DJANGO

Fundamentals

Getting Started · URL Routing · Views

Templates · Models & the ORM

Model Relationships · Forms & Validation

Admin Interface & Settings

Philip Osztromok

Generated with Claude

Table of Contents

Django Fundamentals — 8 Chapters

🎸 Getting Started

Django's own tagline is "the web framework for perfectionists with deadlines" — and its defining trait is right there in the name of this chapter: it comes with almost everything already decided for you. Where FastAPI hands you a thin routing layer and lets you assemble the rest, Django ships an ORM, an admin interface, a templating engine, and a forms system on day one. This chapter gets a project running and makes that philosophy concrete.

Batteries-Included vs Minimal Core

You already know FastAPI's shape from the FastAPI Rebuild project — a handful of decorated functions and whatever libraries you chose to bring in yourself. Django starts from the opposite end:

FastAPI vs Django — Day One

FastAPI: Minimal Core

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello"}
```

No ORM, no admin, no forms system, no templating — you choose and wire up each piece yourself (SQLAlchemy, Jinja2, Pydantic, ...).

Django: Batteries Included

```
django-admin startproject mysite
cd mysite
python manage.py startapp blog
python manage.py runserver
```

Four commands and you already have an ORM, an admin interface, a templating engine, a forms system, and a migration tool — all part of the framework itself.

Convention Over Configuration

Like Rails' philosophy applied to Python — Django expects files in predictable places (`models.py`, `views.py`, `urls.py`) rather than asking you to decide the shape yourself.

The Trade-off

FastAPI is lighter and faster to learn end-to-end for a small API; Django's upfront structure pays off as an application grows past "a few endpoints."

📁 Project & App Structure

Django separates a **project** (the overall site configuration) from an **app** (a self-contained feature — blog, shop, accounts). One project can contain several apps:

```
mysite/              # the project
├── manage.py        # command-line utility — migrations, runserver, shell
├── mysite/
│   ├── settings.py  # all project configuration lives here
│   ├── urls.py      # the project's root URL routing table
│   ├── wsgi.py      # production entry point (sync)
│   └── asgi.py      # production entry point (async)
└── blog/            # an app
    ├── models.py    # database tables, as Python classes
    ├── views.py     # request-handling logic
    ├── urls.py      # the app's own URL patterns (optional but conventional)
    ├── admin.py     # registers models with the auto-generated admin site
    └── migrations/  # auto-generated schema change history
```

Project vs App

A project is the whole site; an app is one reusable piece of it — a well-built app can even be dropped into a different Django project later.

`manage.py`

The command-line entry point for everything: `runserver`, `migrate`, `startapp`, `shell`, `test` — no separate CLI tool to install.

`INSTALLED_APPS`

A list in `settings.py` — an app's models, admin registration, and templates only take effect once it's added here.

`settings.py`

One file for database config, installed apps, middleware, templates, static files — versus assembling that configuration yourself across several files in FastAPI.

From Zero to a Running Server

The Four Commands That Start Every Django Project

```
# 1. Install Django
pip install django

# 2. Create the project (the outer "mysite" folder + config)
django-admin startproject mysite
cd mysite

# 3. Create an app inside it
python manage.py startapp blog

# 4. Apply Django's own built-in migrations (users, sessions, admin tables)
python manage.py migrate

# 5. Run the development server
python manage.py runserver
# -> Starting development server at http://127.0.0.1:8000/
```

That `migrate` step is worth noticing: even before you've written a single model of your own, Django already has database tables it needs for its built-in admin, auth, and session systems — another concrete example of "batteries included."

💻 Coding Challenges

Challenge 1: Create Your First Project and App

Install Django, run `django-admin startproject` to create a project called `library`, then create an app inside it called `catalog` with `startapp`. List the files `startapp` generates and describe what each one is for.

Goal: Get comfortable with the project/app distinction and the files Django scaffolds automatically.

→ Solution

Challenge 2: Register the App

After creating the `catalog` app, explain (and demonstrate with a code snippet) exactly what has to change in `settings.py` before Django will recognize it, and what visibly fails if you forget this step.

Goal: Practice the `INSTALLED_APPS` registration step that's easy to forget coming from FastAPI's import-and-go style.

→ Solution

Challenge 3: Explore `manage.py`

Run `python manage.py help` and pick three subcommands you haven't used yet. For each one, write one sentence describing what it does and when you'd reach for it.

Goal: Build familiarity with `manage.py` as the single entry point for nearly every Django task, instead of memorizing commands one at a time as they come up.

→ Solution

💡 Real-World Tip: Forgetting `INSTALLED_APPS`

The single most common "why isn't this working" moment for a Django newcomer: creating an app with `startapp` and writing models or views in it, then finding they're invisible to migrations, the admin, or template loading. Django only looks at apps listed in `INSTALLED_APPS` in `settings.py` — `startapp` creates the files, but it does not register them for you.

🎯 What's Next

With a project and an app running, the next chapter covers how a request actually finds its way to your code: **URL Routing** — `urls.py`, `path()` patterns, and how Django's URL dispatch compares to FastAPI's decorator-based routes.

🧭 URL Routing

In FastAPI, a route and its handler are the same piece of code — the `@app.get("/users/{id}")` decorator sits directly above the function it triggers. Django keeps these separate on purpose: a dedicated `urls.py` file is the single table mapping every incoming path to the view that handles it. This chapter is about reading and writing that table.

A Separate Routing Table

FastAPI's Decorators vs Django's `urls.py`

FastAPI: Route Lives on the Function

```
@app.get("/books/{book_id}")
def get_book(book_id: int):
    return {"id": book_id}
```

To know every route in the app, you search for every `@app.get`/`@app.post` decorator across all your files.

Django: One Central Table

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

urlpatterns = [
    path("books/<int:book_id>/", views.get_book, name="book-detail"),
]
```

Every route for this app is visible in one file, independent of where `get_book` itself is defined.

`path()` and Converters

`path()` takes a route string, a view to call, and an optional name. Angle brackets capture URL segments and convert them to a specific Python type before your view ever sees them:

```
from django.urls import path
from . import views

urlpatterns = [
    path("", views.book_list, name="book-list"),
    path("<int:book_id>/", views.book_detail, name="book-detail"),
    path("category/<str:slug>/", views.by_category, name="book-category"),
    path("<uuid:token>/confirm/", views.confirm, name="book-confirm"),
]
```

`<int:book_id>`

Matches digits only and converts to a Python `int` — the equivalent of FastAPI's `book_id: int` type hint, just expressed in the route string itself.

`<str:slug>`

Matches any non-empty string except a forward slash — the default converter if you omit a type (`<slug>` alone works too).

`<uuid:token>`

Matches a formatted UUID string and converts it to a Python `UUID` object automatically.

Path Parameter, Not Query String

Just like FastAPI, these are segments of the path itself — query strings (`?page=2`) are read separately from `request.GET` inside the view.

🔗 Wiring Apps Into the Project

Chapter 1 introduced project vs app. Routing is where that split becomes concrete: the project's root `urls.py` doesn't list every route directly — it `include()`s each app's own `urls.py` under a prefix:

Root `urls.py` Delegating to an App

```
# mysite/urls.py — the PROJECT's root routing table
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("books/", include("catalog.urls")),
    path("accounts/", include("accounts.urls")),
]
```

Every pattern inside `catalog/urls.py` is now reachable under `/books/` — `path("<int:book_id>/", ...)` inside that file resolves to `/books/<int:book_id>/` overall. An app's own `urls.py` never needs to know or care what prefix it will eventually sit under.

`re_path()`: Regex Routes

For patterns `path()`'s converters can't express, `re_path()` accepts a full regular expression — the same syntax from the Regular Expressions course, just applied to URL matching instead of text search:

```
from django.urls import re_path
from . import views

urlpatterns = [
    re_path(r"^archive/(?P<year>[0-9]{4})/$", views.year_archive, name="year-archive"),
]
```

Named groups (`(?P<year>...)`) are passed to the view as keyword arguments, the same way `path()`'s converters are — `re_path()` is the escape hatch for the (now rare) cases where a plain `path()` pattern isn't expressive enough.

🏷️ Named URLs: Never Hardcode a Path

Every example above passed a `name=` argument — that name lets templates and Python code build a URL by name instead of typing the literal path, so renaming a route doesn't mean hunting down every hardcoded string:

```
# In a view (Python)
from django.urls import reverse

url = reverse("book-detail", kwargs={"book_id": 42})
# -> "/books/42/" — built from the URL pattern, not a hand-typed string
<!-- In a template (Django Template Language, Chapter 4) -->
<a href="{% url 'book-detail' book_id=42 %}">View Book</a>
```

`reverse()`

The Python function version — used in views, redirects, and tests wherever a URL needs to be built programmatically.

`{% url %}`

The template tag version — used in HTML templates so a link's target is never a brittle, hand-typed string.

💻 Coding Challenges

Challenge 1: Write an App's `urls.py`

For a `catalog` app, write `urlpatterns` with three routes: a book list at the app's root, a book detail using an `int` converter, and a category listing using a `str` (or `slug`) converter — each with a sensible `name`.

Goal: Practice `path()` converters and naming routes from the start rather than as an afterthought.

→ Solution

Challenge 2: Wire It Into the Project

Write the project's root `urls.py` that includes the `catalog` app's URLs under the `/books/` prefix, plus a second app `accounts` included under `/accounts/`. State the final full path for each of the three routes from Challenge 1.

Goal: Practice the project-to-app URL delegation pattern and reasoning about the combined prefix + pattern.

→ Solution

Challenge 3: Use `reverse()` Instead of a Hardcoded Path

Given a view function that currently redirects with a hardcoded string like `redirect("/books/42/")`, rewrite it to use `reverse("book-detail", kwargs={"book_id": 42})` instead, and explain in one sentence why this matters if the `/books/` prefix ever changes.

Goal: Practice never hardcoding a URL that a named route already describes.

→ Solution

⚠️ Gotcha: Forgetting to `include()` a New App's URLs

Writing a perfectly correct `catalog/urls.py` does nothing on its own — Django only knows about it once the project's root `urls.py` actually `include()`s it. Visiting a route that "should" exist and getting a 404 is very often this exact mistake, not a typo in the app's own URL patterns. It's the routing-layer sibling of Chapter 1's `INSTALLED_APPS` gotcha: the file existing on disk and Django actually knowing about it are two separate steps.

🎯 What's Next

URL patterns now know which Python callable to invoke — the next chapter is about what that callable actually looks like: **Views**, comparing Django's function-based views against class-based views, and how the request/response objects differ from FastAPI's.

👁️ Views

A view is the Python callable that `urls.py` hands a request to — the same role a FastAPI path operation function plays. The shape is different in two ways worth learning early: Django's `request` and `response` are their own classes rather than framework-injected parameters and auto-serialized return values, and Django gives you a choice between plain functions and classes for the view itself.

Function-Based Views (FBVs)

The simplest view is just a function that takes a request and returns a response:

```
# catalog/views.py
from django.http import HttpResponse

def book_list(request):
    return HttpResponse("Here are all the books.")
```

Compare that to the FastAPI shape you already know:

FastAPI Path Operation vs Django FBV

FastAPI

```
@app.get("/books")
def book_list():
    return {"books": []}
# A plain dict is auto-serialized to a JSON response.
```

Django

```
from django.http import JsonResponse

def book_list(request):
    return JsonResponse({"books": []})
# A dict must be explicitly wrapped in JsonResponse.
```

The Request Object

Every view's first parameter is Django's `HttpRequest` — where FastAPI hands you individually typed parameters (path params, query params, a Pydantic body model), Django hands you one object and you pull what you need off it:

```
def search_books(request):
    query = request.GET.get("q", "")        # query string: ?q=dune
    method = request.method                 # "GET", "POST", etc.
if request.method == "POST":
        title = request.POST.get("title")       # form-encoded POST data
return HttpResponse(f"Searching for: {query}")
```

`request.GET`

A dict-like object of query string parameters — always present, empty on a request with no query string.

`request.POST`

Form-encoded POST body data — for JSON request bodies, you'd instead parse `request.body` yourself (Chapter 7 covers Forms properly).

`request.method`

One function often handles GET and POST for the same URL by branching on this — versus FastAPI's separate `@app.get`/`@app.post` decorators for the same path.

`request.user`

Once authentication middleware runs (Course 2), the logged-in user is attached here automatically — no manual dependency injection needed.

The Response Object

Every view must return something that is (or behaves like) an `HttpResponse` — Django provides several shortcuts for common cases:

```python
from django.http import HttpResponse, JsonResponse
from django.shortcuts import render, redirect

def plain_text(request):
    return HttpResponse("Just some text", content_type="text/plain")

def as_json(request):
    return JsonResponse({"status": "ok"})

def as_html(request):
    # render() combines a template + context dict + HttpResponse into one call —
# the templating side of this is fully covered in Chapter 4.
return render(request, "catalog/book_list.html", {"books": []})

def after_save(request):
    return redirect("book-list")  # by URL name, per Chapter 2
```

Class-Based Views (CBVs)

Django also lets a view be a class — instead of branching on `request.method` inside one function, each HTTP method gets its own class method:

Function-Based vs Class-Based, Same Route

Function-Based View

```
def book_form(request):
    if request.method == "POST":
        # handle form submission
return redirect("book-list")
    # request.method == "GET"
return render(request, "catalog/book_form.html")
```

Class-Based View

```
from django.views import View

class BookFormView(View):
    def get(self, request):
        return render(request, "catalog/book_form.html")

    def post(self, request):
        # handle form submission
return redirect("book-list")
```

Wiring a CBV into `urls.py` needs one extra step — `.as_view()` converts the class into something `path()` can call:

```
from . import views

urlpatterns = [
    path("new/", views.BookFormView.as_view(), name="book-create"),
]
```

Dispatch by Method Name

`View`'s base class reads `request.method` internally and calls the matching method (`get`, `post`, `put`, ...) — no manual `if`/`elif` chain needed.

Generic CBVs (Preview)

Django ships ready-made CBVs like `ListView` and `DetailView` that implement common patterns almost entirely for you — a deep dive is coming in Course 2.

💻 Coding Challenges

Challenge 1: Write a GET/POST Function-Based View

Write a `subscribe(request)` FBV that returns an HTML form on GET, and on POST, reads an `email` field from `request.POST` and returns a plain-text confirmation via `HttpResponse`.

Goal: Practice branching on `request.method` and reading from `request.POST`.

→ Solution

Challenge 2: Convert It to a Class-Based View

Rewrite the `subscribe` view from Challenge 1 as a `SubscribeView(View)` class with separate `get()` and `post()` methods, and update the corresponding `urls.py` entry to use `.as_view()`.

Goal: Practice the FBV-to-CBV conversion and the `.as_view()` wiring step.

→ Solution

Challenge 3: Return the Right Response Type

Write three tiny views: one returning plain text, one returning JSON, and one returning a redirect to a named URL — using `HttpResponse`, `JsonResponse`, and `redirect()` respectively.

Goal: Build muscle memory for Django's explicit response types, instead of assuming a return value gets auto-converted the way FastAPI would.

→ Solution

⚠️ Gotcha: Forgetting to Return an `HttpResponse`

In FastAPI, returning a dict, a list, or even `None` from a path operation function is handled gracefully — FastAPI serializes whatever you give it. Django is stricter: a view **must** return an `HttpResponse` (or a subclass like `JsonResponse`, or the result of `render()`/`redirect()`, which are HttpResponse under the hood). A view that falls through without an explicit `return` — often because an `if` branch handles POST but there's no matching `return` for GET — raises a very literal `ValueError: The view didn't return an HttpResponse object. It returned None instead.`

🎯 What's Next

Views now return real responses — the next chapter covers where the HTML in `render()` actually comes from: **Templates**, the Django Template Language, template inheritance, and how its auto-escaping compares to what you've seen already.

🎨 Templates

`render()` from the last chapter needs an actual template file to render. Django ships its own templating engine — the Django Template Language (DTL) — which looks a lot like Jinja2 (the templating engine FastAPI projects commonly reach for) but is deliberately more restricted: DTL can't call arbitrary Python, on purpose.

Where Templates Live

By default, Django looks for templates inside each installed app's own `templates/` directory — and convention nests one more folder, named after the app, to avoid collisions between apps that both have a `list.html`:

```
catalog/
├── templates/
│   └── catalog/          # the extra "catalog" folder prevents name clashes
│       ├── base.html
│       ├── book_list.html
│       └── book_detail.html
├── views.py
└── urls.py
```

```python
# catalog/views.py
from django.shortcuts import render

def book_list(request):
    books = [{"title": "Dune"}, {"title": "1984"}]
    return render(request, "catalog/book_list.html", {"books": books})
```

📝 Variables, Tags & Filters

DTL syntax will look immediately familiar if you've used Jinja2 — double curly braces print a value, `{% %}` runs a tag, and `|` applies a filter:

```
<h1>{{ book.title }}</h1>
<p>Published: {{ book.year|default:"unknown" }}</p>
<p>{{ book.title|upper }}</p>
<p>{{ book.description|truncatewords:20 }}</p>
```

Jinja2 (FastAPI) vs Django Template Language

Jinja2

```
{{ book.get_display_name() }}
{% for book in books %}
 {{ book.title }}
{% endfor %}
```

Method calls with parentheses and arguments are allowed directly in the template.

Django Template Language

```
{{ book.get_display_name }}
{% for book in books %}
 {{ book.title }}
{% endfor %}
```

No parentheses — a no-argument method is called automatically *without* them; arbitrary Python calls with arguments aren't allowed at all.

Logic-Light by Design

DTL intentionally can't run arbitrary Python — the idea is that business logic belongs in `views.py` and models, not scattered across templates.

Dot Lookup Does Everything

`book.title` tries, in order: dictionary lookup, attribute lookup, then a no-argument method call — one syntax covers all three.

Control Flow Tags

```
<ul>
{% if books %}
 {% for book in books %}
   <li>{{ book.title }} ({{ forloop.counter }} of {{ books|length }})</li>
 {% endfor %}
{% else %}
 <li>No books yet.</li>
{% endif %}
</ul>
```

`forloop.counter` is a DTL-specific convenience — a 1-indexed loop counter available automatically inside any `{% for %}`, with no separate variable to pass in from the view.

🛡️ Auto-Escaping

Every variable is HTML-escaped by default — the same defense the XSS course covered as the primary fix for output-context injection, applied automatically here without you having to remember to call an escape function:

```
# If book.title == '<script>alert("xss")</script>'
{{ book.title }}
# renders as the literal text &lt;script&gt;alert("xss")&lt;/script&gt; — inert, not executed

{{ book.title|safe }}
# ⚠️ opts OUT of escaping — only use this for content you trust completely,
# e.g. HTML you sanitized yourself with a library, per the XSS course's DOMPurify chapter
```

Template Inheritance: `extends` and `block`

Nearly every page shares a header, nav, and footer — inheritance defines that shared shell once, in a base template, and lets each page override just the parts that differ:

A Base Template and a Child Page

```
<!-- catalog/templates/catalog/base.html -->
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}My Library{% endblock %}</title>
</head>
<body>
  <nav>Home | Books</nav>
  {% block content %}{% endblock %}
  <footer>&copy; 2026</footer>
</body>
</html>

<!-- catalog/templates/catalog/book_list.html -->
{% extends "catalog/base.html" %}

{% block title %}All Books{% endblock %}

{% block content %}
  <ul>
  {% for book in books %}
    <li>{{ book.title }}</li>
  {% endfor %}
  </ul>
{% endblock %}
```

`{% extends %}`

Must be the very first tag in a child template — declares which base template this page fills in.

`{% block %}` / `{% endblock %}`

Marks a named region a child can override; a block left unfilled in the child just uses the base template's default content.

`{% include %}`

Pulls in a separate template fragment inline (`{% include "catalog/_book_card.html" %}`) — for reusable partials rather than a whole-page shell.

Deep Inheritance Chains

A child template can itself be extended by a grandchild — `base.html` → `catalog/base.html` → `book_list.html` is a common real pattern.

💻 Coding Challenges

Challenge 1: Render a List With Filters

Write a template that loops over a `books` context variable, printing each book's title in uppercase and its description truncated to 15 words, with a fallback message if the list is empty.

Goal: Practice `{% for %}`/`{% if %}` tags together with the `upper` and `truncatewords` filters.

→ Solution

Challenge 2: Build a Base Template + Two Child Pages

Write a `base.html` with a `title` block and a `content` block, then two child templates (`book_list.html` and `book_detail.html`) that each `{% extends %}` it and fill in both blocks differently.

Goal: Practice the extends/block inheritance pattern across more than one page.

→ Solution

Challenge 3: Demonstrate Auto-Escaping

Pass a string containing `<script>` tags into a template context, render it two ways — once as plain `{{ value }}` and once as `{{ value|safe }}` — and explain what each one actually sends to the browser and why one is dangerous outside of trusted content.

Goal: Practice recognizing auto-escaping in action and understanding exactly what `|safe` turns off.

→ Solution

⚠️ Gotcha: A Typo Renders as Nothing, Not an Error

Reference `{{ boook.title }}` (misspelled) or a context variable that was never passed in, and DTL doesn't raise an exception — it silently renders an empty string and moves on. A page that's missing text with no error in the console or logs is very often exactly this: a typo'd variable name in a template. When something looks blank that shouldn't be, check the exact variable name against what the view actually passed into `render()`'s context dict before assuming the bug is elsewhere.

🎯 What's Next

Templates can now display data — the next chapter covers where that data actually comes from: **Models & the ORM**, defining database tables as Python classes, migrations, and querying with Django's `QuerySet` API.

🗂️ Models & the ORM

A FastAPI project typically bolts on SQLAlchemy (or another ORM) as a separate decision. Django's ORM ships with the framework itself — one more "batteries included" default from Chapter 1. This chapter defines a database table as a Python class, migrates it into existence, and queries it through Django's `QuerySet` API.

Defining a Model

A model is a Python class whose attributes describe the table's columns — Django infers the SQL types, the column names, and generates the migration for you:

```python
# catalog/models.py
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)
    price = models.DecimalField(max_digits=6, decimal_places=2)
    published_date = models.DateField(null=True, blank=True)
    in_stock = models.BooleanField(default=True)

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

Field Types

`CharField`(short text, requires `max_length`), `TextField` (unlimited text), `IntegerField`, `DecimalField`, `DateField`, `BooleanField` — each maps to an appropriate SQL column type.

`blank` vs `null`

`null=True` allows `NULL` at the database level; `blank=True` allows an empty value in forms/validation — the two are independent and commonly both set together for optional fields.

`__str__`

Not required, but without it, the admin site and shell both show an unhelpful `Book object (1)` instead of the book's actual title.

Implicit `id`

Django adds an auto-incrementing primary key field automatically unless you define your own — no need to declare it yourself.

🔀 Migrations

Django's migrations are **auto-generated by diffing your models** against the last known schema state — you rarely hand-write the SQL yourself, which is a real difference from writing raw migration files by hand:

```
python manage.py makemigrations catalog
# Migrations for 'catalog':
#   catalog/migrations/0001_initial.py
#     - Create model Book

python manage.py migrate
# Applying catalog.0001_initial... OK
```

`makemigrations`

Compares your current models against the migration history and writes a new migration file describing the difference — it does not touch the database.

`migrate`

Actually applies pending migration files to the database — the step that runs the real `CREATE TABLE`/`ALTER TABLE` SQL.

🔍 The QuerySet API

Every model gets a default `.objects` manager — the entry point for every query, returning a `QuerySet` that reads much like a chained, English-ish sentence:

```
# All books
Book.objects.all()

# Filtered — WHERE in_stock = TRUE
Book.objects.filter(in_stock=True)

# Chained filters — WHERE in_stock = TRUE AND price < 20
Book.objects.filter(in_stock=True).filter(price__lt=20)

# Ordering — ORDER BY title ASC
Book.objects.order_by("title")

# A single row, or an exception if zero or more than one match
Book.objects.get(pk=1)

# Excluding rows
Book.objects.exclude(price=0)
```

FastAPI + SQLAlchemy vs Django's QuerySet

FastAPI + SQLAlchemy Session

```
result = session.execute(
    select(Book).where(Book.in_stock == True)
                .order_by(Book.title)
)
books = result.scalars().all()
```

Django QuerySet

```
books = Book.objects.filter(in_stock=True).order_by("title")
```

QuerySets Are Lazy

Writing `Book.objects.filter(...)` does not hit the database — a `QuerySet` only actually runs its query once it's *evaluated*: iterated over, sliced, or converted to a list/bool:

```
qs = Book.objects.filter(in_stock=True)   # no query yet
qs = qs.order_by("title")                 # still no query — just building up the query
for book in qs:                           # NOW the SQL actually runs
    print(book.title)
```

This is why chaining `.filter().exclude().order_by()` across several lines is cheap — Django combines everything into a single SQL query at the moment it's finally evaluated, rather than running a separate query per method call.

Automatic SQL Injection Protection

Every value passed into the QuerySet API is parameterized automatically — the same defense the SQL Injection course spent a full chapter teaching how to do by hand:

```
# Completely safe — Django parameterizes `title` regardless of its content
Book.objects.filter(title=user_supplied_title)

# Even raw() queries are parameterized IF you use placeholders correctly:
Book.objects.raw("SELECT * FROM catalog_book WHERE title = %s", [user_supplied_title])

# ❌ Still dangerous — string-formatting a raw query bypasses the protection entirely,
# exactly the vulnerability pattern from the SQLi course:
# Book.objects.raw(f"SELECT * FROM catalog_book WHERE title = '{user_supplied_title}'")
```

💻 Coding Challenges

Challenge 1: Define and Migrate a Model

Define an `Author` model with `name` (CharField), `bio` (TextField, optional), and `birth_year` (IntegerField, optional), add a proper `__str__`, then write the two commands you'd run to create and apply its migration.

Goal: Practice choosing appropriate field types and the makemigrations/migrate workflow.

→ Solution

Challenge 2: Write Five QuerySet Queries

Against the `Book` model from this chapter, write QuerySet expressions for: all in-stock books ordered by price ascending, books under $15, the single book with `pk=3`, all books excluding out-of-stock ones, and the count of all books.

Goal: Build fluency with `filter`, `exclude`, `order_by`, `get`, and `count`.

→ Solution

Challenge 3: CRUD in the Django Shell

Using `python manage.py shell`, write the Python statements to create a new `Book`, fetch it back with `.get()`, update its price and save it, then delete it — all through the ORM, no raw SQL.

Goal: Practice the full create/read/update/delete cycle through the ORM's object-oriented API.

→ Solution

⚠️ Gotcha: `.get()` Raises When It Doesn't Return Exactly One Row

`Book.objects.get(title="Dune")` looks like it should behave like a dictionary's `.get()` — returning `None` if nothing matches. It does not: zero matching rows raises `Book.DoesNotExist`, and more than one matching row raises `Book.MultipleObjectsReturned`. Use `.get()` only when you're certain exactly one row should match (typically by primary key), and reach for `.filter(...).first()` — which returns `None` gracefully instead of raising — whenever a missing row is a normal, expected outcome rather than a bug.

🎯 What's Next

A single model can now be created, queried, and migrated — the next chapter covers what happens when models need to reference each other: **Model Relationships**, `ForeignKey` and `ManyToMany` fields, and the N+1 query problem this course has run into before, seen here from the ORM's side.

🔗 Model Relationships

Real data rarely lives in one table — a book has an author, a book has many tags. Django expresses those relationships as fields on a model, but the convenience of writing `book.author.name` hides a real performance trap this chapter's second half is dedicated to fixing: the N+1 query problem, seen here from the ORM's side rather than raw SQL.

`ForeignKey`: One-to-Many

An author writes many books; a book has exactly one author — that's a `ForeignKey` on the "many" side:

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

class Author(models.Model):
    name = models.CharField(max_length=150)

    def __str__(self):
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(
        Author,
        on_delete=models.CASCADE,
        related_name="books",
    )
```

`on_delete`

Required for every `ForeignKey` — `CASCADE` deletes dependent books when their author is deleted; `PROTECT` blocks the delete instead; `SET_NULL` requires `null=True` and clears the reference.

`related_name`

Names the reverse accessor — without it, Django defaults to `book_set`; with `related_name="books"` as above, it becomes the more readable `author.books`.

```
# Forward access: book -> author (always singular, always present unless nullable)
book.author.name

# Reverse access: author -> books (a QuerySet, since an author can have many)
author.books.all()
```

`ManyToManyField`: Many-to-Many

A book can have several tags, and a tag applies to several books — Django manages the hidden join table for you automatically:

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

    def __str__(self):
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(Author, on_delete=models.CASCADE, related_name="books")
    tags = models.ManyToManyField(Tag, related_name="books", blank=True)

# Usage
book.tags.add(sci_fi_tag)
book.tags.all()        # every tag on this book
sci_fi_tag.books.all()  # every book with this tag, via related_name
```

For a many-to-many that needs extra data on the relationship itself (e.g. "when was this tag added"), Django supports an explicit `through=` model instead of the automatically-managed hidden table.

Querying Across Relationships

Double-underscore lookups span relationships the same way they reach into a single model's fields:

```
# Books written by an author named "Le Guin"
Book.objects.filter(author__name="Ursula K. Le Guin")

# Books tagged "sci-fi"
Book.objects.filter(tags__name="sci-fi")

# Authors who have written more than one book with a price under 15
Author.objects.filter(books__price__lt=15).distinct()
```

⚠️ The N+1 Problem, From the ORM Side

The relationship access that reads so naturally — `book.author.name` — hides a query every time it runs. Looping over books and touching `.author` on each one is the same N+1 trap the Database Integration and Rails courses warned about:

N+1 Loop vs Fixed Version

N+1: One Query Per Book

```
books = Book.objects.all()  # 1 query
for book in books:
    print(book.author.name)  # 1 MORE query, per book (N queries)
# Total: 1 + N queries for N books
```

Fixed: One Query, Total

```
books = Book.objects.select_related("author")  # 1 query, SQL JOIN
for book in books:
    print(book.author.name)  # no extra query — already loaded
# Total: 1 query, period
```

`select_related()`

For `ForeignKey`/`OneToOne` only — issues a single SQL `JOIN`, since each book has exactly one author, so the rows combine cleanly.

`prefetch_related()`

For `ManyToMany` and reverse `ForeignKey` — runs a *second* separate query and joins the results together in Python, since a JOIN would multiply rows (one book with 3 tags would otherwise return 3 duplicated book rows).

Fixing Both Relationship Types at Once

```
books = Book.objects.select_related("author").prefetch_related("tags")

for book in books:
    print(book.title, book.author.name, [t.name for t in book.tags.all()])
# Exactly 2 queries total, regardless of how many books there are:
# one JOINed query for books+authors, one separate query for all tags.
```

💻 Coding Challenges

Challenge 1: Add a `ForeignKey` Relationship

Add a `Publisher` model with a `name` field, then add a `publisher` `ForeignKey` on `Book` with `on_delete=models.PROTECT` and a sensible `related_name`. Explain in one sentence why `PROTECT` might be the right choice here instead of `CASCADE`.

Goal: Practice choosing an `on_delete` behavior deliberately, not just defaulting to `CASCADE`.

→ Solution

Challenge 2: Query Across a Many-to-Many

Using the `Book`/`Tag` models from this chapter, write a QuerySet expression for "all books tagged either 'sci-fi' or 'fantasy'," making sure the result doesn't contain duplicate books.

Goal: Practice relationship-spanning lookups combined with `.distinct()` where a many-to-many join could otherwise produce repeated rows.

→ Solution

Challenge 3: Fix an N+1 Query

Given a view that loops over `Book.objects.all()` and, for each book, prints its author's name AND every tag's name, rewrite the query to eliminate the N+1 problem for both relationships in a single fixed QuerySet.

Goal: Practice choosing `select_related` vs `prefetch_related` correctly for two different relationship types in the same query.

→ Solution

⚠️ Gotcha: Using `select_related` Where You Needed `prefetch_related`

`select_related()` only works for relationships where the "many" side isn't involved from this model's perspective — `ForeignKey` and `OneToOne`, where a SQL `JOIN` produces exactly one combined row per original row. Try it on a `ManyToManyField` or a reverse `ForeignKey` (`author.books`) and Django raises a `FieldError` — because a JOIN there would multiply rows (one author with 5 books becomes 5 duplicated author rows), which is exactly why those relationships need `prefetch_related()`'s separate-query-then-join-in-Python approach instead.

🎯 What's Next

Relationships between models are now readable, queryable, and efficient — the next chapter turns to getting new data *in*: **Forms & Validation**, Django's Forms and ModelForms, and its built-in CSRF protection compared to the manual setups from FastAPI and Express.

📝 Forms & Validation

FastAPI's Pydantic models validate a request body but leave you to hand-write the HTML form yourself. Django's `Form` and `ModelForm` classes do both jobs at once — rendering the fields *and* validating what comes back — and, without any extra package, protect every form against CSRF the way an entire dedicated course covered doing by hand.

A Basic `Form`

A `Form` class describes fields the same way a model does — but these fields describe input validation, not database columns:

```
# catalog/forms.py
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)


# catalog/views.py
def contact(request):
    if request.method == "POST":
        form = ContactForm(request.POST)
        if form.is_valid():
            data = form.cleaned_data  # validated, type-converted dict
# ... send an email, save a record, etc. ...
return redirect("contact-success")
    else:
        form = ContactForm()
    return render(request, "catalog/contact.html", {"form": form})
```

`is_valid()`

Runs every field's validation — `EmailField` rejects a malformed address, `CharField(max_length=100)` rejects anything longer — and returns `True`/`False`.

`form.cleaned_data`

Only populated after a successful `is_valid()` — already type-converted (an `IntegerField` gives back a real `int`, not a string).

🧬 `ModelForm`: Generated From a Model

Writing a `Form` by hand that mirrors a model's fields is duplicated work — `ModelForm` generates the form directly from the model itself:

```
# catalog/forms.py
from django import forms
from .models import Book

class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ["title", "description", "price", "published_date"]

# catalog/views.py
def create_book(request):
    if request.method == "POST":
        form = BookForm(request.POST)
        if form.is_valid():
            form.save()  # creates AND saves a Book instance directly
return redirect("book-list")
    else:
        form = BookForm()
    return render(request, "catalog/book_form.html", {"form": form})
```

`form.save()` both validates *and* persists the model instance — one line replaces manually reading each field off `request.POST` and constructing a `Book(...)` by hand.

🛡️ Built-in CSRF Protection

The entire CSRF course was dedicated to the synchronizer token pattern — generate a token, embed it in the form, validate it on submit. Django implements exactly that pattern, on by default, for every POST form:

```
<!-- catalog/templates/catalog/book_form.html -->
<form method="post">
 {% csrf_token %}
 {{ form.as_p }}
 <button type="submit">Save</button>
</form>
```

Django's Built-in CSRF vs Express's Manual Setup

Express: Manual Setup Required

```
const csrf = require("csurf");
app.use(csrf());

app.get("/form", (req, res) => {
  res.render("form", { csrfToken: req.csrfToken() });
});
// Requires installing a package, wiring middleware,
// and manually passing the token into every template.
```

Django: On By Default

```
<!-- Just one template tag — CsrfViewMiddleware -->
<!-- is already active in MIDDLEWARE by default. -->
{% csrf_token %}
```

`CsrfViewMiddleware` is already in `settings.py`'s `MIDDLEWARE` list from the moment `startproject` creates it — every POST form just needs the one template tag, and Django rejects any POST request missing a valid token with a 403, automatically.

Custom Validation

Beyond a field's built-in checks, a form can define its own validation logic — per-field with `clean_<fieldname>()`, or across multiple fields with `clean()`:

```
class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ["title", "price", "published_date"]

    def clean_price(self):
        price = self.cleaned_data["price"]
        if price <= 0:
            raise forms.ValidationError("Price must be greater than zero.")
        return price

    def clean(self):
        cleaned_data = super().clean()
        if cleaned_data.get("published_date") and cleaned_data["published_date"].year > 2026:
            raise forms.ValidationError("Published date can't be in the future.")
        return cleaned_data
```

Mass Assignment: `Meta.fields` as an Allowlist

Rails' Forms chapter warned about mass assignment vulnerabilities — Django's protection is `ModelForm.Meta.fields` acting as an explicit allowlist:

Dangerous vs Safe `Meta.fields`

Dangerous: Exposes Everything

```
class Meta:
    model = User
    fields = "__all__"
# Includes is_staff, is_superuser — a user submitting
# extra form fields could grant themselves admin access.
```

Safe: Explicit Allowlist

```
class Meta:
    model = User
    fields = ["username", "email"]
# Only these two fields can ever be set through this form —
# is_staff simply isn't reachable, no matter what's submitted.
```

💻 Coding Challenges

Challenge 1: Write a `ModelForm`

Write a `AuthorForm(forms.ModelForm)` for the `Author` model from Chapter 5, exposing only `name` and `bio` (not `birth_year`), then write the view function that renders it on GET and saves it on a valid POST.

Goal: Practice `ModelForm`'s `Meta.fields` allowlist and the `is_valid()`/`save()` pattern.

→ Solution

Challenge 2: Add Custom Validation

Add a `clean_bio()` method to the `AuthorForm` from Challenge 1 that rejects a bio longer than 500 characters with a clear `ValidationError` message.

Goal: Practice field-level custom validation beyond what the field type checks automatically.

→ Solution

Challenge 3: Spot the Mass Assignment Risk

Given a `ModelForm` for a `UserProfile` model (with fields `bio`, `avatar_url`, and `is_verified`) that uses `fields = "__all__"`, rewrite it to a safe explicit allowlist and explain in one sentence what real-world exploit the original version was vulnerable to.

Goal: Practice recognizing and fixing the Django equivalent of the mass assignment vulnerability the Rails course covered.

→ Solution

⚠️ Gotcha: Forgetting `{% csrf_token %}`

Omit `{% csrf_token %}` from a POST form's template and **every single submission fails** with `403 Forbidden: CSRF verification failed` — not a silent bug, but a confusing one the first time it happens, since GET requests through the same view work fine. Because `CsrfViewMiddleware` is on by default project-wide, this isn't optional configuration to remember per-view the way Express's `csurf` setup is — it's a template detail that's easy to skip when copy-pasting a form from example code that used `{{ form.as_p }}` alone.

🎯 What's Next

The final chapter of this course covers the feature that's arguably Django's biggest single differentiator from FastAPI and Express: **Admin Interface & Settings** — the auto-generated admin site, and how `settings.py` ties every piece covered so far together.

🛠️ Admin Interface & Settings

Neither FastAPI nor Express gives you this for free: a full CRUD interface for every model, generated automatically, with authentication already built in. This final chapter turns on Django's admin site for the models built across this course, then ties together `settings.py` — the file this course has referenced loosely in nearly every chapter without ever looking at directly.

Creating a Superuser

The admin site needs an account with full access — `createsuperuser` is an interactive command that sets one up:

```
python manage.py createsuperuser
# Username: admin
# Email address: admin@example.com
# Password: ********
# Password (again): ********
# Superuser created successfully.

python manage.py runserver
# Visit http://127.0.0.1:8000/admin/ and log in.
```

📋 Registering Models

A model appears in the admin the moment it's registered — one line, versus building an entire CRUD UI by hand:

```
# catalog/admin.py
from django.contrib import admin
from .models import Book, Author, Tag, Publisher

admin.site.register(Book)
admin.site.register(Author)
admin.site.register(Tag)
admin.site.register(Publisher)
```

Hand-Building CRUD vs Django's Admin

> **FastAPI / Express: Build It Yourself**
>
> Routes for list/create/edit/delete, HTML templates for each, form handling, pagination, search, access control for who can reach these pages — all written and maintained by hand, per model.

> **Django: Four Lines**
>
> `admin.site.register(Book)` — full list view, create/edit forms (auto-generated from the model's fields), delete confirmation, search, and login-gated access, all included.

Customizing With `ModelAdmin`

The default registration works, but a `ModelAdmin` subclass controls exactly what the list view shows and how it's searched/filtered:

```
from django.contrib import admin
from .models import Book, Author

@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    list_display = ["title", "author", "price", "in_stock"]
    list_filter = ["in_stock", "author"]
    search_fields = ["title", "author__name"]
```

`list_display`

Which columns appear in the list view — without it, the admin shows only each row's `__str__`.

`list_filter`

Adds a sidebar of clickable filters — Django generates the filter options from the field's actual values automatically.

`search_fields`

Enables a search box; `"author__name"` reaches across the relationship the same way a QuerySet filter would (Chapter 6).

`@admin.register(...)`

A decorator form equivalent to calling `admin.site.register(Book, BookAdmin)` separately — just a more compact way to associate a model with its customization class.

Inline Editing: Related Models on One Page

An `Inline` lets you edit an author's books directly on the author's own admin page — no separate navigation needed:

Editing Books Inline on the Author Page

```
from django.contrib import admin
from .models import Author, Book

class BookInline(admin.TabularInline):
    model = Book
    extra = 1  # how many empty "add a new one" rows to show
@admin.register(Author)
class AuthorAdmin(admin.ModelAdmin):
    list_display = ["name"]
    inlines = [BookInline]
```

⚙️ Tying It Together: `settings.py`

Every chapter has referenced `settings.py` loosely — here's the same file, with everything covered so far pointed out in context:

```
# mysite/settings.py

DEBUG = True # see this chapter's gotcha below
ALLOWED_HOSTS = []    # hostnames allowed to serve this site in production

INSTALLED_APPS = [    # Chapter 1 — an app does nothing until it's listed here
"django.contrib.admin",    # powers this very chapter's admin site
"django.contrib.auth",
   "django.contrib.contenttypes",
   "django.contrib.sessions",
   "django.contrib.messages",
   "django.contrib.staticfiles",
   "catalog",
]

MIDDLEWARE = [        # Chapter 7 — CsrfViewMiddleware lives in this list
"django.middleware.security.SecurityMiddleware",
   "django.contrib.sessions.middleware.SessionMiddleware",
   "django.middleware.csrf.CsrfViewMiddleware",
   "django.contrib.auth.middleware.AuthenticationMiddleware",
   "django.contrib.messages.middleware.MessageMiddleware",
]

TEMPLATES = [{       # Chapter 4 — where DTL looks for app-level templates/ directories
"BACKEND": "django.template.backends.django.DjangoTemplates",
   "APP_DIRS": True,
}]

DATABASES = {        # Chapter 5 — the connection every QuerySet ultimately runs against
"default": {
      "ENGINE": "django.db.backends.sqlite3",
      "NAME": BASE_DIR / "db.sqlite3",
   }
}
```

💻 Coding Challenges

Challenge 1: Register and Customize a Model

Register the `Tag` model (Chapter 6) with a `TagAdmin` that shows `name` in `list_display` and enables searching by `name`.

Goal: Practice the `@admin.register` + `ModelAdmin` customization pattern on a model not already covered in this chapter's examples.

→ Solution

Challenge 2: Add an Inline

Add a `PublisherAdmin` with a `BookInline` so a publisher's books can be added/edited directly from the publisher's own admin page, matching the `AuthorAdmin` pattern from this chapter.

Goal: Practice reusing the same `TabularInline` pattern for a different relationship.

→ Solution

Challenge 3: Audit a Production `settings.py`

Given a `settings.py` with `DEBUG = True` and `ALLOWED_HOSTS = []` deployed to a real production server, explain what's wrong with each setting and what the corrected values should look like.

Goal: Practice recognizing the specific security misconfiguration this chapter's gotcha (and the OWASP course's Security Misconfiguration chapter) warns about.

→ Solution

⚠️ Gotcha: `DEBUG = True` in Production

Django's debug page is genuinely excellent for local development — full stack traces, every local variable's value, the entire request. It is also a severe security exposure if left on in production: an unhandled error on a live site would hand any visitor a detailed view of your source code, settings (including secret keys, unless carefully filtered), and internal application structure. This is precisely the "verbose errors" misconfiguration the OWASP Top 10 course's Security Misconfiguration chapter covered — `DEBUG` must be `False` in production, with `ALLOWED_HOSTS` set to the real domain(s) the site serves (Django refuses to serve requests to unrecognized hosts specifically to prevent HTTP Host header attacks).

🎓 Course Complete

That closes **Django Fundamentals**. Across eight chapters, one contrast ran throughout: where FastAPI and Express hand you a thin core and let you assemble the rest, Django starts from a fully furnished house — routing, an ORM, a templating engine, forms with built-in CSRF protection, and now an entire admin interface, all included from `django-admin startproject` onward. **Course 2 (Intermediate/Advanced)** — authentication, Django REST Framework, testing, Celery, caching, and deployment — is sketched and ready in the course bucket list whenever you want to continue.