DJANGO

Intermediate / Advanced

Authentication · Django REST Framework · Testing

Middleware & Signals · Background Tasks

Caching · Class-Based Views Deep Dive · Deployment

Philip Osztromok

Generated with Claude

Table of Contents

Django Intermediate/Advanced — 8 Chapters

🔐 Authentication

`AuthenticationMiddleware` has been sitting in `MIDDLEWARE` since Chapter 1 of the last course — active the whole time, doing nothing until you actually use it. Where FastAPI and Express typically reach for a JWT library and build the login flow by hand, Django's `django.contrib.auth` ships a complete, battle-tested authentication system: a user model, password hashing, session-based login, and access-control decorators, all included.

The Built-in `User` Model

Django's default `User` model already has the fields most apps need — username, email, and a password field that is **always** stored hashed, never in plain text:

```
from django.contrib.auth.models import User

user = User.objects.create_user(
    username="jane",
    email="jane@example.com",
    password="a-strong-password",
)
# create_user() hashes the password automatically — .password on the
# resulting object is never the plain text, even immediately after creation.
```

`create_user()`, Not `create()`

Bypassing this and using the plain ORM `.create()` (or a `ModelForm` without care) would store the raw password unhashed — always go through `create_user()` or `set_password()`.

Custom User Models

For a production app, swapping in a custom `AUTH_USER_MODEL` from day one (even one that just extends the default) is standard advice — changing it later requires a full migration rebuild.

Password Hashing, Handled For You

The Authentication & Session Security course covered bcrypt/scrypt/Argon2id and *why* passwords must never be stored in plain text. Django's default hasher (PBKDF2) implements that guidance automatically — no manual hashing call required anywhere in your own code:

```
user.set_password("a-new-password")   # hashes and stores it correctly
user.save()

user.check_password("a-new-password")  # True — verifies against the stored hash
```

Switching to Argon2id (the algorithm the security course recommended as the strongest modern default) is a one-line settings change, not a rewrite:

```
# settings.py
PASSWORD_HASHERS = [
    "django.contrib.auth.hashers.Argon2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",  # kept so existing hashes still verify
]
```

🍪 Session-Based Login

Where a typical FastAPI/Express API issues a JWT the client stores and resends, Django's default is session-based — a session ID cookie the server looks up against server-side session data:

```
from django.contrib.auth import authenticate, login, logout
from django.shortcuts import render, redirect

def login_view(request):
    if request.method == "POST":
        username = request.POST.get("username")
        password = request.POST.get("password")
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)  # establishes the session
return redirect("dashboard")
    return render(request, "registration/login.html")

def logout_view(request):
    logout(request)  # destroys the session
return redirect("login")
```

FastAPI/Express JWT Flow vs Django Session Flow

> **Typical FastAPI/Express JWT**
>
> Verify credentials → sign a JWT yourself → client stores it (localStorage or a cookie) → client resends it → you verify the signature on every request.

> **Django Session**
>
> `login()` stores the user's ID server-side and sets a session cookie — `AuthenticationMiddleware` looks it up and attaches `request.user` automatically on every subsequent request.

Notice `login()` does something the Session Security course insisted on doing manually: it rotates the session key on every successful login, preventing session fixation by design — one call, not a step you have to remember.

Protecting Views

Once `login()` has run, every later request in that session has `request.user` populated — `@login_required` is the decorator that turns "must be logged in" into one line:

```
from django.contrib.auth.decorators import login_required

@login_required
def dashboard(request):
    return render(request, "catalog/dashboard.html", {"user": request.user})
# An anonymous request is redirected to LOGIN_URL automatically —
# no manual "if not request.user: redirect" check needed.
```

`request.user.is_authenticated`

A boolean, true for a logged-in user and false for `AnonymousUser` — always present, never `None`, so no null check is needed before reading it.

`@permission_required`

A step further than `@login_required` — checks for a specific permission (`"catalog.add_book"`) rather than just "logged in at all."

💻 Coding Challenges

Challenge 1: Write a Registration View

Write a `register` view that reads `username`, `email`, and `password` from a POST request, creates the user with `User.objects.create_user()`, and logs them in immediately afterward.

Goal: Practice using `create_user()` (never a raw ORM `.create()`) for anything that touches passwords.

→ Solution

Challenge 2: Protect a View

Add `@login_required` to an existing `book_create` view, and explain what happens (in terms of the actual HTTP response) when an anonymous user requests that URL.

Goal: Practice the decorator-based access-control pattern and understand its default redirect behavior.

→ Solution

Challenge 3: Switch to Argon2

Write the `PASSWORD_HASHERS` setting change needed to make Argon2id the primary hashing algorithm for new passwords, while keeping existing PBKDF2-hashed passwords still able to log in.

Goal: Practice a settings-only security upgrade, and explain why the old hasher must stay in the list rather than being removed outright.

→ Solution

⚠️ Gotcha: Don't Roll Your Own Auth

It's tempting, especially coming from a FastAPI project where you built JWT auth by hand, to reach for the same pattern in Django — rolling a custom password check and a hand-written session cookie. Resist it: `django.contrib.auth` already handles password hashing correctly, session fixation prevention, and secure cookie flags, all reviewed by a large security-conscious community over many years. The Authentication & Session Security course's whole thesis — that auth is easy to get subtly wrong — is exactly why using the framework's built-in system, rather than reimplementing it, is the safer default here.

🎯 What's Next

With login working, the next chapter turns to building actual APIs on top of these models: **Django REST Framework** — serializers, viewsets, and how DRF's approach to building an API compares to a plain Django view returning `JsonResponse`.

🔌 Django REST Framework

Chapter 3 of the last course returned `JsonResponse` straight out of a view — fine for one endpoint, repetitive across a whole CRUD API. Django REST Framework (DRF) is Django's answer, the same way FastAPI itself is really "Starlette plus Pydantic bundled together": DRF pairs serializers (validation in, JSON out) with viewsets and routers that generate an entire CRUD API from a handful of lines.

Installing DRF

```
pip install djangorestframework


# settings.py
INSTALLED_APPS = [
    # ...
"rest_framework",
    "catalog",
]
```

Like every app, DRF does nothing until it's registered here — the same `INSTALLED_APPS` discipline from Chapter 1 of the last course.

📦 Serializers: Django's Pydantic Equivalent

A `ModelSerializer` plays exactly the role a Pydantic model plays in FastAPI — it validates incoming data **and** converts model instances to JSON-safe data, generated from the model the same way `ModelForm` generated a form:

```
# catalog/serializers.py
from rest_framework import serializers
from .models import Book

class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ["id", "title", "price", "in_stock"]
```

FastAPI's Pydantic Model vs DRF's Serializer

FastAPI: Pydantic Model

```
class Book(BaseModel):
    id: int
    title: str
    price: float
    in_stock: bool
```

Not tied to a database model — often paired with a separate SQLAlchemy model for the actual table.

DRF: ModelSerializer

```
class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ["id", "title", "price", "in_stock"]
```

Generated directly from the Django model — one less shape to keep manually in sync.

```
# In the Django shell — using a serializer directly
from catalog.serializers import BookSerializer
from catalog.models import Book

book = Book.objects.first()
serializer = BookSerializer(book)
serializer.data  # {'id': 1, 'title': 'Dune', 'price': '12.99', 'in_stock': True}
# Validating incoming data (the "in" direction)
serializer = BookSerializer(data={"title": "1984", "price": "9.99", "in_stock": True})
serializer.is_valid()      # True — same is_valid()/errors pattern as Forms
serializer.save()          # creates and saves a Book, just like ModelForm.save()
```

API Views

The simplest DRF view is a function decorated with `@api_view` — like `@login_required`, it's a decorator adding framework behavior on top of an ordinary function:

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .models import Book
from .serializers import BookSerializer

@api_view(["GET"])
def book_list(request):
    books = Book.objects.all()
    serializer = BookSerializer(books, many=True)
    return Response(serializer.data)
```

🧰 ViewSets & Routers

Writing a separate view for list/create/retrieve/update/destroy is exactly the repetition DRF exists to remove. A `ModelViewSet` implements all five from one class; a `Router` generates the matching `urls.py` entries automatically:

```
# catalog/views.py
from rest_framework import viewsets
from .models import Book
from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer


# catalog/urls.py
from rest_framework.routers import DefaultRouter
from .views import BookViewSet

router = DefaultRouter()
router.register("books", BookViewSet)

urlpatterns = router.urls
# One line generates all five routes:
#   GET    /books/      -> list
#   POST   /books/      -> create
#   GET    /books/{id}/ -> retrieve
#   PUT    /books/{id}/ -> update
#   DELETE /books/{id}/ -> destroy
```

Same Idea as Rails' `resources`

`router.register("books", BookViewSet)` generating 5 routes from one line is directly analogous to Rails' `resources :books` generating 7 — Chapter 2's hand-written `urlpatterns` list was the "before" picture this replaces.

`queryset` + `serializer_class`

The only two things a `ModelViewSet` needs to know — which rows, and how to shape them — everything else (routing to the right action, calling `is_valid()`, calling `save()`) is handled for you.

💻 Coding Challenges

Challenge 1: Write a Serializer

Write an `AuthorSerializer(serializers.ModelSerializer)` for the `Author` model, exposing `id`, `name`, and `bio` (but not `birth_year`), and show the Python code to serialize a single `Author` instance to a dict.

Goal: Practice writing a `ModelSerializer` with an explicit field allowlist.

→ Solution

Challenge 2: Build a Full CRUD API

Using the `AuthorSerializer` from Challenge 1, write an `AuthorViewSet(viewsets.ModelViewSet)` and the router registration that exposes it at `/authors/`.

Goal: Practice the viewset + router pattern that replaces five hand-written views and five hand-written URL patterns.

→ Solution

Challenge 3: Validate Incoming Data

Using a serializer directly (not through a view), demonstrate what `serializer.is_valid()` and `serializer.errors` return for a submitted `price` value that's a negative number, assuming `price` uses DRF's `DecimalField` with a `min_value` validator.

Goal: Practice reasoning about serializer validation independent of any view or HTTP request.

→ Solution

⚠️ Gotcha: `fields = "__all__"` on a Serializer Has the Same Risk as a Form

The mass-assignment danger from the last course's Forms chapter applies identically to `ModelSerializer`: `fields = "__all__"` exposes every model field — including ones like `is_staff` on a `User`-adjacent model — to both reading AND writing through the API. An API is, if anything, a more likely target than an HTML form, since it's trivial to script arbitrary JSON bodies against it. Use an explicit `fields` list on every serializer that touches anything sensitive, exactly as this course has insisted for `ModelForm`.

🎯 What's Next

With a real API in place, the next chapter covers proving it actually works: **Testing Django Apps** — `TestCase`, the Django test client, and fixtures for setting up predictable test data.

🧪 Testing Django Apps

The TypeScript course's Testing chapter argued that types aren't tests — the same is true here: Django's ORM and forms catch shape mistakes, but only a real test proves the actual behavior is correct. Django ships its own test runner, built on Python's `unittest`, with one standout feature most other stacks make you configure yourself: every test's database changes vanish automatically when the test ends.

`TestCase` Basics

A Django test is a class extending `django.test.TestCase`, with test methods starting with `test_` — everything from `unittest` (`assertEqual`, `assertTrue`, …) is available directly:

```
# catalog/tests.py
from django.test import TestCase
from .models import Book, Author

class BookModelTests(TestCase):
    def setUp(self):
        self.author = Author.objects.create(name="Ursula K. Le Guin")

    def test_book_str_returns_title(self):
        book = Book.objects.create(title="The Dispossessed", author=self.author, price=12.99)
        self.assertEqual(str(book), "The Dispossessed")


python manage.py test catalog
# Creating test database for alias 'default'...
# .
# Ran 1 test in 0.012s
# OK
```

🔄 Automatic Per-Test Rollback

The single biggest difference from a typical FastAPI/pytest testing setup: `TestCase` wraps every test method in a database transaction and rolls it back the instant the test finishes — no manual cleanup, no shared state leaking between tests:

Manual Cleanup vs Automatic Rollback

A Typical Manual Setup

Create test data in a fixture/setup step, run the test, then explicitly delete or reset the rows afterward — often in a `teardown` function you have to remember to write and keep correct.

Django's `TestCase`

Every `test_*` method runs inside its own transaction; Django rolls it back automatically when the method returns — objects created in `setUp()` or the test itself are gone before the next test starts, with zero cleanup code.

`setUp()`

Runs before every test method in the class — the natural place for data every test in the class needs, since it's re-created fresh (and rolled back) per test.

Tests Never Pollute Each Other

Because each test's changes roll back, test order never matters and one test's leftover data can never silently affect another's assertions.

The Test Client

`self.client` simulates real HTTP requests against your views — no running server required:

```
class BookViewTests(TestCase):
    def setUp(self):
        self.author = Author.objects.create(name="Ursula K. Le Guin")
        self.book = Book.objects.create(title="The Dispossessed", author=self.author, price=12.99)

    def test_book_list_returns_200(self):
        response = self.client.get("/books/")
        self.assertEqual(response.status_code, 200)

    def test_book_list_shows_book_title(self):
        response = self.client.get("/books/")
        self.assertContains(response, "The Dispossessed")
        self.assertTemplateUsed(response, "catalog/book_list.html")

    def test_book_list_context_contains_books(self):
        response = self.client.get("/books/")
        self.assertIn(self.book, response.context["books"])
```

`response.context`

Direct access to the exact context dict passed to `render()` — a genuinely Django-specific testing convenience, letting you assert on the data a view produced, not just the rendered HTML string.

`assertTemplateUsed`

Confirms which template actually rendered the response — catches a view that returns the right data through the wrong template.

Fixtures

A fixture is a JSON (or YAML) file of pre-defined data, loadable into the test database — useful for larger, shared datasets, though many Django projects prefer building data directly in `setUp()` (as this chapter has done) for anything test-specific and easy to read at a glance:

```
// catalog/fixtures/sample_books.json
[
 {
  "model": "catalog.author",
  "pk": 1,
  "fields": { "name": "Ursula K. Le Guin" }
 },
 {
  "model": "catalog.book",
  "pk": 1,
  "fields": { "title": "The Dispossessed", "author": 1, "price": "12.99" }
 }
]
```

```
class BookFixtureTests(TestCase):
    fixtures = ["sample_books.json"]

    def test_fixture_loaded(self):
        self.assertEqual(Book.objects.count(), 1)
```

Testing Forms Directly

A form (or a DRF serializer, from the last chapter) can be tested without a client or a real request at all:

```
from .forms import BookForm

class BookFormTests(TestCase):
    def test_negative_price_is_invalid(self):
        form = BookForm(data={"title": "Test", "price": -5, "published_date": None})
        self.assertFalse(form.is_valid())
        self.assertIn("price", form.errors)
```

💻 Coding Challenges

Challenge 1: Write a Model Test

Write a `TestCase` for the `Author` model that creates an author in `setUp()` and asserts `str(author)` returns the author's name.

Goal: Practice the basic `setUp()` + `test_*` method structure.

→ Solution

Challenge 2: Write a View Test With the Test Client

Write a test that uses `self.client` to GET a book detail view, asserting the response status is 200, the correct template was used, and the response contains the book's title.

Goal: Practice `self.client.get()`, `assertTemplateUsed`, and `assertContains` together.

→ Solution

Challenge 3: Prove the Rollback Behavior

Write two test methods in the same `TestCase` class: the first creates a `Book` and asserts `Book.objects.count() == 1`; the second (with no `setUp()` creating any books) asserts `Book.objects.count() == 0`. Explain why the second test passes even though it runs after the first.

Goal: Practice reasoning about — and directly observing — the per-test transaction rollback.

→ Solution

⚠️ Gotcha: The Rollback Only Covers the Database

The automatic per-test cleanup is specifically a **database transaction** rollback — it does nothing for Django's cache framework, module-level variables, or any other in-memory state a test might touch. A test that warms the cache (covered in the next-but-one chapter) or mutates a global will leak that state into the next test regardless of `TestCase`'s rollback. Also watch for accidentally subclassing plain `unittest.TestCase` instead of `django.test.TestCase` — the plain version has no database transaction wrapping at all, and a test that touches the ORM under it will leave real, un-rolled-back rows behind for every test that runs afterward.

🎯 What's Next

With a real test suite in place, the next chapter looks at code that runs around every request rather than inside a specific view: **Middleware & Signals** — Django's request/response middleware chain and the signal-dispatch system for decoupled event handling.

🧵 Middleware & Signals

Two Django features answer the same underlying question — "how do I run code without wiring it directly into every view that needs it?" — from opposite directions. Middleware wraps *every* request/response, project-wide, no matter which view handles it. Signals let code react to an event (a model saving, a request finishing) without the code that triggers the event ever knowing who's listening.

The Middleware Chain

`MIDDLEWARE` in `settings.py` (first seen back in Course 1's settings tour) is an ordered list — a request flows through it top-to-bottom, and the response flows back bottom-to-top, the same "onion" model Express's `app.use()` chain uses:

```
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",     # 1st in, last out
"django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware", # sets request.user
"django.contrib.messages.middleware.MessageMiddleware",     # last in, 1st out
]
```

✍️ Writing Custom Middleware

The simplest form is a function that takes `get_response` (the next step in the chain) and returns a function that wraps it — request logic before calling it, response logic after:

```
# catalog/middleware.py
import time
import uuid

def request_timing_middleware(get_response):
    # Runs ONCE, when the server starts — one-time configuration/setup.
def middleware(request):
        # Runs on EVERY request, before the view.
        request.request_id = str(uuid.uuid4())
        start = time.monotonic()

        response = get_response(request)  # calls the next step in the chain (or the view itself)
# Runs on EVERY response, after the view.
        duration_ms = (time.monotonic() - start) * 1000
        print(f"[{request.request_id}] {request.path} took {duration_ms:.1f}ms")
        return response

    return middleware


# settings.py
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    # ...
"catalog.middleware.request_timing_middleware",
]
```

`request.request_id` set here is exactly the correlation ID pattern from the TypeScript course's Logging & Observability chapter — attach one ID per request at the very top of the chain, and every view or log call downstream can reference it.

Express Middleware vs Django Middleware

Express

```
app.use((req, res, next) => {
 req.requestId = crypto.randomUUID();
 const start = Date.now();
 res.on("finish", () => {
  console.log(req.requestId, Date.now() - start);
 });
 next();  // calls the next middleware
});
```

Django

```
def request_timing_middleware(get_response):
  def middleware(request):
    request.request_id = str(uuid.uuid4())
    start = time.monotonic()
    response = get_response(request)  # calls the next middleware
# ... log duration ...
return response
  return middleware
```

Project-Wide, Not Per-View

Unlike `@login_required` (applied per-view), middleware in `MIDDLEWARE` runs on every single request the project handles — the right place for logging, timing, or security headers, not for view-specific access rules.

Class-Based Alternative

Middleware can also be a class with `__init__(self, get_response)` and `__call__(self, request)` — functionally identical, useful when a middleware needs to hold configuration set up once.

📡 Signals: Decoupled Event Notifications

A signal lets one piece of code announce "this happened" without knowing (or caring) who's listening — Django fires several built-in signals automatically, most usefully `post_save` and `post_delete`:

Auto-Creating a Profile When a User Registers

```
# catalog/models.py
from django.db import models
from django.contrib.auth.models import User

class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE, related_name="profile")
    bio = models.TextField(blank=True)

# catalog/signals.py
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.contrib.auth.models import User
from .models import UserProfile

@receiver(post_save, sender=User)
def create_user_profile(sender, instance, created, **kwargs):
    if created:
        UserProfile.objects.create(user=instance)

# catalog/apps.py — signals.py must be imported somewhere, or it never runs
from django.apps import AppConfig

class CatalogConfig(AppConfig):
    default_auto_field = "django.db.models.BigAutoField"
    name = "catalog"
def ready(self):
        import catalog.signals  # noqa
```

`sender=User`

Restricts the receiver to fire only for saves on the `User` model — without it, this function would run for every model's `post_save`, project-wide.

`created`

A boolean passed to every `post_save` receiver — `True` only on the initial `INSERT`, letting this receiver distinguish "a new user just registered" from "an existing user was updated."

💻 Coding Challenges

Challenge 1: Write a Security-Header Middleware

Write a middleware function that adds an `X-Content-Type-Options: nosniff` header to every response, then add it to `MIDDLEWARE` in the correct order relative to Django's own `SecurityMiddleware`.

Goal: Practice the `get_response` closure pattern for a response-modifying (rather than request-modifying) middleware.

→ Solution

Challenge 2: Write a `post_delete` Signal Receiver

Write a signal receiver that logs a message (e.g. `print(f"Book deleted: {instance.title}")`) whenever a `Book` is deleted, including the required `apps.py` wiring to make sure it actually runs.

Goal: Practice the full signal setup, including the easy-to-forget `ready()` import step.

→ Solution

Challenge 3: Middleware vs Signal — Which Fits?

For each of these three requirements, state whether middleware or a signal is the better fit, and why: (a) rejecting all requests from a blocked list of IP addresses, (b) sending a welcome email after a new user registers, (c) adding a `Content-Security-Policy` header to every response.

Goal: Practice recognizing which of the two tools actually matches a given requirement's shape.

→ Solution

⚠️ Gotcha: Signals Are "Spooky Action at a Distance"

A signal receiver connected in some other app's `signals.py` can silently run every time you call `.save()` on a model — with nothing at the call site hinting that anything else happens. This makes signals genuinely hard to trace when debugging: "why did a profile get created here?" sends you hunting across files instead of reading top-to-bottom. Reach for a signal only when the sender genuinely shouldn't know about the receiver (decoupled apps, plugin-style extensibility); for logic within your own app that always needs to happen together, a plain method call or an overridden `save()` is usually clearer. And remember: a `signals.py` file that's never imported from `apps.py`'s `ready()` method never actually connects — the receivers simply don't run, with no error telling you why.

🎯 What's Next

Middleware and signals both run synchronously, inside the request/response cycle. The next chapter covers work that shouldn't block a response at all: **Background Tasks** — Celery basics and task queues for anything too slow to run while a user waits.

⏳ Background Tasks

The TypeScript course's Performance chapter warned about blocking Node's single event loop with slow synchronous work; Django faces the same problem from a different angle — a slow email send or report generation blocks the entire request/response cycle for the one user waiting on it. Celery is Python's standard answer: hand slow work to a separate worker process entirely, and return to the user immediately.

The Architecture: App, Broker, Worker

Celery introduces three separate pieces working together — a genuinely different shape from Node's in-process `worker_threads`:

Your Django App

Calls `my_task.delay(...)` — this doesn't run the task, it just places a message describing the task onto a queue and returns immediately.

The Broker (Redis/RabbitMQ)

A message queue sitting between the app and the workers — it holds pending task messages until a worker is free to pick one up.

Worker Process(es)

One or more separate, long-running processes (often on a different machine entirely) that pull tasks off the broker and actually execute them.

Separate Process, Not Just a Thread

Unlike Node's `worker_threads` (same machine, same process tree), a Celery worker can run anywhere — scale it independently of your web servers entirely.

⚙️ Setting Up Celery

```
pip install celery redis


# mysite/celery.py
import os
from celery import Celery

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "mysite.settings")

app = Celery("mysite")
app.config_from_object("django.conf:settings", namespace="CELERY")
app.autodiscover_tasks()


# catalog/tasks.py
from celery import shared_task
from django.core.mail import send_mail

@shared_task
def send_welcome_email(user_id):
    from django.contrib.auth.models import User
    user = User.objects.get(pk=user_id)  # re-fetch — see this chapter's gotcha
    send_mail("Welcome!", f"Thanks for signing up, {user.username}!", "noreply@example.com", [user.email])
```

Calling a Task: `.delay()`

Blocking vs Backgrounded

Synchronous: Blocks the Response

```
def register(request):
    user = User.objects.create_user(...)
    send_mail("Welcome!", ...)  # the request waits for the SMTP round trip
return redirect("dashboard")
```

Backgrounded: Returns Immediately

```
def register(request):
    user = User.objects.create_user(...)
    send_welcome_email.delay(user.pk)  # queues the task, returns instantly
return redirect("dashboard")
```

`.delay(...)` is shorthand for the more configurable `.apply_async(args=[...])` — reach for `apply_async` directly when you need to set things like a countdown delay or a specific queue.

Upgrading Chapter 4's Signal to Use a Task

The `post_save` welcome-email idea from the last chapter had one problem: a signal receiver runs synchronously, inline with the request that triggered it. Combining it with a task fixes that directly:

```
# catalog/signals.py
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.contrib.auth.models import User
from .tasks import send_welcome_email

@receiver(post_save, sender=User)
def queue_welcome_email(sender, instance, created, **kwargs):
    if created:
        send_welcome_email.delay(instance.pk)  # decoupled AND non-blocking
```

Task Retries

A task can fail — an email provider timing out shouldn't mean the email is lost forever:

```
@shared_task(bind=True, autoretry_for=(ConnectionError,), retry_backoff=True, max_retries=3)
def send_welcome_email(self, user_id):
    from django.contrib.auth.models import User
    user = User.objects.get(pk=user_id)
    send_mail("Welcome!", f"Thanks, {user.username}!", "noreply@example.com", [user.email])
```

`autoretry_for` + `retry_backoff` automatically retries with increasing delays on the listed exception types — the same "don't just fail silently" instinct behind the retry logic in the TypeScript course's Deployment chapter, applied here to a single task instead of a whole deployment.

Periodic Tasks: Celery Beat

Beyond one-off background work, **Celery Beat** schedules a task to run repeatedly — a nightly report, an hourly cache warm-up — configured in `settings.py` rather than a separate cron entry on the server.

💻 Coding Challenges

Challenge 1: Write a Background Task

Write a `generate_book_report(book_id)` task that re-fetches a `Book` by its `pk` and prints a summary line, then call it with `.delay()` from a view instead of running it inline.

Goal: Practice the pass-an-ID-not-an-object pattern for task arguments.

→ Solution

Challenge 2: Add Retry Logic

Extend the task from Challenge 1 to automatically retry up to 3 times with backoff if it raises a `ConnectionError`, using `@shared_task`'s `autoretry_for` and `retry_backoff` options.

Goal: Practice configuring automatic retries instead of letting a transient failure silently drop the task.

→ Solution

Challenge 3: Fix a Task That Passes a Model Instance

Given a broken task defined as `def process_order(order): ...` called as `process_order.delay(order)` (passing the actual `Order` instance), rewrite both the task and the call site to pass `order.pk` instead, and explain what would actually go wrong with the original version.

Goal: Practice recognizing and fixing the model-instance-as-task-argument mistake.

→ Solution

⚠️ Gotcha: Passing a Model Instance Instead of Its ID

Celery task arguments are serialized (typically to JSON) to travel across the broker to a worker — `my_task.delay(book_instance)` either fails outright or silently pickles a stale snapshot of that object as it existed at the moment `.delay()` was called. By the time a worker actually runs the task (seconds, or under load, minutes later), the real database row may have changed or been deleted entirely. Always pass a primary key (`my_task.delay(book.pk)`) and re-fetch the object fresh inside the task. Separately: `.delay()` calls queue silently even with zero workers running — if tasks never seem to execute, check that `celery -A mysite worker` is actually running before assuming the code is broken.

🎯 What's Next

Background tasks handle slow work; the next chapter speeds up *repeated* work instead: **Caching** — Django's cache framework, and per-view and template-fragment caching for data that doesn't need to be recomputed on every single request.

Caching

Chapter 5 offloaded work that was too *slow* to make a user wait for. This chapter is about work that's fast enough to run inline, but wasteful to redo on every single request when the result barely changes — an expensive book list query, a rendered page fragment that's identical for the next thousand visitors. Django's cache framework covers everything from one cached value to an entire cached page.

Cache Backends

Like `DATABASES`, `CACHES` in `settings.py` configures where cached data actually lives:

```
# settings.py
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.redis.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
    }
}
```

Redis / Memcached

The real production choices — an in-memory store separate from Django itself, fast, and shared across every web server process.

`LocMemCache`

A simple in-process cache, fine for local development — but each server process gets its own separate cache, which doesn't reflect production behavior.

`DummyCache`

Implements the same API but never actually stores anything — every `get()` misses. Useful for tests where you deliberately want caching disabled.

🔑 The Low-Level Cache API

```
from django.core.cache import cache

cache.set("featured_book_count", 42, timeout=300)  # cache for 300 seconds
cache.get("featured_book_count")              # 42 — or None if expired/missing
cache.delete("featured_book_count")           # explicit invalidation
```

`get_or_set()` combines the check-then-compute-then-store pattern into one call — the cleanest way to cache an expensive QuerySet result:

Recomputing Every Request vs Caching the Result

Recomputed Every Time

```
def expensive_stats():
    return Book.objects.aggregate(
        avg_price=Avg("price"), total=Count("id")
    )

def stats_view(request):
    stats = expensive_stats()  # runs the aggregate query EVERY request
```

Cached With `get_or_set()`

```
def stats_view(request):
    stats = cache.get_or_set("book_stats", expensive_stats, timeout=600)
    # runs the query ONCE per 10 minutes; every other request within
# that window gets the cached result instantly
```

Per-View Caching: `@cache_page`

The coarsest-grained option — caches an entire view's rendered response:

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)  # cache this view's response for 15 minutes
def book_list(request):
    books = Book.objects.all()
    return render(request, "catalog/book_list.html", {"books": books})
```

Template Fragment Caching

`@cache_page` caches the whole response — `{% cache %}` caches just one part of a template, leaving the rest (a per-user greeting, say) to render fresh every time:

```
{% load cache %}

<h1>Welcome, {{ request.user.username }}!</h1>  <!-- always fresh, never cached -->

{% cache 900 book_list_fragment %}
  <ul>
  {% for book in books %}
    <li>{{ book.title }}</li>
  {% endfor %}
  </ul>
{% endcache %}
```

`900` is the timeout in seconds; `book_list_fragment` is an arbitrary name identifying this cached fragment.

Varying the Cache Key

A page cached for one visitor must not be served to another if the content actually differs per-user — Django's `vary_on_*` decorators fold request details into the cache key itself:

```
from django.views.decorators.vary import vary_on_cookie
from django.views.decorators.cache import cache_page

@vary_on_cookie
@cache_page(60 * 15)
def dashboard(request):
    # now cached SEPARATELY per session cookie — one user's
# cached dashboard is never served to a different user
    ...
```

💻 Coding Challenges

Challenge 1: Cache an Expensive Computation

Write a function `get_catalog_stats()` that aggregates the average price and total count of all books, then use `cache.get_or_set()` to cache the result for 10 minutes inside a view.

Goal: Practice the low-level cache API's `get_or_set()` pattern for an expensive query.

→ Solution

Challenge 2: Cache a Template Fragment

Wrap the book list portion of a template in `{% cache %}` for 5 minutes, while leaving a per-user "Welcome, {{ request.user.username }}" line outside the cached block so it still renders fresh for every request.

Goal: Practice caching only part of a page, not the whole response.

→ Solution

Challenge 3: Invalidate the Cache on Save

Write a `post_save` signal receiver (per Chapter 4's pattern) on the `Book` model that calls `cache.delete("book_stats")` whenever a book is created or updated, so `get_catalog_stats()`'s cached value from Challenge 1 never goes stale for longer than necessary.

Goal: Practice tying cache invalidation to the actual event that makes cached data stale, instead of just waiting out a timeout.

→ Solution

⚠️ Gotcha: Cache Invalidation Is the Hard Part

Caching a book list for 15 minutes means a newly added book — or a price correction — simply won't appear for up to 15 minutes, unless something explicitly clears that cache entry. A timeout alone is a blunt instrument; tying invalidation to the actual event that makes the cache stale (a `post_save` signal calling `cache.delete()`, the same signal pattern from Chapter 4) is what keeps a cache both fast and correct. Also worth knowing: if local development uses `DummyCache` (or never configures a real backend), caching bugs — including a forgotten invalidation — will never surface until the app is running against a real cache backend in production.

🎯 What's Next

With expensive data now cached correctly, the next chapter returns to class-based views from Course 1's introduction and goes much further: **Class-Based Views Deep Dive** — Django's generic views and mixins for building common patterns with even less code.

🏛️ Class-Based Views Deep Dive

Course 1's Views chapter introduced the base `View` class and previewed that `ListView/DetailView` existed. This chapter delivers on that preview — Django's generic views implement entire CRUD patterns from a few class attributes, built from small, composable mixins you can combine, override, and stack with your own.

`ListView`

Hand-Written FBV vs `ListView`

Course 1's Function-Based View

```
def book_list(request):
    books = Book.objects.all()
    return render(request, "catalog/book_list.html", {"books": books})
```

`ListView`

```
from django.views.generic import ListView

class BookListView(ListView):
    model = Book
    template_name = "catalog/book_list.html"
    context_object_name = "books"
    paginate_by = 20
```

`ListView` gives you pagination for free (`paginate_by`) — something the hand-written FBV would need entirely new code to support. Overriding `get_queryset()` customizes what's listed:

```
class InStockBookListView(ListView):
    model = Book
    template_name = "catalog/book_list.html"
    context_object_name = "books"
def get_queryset(self):
        return Book.objects.filter(in_stock=True).select_related("author")
```

`DetailView`

```
from django.views.generic import DetailView

class BookDetailView(DetailView):
    model = Book
    template_name = "catalog/book_detail.html"
    context_object_name = "book"

# catalog/urls.py
path("<int:pk>/", BookDetailView.as_view(), name="book-detail"),
```

`DetailView` reads the primary key from the URL kwarg named `pk` by default (configurable via `pk_url_kwarg`) and calls `get_object_or_404` internally — a missing book returns a proper 404 without writing that check yourself.

✏️ `CreateView`, `UpdateView`, `DeleteView`

These wrap the exact `ModelForm` pattern from Course 1's Forms chapter — `is_valid()`/`save()` handled internally:

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.urls import reverse_lazy

class BookCreateView(CreateView):
    model = Book
    fields = ["title", "description", "price", "author"]
    template_name = "catalog/book_form.html"
    success_url = reverse_lazy("book-list")

class BookUpdateView(UpdateView):
    model = Book
    fields = ["title", "description", "price"]
    template_name = "catalog/book_form.html"
    success_url = reverse_lazy("book-list")

class BookDeleteView(DeleteView):
    model = Book
    template_name = "catalog/book_confirm_delete.html"
    success_url = reverse_lazy("book-list")
```

`fields` here is the same mass-assignment allowlist as `ModelForm.Meta.fields` — the exact same care from the Forms chapter still applies. Overriding `form_valid()` is the standard customization point for anything that needs to happen right when a save succeeds:

```
class BookCreateView(CreateView):
    model = Book
    fields = ["title", "price", "author"]
    template_name = "catalog/book_form.html"
    success_url = reverse_lazy("book-list")

    def form_valid(self, form):
        form.instance.created_by = self.request.user  # set a field before saving
return super().form_valid(form)        # let the parent actually save + redirect
```

Mixins: What Generic Views Are Built From

Every generic view above is itself assembled from small mixins — `SingleObjectMixin`, `MultipleObjectMixin`, `FormMixin` — combined through Python's standard multiple inheritance. You

can add your own mixins to that chain the same way:

`LoginRequiredMixin`

The class-based equivalent of Chapter 1's `@login_required` decorator — added to a generic view's inheritance list, not applied as a decorator.

Method Resolution Order (MRO)

Python resolves multiple inheritance left to right — which mixin comes first in the class definition determines which one's `dispatch()` runs first.

Combining a Mixin With a Generic View

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic.edit import CreateView

class BookCreateView(LoginRequiredMixin, CreateView):
    model = Book
    fields = ["title", "price", "author"]
    template_name = "catalog/book_form.html"
    success_url = reverse_lazy("book-list")
    login_url = "/accounts/login/" # optional — falls back to LOGIN_URL setting
```

💻 Coding Challenges

Challenge 1: Convert an FBV to a `ListView`

Given Course 1's `book_list` function-based view, rewrite it as a `BookListView(ListView)` with pagination set to 10 books per page, and update the corresponding `urls.py` entry.

Goal: Practice the FBV-to-generic-CBV conversion and gain pagination for free.

→ Solution

Challenge 2: Customize `form_valid()`

Write a `ReviewCreateView(CreateView)` for a `Review` model with a `reviewer` foreign key, overriding `form_valid()` to automatically set `form.instance.reviewer = self.request.user` before saving — so the field never needs to appear in the form itself.

Goal: Practice the `form_valid()` override pattern for setting a field the user shouldn't (and can't) submit directly.

→ Solution

Challenge 3: Fix Broken Mixin Order

Given `class BookCreateView(CreateView, LoginRequiredMixin):` (mixin listed second), explain why this doesn't reliably enforce the login requirement, then rewrite it with the correct inheritance order.

Goal: Practice reasoning about Python's MRO in the specific context of Django mixins.

→ Solution

⚠️ Gotcha: Mixin Order Matters

`LoginRequiredMixin` must come **before** the generic view class in the inheritance list — `class BookCreateView(LoginRequiredMixin, CreateView)`, not the reverse. Python's method resolution order checks classes left to right, so `LoginRequiredMixin` needs to appear first to intercept `dispatch()` before `CreateView`'s own dispatch logic runs. Get the order backwards and the access check may not run at all before the view's real logic executes — a classic, quietly broken CBV mistake that "usually works in casual testing" because the symptom is an authorization check silently not firing, not a loud error.

🎯 What's Next

The final chapter of this course is about getting everything built across both courses onto a real server: **Deployment** — WSGI vs ASGI, static file handling, and production settings.

🚀 Deployment

Everything built across both Django courses needs to run somewhere real. This final chapter covers WSGI vs ASGI, why `manage.py runserver` must never touch production, static files (an area Django deliberately does *not* handle efficiently by design), and the environment-driven production settings that tie every prior chapter's loose ends together.

WSGI vs ASGI

Every Django project has generated both entry points since Course 1's `startproject` — this chapter is where they finally matter:

`wsgi.py` — Synchronous

The traditional entry point — one worker handles one request at a time. Fine for the vast majority of Django sites, including everything built in this course.

`asgi.py` — Asynchronous

Needed for async views, WebSockets, or Django Channels — genuinely concurrent request handling, closer to Node's natively async model.

Unlike Node, where async is the default, Django was WSGI-only for over a decade — ASGI and async view support were added later, which is why most existing Django deployments (and everything in this course) still run WSGI.

🖥️ Running Django in Production

`runserver` vs a Real Production Stack

`manage.py runserver`

Single-threaded, auto-reloading, serves static files itself when `DEBUG=True` — built entirely for local development. Django's own documentation explicitly warns against ever using it in production.

Gunicorn + Nginx

Gunicorn is a real WSGI application server — multiple worker processes, no auto-reload, no dev conveniences. **Nginx** sits in front as a reverse proxy, serving static files directly and forwarding dynamic requests to Gunicorn.

```
gunicorn mysite.wsgi:application --bind 0.0.0.0:8000 --workers 3
```

Static Files: `collectstatic`

Django's dev server serves CSS/JS/images automatically under `DEBUG=True` — but that convenience deliberately does **not** carry into production, where a real web server should serve static files instead:

```
# settings.py
STATIC_URL = "/static/"
STATIC_ROOT = BASE_DIR / "staticfiles" # where collectstatic gathers everything into

python manage.py collectstatic
# Copies every app's static/ files (admin CSS, your own catalog/static/, ...)
# into STATIC_ROOT as one combined directory nginx (or WhiteNoise) can serve directly.
```

Nginx Serving `STATIC_ROOT`

The typical production setup — Nginx serves `/static/` directly from disk, never touching Django/Gunicorn for those requests at all.

WhiteNoise (No Nginx Needed)

A Python package letting Gunicorn itself serve static files efficiently — a simpler option for smaller deployments without a separate Nginx layer.

⚙️ Production Settings

Course 1's final chapter flagged `DEBUG` and `ALLOWED_HOSTS` as security-critical — deployment is where every environment-dependent setting needs a real, non-hardcoded source:

Environment-Driven `settings.py`

```
import os

DEBUG = os.environ.get("DJANGO_DEBUG", "False") == "True"
SECRET_KEY = os.environ["DJANGO_SECRET_KEY"]  # raises KeyError immediately if missing — fail fast
ALLOWED_HOSTS = os.environ.get("DJANGO_ALLOWED_HOSTS", "").split(",")

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": os.environ["DB_NAME"],
        "USER": os.environ["DB_USER"],
        "PASSWORD": os.environ["DB_PASSWORD"],
        "HOST": os.environ.get("DB_HOST", "localhost"),
    }
}
```

Compare this to the TypeScript course's Configuration Management chapter — the intent is identical (never hardcode secrets, validate config exists before the app starts), but honestly, Django's approach is looser: `os.environ["DJANGO_SECRET_KEY"]` raises a plain `KeyError` on a missing variable rather than the schema-validated, typed failure a `zod`-based config module produces. Packages like `django-environ` or `python-decouple` add some structure, but nothing in the standard Django toolchain matches that chapter's compile-time-checked config shape.

Django's Built-in Deployment Checklist

A genuinely distinctive feature worth knowing about: Django ships its own automated check for common production misconfigurations:

```
python manage.py check --deploy
# WARNINGS:
# ?: (security.W004) You have not set a value for the SECURE_HSTS_SECONDS setting.
# ?: (security.W008) Your SECURE_SSL_REDIRECT setting is not set to True.
# ?: (security.W018) You should not have DEBUG set to True in deployment.
```

Running this before every deploy catches exactly the class of mistake Course 1's final chapter warned about — and several more — as an explicit, actionable list rather than something discovered after an incident.

💻 Coding Challenges

Challenge 1: Write Environment-Driven Settings

Rewrite a `settings.py` with hardcoded `DEBUG = True`, a hardcoded `SECRET_KEY` string, and an empty `ALLOWED_HOSTS` to instead read all three from environment variables, with `SECRET_KEY` failing fast if it's missing.

Goal: Practice moving hardcoded, environment-dependent values out of source-controlled settings.

→ Solution

Challenge 2: Write a Gunicorn + Static Files Deployment Sequence

Write the ordered shell commands needed to deploy an update to a running Django app: installing dependencies, running migrations, collecting static files, and finally restarting Gunicorn.

Goal: Practice the real deployment sequence, including why `collectstatic` must run before Gunicorn restarts.

→ Solution

Challenge 3: Interpret a `check --deploy` Warning

Given the warning `"security.W018: You should not have DEBUG set to True in deployment,"` explain what setting needs to change and why `check --deploy` is a better safety net than relying on a developer remembering to change it manually before every deploy.

Goal: Practice treating Django's own deployment checklist as a real pre-deploy gate, not just informational output.

→ Solution

⚠️ Gotcha: Forgetting `collectstatic` Before Deploy

Locally, `runserver` under `DEBUG=True` serves every static file automatically — CSS and JS just work, with no extra step. Deploy without running `collectstatic` first, and every one of those files silently 404s in production: pages load, but with no styling and no interactivity, often looking like a completely broken deployment rather than the one missing command it actually is. Equally serious: never commit `SECRET_KEY` to source control — Course 1's Security Misconfiguration warning applies directly here, since a leaked `SECRET_KEY` can be used to forge session cookies and other signed data.

🎓 Course Complete

That closes **Django Intermediate/Advanced** — and with it, the full Django project (16 chapters across both courses). Course 1 built the batteries-included fundamentals against FastAPI and Express; Course 2 layered on everything a real application needs beyond CRUD: authentication built on the framework's own tested system, a full REST API, a real test suite with automatic per-test isolation, decoupled middleware and signals, background work that doesn't block a response, caching with deliberate invalidation, generic class-based views, and finally a real, production-hardened deployment. The same thread ran through both courses: Django's answer to nearly every question is "it's already built in, and it's been reviewed by a large community for years" — the batteries-included philosophy, all the way to the last chapter.