PERSONAL MULTI-TECHNOLOGY PROJECTS

Personal Catalogue: Django & PostgreSQL

A real, working catalogue for books, CDs, DVDs, and Blu-rays — a normalized Django/PostgreSQL schema, the Django admin as a fast entry tool, real public views, tag filtering, PostgreSQL full-text search, and a real deployment mounted onto an existing Astro site, closing with barcode scanning deliberately scoped out as an honest future feature rather than a day-one requirement.

10 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: PROJECT OVERVIEW & DJANGO PROJECT SETUP

Personal Catalogue: Django & PostgreSQL

Chapter 1 · Project Overview & Django Project Setup

This is one of four courses building the exact same personal cataloguing tool in four genuinely different architectures — Personal Catalogue (PHP & MySQL, complete — see `catalog-php1`), Personal Catalogue (React + Express + MongoDB), and Personal Catalogue (React & Firebase) are its siblings. This course's own answer leans into a real, normalized relational schema and Django's own ready-made admin interface — a genuinely different tradeoff from the PHP variant's own bare-metal speed.

What the App Actually Does

The shared spec every course in the quartet builds toward:

- **Catalogue four kinds of item** — books, CDs, DVDs, and Blu-rays — so it's possible to quickly check "do I already own this CD?" or "what Python books do I have?"
- **Tag books** with free-form labels (a FastAPI book might get Python, Web Development, Programming, Web Frameworks) so related books can be found by topic later.
- **Add items manually** for the first release — title, author/artist, format-specific details, tags where relevant.
- **Search and browse** the collection to answer the "do I have this?" question quickly.

Barcode scanning is explicitly out of scope for this first release — a real, obvious way to add items even faster, but deliberately deferred rather than allowed to hold up a release that needs to ship quickly. This course closes with a real, concrete plan for adding it later (Chapter 9), rather than pretending it isn't wanted at all.

Why Django + PostgreSQL for This One

Of the four variants in this set, Django + PostgreSQL is the one built around genuine data integrity and a real, ready-made tool for fast manual entry. Books, CDs, DVDs, and Blu-rays share several real fields (title, format, year) while differing in others, and tags need a genuine many-to-many relationship against books specifically — exactly the shape a normalized relational schema, and PostgreSQL's own real foreign-key enforcement, is built to express

cleanly, covered in full in Chapter 2. Django's own built-in admin interface — generated automatically from the models this course defines — does real double duty here: it's a genuinely fast, already-built manual data-entry tool, directly useful given this project's own real time pressure, covered in Chapter 3.

> **THIS COURSE ASSUMES REAL DJANGO/POSTGRESQL GROUNDWORK ALREADY COVERED ELSEWHERE**
>
> This site's own Django Fundamentals course already covers models, migrations, and the Django ORM in real depth, and the PostgreSQL course covers real relational schema design and query fundamentals. This course doesn't re-teach either — it assumes both, and moves straight into this catalogue's own real schema and features.

A Real Deadline, Not Just a Framing Device

This project genuinely needs a working first release quickly. That constraint shapes real decisions throughout this course — leaning on Django's own built-in admin instead of hand-building a data-entry UI first, and a deliberately un-fancy set of public-facing views before any polish — not because those are always the "correct" choices, but because they're the right choices *given this specific, real time pressure*. Chapter 8 covers styling and templates once the functional core is genuinely working.

Setting Up: Django, PostgreSQL & a Working Connection

A local Python install with `pip`, and a working local PostgreSQL server, are assumed from here on. Install Django and the PostgreSQL adapter into a virtual environment:

```
# Create and activate a virtual environment
python -m venv venv
source venv/bin/activate  # venv\Scripts\activate on Windows

# Install Django and the PostgreSQL adapter
pip install django psycopg2-binary
```

Create the PostgreSQL database this whole course builds on, using `psql` or your own preferred client:

```
CREATE DATABASE catalogue WITH ENCODING 'UTF8';
```

Starting the Django Project

Unlike the PHP variant, there's genuine project scaffolding here — Django generates a real, structured project layout rather than starting from a single connected file:

```
# Start the Django project and the catalogue app
django-admin startproject catalogue_site .
python manage.py startapp catalogue
```

Point the new project at the real PostgreSQL database in `catalogue_site/settings.py`:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'catalogue',
        'USER': 'postgres',
        'PASSWORD': '',  # set a real password in production
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Add `'catalogue'` to `INSTALLED_APPS`, then confirm the connection works by running Django's own initial migrations against the real database:

```
python manage.py migrate
python manage.py runserver
```

REAL SCAFFOLDING TO BUILD BEFORE THE FIRST PAGE LOADS

Unlike the PHP variant in this set — where a single `config.php` plus a working MySQL connection was the entire "setup" — Django genuinely generates a real project structure (settings, URL routing, an app skeleton) before any catalogue-specific code exists at all. That structure is exactly what pays

off from Chapter 3 onward, once Django's own admin interface and ORM start doing real work on top of it.

PSYCOPG2-BINARY IS FOR DEVELOPMENT, NOT PRODUCTION

`psycopg2-binary` bundles its own compiled PostgreSQL client libraries specifically to make local setup fast — genuinely useful for exactly the kind of quick first release this project needs. The real, documented tradeoff is that it isn't recommended for production use, where the plain `psycopg2` package, built against the system's own PostgreSQL client libraries, is the correct choice instead. Chapter 9's own deployment coverage returns to this distinction directly.

Where This Course Is Headed

A real normalized schema for items, item types, and a tags many-to-many (Chapter 2); the Django admin as a fast, ready-made manual entry tool (Chapter 3); the public-facing list and detail views (Chapter 4); a real Django form for adding items outside the admin, handling several genuinely different item shapes (Chapter 5); tags — filtering the catalogue by tag (Chapter 6); real search across the whole catalogue using the Django ORM (Chapter 7); styling and templates (Chapter 8); deployment (Chapter 9); and a capstone on integrating this catalogue into the existing Astro-based site, plus a real, concrete plan for barcode scanning as the next phase (Chapter 10).

Hands-On Exercises

EXERCISE 1

Explain why barcode scanning was deliberately deferred rather than built into this first release, and what real project constraint drove that decision.

View solution

EXERCISE 2

Explain the real, documented difference between `psycopg2-binary` and plain `psycopg2`, and why this project's own deadline makes the binary package the right choice for now.

View solution

EXERCISE 3

Set up the `catalogue` PostgreSQL database and a working Django project connected to it yourself, then write a one-sentence explanation of what `python manage.py migrate` actually confirmed by running successfully.

View solution

CHAPTER 1 QUICK REFERENCE

- **The shared app** — a personal catalogue for books/CDs/DVDs/Blu-rays, tags on books, manual entry first, barcode scanning deferred
- **Why this variant** — a real normalized relational schema plus Django's own ready-made admin interface for fast manual entry
- **Database** — PostgreSQL, created with explicit UTF8 encoding
- **Project** — a real Django project (`catalogue_site`) with a `catalogue` app, connected via `settings.py`'s `DATABASES` dict
- **Assumed groundwork** — Django models/migrations/ORM and PostgreSQL fundamentals, both covered in their own dedicated courses on this site
- **Next chapter:** Data Modeling — a real normalized schema for items, item types, and a tags many-to-many

CHAPTER 2: DATA MODELING

Personal Catalogue: Django & PostgreSQL

Chapter 2 · Data Modeling

Chapter 1 got a Django project genuinely talking to a live PostgreSQL database. This chapter designs the real models that database will actually store — a schema that has to hold four genuinely different kinds of item (books, CDs, DVDs, Blu-rays), let books carry an arbitrary number of tags, and stay fast and simple to query given this project's own real deadline.

The Shape of the Problem

A book has an author. A CD has an artist. A DVD or Blu-ray has a director. All four have a title, a format, and a year. There are two honest ways to model that with Django:

- **One model per item type** — a real, fully normalized `Book` model with an `author` field, a real `MusicCD` model with an `artist` field, and so on. Every field is exactly the right shape for its type, with no unused columns anywhere.
- **One shared `Item` model** — a single model covering every item type, with a generic `creator` field standing in for author/artist/director depending on the row's own type.

This course uses the second approach, and it's worth being honest about why: the app's own core real question is "*do I already own this?*" — a single search across every item type at once. With four separate models, answering that means a real query against each one and combining the results in Python. With one shared model, it's a single Django ORM `filter()` call. Given the real time pressure this project starts with, the shared-model design is the one that lets the search feature (Chapter 7) stay genuinely simple.

> **THIS IS A REAL, DELIBERATE TRADE-OFF**
>
> A fully normalized per-type schema is arguably "more correct" in a formal database-design sense — every field would genuinely apply to every row. The shared-model design accepts a few fields that are meaningless for some rows (a DVD's own `creator` would always be blank) in exchange for a dramatically simpler search query. That's the right call here specifically because search-across-everything is this app's own central feature — it wouldn't necessarily be the right call for every project.

Model 1: ItemType

Rather than hard-coding "book", "cd", "dvd", "bluray" with Django's own `TextChoices` pattern, this course uses a small, real database-backed lookup model instead:

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

class ItemType(models.Model):
    name = models.CharField(max_length=20, unique=True)

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

A `TextChoices` class would work for a fixed, never-changing list — but this collection is never really fixed. Adding "vinyl" or "video game" later with a lookup model is one new row created through the admin; adding it to a hard-coded choices list means editing the model's own source code and shipping a new migration just to add a value. The lookup model costs one extra join later, in exchange for that flexibility staying cheap and entirely data-driven.

Model 2: Item

The main model. Every item in the catalogue — regardless of type — is one row here:

```
class Item(models.Model):
    item_type = models.ForeignKey(ItemType, on_delete=models.PROTECT)
    title = models.CharField(max_length=255)
    creator = models.CharField(max_length=255, blank=True)
    format_detail = models.CharField(max_length=100, blank=True)
    release_year = models.PositiveSmallIntegerField(null=True, blank=True)
    notes = models.TextField(blank=True)
    created_at = models.DateTimeField(auto_now_add=True)

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

A few of these fields are deliberately generic, standing in for something different depending on the item's own type:

FIELD	BOOK	CD	DVD / BLU-RAY
`creator`	Author	Artist	Director
`format_detail`	Hardcover / paperback	Single / album	Region code
`release_year`	Publication year	Release year	Release year

`release_year` is deliberately nullable — the exact publication year isn't always known or worth tracking down on day one, and a real project under time pressure shouldn't make a field mandatory just because it would be nice to have. `created_at`, on the other hand, is genuinely useful and free: Django's own `auto_now_add` fills it in automatically at insert time, giving a real "recently added" ordering with zero extra application code.

`on_delete=models.PROTECT` on `item_type` is a deliberate choice, not the Django default: it stops an `ItemType` from ever being deleted while items of that type still exist, raising a real error instead of silently cascading the deletion (or worse, silently leaving orphaned rows) — a genuinely safer default for a lookup table whose values items actively depend on.

Tags & the Many-to-Many

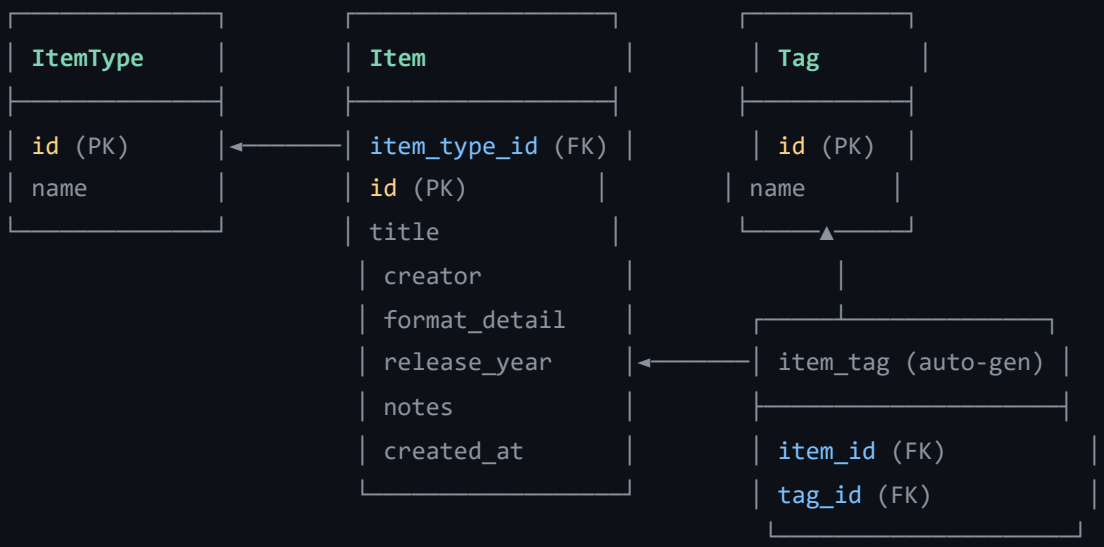
Only books get tags in this project's own spec, but the tags themselves — Python, Web Development, Programming — are reusable across many books, and any one book can carry several. That's a genuine many-to-many relationship:

```
class Tag(models.Model):
    name = models.CharField(max_length=50, unique=True)

    def __str__(self):
        return self.name

class Item(models.Model):
    # ... fields from above ...
    tags = models.ManyToManyField(Tag, blank=True, related_name='items')
```

DJANGO GENERATES THE JOIN TABLE FOR YOU

In the PHP + MySQL variant of this project (`catalog-php1`), the join table connecting items and tags — `item_tags`, with its own composite primary key preventing duplicate assignments — had to be written by hand as a real `CREATE TABLE` statement. Here, `models.ManyToManyField` generates that exact join table automatically the moment a migration runs, including its own uniqueness constraint on the pairing. This is one of the genuinely real conveniences of choosing an ORM-based framework over raw SQL — at the real cost that customizing that generated join table later (adding a "date tagged" column, say) needs Django's own `through=` model syntax rather than a simple column addition, since the table is no longer one this project's own code defines directly.

The Full Schema, in One Picture



Running the Migration

Django's own migration system turns these three model classes into real PostgreSQL tables:

```
python manage.py makemigrations catalogue
python manage.py migrate
```

`makemigrations` reads the model classes and writes a real migration file describing the tables, columns, and constraints needed; `migrate` then applies that file as real SQL against the live

PostgreSQL database. Seed the four real item types this course needs before moving on, either through the Django shell or — once Chapter 3 sets it up — through the admin interface directly:

```
python manage.py shell
>>> from catalogue.models import ItemType
>>> for name in ['book', 'cd', 'dvd', 'bluray']:
...     ItemType.objects.get_or_create(name=name)
```

GET_OR_CREATE AVOIDS A REAL DUPLICATE-SEEDING BUG

Running a plain `ItemType.objects.create(name=name)` loop a second time — say, by accidentally re-running the seed step — would insert four duplicate rows, since nothing stops it. `get_or_create()` checks for an existing row with the given field first, only inserting when one genuinely doesn't exist yet, making the seed step safe to run more than once by accident without corrupting the lookup table.

Hands-On Exercises

EXERCISE 1

Write all three model classes from this chapter in `catalogue/models.py`, run `makemigrations` and `migrate`, and confirm the real PostgreSQL tables were created using `psql`'s own `\dt` command.

View solution

EXERCISE 2

Explain, in your own words, why a shared `Item` model with a generic `creator` field was chosen over four separate per-type models, and name the one real feature that decision makes simpler.

View solution

EXERCISE 3

Using the Django shell, create one book `Item`, create one `Tag`, attach the tag to the book via `item.tags.add(tag)`, then confirm the relationship by printing `item.tags.all()`.

View solution

CHAPTER 2 QUICK REFERENCE

- **Item** — one shared model for every item type, with a generic `creator` field standing in for author/artist/director
- **ItemType** — a lookup model (not `TextChoices`) so new item types can be added as data, with no code change or migration needed
- **Tag** + **ManyToManyField** — a real many-to-many relationship, with Django auto-generating the join table and its own uniqueness constraint
- **Design trade-off** — a slightly less "pure" shared-model schema, chosen specifically because it keeps the app's own core search feature a single ORM query
- **Safety note** — `on_delete=PROTECT` on `item_type`, and `get_or_create()` for safe, re-runnable seeding
- **Next chapter:** the Django admin as a fast, ready-made manual entry tool for this real schema

CHAPTER 3: DJANGO ADMIN AS A FAST, READY-MADE MANUAL ENTRY TOOL

Personal Catalogue: Django & PostgreSQL

Chapter 3 · Django Admin as a Fast, Ready-Made Manual Entry Tool

Chapter 2 built the real schema — `Item`, `ItemType`, and `Tag` — and ran it against a live PostgreSQL database. This chapter gets a genuinely working data-entry interface on top of that schema without writing a single HTML form by hand, using the one real feature that makes this variant of the project move fastest: Django's own built-in admin.

Why the Admin Is the Right Fast-Entry Tool Here

This project's own real deadline means a working way to actually add items matters more, in the short term, than a polished public-facing add-item page. Django generates a complete, permission-gated CRUD interface directly from a model class — list views, detail/edit forms, delete confirmation, and real field-type-aware widgets (a dropdown for the `item_type` foreign key, a proper multi-select for the `tags` many-to-many) — for the cost of a handful of lines in `admin.py`. This isn't a shortcut around building a real app; it's genuinely the fastest correct way to get a real, working manual-entry tool live today, with the custom public-facing form (Chapter 5) built afterward once the core schema is proven out.

Creating a Superuser

The admin requires a real, privileged account to log into:

```
python manage.py createsuperuser
```

Follow the real prompts for a username, email, and password. This account is separate from anything this project's own `Item` / `ItemType` / `Tag` models define — it's part of Django's own built-in `auth` app, already migrated in Chapter 1's own first `migrate` run.

Registering the Models

A bare-minimum registration already produces a working, if plain, admin interface:

```
# catalogue/admin.py
from django.contrib import admin
from .models import Item, ItemType, Tag

admin.site.register(Item)
admin.site.register(ItemType)
admin.site.register(Tag)
```

Run the development server and visit `/admin/` to confirm all three models genuinely show up, each with a working add/list/edit/delete interface already functioning against the real PostgreSQL database.

A Real, Customized ModelAdmin

The bare registration works, but a few genuinely small customizations make the admin dramatically more useful for this project's own real day-to-day task — quickly checking whether an item already exists before adding a new one:

```
@admin.register(Item)
class ItemAdmin(admin.ModelAdmin):
    list_display = ('title', 'item_type', 'creator', 'release_year', 'created_at')
    list_filter = ('item_type', 'tags')
    search_fields = ('title', 'creator')
    filter_horizontal = ('tags',)
```

- `list_display` — shows title, type, creator, year, and date added right on the list page, instead of the model's own bare `__str__` only.
- `list_filter` — adds a real sidebar to filter the list by item type or by tag with a single click.
- `search_fields` — adds a real search box across title and creator, directly supporting this project's own core "do I already own this?" question.
- `filter_horizontal` — replaces the tags field's own default cramped multi-select box with a real, genuinely usable two-column widget for adding and removing tags.

FILTER_HORIZONTAL SPECIFICALLY FOR MANY-TO-MANY FIELDS

Without `filter_horizontal`, Django's own default widget for a `ManyToManyField` is a plain multi-select box — usable, but genuinely awkward once the tag list grows past a handful of entries. `filter_horizontal` is specifically designed for exactly this case: two side-by-side boxes (available tags, selected tags) with a real search filter and add/remove buttons, which is why it's called out here rather than left at Django's own default.

THE REAL PAYOFF OF CHAPTER 1'S OWN SCAFFOLDING

Chapter 1 noted that Django's own project scaffolding was real, genuine setup work the PHP variant of this project skipped entirely. This chapter is where that investment pays off directly: roughly fifteen lines of `admin.py` code produce a complete, working, searchable, filterable CRUD interface — something the PHP variant has to build by hand, page by page, across its own Chapters 3 and 4.

THE ADMIN ISN'T THE PUBLIC-FACING APP

The admin is a real, genuinely useful internal tool — but it's not meant to be the interface this project ships as its own public-facing catalogue browser, and it should never be exposed carelessly once deployed. Chapter 5 builds a dedicated, public-facing add-item form, and Chapter 9's own deployment coverage returns to locking the admin down properly (a non-default URL, real strong credentials) before this project goes anywhere near the internet.

Trying It Out

With `ItemType` rows already seeded in Chapter 2, log into `/admin/` and add a real item through the UI: pick a type from the dropdown, fill in title/creator/year, and — for a book — attach one or more tags using the new `filter_horizontal` widget. Confirm the item appears correctly on the list page, with its tags visible via the `list_filter` sidebar.

Hands-On Exercises

EXERCISE 1

Create a superuser, register all three models with the customized `ItemAdmin` from this chapter, and add three real items through the admin — one book (with at least two tags), one CD, and one DVD.

View solution

EXERCISE 2

Explain what specifically `filter_horizontal` changes about how tags are edited in the admin, and why that change matters more for this project's `tags` field than it would for, say, the single-valued `item_type` field.

View solution

EXERCISE 3

Explain why relying on the Django admin as this project's own primary way of adding items is a reasonable choice given its real deadline, while also explaining why it can't be the app's own final, shipped solution.

View solution

CHAPTER 3 QUICK REFERENCE

- **createsuperuser** — creates the privileged account needed to log into `/admin/`, part of Django's own built-in `auth` app
- **admin.site.register()** — the minimum needed for a working, if plain, CRUD interface per model
- **ModelAdmin customization** — `list_display`, `list_filter`, `search_fields`, and `filter_horizontal` turn that plain interface into a genuinely fast day-to-day tool
- **filter_horizontal** — specifically improves the default many-to-many widget, replacing a cramped multi-select with a real two-column filterable UI
- **Real limit** — the admin is an internal tool, not the public-facing app, and needs locking down before deployment (Chapter 9)
- **Next chapter:** Building the Public-Facing Views — the real list and detail pages visitors (i.e., you) will actually browse

CHAPTER 4: BUILDING THE PUBLIC-FACING VIEWS: LIST & DETAIL

Personal Catalogue: Django & PostgreSQL

Chapter 4 · Building the Public-Facing Views: List & Detail

Chapter 3 gave this project a real, working way to add items — but only through the admin, a login-gated internal tool. This chapter builds the pages the catalogue is actually for: a real list of every item, and a detail page for each one, both genuinely readable without ever touching `/admin/`.

Two Real Views: List & Detail

Django's own built-in generic class-based views handle both of these directly. Rather than writing a function that manually fetches a queryset, passes it to a template, and returns an `HttpResponse`, `ListView` and `DetailView` do that real, repetitive plumbing automatically from a small amount of configuration.

The List View

```
# catalogue/views.py
from django.views.generic import ListView, DetailView
from .models import Item

class ItemListView(ListView):
    model = Item
    template_name = 'catalogue/item_list.html'
    context_object_name = 'items'
    paginate_by = 25
    queryset = Item.objects.select_related('item_type').prefetch_related('tags')
```

`select_related('item_type')` and `prefetch_related('tags')` are both deliberate, not decoration — the reasoning for each is covered directly below, once the template that actually needs them exists.

The Detail View

```
class ItemDetailView(DetailView):
    model = Item
    template_name = 'catalogue/item_detail.html'
    context_object_name = 'item'
```

`DetailView` looks up a single `Item` by its primary key (taken from the URL) and returns a real, genuine HTTP 404 automatically if no item with that ID exists — no manual `get_object_or_404` call needed, since the generic view already wraps that lookup internally.

URL Routing

```
# catalogue/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.ItemListView.as_view(), name='item_list'),
    path('item/<int:pk>/', views.ItemDetailView.as_view(), name='item_detail'),
]
```

Include this app-level `urls.py` from the project's own root `catalogue_site/urls.py`:

```
# catalogue_site/urls.py
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('catalogue.urls')),
]
```

Minimal Templates

A small base template, extended by both real pages:

```
<!-- catalogue/templates/catalogue/base.html -->
<!DOCTYPE html>
<html>
<head><title>{% block title %}My Catalogue{% endblock %}</title></head>
<body>
  <h1><a href="{% url 'item_list' %}">My Catalogue</a></h1>
  {% block content %}{% endblock %}
</body>
</html>
```

```
<!-- catalogue/templates/catalogue/item_list.html -->
{% extends 'catalogue/base.html' %}
{% block content %}
<ul>
  {% for item in items %}
    <li>
      <a href="{% url 'item_detail' item.pk %}">{{ item.title }}</a>
      ({{ item.item_type }}{% if item.creator %}, {{ item.creator }}{% endif %})
      {% for tag in item.tags.all %}<span>#{{ tag.name }}</span>{% endfor %}
    </li>
  {% endfor %}
</ul>
{% endblock %}
```

REAL TEMPLATE INHERITANCE

`{% extends %}` and `{% block %}` are Django's own real template inheritance mechanism — `item_list.html` doesn't repeat the `<html>` / `<head>` /navigation markup at all, it only fills in the `content` block `base.html` already declared. Every later template in this course (detail page, add-item form, Chapter 8's own styled versions) extends the same `base.html`.

A REAL N+1 QUERY BUG, CAUGHT BEFORE IT SHIPPED

The `{% for tag in item.tags.all %}` loop inside `item_list.html` runs once per item on the page — meaning without `prefetch_related('tags')` in the view's own queryset, displaying 25 items would trigger 25 separate database queries just to fetch tags, on top of the one query that fetched the items themselves: a real, classic N+1 query problem. `prefetch_related('tags')` fixes this by

issuing exactly one additional query up front, fetching every tag for every item on the page at once, which Django then serves from memory for each `item.tags.all` call inside the template loop. `select_related('item_type')` solves the same underlying problem one level earlier, for the single-valued `item.item_type` lookup the template also uses — a real `JOIN` folded into the original query instead of a second query per item.

SELECT_RELATED VS. PREFETCH_RELATED — NOT INTERCHANGEABLE

`select_related` only works for single-valued relationships (a `ForeignKey` like `item_type`), since it works by joining the related table directly into the original SQL query. `prefetch_related` is required for multi-valued relationships (a `ManyToManyField` like `tags`), since a single SQL join can't cleanly represent "many tags per item" in one flat result set — it runs as a real, separate second query instead. Using the wrong one, or skipping both, either raises an error or silently reintroduces the N+1 problem this chapter just fixed.

`item_detail.html` follows the identical pattern, extending the same `base.html` and rendering the single `item` the view resolved, including its full tag list — with no N+1 risk here at all, since only one item's own tags are ever fetched on this page.

Trying It Out

Visit the site's own root URL and confirm every item added in Chapter 3 — through the admin — now shows up on this real, public-facing list page too, since both interfaces read from the exact same `Item` table. Click through to a detail page, and confirm visiting a nonexistent item ID (e.g. `/item/9999/`) returns a real 404 rather than an error page or a blank screen.

Hands-On Exercises

EXERCISE 1

Build both views, both templates, and the URL routing from this chapter, then confirm the list page correctly shows every item added through the admin in Chapter 3, tags included.

View solution

EXERCISE 2

Explain, step by step, why removing `prefetch_related('tags')` from the list view's own queryset would produce exactly 26 real database queries to render a page of 25 items with tags — not 25, and not 1.

View solution

EXERCISE 3

Explain why `select_related` can't be used for the `tags` relationship, even though it's used successfully for `item_type` on the exact same queryset.

View solution

CHAPTER 4 QUICK REFERENCE

- **ListView / DetailView** — Django's own generic class-based views, handling querying and 404s automatically from minimal configuration
- **URL routing** — an app-level `urls.py` included from the project's own root `urls.py`
- **Template inheritance** — `{% extends %}` / `{% block %}` share a common `base.html` across every page
- **Real N+1 bug found and fixed** — looping over `item.tags.all` per item without `prefetch_related('tags')` would trigger one extra query per item
- **select_related vs. prefetch_related** — single-valued relationships (`ForeignKey`) use `select_related`; multi-valued relationships (`ManyToManyField`) need `prefetch_related` instead
- **Next chapter:** a real public-facing add-item form, handling several genuinely different item shapes outside the admin

CHAPTER 5: THE ADD-ITEM FORM OUTSIDE THE ADMIN

Personal Catalogue: Django & PostgreSQL

Chapter 5 · The Add-Item Form Outside the Admin: Django Forms & Type-Specific Fields

Chapter 4 gave the catalogue real public list and detail pages, but adding an item still means logging into `/admin/`. This chapter builds a real, public-facing add-item form — and makes that one shared form feel genuinely type-aware, without forking it into four separate forms.

The Core Idea: Data-Driven Labels, Not a Fork in the Form

A tempting but wrong approach here is to build a genuinely separate form for each item type — a "book form," a "CD form," a "DVD form" — each with its own field names. That would mean four times the template markup to maintain, and four separate view branches to validate and save. Since the underlying `Item` model is deliberately shared (Chapter 2), the form stays shared too: one set of fields, with their **labels** — not their underlying model fields — changing based on whatever item type is currently selected.

> **RECAP: ONE SHARED SCHEMA, SEVERAL REAL MEANINGS**
>
> Chapter 2 already established this table — worth having in view again before building the UI on top of it:

FIELD	BOOK	CD	DVD / BLU-RAY
`creator`	Author	Artist	Director
`format_detail`	Hardcover / paperback	Single / album	Region code

A Real Django ModelForm

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

```
class ItemForm(forms.ModelForm):
    class Meta:
        model = Item
        fields = ['item_type', 'title', 'creator', 'format_detail', 'release_year', 'ta
        widgets = {
            'tags': forms.CheckboxSelectMultiple(),
        }
```

A `ModelForm` derives its own fields, widgets, and — critically — its own validation rules directly from the `Item` model itself, rather than requiring them to be redeclared by hand. `release_year`'s own real `PositiveSmallIntegerField` constraint from Chapter 2 is enforced

The CreateView

```
# catalogue/views.py (added to the existing file)
from django.views.generic.edit import CreateView
from django.urls import reverse_lazy
from .models import Item, ItemType
from .forms import ItemForm

class ItemCreateView(CreateView):
    model = Item
    form_class = ItemForm
    template_name = 'catalogue/item_form.html'
    success_url = reverse_lazy('item_list')

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['creator_labels'] = {t.name: LABELS.get(t.name, {}).get('creator', 'Cre
        context['format_labels'] = {t.name: LABELS.get(t.name, {}).get('format', 'Forma
        return context

LABELS = {
    'book':   {'creator': 'Author',              'format': 'Binding (hardcover / paper
    'cd':     {'creator': 'Artist',              'format': 'Release type (single / alb
    'dvd':    {'creator': 'Director',            'format': 'Region code'},
```

```
    'bluray': {'creator': 'Director',            'format': 'Region code'},
}
```

Add the URL to `catalogue/urls.py`:

```
path('add/', views.ItemCreateView.as_view(), name='item_add'),
```

Building the Label Map with json_script

Rather than hand-typing a JSON string directly into the template — a real, documented cross-site-scripting risk if any of that data ever contains untrusted content — Django provides a dedicated template filter built specifically for this exact job:

`catalogue/templates/catalogue/item_form.html`

```
{% extends 'catalogue/base.html' %}
{% block content %}
{{ creator_labels|json_script:"creator-labels-data" }}
{{ format_labels|json_script:"format-labels-data" }}

<form method="post">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Add Item</button>
</form>
{% endblock %}
```

WHAT JSON_SCRIPT ACTUALLY DOES

`{{ creator_labels|json_script:"creator-labels-data" }}` renders a real `<script id="creator-labels-data" type="application/json">` tag, with its own JSON content correctly HTML-escaped by Django — safe to include directly in a template even if the underlying data were ever user-controlled, unlike manually interpolating a JSON string into a page. It's the Django-native equivalent of what PHP Intermediate's own `json_encode()` technique accomplishes in the PHP variant of this project (`catalog-php1`) — the same underlying goal, real server-rendered data handed safely to client-side

JavaScript, achieved through Django's own dedicated tooling rather than a manually-built inline script.

Wiring the Dropdown to the Labels

A small vanilla JS block reads the two JSON script tags and updates the form's own labels whenever the item type dropdown changes:

```
<script>
const creatorLabels = JSON.parse(document.getElementById('creator-labels-data').textCon
const formatLabels  = JSON.parse(document.getElementById('format-labels-data').textCont

const select       = document.getElementById('id_item_type');
const creatorLabel = document.querySelector('label[for="id_creator"]');
const formatLabel  = document.querySelector('label[for="id_format_detail"]');

function updateLabels() {
    const typeName = select.options[select.selectedIndex].text.toLowerCase();
    creatorLabel.textContent = (creatorLabels[typeName] ?? 'Creator') + ':';
    formatLabel.textContent  = (formatLabels[typeName] ?? 'Format detail') + ':';
}

select.addEventListener('change', updateLabels);
updateLabels();
</script>
```

`id_item_type` and `id_creator` are Django's own default, automatically-generated field IDs — `{{ form.as_p }}` already produces them without any extra configuration, which is why the JS above can rely on them directly.

CLIENT-SIDE RELABELING IS COSMETIC ONLY

This JavaScript changes what the form *looks like* — it does not, and cannot, change what data is actually accepted when the form is submitted. A user with JavaScript disabled, or one deliberately tampering with the submitted request, still hits the exact same `ItemForm` validation on the server, defined once in Chapter 2's own model constraints and re-used automatically by the `ModelForm`. The label-swapping script is a real usability improvement, not a security or validation mechanism —

the server-side form is what actually enforces the rules, exactly as it does in the PHP variant of this project.

Trying It Out

Visit `/add/`, select each item type in turn, and confirm the "Creator" and "Format detail" labels genuinely change to Author/Artist/Director and the matching format hint. Submit a real book with a title, author, and at least one tag, then confirm it appears correctly on the list page from Chapter 4.

Hands-On Exercises

EXERCISE 1

Build the form, view, template, and label-swapping script from this chapter, then add one real item of each of the four types through this new public form.

View solution

EXERCISE 2

Explain what `json_script` actually produces in the rendered HTML, and why it's genuinely safer than building the equivalent `<script>` tag by hand-interpolating a JSON string into the template.

View solution

EXERCISE 3

Explain why the label-swapping JavaScript from this chapter is not, and cannot be, a substitute for the server-side validation the `ModelForm` already provides.

View solution

CHAPTER 5 QUICK REFERENCE

- **ItemForm** — a real `ModelForm` deriving its fields and validation directly from the `Item` model
- **ItemCreateView** — Django's own generic `CreateView`, redirecting to the list page via `success_url` on success
- **One shared form, several labels** — labels change per item type via JavaScript; the underlying fields and their names never change
- **json_script** — Django's own dedicated, safely-escaped way to hand server-rendered data to client-side JavaScript
- **Real security note** — client-side relabeling is cosmetic only; the `ModelForm`'s own server-side validation is what actually enforces the rules
- **Next chapter:** Tags — filtering the catalogue by tag

CHAPTER 6: TAGS: FILTERING THE CATALOGUE BY TAG

Personal Catalogue: Django & PostgreSQL

Chapter 6 · Tags: Filtering the Catalogue by Tag

Chapter 5 let items be tagged. This chapter makes those tags actually useful for the app's own real core question — not just "do I already own this?" but the narrower, equally real "what Python books do I have?" — by building a genuine filter-by-tag feature into the list page.

Filtering via a Query Parameter

Rather than a separate page per tag, this course uses one real, shareable URL pattern — `/?tag=python` — read directly off the query string:

```
# catalogue/views.py (ItemListView, updated)
class ItemListView(ListView):
    model = Item
    template_name = 'catalogue/item_list.html'
    context_object_name = 'items'
    paginate_by = 25

    def get_queryset(self):
        queryset = Item.objects.select_related('item_type').prefetch_related('tags')
        tag_name = self.request.GET.get('tag')
        if tag_name:
            queryset = queryset.filter(tags__name=tag_name)
        return queryset

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['active_tag'] = self.request.GET.get('tag', '')
        return context
```

`queryset.filter(tags__name=tag_name)` follows the `ManyToManyField` relationship directly in the ORM — no manual join, no separate query, since Django's own field-lookup syntax (`tags__name`) turns straight into a real SQL join against the auto-generated join table from Chapter 2.

Making Tags Clickable

A small template change turns each tag shown on the list and detail pages into a real link back into this same filter:

```
<!-- inside the item loop in item_list.html -->
{% for tag in item.tags.all %}
  <a href="?tag={{ tag.name }}">#{{ tag.name }}</a>
{% endfor %}
```

And a small banner showing the active filter, with a real way back out of it:

```
{% if active_tag %}
  <p>Filtered by: #{{ active_tag }}
     <a href="{% url 'item_list' %}">(clear)</a></p>
{% endif %}
```

A REAL GOTCHA: FILTERING BREAKS UNDER PAGINATION UNLESS HANDLED

Django's own built-in pagination (from `paginate_by = 25`, Chapter 4) generates its "next page" and "previous page" links from `page_obj` alone — those links only ever carry a `?page=2` query string by default, with no awareness that a `?tag=python` filter might also be active. Clicking "next page" while a tag filter is active would silently drop the filter, landing on page 2 of the *entire unfiltered* catalogue instead of page 2 of the filtered results — a real, easy-to-miss bug that only shows up once a filtered list actually spans more than one page.

The fix is to build pagination links that explicitly carry the active tag forward:

```
{% if page_obj.has_next %}
  <a href="?page={{ page_obj.next_page_number }}{% if active_tag %}&tag={{ active_tag }
{% endif %}
```

EVERY LINK THAT TOUCHES PAGINATION NEEDS THE SAME TREATMENT

"Previous page" needs the identical `&tag={{ active_tag }}` suffix, and so would any other filter this project ever adds later (by item type, say). The real, general rule: any link that changes the page

number, on a page whose own results are already filtered, has to explicitly re-include every active filter in its own query string — Django never does this automatically, since the pagination links and the filtering logic are two genuinely separate pieces of code that don't know about each other unless the template explicitly connects them.

A Small Tag Index Page

A real, genuinely useful companion to filtering: a page listing every tag alongside how many items actually carry it, using Django's own real aggregation support:

```
# catalogue/views.py
from django.db.models import Count
from .models import Tag

class TagIndexView(ListView):
    model = Tag
    template_name = 'catalogue/tag_index.html'
    context_object_name = 'tags'
    queryset = Tag.objects.annotate(item_count=Count('items')).order_by('-item_count')
```

`Count('items')` uses the `related_name='items'` set on the `Item.tags` field back in Chapter 2, letting this query count, for each real `Tag` row, how many `Item` rows actually reference it — a single aggregated query rather than one count query per tag.

Hands-On Exercises

EXERCISE 1

Add the tag-filtering logic to `ItemListView`, make tags clickable in the list template, and confirm visiting `/?tag=python` shows only items actually tagged Python.

View solution

EXERCISE 2

Add enough real items tagged "Python" to force pagination onto a second page, then reproduce the real bug this chapter describes by clicking "next page" before applying the fix — confirm the filter is genuinely lost — then apply the fix and confirm it's preserved.

View solution

EXERCISE 3

Build the `TagIndexView` and its template, and explain in your own words why `Count('items')` is a meaningfully faster approach than looping over every tag and calling `tag.items.count()` individually.

View solution

CHAPTER 6 QUICK REFERENCE

- **Filtering** — `get_queryset()` reads a real `?tag=` query parameter and filters via `tags__name`
- **Clickable tags** — each tag rendered as a link back into the same filter, plus a "clear filter" link
- **Real bug found** — pagination links silently drop the active tag filter unless the query string is explicitly carried forward on every page link
- **TagIndexView** — a real aggregated `Count('items')` query listing every tag with its true item total, in one query
- **Next chapter:** Search — real Django ORM querying across the whole catalogue

CHAPTER 7: SEARCH: REAL DJANGO ORM QUERYING ACROSS THE CATALOGUE

Personal Catalogue: Django & PostgreSQL

Chapter 7 · Search: Real Django ORM Querying Across the Catalogue

Chapter 6 answered "what Python books do I have?" through tags. This chapter answers the app's own broader real question — "do I already own this?" — with a genuine free-text search box that works across every item's title and creator at once, then goes one step further using a real feature this project's own PostgreSQL choice specifically unlocks.

A Real Search Box

A plain GET-based form, matching the same query-string approach the tag filter already uses:

```
<!-- above the item list in item_list.html -->
<form method="get">
  <input type="text" name="q" value="{{ query }}" placeholder="Search title or creator.
  <button type="submit">Search</button>
</form>
```

A GET form is a deliberate choice here, not an oversight — the resulting URL (`/?q=fastapi`) is genuinely shareable and bookmarkable, exactly like the `?tag=` URLs from Chapter 6.

Basic Search with icontains + Q Objects

```
# catalogue/views.py
from django.db.models import Q

class ItemListView(ListView):
    # ... existing config from Chapter 6 ...

    def get_queryset(self):
        queryset = Item.objects.select_related('item_type').prefetch_related('tags')
```

```
        tag_name = self.request.GET.get('tag')
        if tag_name:
            queryset = queryset.filter(tags__name=tag_name)

        query = self.request.GET.get('q')
        if query:
            queryset = queryset.filter(Q(title__icontains=query) | Q(creator__icontains

        return queryset

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['active_tag'] = self.request.GET.get('tag', '')
        context['query'] = self.request.GET.get('q', '')
        return context
```

`icontains` is Django's own case-insensitive "contains" lookup — `title__icontains=query` matches regardless of capitalization, which is the correct default for a personal catalogue search where a user shouldn't have to remember exact casing. `Q(...) | Q(...)` combines the two conditions with a real SQL `OR`, matching an item whose title *or* whose creator contains the search term.

> **FILTERS CHAIN, THEY DON'T RESET**
>
> sequence, visiting `/?tag=python&q=fastapi` genuinely combines both — items must match the tag **and** the search term. This falls out naturally from how the ORM's own `.filter()` chaining works, with no extra code needed to make the two features cooperate.

A Real PostgreSQL-Specific Upgrade: Full-Text Search

`icontains` is a real, working solution — and it's exactly what the PHP + MySQL variant of this project (`catalog-php1`) uses, via a plain SQL `LIKE` clause. Because this variant specifically chose PostgreSQL, a genuinely more capable option exists: Django's own `django.contrib.postgres.search` module, wrapping PostgreSQL's real, built-in full-text search engine.

```python
# settings.py — add to INSTALLED_APPS
INSTALLED_APPS = [
    # ...
    'django.contrib.postgres',
]

# catalogue/views.py
from django.contrib.postgres.search import SearchVector, SearchQuery, SearchRank

def get_queryset(self):
    queryset = Item.objects.select_related('item_type').prefetch_related('tags')
    # ... tag filter unchanged ...

    query = self.request.GET.get('q')
    if query:
        search_query = SearchQuery(query)
        queryset = queryset.annotate(
            search=SearchVector('title', 'creator'),
            rank=SearchRank(SearchVector('title', 'creator'), search_query),
        ).filter(search=search_query).order_by('-rank')

    return queryset
```

A REAL, MEANINGFULLY DIFFERENT SEARCH ENGINE

PostgreSQL's own full-text search does genuinely more than `icontains` / `LIKE` ever can: it applies real language-aware stemming, so searching "programming" also matches items whose title or creator contains "programmer" or "program" — a real linguistic match, not just a substring match. `SearchRank` then orders results by real relevance rather than by whatever arbitrary order the database happens to return rows in, so a title that's a strong match appears above one where the search term only appears incidentally. This is a genuine, direct payoff of choosing PostgreSQL specifically for this variant — the PHP + MySQL sibling has no equivalent built-in feature to reach for.

FULL-TEXT SEARCH WANTS A REAL GIN INDEX AT SCALE

`SearchVector` works correctly with no extra setup, but it recomputes the searchable text from `title` and `creator` on every single query — genuinely fine for a personal catalogue of a few hundred or even a few thousand items, but a real, documented performance cost that grows with the table. A properly indexed setup adds a real PostgreSQL `GIN` index over a precomputed `SearchVectorField`, letting full-text search stay fast even at much larger scale. That's real, additional

setup this course deliberately doesn't build — a personal catalogue's own realistic size doesn't need it — but it's worth knowing the option exists for a much larger dataset.

Trying It Out

Search for a partial, differently-cased word from an item's own title or creator and confirm it's found. Then try a word related to, but not literally contained in, an existing title — like "programmer" when the actual stored title contains "programming" — and confirm the full-text search version still finds it, where the earlier `icontains` version would not have.

Hands-On Exercises

EXERCISE 1

Build the `icontains` / `Q`-object version of search from this chapter, then confirm `/?tag=python&q=fastapi` genuinely returns only items matching both conditions at once.

View solution

EXERCISE 2

Switch to the PostgreSQL full-text search version, add an item titled "Learning Python Programming", and confirm a search for "programmer" (not "programming") still finds it — then explain why `icontains` alone would not have.

View solution

EXERCISE 3

Explain why this project deliberately doesn't build a GIN index for the full-text search field, and under what real condition that decision should be revisited.

View solution

CHAPTER 7 QUICK REFERENCE

- **Basic search** — `Q(title__icontains=query) | Q(creator__icontains=query)`, a case-insensitive OR across two fields
- **Filters chain** — search and the Chapter 6 tag filter combine automatically via sequential `.filter()` calls on the same queryset
- **PostgreSQL full-text search** — `SearchVector` / `SearchQuery` / `SearchRank` from `django.contrib.postgres.search`, giving real stemming and relevance ranking `icontains` can't provide
- **Real tradeoff** — full-text search recomputes the search vector per query without a GIN index; fine at personal-catalogue scale, worth revisiting at real scale
- **Next chapter:** Styling & Templates

CHAPTER 8: STYLING & TEMPLATES

Personal Catalogue: Django & PostgreSQL

Chapter 8 · Styling & Templates

The functional core is done: real schema, a real admin, real public list/detail pages, a real add-item form, tag filtering, and search. This chapter deliberately comes after all of that, not before — matching this project's own real deadline-driven priority order from Chapter 1, function before polish.

Django's Static Files System

CSS, images, and any other static asset live in a dedicated app-level directory, wired up through a small, one-time settings change:

```
# catalogue_site/settings.py
STATIC_URL = 'static/'
STATICFILES_DIRS = [BASE_DIR / 'catalogue' / 'static']
```

Create `catalogue/static/catalogue/style.css` — the extra `catalogue/` nesting inside `static/` is a real, deliberate Django convention (namespacing static files by app), avoiding a real file-name collision if a second app in this project ever also defines a `style.css`.

A Real Base Template With Actual Styling

```
<!-- catalogue/templates/catalogue/base.html -->
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}My Catalogue{% endblock %}</title>
  <link rel="stylesheet" href="{% static 'catalogue/style.css' %}">
</head>
<body>
```

```
    <nav>
      <a href="{% url 'item_list' %}">My Catalogue</a>
      <a href="{% url 'item_add' %}">Add Item</a>
      <a href="{% url 'tag_index' %}">Browse Tags</a>
    </nav>
    <main>{% block content %}{% endblock %}</main>
  </body>
  </html>
```

`{% load static %}` must appear before any `{% static %}` tag is used in the template — a real, easy-to-forget requirement, since forgetting it produces a confusing `TemplateSyntaxError` pointing at the `{% static %}` line itself rather than at the missing `{% load %}` that actually caused it.

A Reusable Item Card Partial

Both the list page and the tag-filtered list page render items the same way — a real, genuine case for Django's own template composition, rather than maintaining the same markup twice:

```
<!-- catalogue/templates/catalogue/_item_card.html -->
<article class="item-card">
  <h3><a href="{% url 'item_detail' item.pk %}">{{ item.title }}</a></h3>
  <p>{{ item.item_type }}{% if item.creator %} &middot; {{ item.creator }}{% endif %}</
  {% for tag in item.tags.all %}
    <a class="tag-pill" href="?tag={{ tag.name }}">#{{ tag.name }}</a>
  {% endfor %}
</article>
```

And in `item_list.html`:

```
{% for item in items %}
  {% include 'catalogue/_item_card.html' %}
{% endfor %}
```

WHY {% INCLUDE %} INSTEAD OF REPEATING THE MARKUP

A leading underscore on `_item_card.html` is a real, common Django community convention — not a Django feature itself — marking a template as a partial, meant to be included rather than rendered directly as a page. Pulling this markup into its own file means a future styling change to how an item card looks only needs to happen in one place, rather than being kept in sync across the list page and anywhere else an item might eventually be rendered the same way.

Real Dark-Theme Styling

```
/* catalogue/static/catalogue/style.css */
body { background: #0d1117; color: #c9d1d9; font-family: system-ui, sans-serif; margin:
nav { background: #161b22; padding: 1rem 1.5rem; display: flex; gap: 1.5rem; }
nav a { color: #7fd6b4; text-decoration: none; }
main { max-width: 800px; margin: 0 auto; padding: 1.5rem; }
.item-card { background: #161b22; border: 1px solid #30363d; border-radius: 8px; paddin
.tag-pill { display: inline-block; background: #0c1f18; color: #7fd6b4; border: 1px sol
            border-radius: 999px; padding: 2px 10px; margin-right: 4px; font-size: 0.8r
```

STATICFILES_DIRS ISN'T WHERE FILES LIVE IN PRODUCTION

`STATICFILES_DIRS` is where Django's own development server looks for static files while `DEBUG = True` — genuinely convenient locally, but Django's development server is explicitly documented as unsuitable for serving static files in production. A real production deployment instead runs `python manage.py collectstatic`, which copies every static file from every app into one single folder — `STATIC_ROOT`, a separate setting not yet defined in this project — for a real web server (or a service like WhiteNoise) to serve directly. This distinction doesn't matter yet, but it matters directly once Chapter 9 covers real deployment.

Trying It Out

Reload the list page and confirm the nav bar, dark theme, and pill-styled tags all render correctly, and that clicking a tag pill still filters exactly as it did before this chapter — styling changed nothing about the underlying filtering logic from Chapter 6.

Hands-On Exercises

EXERCISE 1

Build `style.css`, the updated `base.html`, and the `_item_card.html` partial, then update `item_list.html` to use `{% include %}` instead of its own inline item markup.

View solution

EXERCISE 2

Deliberately remove `{% load static %}` from the top of `base.html`, reload the page, and record the exact error Django produces — then explain why the error points at the `{% static %}` line rather than at the missing `{% load %}` tag itself.

View solution

EXERCISE 3

Explain, in your own words, why `STATICFILES_DIRS` being sufficient in development doesn't mean it's sufficient in production, and what `collectstatic` and `STATIC_ROOT` actually do differently.

View solution

CHAPTER 8 QUICK REFERENCE

- **Static files** — `STATICFILES_DIRS` plus `{% load static %}` and `{% static %}` wire up CSS in development
- **_item_card.html** — a reusable partial included via `{% include %}`, keeping item-card markup defined in exactly one place
- **Real gotcha** — forgetting `{% load static %}` produces an error pointing at the wrong line
- **Real production distinction** — `STATICFILES_DIRS` serves files in development only; `collectstatic` and `STATIC_ROOT` handle real production serving, covered directly in Chapter 9
- **Next chapter:** Deployment — taking this project from a local dev server to something actually live

CHAPTER 9: DEPLOYMENT

Personal Catalogue: Django & PostgreSQL

Chapter 9 · Deployment

Every chapter so far has run against a local dev server. This chapter moves the finished catalogue onto a real server — reusing the Debian/web-server groundwork this site's own *Setting Up a Web Server on Debian* and *Securing Your Web Server* courses already cover, rather than re-deriving general server setup here.

Settings Never Travel With the Code

Chapter 1's own `settings.py` has a real database password and a hardcoded `SECRET_KEY` sitting directly in it — fine for local development, but neither should ever be committed to version control the way the rest of the codebase is. A committed secret stays in a Git repository's own history forever, even once deleted in a later commit.

```
# Install a real environment-variable loader
pip install python-decouple
```

catalogue_site/settings.py (updated)

```
from decouple import config

SECRET_KEY = config('DJANGO_SECRET_KEY')
DEBUG = config('DJANGO_DEBUG', default=False, cast=bool)
ALLOWED_HOSTS = config('DJANGO_ALLOWED_HOSTS', default='').split(',')

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': config('DB_NAME'),
        'USER': config('DB_USER'),
        'PASSWORD': config('DB_PASSWORD'),
        'HOST': config('DB_HOST', default='localhost'),
        'PORT': config('DB_PORT', default='5432'),
```

```
    }
}
```

The real values live in a local `.env` file, added to `.gitignore`, with a placeholder-only `.env.example` committed in its place — the same real config/example split the PHP variant of this project (`catalog-php1`) uses for its own `config.php` / `config.example.php` pair:

```
# .env.example — committed, no real secrets
DJANGO_SECRET_KEY=REPLACE_ME
DJANGO_DEBUG=False
DJANGO_ALLOWED_HOSTS=catalogue.example.com
DB_NAME=catalogue
DB_USER=REPLACE_ME
DB_PASSWORD=REPLACE_ME
DB_HOST=localhost
DB_PORT=5432
```

DEBUG = TRUE IS A REAL INFORMATION LEAK IN PRODUCTION

With `DEBUG = True`, an unhandled Django error shows the visitor a full, detailed traceback — real file paths, real settings values, and in a worse case, a database query. `DEBUG` defaults to `False` in the settings above specifically so a missing or misconfigured environment variable fails safe, rather than silently leaving debug output enabled on a real, publicly reachable server. Setting `ALLOWED_HOSTS` correctly also matters directly here — with `DEBUG = False`, Django refuses to serve any request whose `Host` header isn't in that list, a real protection against a class of HTTP Host header attacks.

Migrating the Schema to the Production Database

Rather than re-running every migration by hand and re-seeding data manually, the local database can be exported and imported directly:

```
# On the local machine, dump structure and data together
pg_dump -U postgres catalogue > catalogue_export.sql

# Copy it to the server
scp catalogue_export.sql user@server:/tmp/
```

```
# On the server, create an empty database first...
createdb -U postgres catalogue

# ...then import the exported structure and data into it
psql -U postgres catalogue < /tmp/catalogue_export.sql
```

Alternatively, for a genuinely fresh production database with no data to migrate yet, running `python manage.py migrate` directly against the production database (once `.env` points at it) recreates the real schema from Chapter 2's own migration files without needing a dump/restore step at all.

A Dedicated, Least-Privilege Database Role

Connecting as PostgreSQL's own `postgres` superuser role — even locally, as Chapter 1's example did — is a shortcut worth dropping before this goes anywhere real:

```
CREATE ROLE catalogue_app WITH LOGIN PASSWORD 'a-real-strong-password-here';
GRANT CONNECT ON DATABASE catalogue TO catalogue_app;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO catalogue_app;
```

DELIBERATELY NOT A SUPERUSER

`catalogue_app` gets exactly what the running app actually needs — `SELECT` / `INSERT` / `UPDATE` / `DELETE` on the real tables the app queries — and nothing more. It deliberately can't `DROP` a table, alter the schema, or create new roles, so a bug or a successfully exploited vulnerability in the running app genuinely couldn't do that kind of damage, even in the worst case. `DB_USER` and `DB_PASSWORD` in the real production `.env` point at this role, never at `postgres`.

Static Files: Real Payoff of Chapter 8

Chapter 8 flagged that `STATICFILES_DIRS` only serves files in development. Production needs a real `STATIC_ROOT` plus `collectstatic`:

```
# settings.py
STATIC_ROOT = BASE_DIR / 'staticfiles'

# pip install whitenoise, then add to MIDDLEWARE (right after SecurityMiddleware)
'whitenoise.middleware.WhiteNoiseMiddleware',
```

```
python manage.py collectstatic --noinput
```

WhiteNoise lets Gunicorn itself serve static files efficiently, directly from `STATIC_ROOT`, without needing a fully separate nginx static-file configuration for a project this size — a real, deliberately lightweight choice matching the same small-project pragmatism this course has followed since Chapter 1.

Gunicorn as a Real Production WSGI Server

`python manage.py runserver` is explicitly documented as unsuitable for production. A real WSGI server runs the actual application instead:

```
pip install gunicorn
gunicorn catalogue_site.wsgi:application --bind 127.0.0.1:8000
```

A real deployment runs this under a process manager (systemd is the natural choice, already covered by this site's own *Setting Up a Web Server on Debian* course), with nginx or Apache configured as a reverse proxy in front of it — genuine, general reverse-proxy setup this course doesn't re-derive, since it's already covered in full there.

Locking Down the Admin

Chapter 3's own warn-box flagged this directly: the admin shouldn't sit at its default, well-known URL on a real, publicly reachable deployment.

```
# catalogue_site/urls.py
```

```
path(config('ADMIN_URL_PATH', default='admin/'), admin.site.urls),
```

Setting a real, non-obvious `ADMIN_URL_PATH` in production's own `.env` — something like `a8x92-manage/` — doesn't replace real authentication as a security measure, but it does genuinely reduce exposure to the automated bots that specifically probe `/admin/` on every Django site they find, since this project's own admin no longer sits at the address they're already looking for.

HTTPS Is Not Optional Here

This project's own admin and add-item form accept real form submissions, and real database credentials flow through the deployment process. Serving any of it over plain HTTP sends that traffic unencrypted. This site's own *HTTPS/TLS Fundamentals* course covers obtaining and configuring a certificate in full — a real prerequisite for this project going live, not an optional hardening step to add later.

A Simple, Real Backup Habit

```
# Add to crontab -e, runs daily at 2am
0 2 * * * pg_dump -U catalogue_app catalogue > /home/user/backups/catalogue_$(date +\%Y
```

A REAL, HONEST SCOPE NOTE

A cron-scheduled `pg_dump` writing to the same server's own disk isn't a complete backup strategy — a real disk failure would take the backups down with the live data. For a small personal project, it's still a genuine, meaningful improvement over no backup at all, and copying the resulting `.sql` files somewhere off that same machine occasionally closes most of the realistic gap without needing a dedicated backup service.

Hands-On Exercises

EXERCISE 1

Create `.env.example` with placeholder values, add `.env` to a real `.gitignore`, and confirm (with `git status`, assuming the project is a Git repository) that `.env` genuinely doesn't appear as a trackable file once `.gitignore` is in place.

View solution

EXERCISE 2

Create the `catalogue_app` PostgreSQL role with exactly the privileges shown in this chapter, then confirm — by trying and expecting it to fail — that a query like `DROP TABLE catalogue_item;` run as that role is correctly rejected with a real permissions error.

View solution

EXERCISE 3

Set `DEBUG=False` and a deliberately incomplete `ALLOWED_HOSTS` in a local test setup, then explain the real error Django returns when visiting the site through a hostname not included in that list.

View solution

CHAPTER 9 QUICK REFERENCE

- **.env / .env.example** — real secrets gitignored, placeholders committed, loaded via `python-decouple`
- **DEBUG=False by default** — fails safe if the environment variable is ever missing; paired with a real `ALLOWED_HOSTS`
- **catalogue_app role** — `SELECT` / `INSERT` / `UPDATE` / `DELETE` only, never the app connecting as `postgres`
- **Static files** — `STATIC_ROOT` + `collectstatic` + WhiteNoise, the real production payoff of Chapter 8's own setup
- **Gunicorn** — a real WSGI server replacing `runserver`, run behind a reverse proxy (nginx/Apache, per this site's own web-server courses)
- **Admin hardening** — a non-default admin URL, resolving Chapter 3's own warn-box

- **Next chapter:** the capstone — integrating this catalogue into the existing Astro site, and a real plan for barcode scanning as the next phase

CHAPTER 10: CAPSTONE — INTEGRATING THE CATALOGUE INTO THE EXISTING SITE

Personal Catalogue: Django & PostgreSQL

Chapter 10 (Capstone) · Integrating the Catalogue Into the Existing Astro Site, and Planning Barcode Scanning as a Later Phase

Nine chapters have built a genuinely working catalogue — a real normalized schema, the admin as a fast entry tool, real public views, a type-aware add form, tag filtering, search (including a real PostgreSQL-specific full-text upgrade), styling, and a real deployment plan. This closing chapter covers the actual point of building it: getting it in front of the user, mounted onto their real, existing Astro-based site, then reviews the whole course chapter by chapter.

Three Real Ways to Mount a Django App Onto an Astro Site

Astro's own live site is a static build served by a web server — it has no Python runtime of its own, and doesn't need one for its own pages. Integrating this project isn't a matter of "adding it into Astro" the way a new Astro page would be added — it's a matter of getting both apps served correctly, side by side, behind the same web server:

OPTION	WHAT IT LOOKS LIKE	REAL EFFORT
Subdomain	`catalogue.example.com`, a fully separate vhost	Lowest — no code changes at all, just DNS + a new nginx server block
Path-mounted reverse proxy	`example.com/catalogue/`, same domain as the Astro site	Low — one nginx `location` block plus one real Django setting
JSON API + Astro frontend	Django returns data only; an Astro page/island renders it	Highest — every page effectively rebuilt in Astro/React

The Realistic Choice: Path-Mounted Reverse Proxy

For a personal project like this, a subdomain works but feels like more infrastructure than the project needs, and a full JSON-API rewrite defeats much of the point of having already built a working Django app across nine chapters. A path-mounted reverse proxy is the real middle

ground: one domain, the existing Astro site untouched, and the catalogue reachable at its own clean path.

Assuming nginx is already serving the Astro site's static build (per this site's own *Setting Up a Web Server on Debian* and *Nginx In Depth* courses), and Gunicorn (Chapter 9) is running the catalogue on a local port, the real config addition is one `location` block using `proxy_pass` rather than PHP's own `fastcgi_pass`:

/etc/nginx/sites-available/example.com (excerpt)

```
server {
    listen 443 ssl;
    server_name example.com;

    # The existing Astro static site
    root /var/www/example.com/dist;
    index index.html;

    location / {
        try_files $uri $uri/ =404;
    }

    # The catalogue, mounted at /catalogue/, proxied to Gunicorn
    location /catalogue/ {
        proxy_pass http://127.0.0.1:8000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

DJANGO NEEDS TO KNOW IT'S MOUNTED UNDER A SUB-PATH

Unlike a plain PHP app, where absolute links break unless manually prefixed, Django's own `{% url %}` tags and `reverse()` calls — used throughout every template built since Chapter 4 — generate URLs based on what Django itself believes its own root path is. Proxied at `/catalogue/` with no further configuration, Django still generates links starting from `/`, which would resolve to `example.com/item/1/` instead of the real, correctly mounted `example.com/catalogue/item/1/`. The fix is Django's own real `FORCE_SCRIPT_NAME` setting:

```
# settings.py
FORCE_SCRIPT_NAME = config('FORCE_SCRIPT_NAME', default=None)
```

Setting `FORCE_SCRIPT_NAME=/catalogue` in production's own `.env` tells Django, at the framework level, that every URL it generates needs that prefix — every `{% url 'item_detail' item.pk %}` call across every template built since Chapter 4 then correctly produces `/catalogue/item/1/` with zero changes to any individual template. This is the real Django-specific counterpart to the PHP variant's own absolute-vs-relative-redirect problem — a genuinely different mechanism, but solving the identical underlying "the app doesn't know it's not at the web root" issue.

Making It Feel Like Part of the Same Site

A reverse proxy solves reachability, not visual consistency. Chapter 8's own `style.css` already gives the catalogue a real dark theme — matching its own CSS custom properties (background/accent colors) to whatever the Astro site's own build already uses is a small, one-time styling pass rather than a full visual redesign, and the `base.html` nav bar built in Chapter 8 is the natural place to add a real "← Back to Main Site" link.

A Real Alternative for Deeper Integration: A JSON Endpoint

If, later, the catalogue's own search results need to appear directly on an Astro page rather than linking out to a separate Django page, the reverse-proxy setup already built supports that too — a small Django view can return JSON instead of HTML, and an Astro island can `fetch()` it client-side:

catalogue/views.py

```
from django.http import JsonResponse
from django.db.models import Q

def api_search(request):
    query = request.GET.get('q', '').strip()
    items = Item.objects.filter(
        Q(title__icontains=query) | Q(creator__icontains=query)
```

```
    ).values('id', 'title', 'creator')[:20]
    return JsonResponse(list(items), safe=False)
```

This isn't built out further in this course — it's flagged here honestly as the real next step if the project's own scope ever grows toward a tighter Astro-embedded experience, reusing exactly the same ORM querying discipline Chapter 7 already established.

Capstone: What Each Chapter Actually Built

The finished project is a genuine sum of nine real prior chapters, not a single new piece written for this capstone:

CHAPTER	WHAT IT CONTRIBUTED
1	The real Django project, its `catalogue` app, and a working PostgreSQL connection
2	The shared `Item` model, the `ItemType` lookup, and the `Tag` many-to-many
3	A real, customized Django admin as the fastest working data-entry tool
4	The real public list and detail views, plus the N+1 query bug found and fixed with `prefetch_related`
5	A public add-item form with type-aware labels via `json_script`, and the real server-vs-client validation distinction
6	Tag filtering, a real pagination-drops-the-filter bug found and fixed, and an aggregated tag index
7	Basic search, chained with the tag filter, plus a real PostgreSQL-specific full-text search upgrade
8	Real static-file setup, a reusable `_item_card.html` partial, and dark-theme styling
9	Environment-based settings, a least-privilege database role, `collectstatic`, Gunicorn, and admin hardening
10	Mounting the finished app behind the same domain as the real, existing Astro site

Where Barcode Scanning Fits Now

Chapter 1's own real deadline pushed barcode scanning out of scope for this course entirely — items get added manually, through the admin (Chapter 3) or the public form (Chapter 5). With the core catalogue now genuinely working and deployed, barcode scanning becomes a real, concrete next feature rather than an abstract "someday": a barcode's ISBN or UPC number looked up against a free API (Open Library for books is a natural real fit, matching this site's own established Food Tracker courses' use of Open Food Facts for a very similar lookup-by-code pattern), pre-filling `ItemForm`'s own fields instead of requiring every field typed by hand.

Hands-On Exercises

EXERCISE 1

Write the nginx `location /catalogue/` block from this chapter, set `FORCE_SCRIPT_NAME=/catalogue` in a local test `.env`, and confirm a rendered `{% url 'item_detail' item.pk %}` link now correctly includes the `/catalogue` prefix.

View solution

EXERCISE 2

Explain, in your own words, why `FORCE_SCRIPT_NAME` is the real Django-specific counterpart to the PHP variant's own absolute-vs-relative-redirect problem — what underlying issue do both actually solve?

View solution

EXERCISE 3

Build the `api_search` JSON view from this chapter, confirm it returns real JSON via a browser or `curl` request, and explain why it deliberately reuses the exact same `Q`-object query already built in Chapter 7 rather than writing new search logic.

View solution

CHAPTER 10 QUICK REFERENCE

- **Path-mounted reverse proxy** — one nginx `location /catalogue/` block using `proxy_pass` to Gunicorn
- **FORCE_SCRIPT_NAME** — the real Django setting that keeps every `{% url %}`-generated link correct once the app is mounted at a sub-path
- **Visual integration** — matching Chapter 8's own CSS custom properties to the Astro site's own theme
- **JSON endpoint** — a real, honest next step for tighter Astro-embedded integration, reusing Chapter 7's own query logic unchanged
- **Barcode scanning** — a real, concrete next phase now that the core catalogue is live, looked up via a free API and pre-filling `ItemForm`

Course Complete — Personal Catalogue: Django & PostgreSQL (10/10 chapters)