching.

THE SHADOW-FIELD PATTERN GENERALIZES

Any field ever needing case-insensitive querying in Firestore needs its own lowercase shadow field, following `nameLower`'s own pattern exactly — there's no query-time function that does this for you, unlike SQL's `LOWER()`.

ADDING A SHADOW FIELD AFTER REAL DATA ALREADY EXISTS IS A REAL MIGRATION PROBLEM

If `nameLower` is added to the add-item flow only after the app already has real items in production, every document created before that change is missing the field entirely — and per Chapter 2's own "absence, not null" lesson, those documents simply won't be found by any search built against `nameLower`. Fixing this needs a one-time backfill script that reads every existing document, computes `nameLower` from its `name`, and writes it back. A schema-on-read database doesn't retroactively update old documents just because the application's own idea of the schema changed.

Where This Course Is Headed

Marking items used, recipe lookup for items nearing expiry, Firebase Authentication, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain the two real limits of the `where(">=", term).where("<", term + "\uf8ff")` prefix trick, and what field this chapter adds — and why — to fix the case-sensitivity limit specifically.

View solution

EXERCISE 2

Explain why this chapter's own client-side filtering approach needs no debounce, unlike a typical live-search feature hitting a network API on every keystroke. At what point would this chapter say client-side filtering stops being the right approach?

View solution

EXERCISE 3

Explain why adding nameLower after the app already has real production data creates a genuine problem, connecting it back to Chapter 2's own "field absence, not null" lesson. What's the actual fix?

View solution

CHAPTER 8 QUICK REFERENCE

- **No native full-text search** — a genuine Firestore limitation, not a missing feature to route around silently

- **Prefix trick** — `where(">=",term).where("<",term+"")`; case-sensitive, prefix-only
- `nameLower` — a lowercase shadow field, fixing case-sensitivity; needs a backfill if added after real data exists
- **Client-side filter vs. a search service** — client-side is right for this app's realistic data volume; a service like Algolia is the honest answer only at real scale
- **No debounce needed here** — the small dataset loads once; filtering afterward is pure in-memory JS, no per-keystroke network call
- **Next chapter:** Marking Items Used

CHAPTER 9 OF 13

Marking Items Used

Food Tracker (React + Firebase)

Chapter 9 · Marking Items Used

A short chapter, and a satisfying one — this is where Chapter 2's original data-modeling decision finally gets exercised as a real transition, not just a starting shape.

The Transition: Active to Used

Chapter 2 modeled a used item as one with `status: "used"` and **no** `expiryDate` **field at all** — not a null value, a genuine absence. Marking an active item used has to actually produce that same shape, which means removing a field on an update, not merely creating a new document without it in the first place.

```
import { doc, updateDoc, deleteField, serverTimestamp } from "firebase/firestore";

async function markItemUsed(itemId) {
  const itemRef = doc(db, "items", itemId);
  await updateDoc(itemRef, {
    status: "used",
    usedAt: serverTimestamp(),
    expiryDate: deleteField(),
  });
}
```

Why Not Just Set `expiryDate` to `null`?

Setting it to `null` would leave the field genuinely present, with a null value — a different shape from the one Chapter 2 designed, and a different shape from a used item that started out used from the very beginning (Chapter 2's own creation example never included the field at all). Using `deleteField()` keeps both paths to "used" — created used, or transitioned to used — producing the identical document shape, rather than the app quietly having two different ways of representing the same "no expiry" state depending on how an item got there.

Extending Chapter 6's Rules to Cover This Update

Chapter 6 validated shape at *create* time and left `update` / `delete` permissive until real auth exists (Chapter 11). That's still the right call for *who* can update — but rules can validate the *shape of an update* independently of who's making it:

```
allow update: if request.resource.data.status in ['active', 'used']
              && (request.resource.data.status != 'used'
                  || !('expiryDate' in request.resource.data));
```

This says: if the update's own resulting document has `status == 'used'`, that same resulting document must **not** contain an `expiryDate` field at all. It's not an authorization check — anyone can still perform this update, per Chapter 6's own deferred-to-Chapter-11 stance — it's a **data invariant**, enforced at the database layer regardless of which client code path triggered the update, or whether that code even remembered to call `deleteField()` correctly.

RULES VALIDATE TRANSITIONS, NOT JUST CREATION

Chapter 6 only showed rules constraining what a brand-new document could look like. This chapter's own rule constrains what an *update* is allowed to turn a document into — a genuinely different kind of check, comparing the resulting shape against an invariant rather than checking who's allowed to act. Security Rules aren't only about authorization; they're a real place to enforce that the data itself stays internally consistent, no matter what code path produced the write.

`DELETEFIELD()` NEEDS A MERGE, NOT A REPLACE

`deleteField()` works inside `updateDoc()`, and inside `setDoc(ref, data, { merge: true })` — but not a plain, non-merge `setDoc()`, which replaces the entire document outright. "Delete this one field" only means something when the rest of the document is being preserved around it.

AN "UNDO" FEATURE WOULD NEED TO RESTORE THE EXPIRY DATE TOO

If this app ever adds an undo action for an accidental "mark used," simply flipping `status` back to `"active"` is not enough — the item would then be active with no `expiryDate` at all, since `deleteField()` already removed it and undoing the status change doesn't bring it back. A genuine undo needs to either have retained the original expiry date somewhere to restore, or explicitly ask the user for a new one.

Where This Course Is Headed

Recipe lookup for items nearing expiry, Firebase Authentication (which finally tightens the `update` / `delete` rules this chapter left permissive), deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain why marking an item used calls deleteField() on expiryDate rather than setting it to null, tying the reason back to Chapter 2's own data model.

View solution

EXERCISE 2

Explain what invariant this chapter's own update rule enforces, and how it differs in kind from Chapter 6's create rule — one validates creation shape, the other validates something else. What is that something else?

View solution

EXERCISE 3

Explain what would go wrong if a future "undo" feature simply flipped status back to active without also restoring expiryDate, and why deleteField() is the reason this specific problem exists.

View solution

CHAPTER 9 QUICK REFERENCE

- `deleteField()` — removes a field entirely on update, matching Chapter 2's "absence, not null" design
- **Consistent shape** — created-used and transitioned-to-used items end up identical, not two different representations of "no expiry"
- **Update-time rules** — Security Rules can validate a transition's resulting shape, not just a new document's creation shape
- **A data invariant, not an auth check** — this chapter's rule enforces "used items never have expiryDate," regardless of who performs the update
- `deleteField()` **needs a merge** — works in `updateDoc()` or a merged `setDoc()`, not a plain replace
- **Next chapter:** Recipe Lookup with TheMealDB

CHAPTER 10 OF 13

Recipe Lookup with TheMealDB

Food Tracker (React + Firebase)

Chapter 10 · Recipe Lookup with TheMealDB

This chapter delivers the last named feature from the very first chapter's own spec: recipes for what's about to expire. It reuses Chapter 3's own Cloud-Function-as-proxy pattern — and needs it for a genuinely new reason this time, not just the old one.

The Same Question, a New Answer

TheMealDB's basic filter-by-ingredient search is also free and keyless, so Chapter 3's own question applies again: does this need a Cloud Function at all? The caching/consistency/future-proofing reasons from Chapter 3 still apply — but there's a new, additional reason this time: this feature needs to search **multiple** expiring ingredients at once and combine the results into one meaningful list, and that kind of fan-out-and-merge step belongs server-side, not spread across several separate client-side `fetch()` calls the browser would otherwise have to coordinate itself.

TheMealDB's Real Limitation: One Ingredient at a Time

TheMealDB's filter endpoint (`filter.php?i={ingredient}`) only searches by a **single** ingredient per request — there's no "match any of these ingredients" query available on the free tier. Searching across everything expiring soon genuinely requires one request per ingredient, then merging the results afterward.

Writing the Cloud Function

```
const { onCall, HttpsError } = require("firebase-functions/v2/https");

exports.suggestRecipes = onCall(async (request) => {
  const ingredients = request.data.ingredients; // item names from Chapter 7's expiring-soon list
  if (!Array.isArray(ingredients) || ingredients.length === 0) {
    throw new HttpsError("invalid-argument", "ingredients array is required");
  }

  const results = await Promise.all(
    ingredients.map(async (ingredient) => {
      const res = await fetch(
        `https://www.themealdb.com/api/json/v1/1/filter.php?i=${encodeURIComponent(ingredient)}`
      );
      const data = await res.json();
      return data.meals || [];
    })
  );

  // merge and deduplicate by meal ID, tracking which ingredient(s) matched each recipe
  const merged = new Map();
  results.forEach((meals, i) => {
    meals.forEach((meal) => {
      if (!merged.has(meal.idMeal)) {
        merged.set(meal.idMeal, { ...meal, matchedIngredients: [] });
      }
      merged.get(meal.idMeal).matchedIngredients.push(ingredients[i]);
    });
  });

  return Array.from(merged.values());
});
```

Wiring In Chapter 7's Own Data

The expiring-items list Chapter 7 already builds is exactly what feeds this function — no new query needed, just the item names passed straight through:

```
const suggestRecipes = httpsCallable(functions, "suggestRecipes");

async function loadRecipeIdeas(expiringItems) {
  const result = await suggestRecipes({
    ingredients: expiringItems.map(item => item.name),
  });
  return result.data;
}
```

Sorting by Relevance

Since every merged recipe carries its own `matchedIngredients` array, sorting by `matchedIngredients.length` descending surfaces recipes that use the **most** expiring items first — directly serving the app's original point: using up as much of what's about to go to waste as possible in one meal, not just finding any recipe that happens to include one soon-to-expire ingredient.

THE SAME PATTERN, A GENUINELY NEW REASON

Chapter 3's Cloud Function existed for caching, consistency, and future-proofing — a single barcode always meant a single lookup. This chapter's Cloud Function needs those same reasons *and* a new one: server-side fan-out across multiple ingredient queries, merged into one coherent result before the client ever sees it. Recognizing when a familiar pattern still applies, and when it's being stretched to cover a genuinely new job, is worth noticing explicitly rather than assuming "we already built a Cloud Function for this kind of thing" covers every reason to build another one.

CACHE RECIPE RESULTS THE SAME WAY CHAPTER 3 CACHED BARCODES

Common ingredients — chicken, eggs, milk — will be searched constantly across every user of this app. A `recipeCache` collection, keyed by ingredient name exactly like Chapter 3's own `barcodeCache`, avoids re-querying TheMealDB for the same ingredient over and over. Same technique, same reasoning, a second time.

FREE, KEYLESS APIS CAN STILL HAVE REAL LIMITS IN PRACTICE

TheMealDB's free test key has no clearly documented rate limit, but a shared Cloud Function fanning out one request per expiring ingredient, multiplied across every user of this app, could realistically trigger undocumented throttling once usage grows — caching helps, but doesn't eliminate the risk entirely for ingredients that genuinely differ user to user. Worth monitoring for failures in practice, and worth knowing TheMealDB's own paid tier exists as a real option if this app ever had meaningfully many simultaneous users.

Where This Course Is Headed

Firebase Authentication (finally tightening the permissive rules left open since Chapter 6), deployment, and a capstone tying every chapter into one complete, working app.

Hands-On Exercises

EXERCISE 1

Explain why this Cloud Function fans out one request per ingredient rather than making a single combined call to TheMealDB, and name the specific API limitation that makes this necessary.

View solution

EXERCISE 2

Explain the matchedIngredients.length sorting heuristic, and how it serves the original recipe-lookup feature's own intent described back in Chapter 1.

View solution

EXERCISE 3

Explain why the recipeCache pattern from the tip box doesn't fully eliminate the rate-limit risk described in the warn-box, even though it helps.

View solution

CHAPTER 10 QUICK REFERENCE

- **One ingredient per request** — TheMealDB's real limitation, requiring a fan-out of parallel calls
- `suggestRecipes` — a second Cloud Function, reusing Chapter 3's pattern for a new reason: server-side merge/dedupe
- **Chapter 7's data feeds this feature directly** — the expiring-items list becomes the ingredients array
- **Sort by** `matchedIngredients.length` — surfaces recipes using the most expiring items first
- `recipeCache` — the same caching technique as `barcodeCache`, applied a second time
- **Next chapter:** Firebase Authentication

CHAPTER 11 OF 13

Firebase Authentication

Food Tracker (React + Firebase)

Chapter 11 · Firebase Authentication

Four separate chapters have been quietly waiting for this one. Chapter 6 left `read` / `update` / `delete` permissive "until Chapter 11." Chapter 3 named Firebase Authentication as the real answer to "who's allowed to call this function." Chapter 9 added a data invariant but no ownership check. Chapter 10 said this chapter would finally tighten everything. It's time.

Why This Course Can Afford Auth, Genuinely Cheaply

Adding real sign-in to Food Tracker (FastAPI), Food Tracker (Django), or Food Tracker (React + Express) means building password hashing, session or token management, and a users table or model — real, substantial work each of those courses would be right to treat as a separate topic. This course already has Firebase Authentication sitting right there, provisioned the moment the Firebase project was created back in Chapter 1. Adding real accounts here is a natural extension of infrastructure already paid for and already set up, not a new subsystem to build from scratch — which is exactly why this course's own outline includes it as a real chapter while the other three reasonably treat it as out of scope.

Setting Up Email/Password Sign-In

Enable the Email/Password provider in the Firebase console, then:

```
import {
  getAuth, createUserWithEmailAndPassword,
  signInWithEmailAndPassword, onAuthStateChanged
} from "firebase/auth";

const auth = getAuth();

function signUp(email, password) {
  return createUserWithEmailAndPassword(auth, email, password);
}

function signIn(email, password) {
  return signInWithEmailAndPassword(auth, email, password);
}

// track the current user across the whole app
function useAuthUser() {
  const [user, setUser] = useState(null);
  useEffect(() => onAuthStateChanged(auth, setUser), []);
  return user;
}
```

Making Item Data Per-User

Every item now needs to record who it belongs to. Chapter 5's own `addItem` gets one more field:

```
await addDoc(collection(db, "items"), {
  name,
  nameLower: name.toLowerCase(),
  userId: auth.currentUser.uid, // new
  // ...everything else, unchanged since Chapters 5 and 8
});
```

And every query — Chapter 7's expiry check, Chapter 8's history — adds one more filter: `where("userId", "==", auth.currentUser.uid)`.

The Real Rules Tightening

Chapter 6's own `create` validation and Chapter 9's own `update` invariant don't get replaced — they get **extended**, with an ownership check added alongside what was already there:

```
match /items/{itemId} {
  allow read: if request.auth != null
              && request.auth.uid == resource.data.userId;

  allow create: if request.auth != null
                && request.auth.uid == request.resource.data.userId
                && request.resource.data.name is string
                && request.resource.data.name.size() > 0
                && request.resource.data.status in ['active', 'used']
                && request.resource.data.addedAt == request.time;

  allow update: if request.auth != null
                && request.auth.uid == resource.data.userId
                && request.resource.data.status in ['active', 'used']
                && (request.resource.data.status != 'used'
                    || !('expiryDate' in request.resource.data));

  allow delete: if request.auth != null
                && request.auth.uid == resource.data.userId;
}
```

Chapter 6's field-shape validation is still exactly there in `create`. Chapter 9's used-item invariant is still exactly there in `update`. Both simply gained a `request.auth` check alongside what they already validated — the rules compose cleanly rather than needing to be rewritten from scratch.

Closing Chapter 3's Own Loop

Callable Cloud Functions automatically receive the caller's `request.auth`, exactly like Firestore rules do. `lookupBarcode` and `suggestRecipes` can now genuinely enforce who's allowed to invoke them, answering Chapter 3's own warning directly:

```
exports.lookupBarcode = onCall(async (request) => {
  if (!request.auth) {
    throw new HttpsError("unauthenticated", "Sign in required");
  }
  // ...unchanged from Chapter 3
});
```

FOUR CHAPTERS, ONE PAYOFF

Chapter 3 named Authentication as the eventual answer to "who's allowed to call this." Chapter 6 left rules permissive specifically until this chapter. Chapter 9 added a data invariant with no ownership check yet. Chapter 10 pointed here

directly. Nothing about this chapter invented a new idea — it's the single point where every one of those earlier, deliberately incomplete answers becomes complete at once.

TEST SIGN-UP AND SIGN-IN AGAINST THE EMULATOR

The Firebase Auth emulator, part of the same emulator suite set up in Chapter 1, lets you create and sign in test accounts locally without touching real user records in the actual Firebase project.

EVERY ITEM CREATED BEFORE THIS CHAPTER HAS NO OWNER

Every item created while working through Chapters 2 through 10 has no `userId` field at all — it never existed as a concept until now. Once the new rules require `request.auth.uid == resource.data.userId`, those older documents fail that check against every possible user, since there's no value there to match at all — they become permanently inaccessible through the app. This is exactly the same backfill problem Chapter 8 already taught, for `nameLower`, applied here to ownership instead of search: a field added after real documents already exist needs a one-time script to backfill it, or those documents are effectively lost to any rule that now depends on it.

Where This Course Is Headed

Deployment — Firebase Hosting and deploying the Cloud Functions built across this course — and a capstone tying every chapter together into one complete, working, per-user app.

Hands-On Exercises

EXERCISE 1

Explain why this course can add real Authentication relatively cheaply compared to what Food Tracker (FastAPI), Food Tracker (Django), and Food Tracker (React + Express) would each need to build for the same feature.

View solution

EXERCISE 2

Explain how this chapter's rules extend Chapter 6's create rule and Chapter 9's update rule rather than replacing them. What stayed exactly the same, and what was added?

View solution

EXERCISE 3

Explain why items created before this chapter become permanently inaccessible once the new rules take effect, and connect this directly to Chapter 8's own earlier example of the same underlying problem.

View solution

CHAPTER 11 QUICK REFERENCE

- **Firebase Authentication** — near-free here since the platform already provides it; a bigger lift for every sibling course
- `userId` — added to every item at creation, filtered on in every query
- **Rules extended, not replaced** — Chapter 6's create validation and Chapter 9's update invariant both gain an ownership check alongside what they already checked
- **Callable functions get** `request.auth` **too** — closing Chapter 3's own "who's allowed to call this" loop
- **Pre-existing items lose access** — the same backfill problem Chapter 8 already taught for `nameLower`, now for `userId`
- **Next chapter:** Deployment

CHAPTER 12 OF 13

Deployment

Food Tracker (React + Firebase)

Chapter 12 · Deployment

Four genuinely different things need to reach production: the React app itself, three Cloud Functions, Firestore's Security Rules, and a composite index that's only ever existed as a console click-through so far.

Building and Configuring Hosting

`npm run build` produces a static output folder; `firebase.json` tells Hosting where to find it and how to handle client-side routes:

```
{
  "hosting": {
    "public": "dist",
    "rewrites": [{ "source": "**", "destination": "/index.html" }]
  }
}
```

The `rewrites` rule is not optional decoration — every route that isn't the root path only exists as far as React Router (running inside `index.html`) understands it. Hosting itself has no file at `/history` or `/scan`; without this rewrite, refreshing or directly navigating to any route but the root returns a genuine 404.

Deploying — All of It, or Just What Changed

```
firebase deploy                          # everything
firebase deploy --only hosting           # just the React build
firebase deploy --only functions         # just the Cloud Functions
firebase deploy --only firestore:rules   # just Chapters 6/9/11's own rules
firebase deploy --only firestore:indexes # just the composite index — see below
```

Scoped deploys aren't just a shortcut — redeploying Hosting when only a Cloud Function changed adds time and risk for no reason. Deploying only what actually changed keeps each deploy small, fast, and easy to reason about.

Codifying Chapter 7's Own Index

Chapter 7's composite index was created by clicking the link in Firestore's own error message — fine for one developer's local project, but a fresh environment, a teammate's machine, or a CI pipeline has no such link to click. `firestore.indexes.json`, checked into source control, makes that index a deployable artifact instead of a one-time manual step:

```
{
  "indexes": [
    {
      "collectionGroup": "items",
      "fields": [
        { "fieldPath": "status", "order": "ASCENDING" },
        { "fieldPath": "expiryDate", "order": "ASCENDING" }
      ]
    }
  ]
}
```

Config for Secrets — Not Needed Yet, Worth Knowing

Neither Open Food Facts nor TheMealDB has ever needed a key in this course — but if that ever changed (a paid nutrition API, say), `firebase-functions/params`' `defineSecret` is where a real key would live, kept entirely out of source control and out of the client:

```
const { defineSecret } = require("firebase-functions/params");
const nutritionApiKey = defineSecret("NUTRITION_API_KEY");
// firebase functions:secrets:set NUTRITION_API_KEY
```

CHAPTER 4'S REQUIREMENT, FINALLY GENUINELY MET

Chapter 4 noted that `getUserMedia` only works over HTTPS or `localhost`, and left it at that since local development doesn't need to worry about it yet. Firebase Hosting serves every deployment over HTTPS by default, on both its own `*.web.app` domain and any custom domain attached later. This chapter is where that early requirement stops being an assumption and becomes something the deployed app actually satisfies.

PREVIEW CHANNELS BEFORE GOING LIVE

`firebase hosting:channel:deploy preview` publishes the current build to a separate, temporary URL — genuinely useful for trying a change (or sharing it for feedback) before merging it into the production channel everyone else sees.

FORGETTING THE REWRITE RULE IS THE SINGLE MOST COMMON FIRST-DEPLOY MISTAKE

Without the `rewrites` block above, the app works perfectly at its root URL and then appears completely broken the moment someone refreshes on any other page, or shares a direct link to one — a confusing, easy-to-miss failure mode for anyone deploying a single-page app to Firebase Hosting for the first time.

Where This Course Is Headed

One chapter left: a capstone tying together this course's own thread — from Chapter 1's architectural framing through this chapter's own deployment — into one complete, working, per-user Food Tracker.

Hands-On Exercises

EXERCISE 1

Explain what the Hosting rewrite rule actually does, and describe precisely what breaks for a user if it's left out.

View solution

EXERCISE 2

Explain why deploying only what changed (a scoped `firebase deploy --only ...`) is preferable to running a full `firebase deploy` every time, even though the full deploy would technically also work.

View solution

EXERCISE 3

Explain why Chapter 7's composite index needs to be codified in firestore.indexes.json for a real deployment, rather than just clicking the console link once the way it was created during development.

View solution

CHAPTER 12 QUICK REFERENCE

- **Hosting rewrite rule** — every route rewrites to `index.html`, or non-root routes 404 on direct navigation/refresh
- **Scoped deploys** — `--only hosting/functions/firestore:rules/firestore:indexes`, faster and lower-risk than deploying everything every time

- `firestore.indexes.json` — codifies Chapter 7's console-created index so new environments don't need the manual click-through
- `defineSecret` — where a real API key would live, if this app's external APIs ever required one
- **HTTPS by default** — Firebase Hosting finally, genuinely satisfies Chapter 4's own `getUserMedia` requirement
- **Next chapter:** Capstone — A Complete, Working Food Tracker

CHAPTER 13 OF 13

Capstone: A Complete, Working Food Tracker

Food Tracker (React + Firebase)

Chapter 13 · Capstone: A Complete, Working Food Tracker

Maya has just installed the finished app on her phone, at its real deployed URL. Every step below is a real feature, built in a specific earlier chapter, used exactly the way it was designed to be used.

STEP 1 — SIGNING UP

Maya creates an account with her email and a password. From this moment on, her pantry is genuinely hers — not merely hidden behind a login screen, but enforced by the Security Rules built across Chapters 6, 9, and 11: every read, write, and update on her items now checks `request.auth.uid == resource.data.userId`, on Firebase's own servers, regardless of what the app's UI does or doesn't show.

STEP 2 — SCANNING HER FIRST ITEM

She points her phone's camera at a yogurt carton's barcode. The `getUserMedia`-based scanning component (Chapter 4) decodes it and hands it to `lookupBarcode` (Chapter 3), which checks `barcodeCache` first, finds nothing (a genuinely new product), queries Open Food Facts, and caches the result for the next person who scans the same barcode. She confirms the name, picks an expiry date, and saves — Chapter 5's `addItem`, now including her own `userId` per Chapter 11.

STEP 3 — WHAT ACTUALLY GOT STORED

Behind the scenes, the real document reflects every data-modeling decision from Chapter 2: `name` and `nameLower`, `barcode`, `category`, `status: "active"`, a genuine Firestore `Timestamp` for `expiryDate`, `addedAt` set by `serverTimestamp()` (and enforced by Chapter 6's own rule requiring it), and `userId`. Nothing here was left to chance — every field exists because an earlier chapter deliberately decided it should.

STEP 4 — AN ALERT, A FEW DAYS LATER

Opening the app later that week, the dashboard flags the yogurt as expiring soon — Chapter 7's own query, running cleanly now that the composite index it needs was codified in `firestore.indexes.json`

and deployed for real in Chapter 12, rather than existing only as a console click-through on one developer's machine.

STEP 5 — SEARCHING HER HISTORY

Wanting to buy more of something she'd tracked before, Maya types `"yogurt"` into search. This finds **"Greek Yogurt"** — a genuine mid-word match, something the raw Firestore prefix trick Chapter 8 first demonstrated could never do. It works because Chapter 8's actual chosen implementation filters the loaded history client-side with `.includes()`, not the more limited prefix-only Firestore query — the more capable of the chapter's own two approaches, deliberately chosen over the more constrained one.

STEP 6 — MARKING THE YOGURT USED

She finishes it. One tap calls Chapter 9's `markItemUsed`: `status` flips to `"used"`, `usedAt` is stamped, and `expiryDate` is genuinely removed via `deleteField()` — not nulled. The item drops off the active dashboard immediately, but per Chapter 2's original design, it never disappears from her history at all.

STEP 7 — A RECIPE SUGGESTION

With chicken and eggs both nearing expiry, Maya opens the recipe tab. Chapter 10's `suggestRecipes` fans out one TheMealDB query per ingredient, merges the results, and sorts by how many of her own expiring items each recipe actually uses — a recipe using both chicken and eggs together surfaces above one using only either alone.

STEP 8 — ALL OF THIS, AT A REAL ADDRESS

Every step above happened at the app's actual Firebase Hosting URL, over HTTPS by default — the exact requirement Chapter 4 flagged early and left as an assumption, now genuinely satisfied since Chapter 12's deployment.

Chapter Attribution

STEP	CHAPTER(S) APPLIED
1 — Signing up	Chapter 11 (Firebase Authentication), Chapters 6 & 9 (rules extended by ownership)

STEP	CHAPTER(S) APPLIED
2 — First scan	Chapter 4 (camera scanning), Chapter 3 (barcode lookup + cache), Chapter 5 (add-item flow), Chapter 11 (userId)
3 — What got stored	Chapter 2 (data model), Chapter 6 (serverTimestamp enforcement)
4 — Expiry alert	Chapter 7 (expiry query), Chapter 12 (deployed composite index)
5 — Search	Chapter 8 (item history & search)
6 — Marking used	Chapter 9 (deleteField, the used-item invariant)
7 — Recipe suggestion	Chapter 10 (TheMealDB fan-out and merge)
8 — Real deployed URL	Chapter 12 (Hosting, HTTPS), Chapter 1 (the architecture this all rests on), Chapter 4 (the requirement finally met)

WHAT THIS WHOLE COURSE WAS REALLY ABOUT

Chapter 1 opened with a claim: this app would have no server the developer writes, runs, and deploys themselves. Every single step above — authentication, a barcode lookup, a database write, an alert, a search, a status change, a multi-API recipe search, and the deployment serving all of it — happened without that claim ever being broken. Not because the app does less than its siblings, but because Firestore, Security Rules, and Cloud Functions turned out to be enough, chapter after chapter, for a real, complete, multi-feature application.

HONEST SCOPE NOTE

This capstone deliberately stops short of several things: the weekly meal planner named as future work all the way back in Chapter 1 was never built here; there's no offline/PWA support, so the app requires a live connection; each account's pantry is fully private and single-user, with no shared-household or multi-user pantry support; Chapter 7's proactive scheduled notification was built as an optional stretch feature and was never wired up to an actual push-notification delivery mechanism (like Firebase Cloud Messaging tokens); and no automated test suite or CI pipeline was covered anywhere in this course. Each is a genuine, reasonable next step — none of them were quietly assumed to already be done.

Hands-On Exercises

EXERCISE 1

In Step 5, Maya's search for "yogurt" finds "Greek Yogurt" — a mid-word match. Explain specifically why this works, referencing which of Chapter 8's two approaches actually powers the real search feature.

View solution

EXERCISE 2

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

View solution

EXERCISE 3

Explain why the honest scope note excludes push notifications specifically, even though Chapter 7 already built a scheduled Cloud Function that checks for expiring items.

View solution

CHAPTER 13 QUICK REFERENCE — COURSE COMPLETE

- **8 steps, 12 prior chapters** — one continuous, realistic session with the finished, deployed app
- **This course's own throughline, closed out:** no server the developer writes, runs, or deploys themselves — genuinely true across every single feature built
- **Honest scope note:** no meal planner, no offline/PWA support, no shared-household pantries, no push-notification delivery, no automated tests/CI
- **Food Tracker (React + Firebase) is now complete** — 13/13 chapters