Food Tracker (Django)

A Complete 13-Chapter Batteries-Included Course

Topics covered:

Why Django · model design · the free admin

Barcode lookup & caching · camera scanning · forms

Expiry alerts · native search · recipe lookup · Django REST Framework

Deployment

Capstone: one complete, working 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, React + Firebase, and React + Express editions

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 13

Project Overview & Why Django

Food Tracker (Django)

Chapter 1 · Project Overview & Django Project/App Setup

This is one of four courses building the exact same app in four genuinely different architectures — Food Tracker (FastAPI), Food Tracker (React + Express), and Food Tracker (React + Firebase) are its siblings. Every one of them scans a barcode, tracks a use-by date, and alerts before something goes to waste. This course's own answer leans all the way into the opposite philosophy from its FastAPI sibling: one single, integrated framework, rather than a thin layer you assemble yourself from smaller pieces.

What the App Actually Does

The shared spec every course in the quartet builds 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.
- **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.

Batteries Included, vs. a Thin API Layer

FastAPI, this quartet's other Python course, gives you essentially one thing: a fast, well-typed way to define HTTP routes. Everything else — the database layer, the templating, an admin interface — is a separate library you choose and wire in yourself. Django takes the opposite position entirely: an ORM, a templating engine, a full admin interface, a forms system, and authentication all ship together, pre-integrated, from the very first `startproject` command. Neither philosophy is simply "better" — they represent two real, different answers to how much a framework should decide for you up front.

Project vs. App: Two Different Things

```
django-admin startproject foodtracker .
python manage.py startapp pantry
```

A Django **project** (`foodtracker`) is the overall configuration — settings, URL routing, WSGI/ASGI entry points. An **app** (`pantry`) is a self-contained component holding its own models, views, and templates for one piece of functionality. A single project can hold several apps; this course's entire feature set lives inside one, `pantry`, since the app is genuinely small enough not to need splitting further.

MVT: Model-View-Template

Django's own name for its architecture is **MVT**, not the more familiar MVC — and the naming difference matters, because it's a real, common point of confusion. Django's **Model** is the ORM layer, same idea as MVC's Model. Django's **Template** is the presentation layer, same idea as MVC's View. Django's own **View**, confusingly, is actually the *controller* in classic MVC terms — the Python function or class that receives a request, talks to the Model, and picks a Template to render. Anyone arriving from traditional MVC terminology (or from FastAPI, which has no equivalent three-part naming at all) has to consciously remember that Django's "View" isn't what "view" means almost everywhere else.

Setting Up and Running It

```
# settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "pantry",  # the new app, added by hand
]

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}
```

SQLite is the genuine, appropriate choice here, not a placeholder to "upgrade later" — this is a personal, single-user app, exactly the case SQLite was designed for.

```
python manage.py runserver
```

> **A WORKING ADMIN PANEL, BEFORE WRITING ANY BUSINESS LOGIC AT ALL**

Once `pantry`'s models exist (Chapter 2), Django's admin site gives a genuine, working CRUD interface for them for free, with zero custom code — something FastAPI has no equivalent of at all without building it by hand. This is the "batteries included" philosophy made concrete: an entire category of work Django considers part of the framework itself, that a thin-API-layer framework considers entirely the developer's own problem.

NAME APPS AFTER THE DOMAIN, NOT GENERICALLY

`pantry` reads clearly everywhere Django references app names — `INSTALLED_APPS`, migration folders, template lookup paths. A generic name like `app1` would work identically but read as meaningless the moment the project has more than one app.

FORGETTING INSTALLED_APPS IS THE CLASSIC FIRST DJANGO MISTAKE

A new app's models won't be picked up by migrations, and its templates or static files won't be found, until it's added to `INSTALLED_APPS` — and the resulting errors are often confusing rather than direct: a "template does not exist" error, for instance, rather than a clear "app not registered" message. If something Django-related seems to silently not exist, checking `INSTALLED_APPS` first is worth making a reflex.

Where This Course Is Headed

Modeling the Pantry Item with Django's ORM, the Django admin as an instant CRUD tool, barcode lookup via a Django view, camera-based scanning, the add-item flow via Django Forms, expiry alerts, item history and search, marking items used, recipe lookup, Django REST Framework for the interactive features that need real JSON, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain what Django's "View" actually corresponds to in classic MVC terms, and why this chapter calls it a common point of confusion for newcomers.

View solution

EXERCISE 2

Explain the difference between a Django project and a Django app, using this chapter's own startproject/startapp commands as your example.

View solution

EXERCISE 3

Explain what breaks if a newly created app is never added to INSTALLED_APPS, and why the resulting error can be confusing to diagnose.

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
- **Batteries included** — ORM, admin, templating, forms, auth all ship together, unlike FastAPI's thin-layer approach
- **Project vs. app** — `foodtracker` (config) vs. `pantry` (the feature itself)
- **MVT** — Model (ORM), View (actually the controller), Template (presentation) — not classic MVC naming
- **SQLite** — a genuine, appropriate choice for this personal, single-user app
- **This course's own throughline:** one integrated framework deciding more up front, vs. FastAPI's compose-it-yourself approach
- **Next chapter:** Data Modeling with Django's ORM

CHAPTER 2 OF 13

Modeling the Item

Food Tracker (Django)

Chapter 2 · Data Modeling with Django's ORM

Chapter 1 set up the project and the app. This chapter defines the one Model that everything else in this course builds on.

The Model

```
# pantry/models.py
from django.db import models

class Item(models.Model):
    STATUS_CHOICES = [
        ("active", "Active"),
        ("used", "Used"),
    ]

    name = models.CharField(max_length=200)
    barcode = models.CharField(max_length=64, blank=True)
    category = models.CharField(max_length=100, blank=True)
    status = models.CharField(max_length=10, choices=STATUS_CHOICES, default="active")
    expiry_date = models.DateField(null=True, blank=True)
    added_at = models.DateTimeField(auto_now_add=True)
    used_at = models.DateTimeField(null=True, blank=True)

    def __str__(self):
        return self.name
```

`null=True` and `blank=True` Are Two Different Settings

`null=True` is a **database-level** setting — it lets the actual SQL column store `NULL`. `blank=True` is a **form/admin-level** setting — it lets Django's forms and the admin site accept an empty value without raising a validation error. They're independent, and both are needed on `expiry_date` for two separate reasons: `null=True` so a used item's row can genuinely have no expiry date stored, and `blank=True` so the admin form doesn't reject an empty expiry date as a validation failure, even though the database would happily accept it.

This app's own Firestore-based sibling course models the identical "no expiry" case by omitting the field from the document entirely — a genuine structural difference between schema-on-write (this course: an always-present column, sometimes holding `NULL`) and schema-on-read (that course: the field simply isn't there at all).

`DateField` vs. `DateTimeField`

`expiry_date` uses `DateField` — a use-by date has no meaningful time component. `added_at` and `used_at` use `DateTimeField`, since knowing roughly *when* within a day something happened is genuinely useful for those two fields specifically.

`auto_now_add`: Django's Own Trustworthy Timestamp

`auto_now_add=True` sets a field's value **once**, at creation, using the server's own clock — and makes the field non-editable through ordinary forms. This solves the identical problem the Firebase sibling course's own `serverTimestamp()` solves: never trust a client-supplied creation time. `auto_now_add` is easy to confuse with the similarly-named `auto_now`, which instead updates the field on **every** save — the wrong choice for `added_at`, which should only ever be set the one time the row is created.

`choices=` Is a Voluntary Constraint, Not a Guarantee

`STATUS_CHOICES` restricts what the admin site and Django Forms will offer as valid options for `status` — but whether that also becomes a real database-level constraint depends on the Django version in use, and shouldn't be assumed either way without checking. A raw SQL statement, or code that bypasses Django's own forms and admin validation, may still be able to write a value outside `STATUS_CHOICES` depending on that. `choices=` is genuinely useful for guiding the admin and forms layer; treating it as an unconditional database-level guarantee is the kind of assumption worth verifying rather than trusting blindly.

Migrations: Two Separate Steps, On Purpose

```
python manage.py makemigrations pantry
python manage.py migrate
```

`makemigrations` compares the current models against the last recorded schema state and **generates** a migration file — a versioned, reviewable Python description of the change. `migrate` is the separate step that actually **applies** pending migrations to a real database. Keeping "describe the change" and "apply the change" as two distinct commands means a migration file can be reviewed and committed to version control before it ever touches a database, and the exact same migration can be applied identically across a developer's machine, staging, and production.

THE SAME TRADEOFF, FROM THE OTHER SIDE

Food Tracker (React + Firebase)'s own Chapter 2 explained schema-on-read as trading database-enforced consistency for flexibility. This chapter is the mirror image of that same tradeoff: Django's ORM enforces column types and nullability at write time, in exchange for needing an explicit migration every time the shape of the data changes at all. Neither approach is free — each simply decided which cost to accept.

TRY THE MODEL OUT BEFORE BUILDING ANY VIEWS

`python manage.py shell` opens an interactive Python shell with Django's app registry already loaded — `Item.objects.create(name="Test", status="active")` works immediately, letting the model itself be exercised before any URL, view, or template exists yet.

THE SINGLE MOST LIKELY MISTAKE IN THIS EXACT MODEL

Setting `null=True` without also setting `blank=True` (or the reverse) is the most common source of confusing bugs on a field like this one. A field that genuinely should be optional will still throw a "this field is required" validation error in a form or the admin if `blank=True` is missing — even though the database column itself would happily accept `NULL`. If a field behaves like it's required when it clearly shouldn't be, check `blank` before anything else.

Where This Course Is Headed

The Django admin as an instant CRUD tool for this exact model, barcode lookup via a Django view, camera-based scanning, the add-item flow via Django Forms, expiry alerts, item history and search, marking items used, recipe lookup, Django REST Framework, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain the difference between null=True and blank=True, and why expiry_date needs both rather than just one.

View solution

EXERCISE 2

Explain the difference between auto_now_add and auto_now, and why added_at specifically needs the former, not the latter.

View solution

EXERCISE 3

Explain what makemigrations does versus what migrate does, and why Django keeps them as two separate commands rather than one combined step.

View solution

CHAPTER 2 QUICK REFERENCE

- `null=True` — database-level; allows a real SQL `NULL`
- `blank=True` — form/admin-level; allows an empty value in forms and the admin
- `auto_now_add` — set once at creation, server clock, non-editable; `auto_now` updates on every save instead
- `choices=` — constrains forms/admin; don't assume it's an unconditional database guarantee without checking
- `makemigrations` **vs.** `migrate` — describe the change, then separately apply it; keeps changes reviewable and reproducible
- **Next chapter:** The Django Admin: Instant CRUD for Free

CHAPTER 3 OF 13

The Django Admin, Free

Food Tracker (Django)

Chapter 3 · The Django Admin: Instant CRUD for Free

Chapter 1 promised a working CRUD interface before writing any business logic at all. Here's that promise, delivered.

Registering the Model

```
# pantry/admin.py
from django.contrib import admin
from .models import Item

admin.site.register(Item)
```

```
python manage.py createsuperuser
```

`createsuperuser` creates the admin's own login account — a separate concept entirely from any future end-user account this app might have. Visiting `/admin/` and signing in with it now shows a complete, working list/add/edit/delete interface for `Item`, generated from just those two lines of code.

Customizing It With `ModelAdmin`

Plain registration gives a genuinely functional but generic list view. A `ModelAdmin` subclass shapes it into something actually useful day to day:

```
@admin.register(Item)
class ItemAdmin(admin.ModelAdmin):
    list_display = ("name", "category", "status", "expiry_date", "added_at")
    list_filter = ("status", "category")
    search_fields = ("name", "barcode")
    readonly_fields = ("added_at",)
```

- `list_display` — which columns actually show in the list view, instead of just each row's `__str__`.
- `list_filter` — a sidebar of quick filters, genuinely useful for jumping straight to, say, everything with `status = "active"`.
- `search_fields` — the admin's own search box, which performs a real substring (SQL `LIKE`) match by default. Worth flagging honestly: this is a happy coincidence, not the same thing as the live search-as-you-

type feature this course builds for the app itself later — the admin's search exists purely for the admin's own list view.

- `readonly_fields` — `added_at` is already non-editable (Chapter 2's `auto_now_add` implicitly sets `editable=False`), and Django's admin would normally just omit a non-editable field from the form entirely. Listing it in `readonly_fields` instead keeps its actual value visible — read-only — right there in the change form, rather than making it invisible.

A Real Tool, Not Just a Demo

Before Chapter 4's barcode lookup or Chapter 5's real add-item flow exist, this admin already lets real test `Item` rows be created by hand — genuinely useful for exercising Chapter 7's expiry query, or Chapter 8's search, against real data long before the app's own user-facing screens are built.

A Real Limit Worth Naming

The admin is a tool for whoever manages the site's data directly — it was never meant to be, and won't become, the pantry app's own end-user interface. There's no camera scanning here, no barcode-triggered lookup, no live search-as-you-type UX. Having a working admin is not the same as having a finished app; the two serve genuinely different audiences.

A GENUINE, UNIQUELY DJANGO ADVANTAGE

Food Tracker (FastAPI), Food Tracker (React + Express), and Food Tracker (React + Firebase) would each need real, hand-written code — routes, a UI, or both — just to get this same day-one capability: a basic list/add/edit interface for managing data directly. Here, it exists after two lines in `admin.py`. This is exactly the "batteries included" tradeoff Chapter 1 named, made concrete rather than abstract.

GOING FURTHER: `LIST_EDITABLE`

`list_editable = ("status",)` lets specific fields be edited directly from the list view itself, without opening the full change form — useful for a quick bulk "mark several as used" pass, once there's real data to try it on.

THE ADMIN IS ONLY AS SECURE AS WHO CAN LOG INTO IT

Once deployed publicly, `/admin/` is a powerful, largely unrestricted interface for anyone who successfully logs in — it isn't sandboxed the way the app's own end-user views might be. Real deployment deserves real consideration here: a genuinely strong password, restricting network access to it where feasible, and third-party two-factor packages are all common, real mitigations — treating the admin as purely a development convenience with no real-world exposure risk, once it's live, would be a mistake.

Where This Course Is Headed

Barcode lookup via a Django view, camera-based scanning, the add-item flow via Django Forms, expiry alerts, item history and search, marking items used, recipe lookup, Django REST Framework, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain what admin.site.register(Item) alone provides, and what a customized ModelAdmin subclass adds on top of it. Why do list_display, list_filter, and search_fields matter in practice rather than just being nice-to-haves?

View solution

EXERCISE 2

Explain why readonly_fields is still useful for added_at, given that auto_now_add already makes the field non-editable via editable=False.

View solution

EXERCISE 3

Explain the real security consideration this chapter raises about exposing /admin/ on a public deployment, and name at least two real mitigations it mentions.

View solution

CHAPTER 3 QUICK REFERENCE

- `admin.site.register(Item)` — a full generic CRUD interface, two lines of code
- `ModelAdmin` — `list_display`, `list_filter`, `search_fields`, `readonly_fields` shape it into something actually usable
- **The admin's search ≠ the app's own live search** — a happy coincidence of substring matching, not the same feature built later
- **A dev tool, not the end-user UI** — no camera scanning, no live search UX, a genuinely different audience
- **Real production consideration** — a public `/admin/` needs a real password policy, restricted access, or 2FA

- **Next chapter:** Barcode Lookup: Integrating Open Food Facts

CHAPTER 4 OF 13

Barcode Lookup & BarcodeCache

Food Tracker (Django)

Chapter 4 · Barcode Lookup: Integrating Open Food Facts

The admin gave a working CRUD interface for free. This chapter writes the first genuinely custom code in this course: a view that turns a scanned barcode into a real product name.

URLs and Views: Two Separate Files, on Purpose

Django deliberately keeps "which URL triggers this code" (`urls.py`) separate from "what that code actually does" (`views.py`) — two files, wired together explicitly. FastAPI's own decorator-based routing (`@app.get("/path")`) puts both in the same place, directly above the function it decorates. Neither is objectively better; it's a real, structural difference in where that one piece of wiring lives, worth knowing rather than assuming every framework does it the same way.

A Second Model: `BarcodeCache`

Open Food Facts needs no API key here either, so the same reasoning that justified a Cloud Function in this app's own Firebase sibling course applies again: caching, consistent error handling, future-proofing — nothing about secrecy.

```
class BarcodeCache(models.Model):
    barcode = models.CharField(max_length=64, primary_key=True)
    name = models.CharField(max_length=200)
    category = models.CharField(max_length=100, blank=True)
```

Note `primary_key=True` on `barcode` here — the exact opposite of Chapter 2's own `Item` model, where barcode is deliberately an ordinary field, never the primary key, because the same product can be purchased and tracked more than once. `BarcodeCache` genuinely is one record per product, so keying it directly by barcode is correct here.

The View

```python
# pantry/views.py
import requests
from django.http import JsonResponse
from .models import BarcodeCache

def lookup_barcode(request, barcode):
    cached = BarcodeCache.objects.filter(barcode=barcode).first()
    if cached:
        return JsonResponse({"name": cached.name, "category": cached.category})

    response = requests.get(f"https://world.openfoodfacts.org/api/v2/product/{barcode}.json")
    data = response.json()

    if data.get("status") != 1:
        return JsonResponse({"error": "Product not found"}, status=404)

    product = data["product"]
    name = product.get("product_name", "Unknown item")
    category = (product.get("categories_tags") or [""])[0].replace("en:", "")

    BarcodeCache.objects.create(barcode=barcode, name=name, category=category)
    return JsonResponse({"name": name, "category": category})
```

Wiring the URL

```python
# pantry/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path("lookup/<str:barcode>/", views.lookup_barcode, name="lookup_barcode"),
]
```

`<str:barcode>` is Django's own path converter — it types and captures the URL segment directly inside the route pattern string, rather than as a Python function type annotation the way FastAPI declares a path parameter's type.

Migrating the New Model

```bash
python manage.py makemigrations pantry
python manage.py migrate
```

`requests` Isn't a Django Battery

Worth being precise about what "batteries included" actually covers: Django ships an ORM, an admin, forms, and templating — everything for building the app's *own* request/response cycle. Making an outbound call to someone else's API is a different job entirely, and needs a separate library (`pip install requests`) regardless of how much Django itself provides natively.

THE SAME DESIGN QUESTION, RESOLVED IDENTICALLY

Whether to use a barcode as a primary key isn't answered once for the whole app — it depends entirely on what the collection or table actually represents. `Item` tracks purchased instances (many can share a barcode); `BarcodeCache` tracks canonical products (exactly one per barcode). This app's own Firebase sibling course faced the identical question for its `items` vs. `barcodeCache` collections, and landed on the same answer for the same underlying reason — proof this is a genuine data-modeling principle, not a coincidence of one particular database technology.

TEST THE VIEW BEFORE BUILDING ANYTHING ELSE

Visiting `/lookup/<a-real-barcode>/` directly in a browser returns the raw JSON response — a genuinely useful way to confirm this view works correctly before any frontend exists at all, the same "test one layer in isolation" habit Chapter 3's `manage.py shell` tip already established.

NO ERROR HANDLING YET FOR A FAILING EXTERNAL REQUEST

This view has no `try` / `except` around the `requests.get()` call at all. If Open Food Facts is slow, times out, or is simply down, this raises an unhandled exception, and whoever called this endpoint gets a raw Django 500 error page instead of a clean, meaningful response. This is a genuine, known gap in this simple version — a production-ready version would wrap the request in error handling and return a proper JSON error response instead of letting the exception propagate.

Where This Course Is Headed

Camera-based barcode scanning, the add-item flow via Django Forms, expiry alerts, item history and search, marking items used, recipe lookup, Django REST Framework, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain the structural difference between Django's separated urls.py/views.py routing and FastAPI's decorator-based routing. Where does "this URL maps to this code" actually live in each?

View solution

EXERCISE 2

Explain why BarcodeCache uses barcode as its primary key while Item does not, and connect this to how the Firebase sibling course resolved the identical design question.

View solution

EXERCISE 3

Explain what actually happens right now if Open Food Facts is down or times out while this view is running, and why this is described as a genuine, known gap rather than an acceptable final state.

View solution

CHAPTER 4 QUICK REFERENCE

- **urls.py / views.py** — routing and logic deliberately kept separate, unlike FastAPI's combined decorator approach
- `BarcodeCache` — `barcode` as primary key, correctly, since this collection is one-per-product
- `<str:barcode>` — Django's own URL path converter/typing system
- `requests` — not a Django battery; outbound API calls still need a separate library
- **Known gap** — no error handling yet for a failing Open Food Facts request; a real 500 error results today
- **Next chapter:** Camera-Based Barcode Scanning (Frontend)

CHAPTER 5 OF 13

Camera-Based Barcode Scanning

Food Tracker (Django)

Chapter 5 · Camera-Based Barcode Scanning (Frontend)

Chapter 4 built the endpoint; this chapter builds the page a phone camera actually talks to. The underlying browser APIs are identical to what this app's React-based sibling courses use — but how that code gets packaged looks genuinely different here.

A Template, Not a Component

Django has no equivalent of a portable React component or hook — no lightweight, reusable frontend unit that carries its own logic and can simply be dropped into two different apps unmodified. What it has instead is **template inheritance and includes** (`{% extends %}` , `{% include %}`) — a coarser-grained way to reuse markup, not JavaScript behavior. The scanning logic in this course lives as page-specific script inside one template, not as something this course could hand to a sibling the way the two React-based courses share one component outright.

The View

```
def scan_page(request):
    return render(request, "pantry/scan.html")
```

```
path("scan/", views.scan_page, name="scan_page"),
```

The Template

```
<!-- pantry/templates/pantry/scan.html -->
{% extends "pantry/base.html" %}
{% block content %}
<video id="scanner-video" autoplay playsinline muted></video>
<script src="{% static 'pantry/scan.js' %}"></script>
{% endblock %}
```

```
// pantry/static/pantry/scan.js
(async function () {
  const video = document.getElementById("scanner-video");
  const stream = await navigator.mediaDevices.getUserMedia({ video: { facingMode: "environment" } });
  video.srcObject = stream;
  await video.play();

  if ("BarcodeDetector" in window) {
    const detector = new BarcodeDetector({ formats: ["ean_13", "upc_a"] });
    const scan = async () => {
      const barcodes = await detector.detect(video);
      if (barcodes.length > 0) {
        stream.getTracks().forEach(t => t.stop());
        window.location.href = `/lookup-redirect/${barcodes[0].rawValue}/`;
        return;
      }
      requestAnimationFrame(scan);
    };
    scan();
  } else {
    // fall back to ZXing's BrowserMultiFormatReader here
  }
})();
```

A Real Full-Page-Navigation Consequence

Detecting a barcode here navigates to an entirely new URL — a genuine server round-trip and full page reload, not an in-memory state update. This is a real, honest consequence of the server-rendered-template default, not a mistake: the two React-based sibling courses handle a successful scan by updating component state directly, with no navigation at all. Chapter 11's Django REST Framework chapter is exactly where this could change — a `fetch()` call to a JSON endpoint instead of a URL redirect would avoid the page reload entirely, at the cost of writing more client-side JavaScript to manage that state by hand.

Camera Cleanup: The Same Gotcha, a Genuinely Harder Problem Here

`stream.getTracks().forEach(t => t.stop())` still has to run before the camera is released — same underlying requirement as any camera-scanning code, in any framework. But without a component lifecycle, this template can only reliably guarantee cleanup on the one exit path it explicitly coded: a successful scan. A user who instead clicks a "Cancel" link, uses the browser's back button, or simply closes the tab leaves the camera running, with no equivalent of a React `useEffect` cleanup function guaranteed to fire regardless of *how* the component goes away. This is a genuine structural disadvantage of the plain-template approach for this specific concern, not something this chapter glosses over.

`BarcodeDetector` Support: The Same Real Limitation

Chrome/Edge on Android and desktop support it natively; Safari on iOS does not, and needs the same ZXing fallback this app's React-based courses rely on. This is a browser API limitation, entirely independent of which backend framework is serving the page.

THE SAME LOGIC, PACKAGED COMPLETELY DIFFERENTLY

The underlying browser APIs — `getUserMedia`, `BarcodeDetector`, the ZXing fallback — don't care which backend framework is involved, and are genuinely reusable in spirit across all four Food Tracker courses. What differs is packaging: two React-based sibling courses share one literal, portable component; this course's identical logic lives as a page-specific script, because Django has no equivalent lightweight frontend-component primitive to package it as.

A SEPARATE STATIC FILE, NOT INLINE SCRIPT

Keeping the scanning logic in its own `pantry/static/pantry/scan.js`, rather than inline inside the template, means Django's own static-file handling (collected and served properly once Chapter 12 covers deployment) applies to it, and it's ready to reuse directly if a second page in this app ever needs scanning too.

NOT EVERY EXIT PATH STOPS THE CAMERA

Only the successful-scan path in this chapter's own script stops the camera stream. Navigating away any other way — Cancel, back button, closing the tab — leaves it running, with no framework-level guarantee catching every case the way a component's own unmount lifecycle would. Worth being explicit about with users (a visible "Cancel" control that also stops the stream) rather than assuming this is fully solved.

Where This Course Is Headed

The add-item flow via Django Forms, expiry alerts, item history and search, marking items used, recipe lookup, Django REST Framework — which directly revisits this chapter's own full-page-reload limitation — deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain why this course's camera-scanning JavaScript can't be shared as a single reusable unit the way the two React-based sibling courses share one component. What does Django offer instead, and why is it a coarser-grained kind of reuse?

View solution

EXERCISE 2

Explain what happens, technically, when this chapter's scan.js detects a barcode, and how that differs from what happens in the React-based sibling courses on a successful scan. What would Chapter 11's DRF work change about this?

View solution

EXERCISE 3

Explain why camera cleanup is honestly a harder problem in this chapter's template-based approach than in a React component, referencing specifically what a React lifecycle hook guarantees that a plain template script cannot.

View solution

CHAPTER 5 QUICK REFERENCE

- **No portable component** — Django's reuse tools (`{% extends %}` / `{% include %}`) work at the markup level, not the JS-behavior level
- **Same browser APIs as the React courses** — `getUserMedia` , `BarcodeDetector` , ZXing fallback — just packaged differently
- **A successful scan triggers a full page reload** — a genuine server round-trip, not an in-memory update; Chapter 11's DRF work is where this could change
- **Cleanup is genuinely harder here** — only the coded exit path stops the camera; no lifecycle guarantee covers every way a user might leave
- **Next chapter:** Building the Add-Item Flow with Django Forms

CHAPTER 6 OF 13

The Add-Item Form

Food Tracker (Django)

Chapter 6 · Building the Add-Item Flow with Django Forms

Chapter 5's scan flow redirects here once a barcode's been looked up. This chapter builds the actual confirm-and-save step — and shows one of Django's most concrete "batteries included" payoffs yet.

`ModelForm`: Deriving a Form From the Model, Not Duplicating It

```
# pantry/forms.py
from django import forms
from .models import Item

class ItemForm(forms.ModelForm):
    class Meta:
        model = Item
        fields = ["name", "barcode", "category", "expiry_date"]
        widgets = {
            "expiry_date": forms.DateInput(attrs={"type": "date"}),
        }
```

`ModelForm` generates both the form fields *and* their validation rules directly from Chapter 2's own `Item` model — `max_length`, `blank`, `choices`, all of it carries over automatically. FastAPI's own equivalent needs a separate Pydantic schema class, hand-written to mirror the same fields the SQLAlchemy model already defines — a real duplication FastAPI's own philosophy accepts as a reasonable tradeoff for its thinner-layer design. Django's `ModelForm` reads the single source of truth directly instead of restating it.

One View, Both GET and POST

```
def add_item(request, name="", category=""):
    if request.method == "POST":
        form = ItemForm(request.POST)
        if form.is_valid():
            item = form.save(commit=False)
            item.status = "active"
            item.save()
            return redirect("item_list")
    else:
        form = ItemForm(initial={"name": name, "category": category})
    return render(request, "pantry/add_item.html", {"form": form})
```

One function handles the entire request cycle for this URL: a `GET` shows a blank or pre-filled form, a `POST` validates and saves it. A typical FastAPI/REST-style app usually splits this into two distinct endpoints instead — a genuine structural difference beyond just where routing lives (Chapter 4), extending into how the request/response cycle itself is organized.

`commit=False` returns the unsaved model instance the form built, without writing it to the database yet — giving a chance to set `status = "active"` directly in code, a field deliberately left out of the form entirely so the user never controls it, before the actual `.save()` happens.

The Template

```
{% extends "pantry/base.html" %}
{% block content %}
<form method="post">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Save</button>
</form>
{% endblock %}
```

Where `ModelForm` Stops Being the Right Tool

Chapter 5 already flagged that a full-page redirect isn't the only way this could work — a future JSON-driven version of this flow would want a `fetch()` call instead of a traditional form POST. `ItemForm` genuinely isn't suited to that: it's built to render HTML and consume form-encoded POST bodies, not to produce or consume JSON. That's specifically what Django REST Framework's own **Serializers** are for — JSON in, JSON out, no HTML rendering assumed. This chapter's `ModelForm` approach is a real, complete, working feature on its own terms, not a placeholder DRF simply replaces; Chapter 11 picks the serializer-based path for a genuinely different reason (the interactive features that need real JSON), not because this chapter's approach was wrong.

> **THE SAME THEME, NOW VERY CONCRETE**

`ModelForm` reading directly from Chapter 2's model instead of restating its fields, and `{% csrf_token %}` requiring zero extra library to get real CSRF protection, are both direct continuations of Chapter 1's own "batteries included" framing — not abstract philosophy anymore, but two specific, working pieces of code that exist because the framework decided to provide them.

VALIDATION ERRORS RENDER THEMSELVES

`{{ form.as_p }}` automatically displays any field-level validation errors back to the user, right alongside the offending field — no extra template code needed to wire that up.

A MISSING CSRF_TOKEN ISN'T A COSMETIC OMISSION

Forgetting `{% csrf_token %}` in a POST form causes Django's own CSRF middleware to reject the submission outright with a 403 Forbidden — a confusing error for anyone who doesn't yet know why, but genuinely the correct, secure default behavior working as designed, not a bug. This protection exists automatically, for every POST form, without installing anything extra.

Where This Course Is Headed

Expiry alerts, item history and search, marking items used, recipe lookup, Django REST Framework — which revisits this chapter's own ModelForm-vs-serializer fork directly — deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain why Django's ModelForm avoids a duplication that FastAPI's own Pydantic-schema approach accepts, tying your answer back to Chapter 2's own model definition.

View solution

EXERCISE 2

Explain what commit=False actually does, and why status is set directly in the view rather than being included as a field in ItemForm itself.

View solution

EXERCISE 3

Explain exactly what happens if {% csrf_token %} is omitted from this form's template, and why this chapter treats Django's default CSRF behavior as a real security feature rather than boilerplate to work around.

View solution

CHAPTER 6 QUICK REFERENCE

- `ModelForm` — derives fields and validation from Chapter 2's model directly; no separate schema to duplicate
- **One view, GET and POST** — a single function handles both, unlike a typical split REST endpoint pair
- `commit=False` — set fields the user shouldn't control (like `status`) before the actual save
- `{% csrf_token %}` — required for every POST form; omitting it means a 403, by design, not a bug
- **ModelForm vs. DRF serializers** — a real, honest fork; this chapter's approach is complete on its own terms, not a placeholder
- **Next chapter:** Expiry Alerts

CHAPTER 7 OF 13

Expiry Alerts

Food Tracker (Django)

Chapter 7 · Expiry Alerts

With items actually being added, this chapter builds the first payoff feature: surfacing what's about to go to waste.

The QuerySet

```
from datetime import timedelta
from django.utils import timezone

def dashboard(request):
    threshold = timezone.now().date() + timedelta(days=3)
    expiring_items = Item.objects.filter(
        status="active", expiry_date__lte=threshold
    ).order_by("expiry_date")
    return render(request, "pantry/dashboard.html", {"items": expiring_items})
```

`expiry_date__lte` is Django's own field-lookup syntax — a double-underscore suffix mapping directly to a SQL comparison (`__lte` → `<=`, `__gte` → `>=`, `__contains` → `LIKE`, and so on). SQLAlchemy, Python's other major ORM, takes a genuinely different syntactic approach — operator overloading, so the identical comparison reads as `Item.expiry_date <= threshold` directly. Both are real, working styles for the same underlying idea; recognizing Django's own double-underscore convention on sight is worth having, since it looks unusual coming from any other ORM.

`timezone.now()`, Not `datetime.now()`

Django's modern default, `USE_TZ=True`, makes the project timezone-aware. Using plain `datetime.now()` in a timezone-aware project produces a **naive** datetime with no timezone attached at all — comparisons against Django's own timezone-aware stored values can then silently be wrong, especially once the deployed server and its users don't share a timezone. `django.utils.timezone.now()` is Django's own answer to exactly the same category of problem every Food Tracker course has hit in its own way — never trust the wrong clock.

No Composite Index Drama Here — a Genuinely Fair Point for This Course

Food Tracker (React + Firebase)'s own Chapter 7 needed an entire section on Firestore's composite-index requirement — a query combining an equality filter and a range filter on two different fields simply refuses to run at all until an index is explicitly declared. Django's ORM, running on a real relational database, has no equivalent hard requirement: the identical two-field filter here just runs. It might run slowly without an index as the table grows — ordinary SQL's own honest tradeoff, matching that same comparison the Firebase course already drew — but it never refuses outright the way Firestore does. This is one of the few places in the whole quartet where the relational side genuinely has the easier time, worth saying plainly rather than only ever finding fault with it.

```
# Optional — a real performance index, not a functional requirement
class Meta:
    indexes = [models.Index(fields=["status", "expiry_date"])]
```

The Template

```
{% extends "pantry/base.html" %}
{% block content %}
<h1>Expiring Soon</h1>
<ul>
  {% for item in items %}
    <li>{{ item.name }} — expires {{ item.expiry_date }}</li>
  {% empty %}
    <li>Nothing expiring soon.</li>
  {% endfor %}
</ul>
{% endblock %}
```

`{% empty %}` is Django's own built-in "if the loop had nothing to iterate" branch, right inside the `{% for %}` tag — no separate `{% if items %}` wrapper needed around it.

A FAIR ADVANTAGE, STATED PLAINLY

This is the one chapter in the whole quartet where the relational courses genuinely have it easier than the document-database one: a two-field filter query here simply runs, where the same shape of query on Firestore needs an index declared ahead of time or it fails outright. Every architecture in this comparison has real strengths and real costs — this is one of the SQL side's real strengths, worth naming without hedging.

TEST THE QUERYSET BEFORE THE VIEW EXISTS

`python manage.py shell`, then `Item.objects.filter(status="active", expiry_date__lte=timezone.now().date() + timedelta(days=3))` — the same isolate-and-verify habit already established for the model (Chapter 3) and the barcode view (Chapter 4).

A NAIVE DATETIME CAN SILENTLY PRODUCE A WRONG COMPARISON

With `USE_TZ=True` (Django's modern default), mixing a naive `datetime.now()` value into a comparison against timezone-aware stored values doesn't necessarily raise an error — it can just as easily produce a subtly wrong result, especially once the deployed server and its users span different timezones. `timezone.now()` avoids the ambiguity entirely by staying timezone-aware throughout.

Where This Course Is Headed

Item history and search, marking items used, recipe lookup, Django REST Framework, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain Django's expiry_date__lte field-lookup syntax, and show how SQLAlchemy would express the identical comparison using its own operator-overloading style.

View solution

EXERCISE 2

Explain why this chapter's own two-field filter query doesn't need anything like the Firebase sibling course's composite-index requirement, and what tradeoff still exists if no database index is ever added.

View solution

EXERCISE 3

Explain concretely what could go wrong if datetime.now() were used instead of timezone.now() in this chapter's own dashboard view, given USE_TZ=True.

View solution

CHAPTER 7 QUICK REFERENCE

- `__lte` — Django's own double-underscore field-lookup syntax, vs. SQLAlchemy's operator overloading
- `timezone.now()` — always, not `datetime.now()`, once `USE_TZ=True`
- **No composite index required** — a genuine, fair SQL advantage over the Firebase sibling course's own Chapter 7
- `Meta.indexes` — an optional performance optimization here, not a hard functional requirement

- `{% empty %}` — a built-in empty-loop branch inside `{% for %}`
- **Next chapter:** Item History & Live Search-as-You-Type

CHAPTER 8 OF 13

Item History & Live Search

Food Tracker (Django)

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 lands on a genuinely different architecture than this app's Firebase-based sibling course did.

The History View

```
def history(request):
    items = Item.objects.all().order_by("-added_at")
    return render(request, "pantry/history.html", {"items": items})
```

A Thin JSON Endpoint — Deliberately Not DRF Yet

Live search genuinely needs the frontend to ask the server "what matches this text?" repeatedly, without a full page reload per keystroke — exactly the kind of interactive feature Chapter 6 flagged as DRF's own eventual territory. But standing up Django REST Framework's full machinery for one simple, read-only endpoint would be premature. This chapter takes the honest middle path: a single minimal view returning `JsonResponse` directly, no serializer class involved. DRF's real payoff (Chapter 11) arrives once several endpoints and genuinely richer validation actually justify it — not before.

```
def search_items(request):
    query = request.GET.get("q", "")
    items = Item.objects.filter(name__icontains=query)[:20] if query else Item.objects.none()
    results = [{"id": i.id, "name": i.name, "status": i.status} for i in items]
    return JsonResponse(results, safe=False)
```

`name__icontains` is a real, native, case-insensitive substring match, built directly into Django's ORM — one line, no workaround, no shadow field. `safe=False` is required because `JsonResponse` defaults to expecting a dict (a historical safeguard against a JSON-hijacking risk that applied to older browsers when a bare array was the top-level response); returning a plain list means opting out of that default explicitly.

A Genuinely Fair, Direct Contrast

Food Tracker (React + Firebase)'s own Chapter 8 needed real, honest work to get case-insensitive, substring-anywhere matching at all — Firestore has no equivalent of SQL's `LIKE`, so that course built a `nameLower` shadow field and ultimately chose client-side filtering specifically to get genuine mid-word matching. Here, on a real relational database, `icontains` already does exactly that, natively, in the query itself. This is a genuine SQL-side advantage worth crediting plainly, the same way Chapter 7 already credited SQL honestly for not needing a composite index.

The Frontend Genuinely Needs Debouncing This Time

This is the one place this course's own approach is more expensive per keystroke than the Firebase course's own choice, and worth being precise about why: this view hits the real database on every single request, rather than filtering an already-loaded, small array in memory. Debouncing — waiting for typing to pause before actually firing the request — genuinely matters here, unlike the Firebase course's own chosen approach, which loaded the full history once and needed no debounce at all because filtering afterward never touched the network again.

```
// pantry/static/pantry/search.js
let debounceTimer;
document.getElementById("search-box").addEventListener("input", (e) => {
  clearTimeout(debounceTimer);
  const query = e.target.value;
  debounceTimer = setTimeout(async () => {
    const response = await fetch(`/search/?q=${encodeURIComponent(query)}`);
    const results = await response.json();
    renderResults(results);
  }, 300);
});
```

TWO LEGITIMATE ARCHITECTURES, SHAPED BY TWO DIFFERENT CONSTRAINTS

Food Tracker (React + Firebase) chose client-side filtering: zero network cost per keystroke, no debounce needed, but explicitly scoped to a realistically small personal dataset. This course chooses a server round-trip per search: it needs debouncing, but it scales cleanly to a history far larger than "load it all into memory once" could ever comfortably handle, and gets substring/case-insensitive matching for free via `icontains` rather than needing a workaround field at all. Neither is the universally "right" search architecture — each is the correct answer to a different actual constraint.

TEST THE ENDPOINT DIRECTLY FIRST

Visiting `/search/?q=chedd` in a browser returns the raw JSON — confirming the query itself works before writing a single line of the debounced frontend JS.

`[:20]` LIMITS THE RESPONSE SIZE, NOT THE QUERY COST

Slicing to 20 results controls how much comes *back*, but a leading-wildcard `LIKE '%...%'` query (which is what `icontains` compiles to) generally can't use a standard B-tree index efficiently regardless of how few rows are ultimately returned — the database may still have to scan a large portion of the table to find those 20 matches. At real scale, a genuine full-text or trigram search feature (PostgreSQL's own trigram extension, for instance) would be the actual fix, not a smaller result slice.

Where This Course Is Headed

Marking items used, recipe lookup, Django REST Framework — now genuinely justified by more than one interactive endpoint — deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain what name__icontains gives this course for free, and what workaround the Firebase sibling course's own Chapter 8 needed to build to get roughly the same capability.

View solution

EXERCISE 2

Explain why this course's search needs debouncing while the Firebase sibling course's own chosen search approach didn't. What's the actual underlying difference in where the filtering happens?

View solution

EXERCISE 3

Explain why slicing the queryset to [:20] doesn't actually solve the performance concern with a LIKE '%...%' query, and what this chapter names as the real fix at genuine scale.

View solution

CHAPTER 8 QUICK REFERENCE

- `name__icontains` — free, native, case-insensitive substring search; no shadow field needed, unlike the Firebase sibling course
- **A thin** `JsonResponse` **view** — deliberately not DRF yet; one endpoint doesn't justify the full framework

- `safe=False` — required to return a plain JSON list rather than a dict
- **Debouncing genuinely needed here** — every search hits the real database, unlike the Firebase course's own load-once-filter-in-memory approach
- `[:20]` **limits the response, not the query cost** — a leading-wildcard `LIKE` can't use a standard index regardless
- **Next chapter:** Marking Items Used

CHAPTER 9 OF 13

Marking Items Used

Food Tracker (Django)

Chapter 9 · Marking Items Used

A short chapter — and a good place to see how a decision this app's Firebase-based sibling course treated very carefully turns out to need almost no ceremony at all in a relational model.

The View

```
def mark_used(request, item_id):
    item = get_object_or_404(Item, id=item_id)
    item.status = "used"
    item.expiry_date = None
    item.used_at = timezone.now()
    item.save()
    return redirect("item_history")
```

`get_object_or_404` is a small, genuinely useful Django shortcut — it combines the lookup and an automatic 404 response if nothing matches, instead of a manual `try` / `except Item.DoesNotExist`.

`expiry_date = None`: Simply the Right Answer Here

Food Tracker (React + Firebase)'s own Chapter 9 spent real effort explaining why `deleteField()`, not `null`, was the correct choice — because in Firestore's schema-on-read model, a document created used-from-the-start never had the field at all, and setting it to `null` on an existing document would have created a second, inconsistent representation of "no expiry." That entire problem simply doesn't exist here. Chapter 2's `expiry_date` column is **always present**, on every row, regardless of history — a used item created directly as used already has `expiry_date = NULL` from the very start, exactly matching what this view produces on a transition. There's only ever one shape for "no expiry" in a schema-on-write model, because every row already shares the identical set of columns.

TWO DIFFERENT MECHANICS, THE SAME CORRECT UNDERLYING PRINCIPLE

Neither course's approach is more "correct" than the other — `deleteField()` and `None` are each the right tool for representing an identical idea ("this item has no expiry date") inside two genuinely different storage models. Schema-on-read has to actively choose between absence and a stored null, and picking wrong creates two different

representations. Schema-on-write never has that choice to make at all — every row already has the same columns, so there's exactly one way to represent "no expiry," and it's simply `NULL`.

Should This Even Be a Plain Link?

`mark_used` takes no user input — no form fields, nothing to fill in. It's tempting to trigger it from a simple `<a href="/mark-used/<id>/">` link. Resist that: a state-*changing* action behind a plain `GET` request is a real, substantive correctness problem, not a stylistic nitpick — browser prefetching, crawlers following links, or the browser's own back/forward cache can all trigger a `GET` request without the user ever intending to. `GET` is supposed to be safe and non-mutating per HTTP's own semantics; this action should be a `POST`, even with no actual form fields to collect:

```
<form method="post" action="{% url 'mark_used' item.id %}">
  {% csrf_token %}
  <button type="submit">Mark Used</button>
</form>
```

REACH FOR `GET_OBJECT_OR_404` EVERYWHERE A LOOKUP CAN FAIL

Any view fetching a specific row by ID benefits from the same pattern — a clean, automatic 404 instead of an unhandled exception if the ID doesn't exist.

A GET LINK FOR A MUTATING ACTION IS A REAL BUG WAITING TO HAPPEN

A search engine crawler, a browser preloading a hovered link, or simply pressing back after marking something used could all silently re-trigger this exact action if it's reachable via `GET` — genuinely different from a cosmetic style preference, since it can cause real, unintended state changes with no user action actually behind them.

AN "UNDO" WOULD STILL NEED TO RESTORE THE VALUE FROM SOMEWHERE

Once `expiry_date` is set to `None` and saved, the original value is genuinely gone from that row — exactly the same limitation Food Tracker (React + Firebase)'s own Chapter 9 already named for `deleteField()`. A real undo feature would need to have captured the original date somewhere before clearing it (briefly, in the session, for an "undo" window) or simply ask the user to re-enter it — the same underlying lesson, now confirmed a second time in a completely different storage model.

Where This Course Is Headed

Recipe lookup, Django REST Framework, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain why expiry_date = None is sufficient here, while the Firebase sibling course specifically needed deleteField() rather than null. What single underlying difference between the two courses' storage models explains this?

View solution

EXERCISE 2

Explain the real problem with triggering mark_used from a plain GET link, giving at least one concrete scenario where this could cause an unintended state change.

View solution

EXERCISE 3

Explain why a future "undo" feature can't simply set expiry_date back to a value after this view has already run, and what a genuine undo implementation would need to do instead.

View solution

CHAPTER 9 QUICK REFERENCE

- `expiry_date = None` — sufficient here; schema-on-write guarantees one consistent shape for "no expiry" regardless of history
- `get_object_or_404` — lookup + automatic 404, instead of a manual exception handler
- **A mutating action must be POST** — never a plain GET link, even with no form fields to collect
- **Undo still isn't free** — the original value is genuinely gone once saved, the same limitation as the Firebase course's own `deleteField()`
- **Next chapter:** Recipe Lookup with TheMealDB

CHAPTER 10 OF 13

Recipe Lookup with TheMealDB

Food Tracker (Django)

Chapter 10 · Recipe Lookup with TheMealDB

This chapter delivers the last named feature from Chapter 1's original spec, reusing the same integration pattern Chapter 4 already established — and running into a real, honest limitation specific to how this course's views are built.

The Same Limitation, Regardless of Framework

TheMealDB's filter-by-ingredient endpoint searches exactly one ingredient per request — this isn't a Django, FastAPI, or Firebase-specific constraint at all, it's simply a property of the external API itself, identical across all four Food Tracker courses. Every one of them has to fan out one request per expiring ingredient and merge the results afterward.

A View That Fans Out

```
def suggest_recipes(request):
    expiring = Item.objects.filter(status="active", expiry_date__lte=threshold)
    ingredients = [item.name for item in expiring]

    merged = {}
    for ingredient in ingredients:
        response = requests.get(
            f"https://www.themealdb.com/api/json/v1/1/filter.php?i={ingredient}"
        )
        data = response.json()
        for meal in (data.get("meals") or []):
            meal_id = meal["idMeal"]
            if meal_id not in merged:
                merged[meal_id] = {**meal, "matched_ingredients": []}
            merged[meal_id]["matched_ingredients"].append(ingredient)

    results = sorted(
        merged.values(), key=lambda m: len(m["matched_ingredients"]), reverse=True
    )
    return JsonResponse(results, safe=False)
```

A Real Cost of This Sequential Loop

Each `requests.get()` call here runs one after another, and this ordinary Django view blocks entirely until every single one finishes — five expiring ingredients means roughly five times the wait of firing them all at once. This is a genuine, honest cost specific to how this view is written: Food Tracker (FastAPI)'s own async patterns and Food Tracker (React + Firebase)'s `Promise.all()` both fire every ingredient lookup **concurrently** instead. Django does support async views (`async def`, since Django 3.1), but doing this properly would also require swapping the synchronous `requests` library for an async-compatible HTTP client — real additional complexity genuinely beyond this chapter's own scope, named honestly rather than glossed over.

Relevance Sorting — The One Place All Four Courses Converge

Sorting by `len(matched_ingredients)` descending surfaces recipes using the most expiring items first — directly serving Chapter 1's original point: using up as much soon-to-expire food as possible in one meal. This particular piece of logic is functionally identical across all four Food Tracker courses, each expressed in its own language's idioms — a genuine convergence point after several chapters of real, honest divergence.

Caching, Same Reasoning, a New Kind of Column

```
class RecipeCache(models.Model):
    ingredient = models.CharField(max_length=200, primary_key=True)
    meals_json = models.JSONField()
```

`JSONField` (native since Django 3.1, backed by real JSON column support in modern PostgreSQL, MySQL, and SQLite) is a genuinely interesting nuance worth naming directly: even a relational, schema-on-write framework has its own escape hatch for storing semi-structured, document-like data in a single column, when that's honestly the better fit — a list of matched-meal dictionaries here doesn't cleanly decompose into further normalized tables for this app's own modest needs. Relational and document approaches aren't a strict either/or; a real relational database can hold document-shaped data exactly where that's the more sensible choice.

THIS TIME, THE OTHER COURSES HAVE THE ADVANTAGE

Chapters 7 and 8 credited this course's own SQL-based approach with real, fair advantages over the Firebase sibling. This chapter is the honest opposite case: a naive sequential loop here is genuinely slower than the concurrent fan-out both the FastAPI and Firebase courses use for the identical task. Keeping the comparison honest means naming a real cost when there is one, not just the wins.

`JSONFIELD` IS WORTH KNOWING ABOUT GENERALLY

Beyond this one cache, it's the right tool anytime a piece of data is genuinely variable in shape or doesn't need its own normalized table — a small, real escape hatch inside an otherwise strict, columnar model.

THIS VIEW IS GENUINELY SLOWER THAN IT NEEDS TO BE

The sequential loop above blocks on every single TheMealDB request in turn. This is a known, real limitation of the simple version built in this chapter — not a hidden defect, but a cost worth being honest about rather than assuming a synchronous view is automatically "fine" regardless of how many external calls it makes.

Where This Course Is Headed

Django REST Framework, deployment, and a capstone.

Hands-On Exercises

EXERCISE 1

Explain why this view takes roughly N times as long as necessary for N expiring ingredients, and what the FastAPI and Firebase sibling courses do differently to avoid this specific cost.

View solution

EXERCISE 2

Explain how the matched_ingredients sort serves Chapter 1's original recipe-lookup intent, and why this chapter calls this one piece of logic essentially identical across all four Food Tracker courses.

View solution

EXERCISE 3

Explain what JSONField is, and why using it for RecipeCache doesn't actually contradict Django's own relational, schema-on-write identity.

View solution

CHAPTER 10 QUICK REFERENCE

- **One ingredient per request** — TheMealDB's own limitation, identical across all four courses

- **Sequential, blocking loop** — a real, honest cost; FastAPI's async and Firebase's `Promise.all()` both avoid it by running concurrently
- `matched_ingredients` **sort** — the one piece of logic functionally identical across the whole quartet
- `JSONField` — Django's own escape hatch for document-shaped data inside a relational schema
- **Next chapter:** Django REST Framework: Exposing an API Layer

CHAPTER 11 OF 13

Django REST Framework

Food Tracker (Django)

Chapter 11 · Django REST Framework: Exposing an API Layer

Chapter 8 deliberately avoided DRF for one simple search endpoint. By now there are four genuinely JSON-shaped features — search, add-item, mark-used, recipe lookup — plus Chapter 5's own full-page-reload cost still unresolved. That's the actual threshold this chapter waits for.

Installing DRF

```
pip install djangorestframework
```

```
INSTALLED_APPS = [
    ...,
    "rest_framework",
    "pantry",
]
```

A Serializer Isn't a Second `ModelForm`

```
from rest_framework import serializers
from .models import Item

class ItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = Item
        fields = ["id", "name", "barcode", "category", "status", "expiry_date", "added_at", "used_at"]
        read_only_fields = ["status", "added_at", "used_at"]
```

Like Chapter 6's `ModelForm`, `ModelSerializer` derives its fields directly from the same `Item` model — no separate schema class to duplicate. But it's built for JSON in, JSON out, with no HTML rendering assumption at all. And `read_only_fields` works differently from Chapter 6's own `commit=False` pattern: `commit=False` lets the view set server-controlled fields *after* the form validates but *before* saving; `read_only_fields` instead tells the serializer itself to include a field in *output* (a GET response) while silently ignoring any attempt to set it via *input* — a genuinely different mechanism arriving at a similarly-shaped safety outcome, not the identical technique wearing a new name.

ViewSets and Routers: The Pattern Repeats, One Level Up

```python
from rest_framework import viewsets
from .models import Item
from .serializers import ItemSerializer

class ItemViewSet(viewsets.ModelViewSet):
    queryset = Item.objects.all()
    serializer_class = ItemSerializer
```

```python
# urls.py
from rest_framework.routers import DefaultRouter

router = DefaultRouter()
router.register("items", ItemViewSet)
urlpatterns = [path("api/", include(router.urls))]
```

One `ViewSet` class, registered with a router, auto-generates a full set of RESTful endpoints — list, create, retrieve, update, delete — with no individual view functions written by hand for any of them. It's Django's own "batteries included" philosophy repeating itself one level deeper, this time inside DRF specifically.

The Honest Hybrid Shape

This chapter doesn't throw away Chapters 3 through 10 and replace everything with DRF. The admin (Chapter 3) stays exactly what it was. The dashboard and history pages could keep their server-rendered templates for the parts that don't need deep interactivity, while search, add-item, mark-used, and recipe lookup can genuinely migrate to these new `/api/` endpoints, called via `fetch()` instead of a full-page form POST or redirect. Some pages stay templates; some features become a real JSON API. This hybrid shape is the honest, common reality most substantial Django applications actually settle into — not a failure to fully commit to one philosophy.

CHAPTER 8'S OWN REASONING, RESOLVED THE OTHER WAY

Chapter 8 declined DRF because one endpoint didn't justify it. This chapter reaches for it precisely because there are now enough genuinely interactive features, sharing enough real serialization and validation needs, that the earlier cost/benefit calculation flips. Neither decision was wrong — they were the correct answer at two different points in the same app's own growth.

THE BROWSABLE API IS WORTH SEEING DIRECTLY

Visiting `/api/items/` in an ordinary browser renders a genuinely pleasant, interactive HTML exploration UI for the JSON API itself — automatically, with zero extra code required to get it.

SWITCHING TO JSON DOESN'T SWITCH OFF CSRF PROTECTION

Migrating Chapter 5's scan-and-redirect flow to instead `fetch()` a DRF endpoint directly from the browser doesn't exempt that call from Django's own CSRF protection — a same-origin, session-authenticated browser request still needs the CSRF token included in the request's own headers for any unsafe method (POST/PUT/DELETE). Forgetting it produces the exact same 403 rejection Chapter 6 already covered for a template form, just now from JavaScript instead of an HTML `<form>`.

Where This Course Is Headed

Deployment, and a capstone tying every chapter together into one complete, working app.

Hands-On Exercises

EXERCISE 1

Explain why this chapter is genuinely the right point to introduce DRF, tying your answer back to Chapter 8's own reasoning for avoiding it. What specifically changed between Chapter 8 and now?

View solution

EXERCISE 2

Explain the difference between Chapter 6's commit=False pattern and this chapter's read_only_fields. Both prevent user control of a field — how do they actually achieve that differently?

View solution

EXERCISE 3

Explain why switching Chapter 5's scan flow from a template form POST to a fetch() call against a DRF endpoint doesn't bypass Django's CSRF protection, and what would happen if the CSRF token were left out of that fetch() call.

View solution

CHAPTER 11 QUICK REFERENCE

- `ModelSerializer` — derives from the same model as `ModelForm`, but for JSON, not HTML
- `read_only_fields` **vs.** `commit=False` — genuinely different mechanisms, similar safety outcome
- `ViewSet` **+ router** — one class auto-generates a full RESTful endpoint set
- **The hybrid shape** — templates and a JSON API coexisting is the honest norm, not a compromise

- **CSRF still applies to same-origin `fetch()` calls** — the token must be sent explicitly in headers
- **Next chapter:** Deployment

CHAPTER 12 OF 13

Deployment

Food Tracker (Django)

Chapter 12 · Deployment

Every earlier chapter's "batteries included" moments were real. This one is where that story honestly stops covering everything.

`DEBUG=False`: The Single Most Important Flag

Django's development default, `DEBUG=True`, shows full, detailed error pages — stack traces, local variable values, even source snippets. Genuinely useful while developing; genuinely dangerous left on in production, since it hands anyone who triggers an error a real window into secrets, internal paths, and database structure. `DEBUG=False` swaps that for a generic error page instead — but it also activates a real requirement that trips up nearly every first-time Django deployment:

```
# settings.py (production)
DEBUG = False
ALLOWED_HOSTS = ["foodtracker.example.com"]
SECRET_KEY = os.environ["DJANGO_SECRET_KEY"]
```

With `DEBUG=False`, Django refuses to serve *any* request at all if its `Host` header doesn't match something in `ALLOWED_HOSTS` — everything that worked perfectly with `runserver` locally can appear completely broken the moment `DEBUG` flips to `False`, purely because `ALLOWED_HOSTS` was never configured for the real domain.

`SECRET_KEY`: This Course's First Genuine Secret

Every external API this course has touched — Open Food Facts, TheMealDB — never needed a real secret at all. Django's own `SECRET_KEY` is different: it signs sessions, CSRF tokens (Chapter 6's own protection depends on it), and password-reset tokens. If it leaks, an attacker can forge valid session cookies directly. It must be loaded from an environment variable, never hardcoded or committed to source control — the one place in this entire course where an actual secret genuinely exists, and it belongs to Django itself, not to anything this app integrates with.

Static Files vs. Media Files

Static files are the app's own assets — CSS, and Chapter 5's own `scan.js`. **Media** files would be user-uploaded content (this app has none currently, but the distinction is worth knowing since it's a common point of confusion). `python manage.py collectstatic` gathers every app's static files into one directory for production serving — a step Django's own dev server never required, since it serves static files automatically during development without it.

Django itself explicitly does not serve static files efficiently in production. A real deployment needs either a dedicated web server (nginx) in front, or a library like WhiteNoise letting the Django app serve them itself reasonably well — a genuine decision point this chapter names rather than glosses over.

SQLite in Production: An Honest, Calibrated Note

Chapter 2 called SQLite the genuine, appropriate choice for this app, not a placeholder — and for this app's own realistically small, personal-use scope, that can remain true in production too. The honest caveat: SQLite's own file-level locking becomes a real concurrency bottleneck under genuine multi-user, high-write-volume load. If this were ever a shared, heavily-used deployment, PostgreSQL is the standard upgrade path — not because SQLite is "never production-ready" (a common overstatement), but because its real limits are concurrency-shaped, and worth knowing about honestly rather than either dismissing or overselling.

Running It For Real

```
pip install gunicorn
gunicorn foodtracker.wsgi:application
```

Chapter 1's `manage.py runserver` was always explicitly a development-only server — Django's own documentation says directly not to use it in production, since it offers neither real concurrency nor production security hardening. Gunicorn (or another genuine WSGI server) is what actually serves production traffic.

WHERE "BATTERIES INCLUDED" HONESTLY STOPS

Django ships an ORM, an admin, forms, and — as of Chapter 11 — a REST framework, all genuinely included. It does not ship a production-grade web server, an efficient static-file server, or a database that scales to arbitrary concurrent load. No framework's "batteries included" promise extends infinitely — this chapter is simply the honest, unavoidable place that becomes visible.

LET DJANGO CHECK ITS OWN CONFIGURATION

`python manage.py check --deploy` scans the current settings for common production misconfigurations — `DEBUG` still `True`, missing `SECURE_*` settings, and more — worth running before trusting a deployment is actually ready.

THE CLASSIC FIRST-DEPLOYMENT MISTAKE

Flipping `DEBUG` to `False` without also setting `ALLOWED_HOSTS` correctly makes the entire site appear to stop working — every request gets rejected outright, with no obvious explanation unless the connection between these two settings is already understood.

Where This Course Is Headed

One chapter left: a capstone tying together this course's own thread — from Chapter 1's batteries-included framing through this chapter's own real production considerations — into one complete, working app.

Hands-On Exercises

EXERCISE 1

Explain why DEBUG=True is appropriate in development but a real security risk in production, and explain exactly what happens (and why) once DEBUG=False if ALLOWED_HOSTS hasn't been configured for the real domain.

View solution

EXERCISE 2

Explain why SECRET_KEY is a genuine secret in a way Open Food Facts and TheMealDB never required, and name at least two things it's actually used to protect.

View solution

EXERCISE 3

Explain why manage.py runserver and SQLite were both genuinely appropriate choices for earlier chapters, and why deployment is specifically where each one's own real limits actually become visible.

View solution

CHAPTER 12 QUICK REFERENCE

- `DEBUG=False` + `ALLOWED_HOSTS` — the classic first-deployment mistake if not set together
- `SECRET_KEY` — this course's first genuine secret; signs sessions, CSRF tokens, password resets
- `collectstatic` — a real production-only step; the dev server never needed it
- **SQLite in production** — genuinely fine for this app's own scope; PostgreSQL is the real upgrade path for concurrency, not a blanket "never production-ready" rule

- **Gunicorn, not** `runserver` — `runserver` was always explicitly dev-only
- **Next chapter:** Capstone — A Complete, Working Food Tracker

CHAPTER 13 OF 13

Capstone: A Complete, Working Food Tracker

Food Tracker (Django)

Chapter 13 · Capstone: A Complete, Working Food Tracker

Aisha manages this deployment day to day. Every step below is a real, working feature, each one built in a specific earlier chapter.

STEP 1 — SEEDING THE FIRST TEST DATA VIA THE ADMIN

Before a single real user touches the app, Aisha logs into `/admin/` and adds a couple of test items by hand — Chapter 1's own promise, delivered concretely by Chapter 3's two-line `admin.site.register(Item)`, exercised here exactly as intended: real CRUD, before any custom view existed.

STEP 2 — SCANNING A REAL ITEM

A user scans a yogurt carton's barcode. Chapter 5's `scan.js` decodes it and navigates to Chapter 4's lookup view, which checks `BarcodeCache`, finds nothing, queries Open Food Facts, and caches the result. The confirm-and-save form (Chapter 6) pre-fills name and category; `commit=False` sets `status="active"` before the actual save.

STEP 3 — WHAT'S ACTUALLY IN THE ROW

The real database row reflects every decision Chapter 2 made: `name`, `barcode`, `category="dairy"`, `status="active"`, a real `expiry_date`, and `added_at` set automatically by `auto_now_add` — never trusting a client-supplied value for that field.

STEP 4 — THE DASHBOARD FLAGS IT

A few days later, Chapter 7's dashboard query — `status="active"`, `expiry_date__lte` a `timezone.now()`-based threshold — surfaces the item as expiring soon, with no composite-index drama at all, exactly the fair SQL advantage that chapter named.

STEP 5 — SEARCHING HER HISTORY

Wanting to buy more of something bought before, a user types `"yog"`. Chapter 8's debounced JS hits `/search/`, which uses `name__icontains` — genuine, native, case-insensitive substring matching, no shadow field ever needed — finding **"Greek Yogurt"** instantly.

STEP 6 — MARKING IT USED

The yogurt gets finished. A POST-only "Mark Used" button — never a plain GET link, per Chapter 9's own correctness point — sets `status="used"`, stamps `used_at`, and sets `expiry_date = None`. The item stays in history forever, exactly as Chapter 2 designed, just without an expiry date attached anymore.

STEP 7 — A RECIPE SUGGESTION

With chicken and eggs both nearing expiry, Chapter 10's `suggest_recipes` view fans out one request per ingredient to TheMealDB — sequentially, with the honest performance cost that chapter named — merges the results, and sorts by `matched_ingredients` count, surfacing a recipe using both chicken and eggs above one using only either alone.

STEP 8 — FASTER THAN IT USED TO BE

By now, the team has followed Chapter 11's own hybrid path: the admin and dashboard stayed exactly as they were, but search, add-item, mark-used, and recipe lookup moved behind real DRF `ViewSet`s, called via `fetch()` with the CSRF token included correctly. Chapter 5's own full-page-reload limitation — named honestly back when it was first built — is genuinely resolved here, not by rewriting that chapter's logic, but by Chapter 11 giving the app somewhere better to send its requests.

STEP 9 — ALL OF THIS, IN PRODUCTION

The whole session happens on a real deployment: `DEBUG=False` with `ALLOWED_HOSTS` set correctly, `SECRET_KEY` loaded from the environment, static files collected and served, gunicorn handling real traffic — SQLite still genuinely appropriate at this app's own realistic scale, exactly as Chapter 12 concluded.

Chapter Attribution

STEP	CHAPTER(S) APPLIED
1 — Admin seeding	Chapter 1 (batteries included), Chapter 3 (admin registration)
2 — Scanning an item	Chapter 5 (scanning), Chapter 4 (lookup + cache), Chapter 6 (add-item form, commit=False)
3 — The stored row	Chapter 2 (model design, auto_now_add)
4 — Expiry alert	Chapter 7 (queryset, timezone.now())
5 — Search	Chapter 8 (icontains, debounced JS)
6 — Marking used	Chapter 9 (expiry_date=None, POST-only action)
7 — Recipe suggestion	Chapter 10 (fan-out, relevance sort, JSONField cache)
8 — Faster interactions	Chapter 11 (DRF, resolving Chapter 5's own limitation)
9 — Real production	Chapter 12 (deployment)

WHAT THIS WHOLE COURSE WAS REALLY ABOUT

Chapter 1 opened by contrasting Django's batteries-included philosophy against FastAPI's thin-API-layer approach. Every step above is that same claim made concrete: an admin panel free, a form system reading directly from the same model, an ORM with native substring search and no composite-index requirement, and DRF as the exact same philosophy escalating one level further once genuinely justified. Where Food Tracker (FastAPI) would hand-write a serializer schema, where Food Tracker (React + Firebase) would need a shadow field for search, this course's own answer was already sitting inside the framework — not because Django is simply "better," but because it made a real, different bet about how much to decide up front.

HONEST SCOPE NOTE

This capstone deliberately stops short of several things: the weekly meal planner named as future work all the way back in Chapter 1 was never built; there's no offline/PWA support; no multi-user ownership model was ever built into this course at all — `django.contrib.auth` is genuinely available and easy to add, but this course never actually wired it in, so every `Item` row remains ownerless, unlike the Firebase sibling course's own Chapter 11; Chapter 10's own sequential recipe-lookup performance cost was named honestly but never actually fixed; and no automated test suite or CI pipeline was covered anywhere in this course. Each is a genuine, reasonable next step — none were quietly assumed to already be done.

Hands-On Exercises

EXERCISE 1

Explain how Step 8 resolves Chapter 5's own full-page-reload limitation without rewriting that chapter's own scanning logic. What specifically changed to make this possible?

View solution

EXERCISE 2

Pick any two steps from Aisha'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 specifically calls out the lack of a multi-user ownership model as a genuine gap, rather than assuming Django's built-in django.contrib.auth makes this a non-issue.

View solution

CHAPTER 13 QUICK REFERENCE — COURSE COMPLETE

- **9 steps, 12 prior chapters** — one continuous, realistic session with the finished, deployed app
- **This course's own throughline, closed out:** batteries included, one integrated framework deciding more up front than FastAPI's thin-layer approach
- **Honest scope note:** no meal planner, no offline/PWA, no multi-user ownership model despite `django.contrib.auth` being available, no fix for Chapter 10's own sequential recipe lookup, no automated tests/CI
- **Food Tracker (Django) is now complete** — 13/13 chapters