LARAVEL

Intermediate / Advanced

Authentication · API Resources · Testing

Events & Listeners · Queues

Caching · Authorization · Deployment

Philip Osztromok

Generated with Claude

Table of Contents

Laravel Intermediate/Advanced — 8 Chapters

🔐 Authentication

Django's auth system is always there, active from the first request, entirely inside the framework. Laravel takes a genuinely different approach: auth is **opt-in scaffolding** you install via a starter kit — and crucially, it hands you real, editable code rather than hiding the implementation inside the framework.

Laravel Breeze: Scaffolding, Not a Closed Box

```
composer require laravel/breeze --dev
php artisan breeze:install
# Which stack would you like to install?
# > blade / react / vue / api
npm install && npm run build
php artisan migrate
```

`breeze:install` generates actual files into your project — login/register controllers, Blade views, routes — every one of them a normal, editable file sitting in `app/` and `resources/`, not framework internals you're expected to leave alone.

Django's Built-in Auth vs Laravel's Installed Scaffolding

Django

`django.contrib.auth` is always active. Login/logout views, the `User` model, and password hashing all live inside the framework — you use them, but generally don't edit their internals.

Laravel

`breeze:install` generates real controller and view files into *your* codebase — editing `app/Http/Controllers/Auth/RegisteredUserController.php` directly is completely normal and expected.

The `User` Model

Unlike Django (where auth is a separate contrib app), every new Laravel project already ships with `App\Models\User` from `create-project` — auth-readiness is baked in from day one, even before Breeze is installed:

```php
// app/Models/User.php (already present in a fresh Laravel install)
namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    protected $fillable = ['name', 'email', 'password'];

    protected function casts(): array
    {
        return ['password' => 'hashed'];  // automatic hashing on assignment
    }
}
```

`Authenticatable`

The base class that gives `User` everything the auth system needs — password checking, "remember me" tokens — the same role Django's `AbstractUser` plays.

`'password' => 'hashed'` Cast

Setting `$user->password = 'plaintext'` hashes it automatically on save — no separate `create_user()`-style method required the way Django's `User.objects.create_user()` is.

Password Hashing

Laravel's default algorithm is bcrypt, not Django's PBKDF2 — same underlying goal (the Auth & Session Security course's guidance), different default choice:

```php
use Illuminate\Support\Facades\Hash;

Hash::make('plaintext');           // bcrypt hash, by default
Hash::check('plaintext', $user->password);  // verify — same idea as Django's check_password()
```

Switching to Argon2id (the algorithm the security course recommended) is a config change, exactly like Django's `PASSWORD_HASHERS`:

```
// config/hashing.php
'driver' => 'argon2id',
```

🍪 Session-Based Login

Breeze's generated login controller uses the `Auth` facade — conceptually identical to Django's `authenticate()`/`login()`/`logout()`:

```
use Illuminate\Support\Facades\Auth;

public function store(Request $request)
{
    $credentials = $request->validate([
        'email' => 'required|email',
        'password' => 'required',
    ]);

    if (Auth::attempt($credentials)) {
        $request->session()->regenerate();  // prevents session fixation — same as Django's login()
return redirect()->intended('dashboard');
    }

    return back()->withErrors(['email' => 'Invalid credentials.']);
}
```

`$request->session()->regenerate()` is the explicit call Laravel requires for the same session-fixation protection Django's `login()` applies automatically — a real difference worth noticing: Django rotates the session key for you, Breeze's generated code does it as one visible line you own and could (incorrectly) remove.

Protecting Routes: The `auth` Middleware

Chapter 8 of Course 1 covered route-scoped middleware — `auth` is the built-in alias for exactly this:

```
Route::middleware('auth')->group(function () {
    Route::get('/dashboard', [DashboardController::class, 'index'])->name('dashboard');
});
```

This is the same job as Django's `@login_required`, expressed as middleware attached to routes rather than a decorator on view functions — an anonymous request redirects to the login route automatically, same behavior, different mechanism.

🎟️ API Authentication: Sanctum

Session-based auth (via Breeze) is for browser-facing web routes. For APIs, Laravel Sanctum issues tokens instead — a brief preview before Course 2's API Resources chapter goes further:

```php
// Issuing a token (e.g. from a mobile app login endpoint)
$token = $user->createToken('mobile-app')->plainTextToken;

// Protecting an API route with token auth instead of session auth
Route::middleware('auth:sanctum')->get('/api/user', function (Request $request) {
    return $request->user();
});
```

Session Guard vs Sanctum Guard

Laravel's `auth` middleware defaults to the session guard for web routes; `auth:sanctum` switches to token-based auth — one framework, two auth strategies selected per-route.

Comparable to FastAPI/Express JWT

Sanctum tokens play a similar role to the JWTs seen in the FastAPI/Express comparisons throughout this project — a bearer credential the client resends, rather than a server-side session.

💻 Coding Challenges

Challenge 1: Trace a Breeze Login Flow

After installing Breeze in a fresh project, locate the generated `AuthenticatedSessionController`'s `store()` method and identify: where credentials are validated, where `Auth::attempt()` is called, and where the session is regenerated.

Goal: Practice reading generated scaffolding as real, ownable code rather than a black box.

→ Solution

Challenge 2: Protect a Route Group

Wrap an `/orders` route group (from Course 1's middleware challenge) with the `auth` middleware so only logged-in users can reach it, and explain what response an anonymous request receives.

Goal: Practice applying `auth` middleware to a route group, reusing Course 1's grouping pattern.

→ Solution

Challenge 3: Issue and Use a Sanctum Token

Write the code to issue a Sanctum token for a user after a successful API login, and a route protected with `auth:sanctum` that returns the authenticated user's data.

Goal: Practice the token-based auth pattern for an API, separate from session-based web auth.

→ Solution

⚠️ Gotcha: Editing Generated Auth Code Is Normal Here — Unlike Django

A developer coming from Django often hesitates to touch anything Breeze generated, out of a (reasonable, in Django's world) instinct that "framework auth code" shouldn't be edited. In Laravel, the opposite is true: Breeze's generated controllers and views are *yours* the moment they're generated — customizing `RegisteredUserController` to add an extra field, or restyling the login Blade view, is the expected workflow, not a maintenance risk. The trade-off is real too: because you own this code, forgetting something Django would have handled automatically (like the explicit `session()->regenerate()` call above) is now your responsibility to get right, not the framework's.

🎯 What's Next

With login working, the next chapter turns to building actual APIs on top of these models: **API Resources** — Laravel's answer to DRF's serializers, for shaping Eloquent models into JSON responses.

📦 API Resources

A real architectural difference from DRF, worth stating up front: a DRF `ModelSerializer` does double duty — validating incoming data *and* serializing outgoing data, in one class. Laravel splits these into two: `FormRequest` (Course 1, Chapter 7) validates input; **API Resources** handle output only. Neither replaces the other.

Creating a Resource

```
php artisan make:resource BookResource

// app/Http/Resources/BookResource.php
namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class BookResource extends JsonResource
{
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'title' => $this->title,
            'price' => $this->price,
        ];
    }
}
```

```php
// app/Http/Controllers/BookController.php
use App\Http\Resources\BookResource;

public function show(Book $book)
{
    return new BookResource($book);
}

public function index()
{
    return BookResource::collection(Book::all());
}
```

DRF's One Class vs Laravel's Two-Class Split

DRF: One Serializer, Both Directions

```python
class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ["id", "title", "price"]
# Same class validates incoming data AND serializes outgoing data.
```

Laravel: Two Classes, One Direction Each

```php
// StoreBookRequest (Ch.7) — validates INPUT only
// BookResource (this chapter) — shapes OUTPUT only
// Neither class knows about the other.
```

🔀 Conditional Attributes

`whenLoaded()` is a genuinely Resource-specific feature — it includes a relationship only if it was *already* eager-loaded, and never triggers a query of its own:

```
class BookResource extends JsonResource
{
  public function toArray(Request $request): array
  {
    return [
      'id' => $this->id,
      'title' => $this->title,
      'author' => new AuthorResource($this->whenLoaded('author')),
      'is_premium' => $this->when($this->price > 50, true),
    ];
  }
}
```

`whenLoaded('author')`

Includes the `author` key only if `with('author')` was called upstream — omits it entirely (not `null`) if the relationship was never eager loaded.

`when($condition, $value)`

Includes a field only if a condition is true — useful for fields that only make sense in certain states (e.g. an admin-only field, or a computed flag).

Nesting Resources

```
// app/Http/Resources/AuthorResource.php
class AuthorResource extends JsonResource
{
  public function toArray(Request $request): array
  {
    return [
      'id' => $this->id,
      'name' => $this->name,
    ];
  }
}
```

Nesting `AuthorResource` inside `BookResource`'s `toArray()` (as shown above) produces a JSON payload with the author embedded as a sub-object — the same nested-shape idea as a DRF serializer with a nested serializer field.

Resource Collections & Pagination

```
public function index()
{
    return BookResource::collection(Book::paginate(20));
}
// Response automatically includes "data", "links", and "meta"
// (current_page, total, per_page, ...) — no manual pagination wiring needed.
```

⚠️ Resources Don't Prevent N+1 — They Just Format It

A critical, easy-to-miss trap: a `Resource` transforms whatever data is on the model — it does nothing to prevent the N+1 problem from Chapter 6 if the underlying query never eager-loaded the relationship in the first place:

N+1 Through a Resource vs Eager-Loaded First

Still N+1

```
public function index()
{
    return BookResource::collection(Book::all());
    // If BookResource::toArray() accesses $this->author,
// that's still 1 query PER book — the Resource doesn't know or care.
}
```

Fixed at the Query, Not the Resource

```
public function index()
{
    return BookResource::collection(
        Book::with('author')->get()
    );
    // Eager loading happens BEFORE the Resource ever runs.
}
```

`whenLoaded()` is the safety net for this specific trap — using it instead of a direct `$this->author` access means a forgotten `with('author')` silently omits the field rather than silently triggering N+1 queries.

💻 Coding Challenges

Challenge 1: Write a Basic Resource

Write an `AuthorResource` exposing `id`, `name`, and `bio`, and a controller `index()` method returning `AuthorResource::collection(Author::all())`.

Goal: Practice the basic `toArray()` shape and collection usage.

→ Solution

Challenge 2: Nest a Resource With `whenLoaded()`

Extend `BookResource` to include a nested `PublisherResource` using `whenLoaded('publisher')`, and demonstrate the difference in output between a query that eager-loads `publisher` and one that doesn't.

Goal: Practice the safe, N+1-avoiding pattern for including a relationship.

→ Solution

Challenge 3: Fix an N+1 Hiding Behind a Resource

Given a controller `index()` method returning `BookResource::collection(Book::all())` where `BookResource::toArray()` directly accesses `$this->author->name`, identify the N+1 problem and fix it — without changing `BookResource` itself.

Goal: Practice recognizing that the fix belongs at the query level, not inside the Resource.

→ Solution

⚠️ Gotcha: A Resource Formats What's There — It Doesn't Fetch Efficiently

It's tempting to assume a nicely encapsulated `Resource` class handles performance concerns the way it handles shape concerns — it doesn't. `$this->author->name` inside a `toArray()` method triggers a real Eloquent lazy load if `author` wasn't eager-loaded by the query that produced the model in the first place, exactly the same N+1 trap from Chapter 6, just one layer removed and easier to miss because the Resource class itself looks completely correct in isolation. Always check the query feeding a Resource collection, not just the Resource's own code, when investigating N+1 issues on an API endpoint.

🎯 What's Next

With a real API taking shape, the next chapter covers proving it works: **Testing Laravel Apps** — PHPUnit, feature tests, and factories for generating test data.

🧪 Testing Laravel Apps

Laravel tests run on PHPUnit — a real, standard PHP testing framework Laravel didn't invent, just wraps with a helpful base class and a fluent HTTP testing API. One real gap from Django worth flagging immediately: Django's per-test database rollback is automatic by inheriting `TestCase`; Laravel's equivalent is a **trait you have to remember to add**.

Feature Tests vs Unit Tests

Laravel draws an explicit line Django doesn't — `tests/Feature/` exercises the full stack through an HTTP request, `tests/Unit/` tests a single class in isolation with no framework bootstrapping at all:

Feature Test

Simulates a real HTTP request through routing, middleware, controller, and response — Django's `self.client.get()`-style tests fill this same role.

Unit Test

Tests one class or method directly — a plain PHPUnit test with no Laravel application booted, useful for a pure calculation or validator method.

🔄 `RefreshDatabase`: Opt-In, Not Automatic

This is the sharpest difference from Django in this whole course. Django's `TestCase` wraps *every* test in a rollback transaction automatically, just by inheriting it. Laravel requires an explicit trait:

```php
// tests/Feature/BookTest.php
namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class BookTest extends TestCase
{
    use RefreshDatabase; // ⚠️ without this line, tests share a real, un-reset database
public function test_book_list_returns_200(): void
    {
        $response = $this->get('/books');
        $response->assertStatus(200);
    }
}
```

Automatic (Django) vs Opt-In (Laravel)

> **Django**
>
> `class BookTests(TestCase):` — inheriting `TestCase` is the whole story. Every test method is automatically wrapped in a rollback transaction; there's no separate step to remember.

> **Laravel**
>
> `use RefreshDatabase;` must be added explicitly inside the class — extending `Tests\TestCase` alone does **not** reset the database between tests.

The HTTP Testing API

Laravel's fluent testing methods read close to Django's `self.client`, with more built-in assertion helpers:

```php
$response = $this->get('/books');
$response->assertStatus(200);
$response->assertSee('The Dispossessed');
$response->assertViewIs('books.index');

// Testing an authenticated route (Chapter 1's auth middleware)
$response = $this->actingAs($user)->get('/dashboard');
$response->assertStatus(200);

// Testing a JSON API endpoint (Chapter 2's API Resources)
$response = $this->getJson('/api/books');
$response->assertJson(['data' => []]);
```

`actingAs($user)`

Simulates a logged-in session for the test — the direct equivalent of manually calling `self.client.login()` in Django, just one method instead of two steps.

`assertViewIs()`

Confirms which Blade view actually rendered — the same job as Django's `assertTemplateUsed`.

🏭 Model Factories

A genuinely nice built-in feature Django doesn't have a core equivalent for — Django developers commonly reach for a third-party library like `factory_boy`; Laravel ships factories as part of the framework itself:

```
php artisan make:factory BookFactory

// database/factories/BookFactory.php
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

class BookFactory extends Factory
{
    public function definition(): array
    {
        return [
            'title' => fake()->sentence(3),
            'price' => fake()->randomFloat(2, 5, 50),
        ];
    }
}


// In a test:
$book = Book::factory()->create();            // one realistic, saved Book
$books = Book::factory()->count(10)->create(); // ten of them
$book = Book::factory()->make(['price' => 0]); // override one field, don't save
```

💻 Coding Challenges

Challenge 1: Write a Feature Test

Write a feature test class for the book detail route that uses `RefreshDatabase`, creates a book with a factory, and asserts the response is a 200 containing the book's title.

Goal: Practice the full `RefreshDatabase` + factory + HTTP assertion workflow.

→ Solution

Challenge 2: Test an Authenticated Route

Write a test using `actingAs()` to confirm a logged-in user can reach `/dashboard`, and a second test confirming an anonymous request is redirected instead.

Goal: Practice testing both the success and the access-control-denied path for a protected route.

→ Solution

Challenge 3: Diagnose a Test Suite Missing `RefreshDatabase`

Given a test class that extends `Tests\TestCase` but omits `use RefreshDatabase;`, and two tests that each create a `Book` and assert `Book::count()` equals 1 — explain why the second test might fail when run after the first, and fix it.

Goal: Practice recognizing test pollution caused by the missing trait, and understand exactly why it happens.

→ Solution

⚠️ Gotcha: Forgetting `RefreshDatabase`

This is the sharpest version of a pattern this project has built toward — Django simply doesn't let you make this mistake, because the rollback is baked into `TestCase` itself. In Laravel, a test class that extends `Tests\TestCase` but omits `use RefreshDatabase;` runs every test against the **same real database**, with no cleanup between tests at all. Data created in one test persists into the next, test order starts mattering, and a suite that passes in isolation can fail unpredictably when run as a whole — classic flaky-test symptoms, all traceable to one missing line. Add the trait to every feature test that touches the database, as a default habit, not an afterthought.

🎯 What's Next

With a real test suite in place, the next chapter looks at code that reacts to events rather than sitting inline in a controller: **Events & Listeners** — Laravel's decoupled event system compared to Django's signals.

📡 Events & Listeners

Same decoupling goal as Django's signals — code reacts to something happening without the triggering code needing to know who's listening — expressed differently. Laravel uses explicit `Event` and `Listener` classes instead of a `@receiver`-decorated function, and (in modern Laravel) auto-discovers listeners with no manual wiring step at all.

Creating an Event and a Listener

```
php artisan make:event OrderPlaced
php artisan make:listener SendOrderConfirmation --event=OrderPlaced


// app/Events/OrderPlaced.php
namespace App\Events;

use App\Models\Order;
use Illuminate\Foundation\Events\Dispatchable;

class OrderPlaced
{
    use Dispatchable;

    public function __construct(
        public Order $order // a plain, typed data-carrying property
    ) {}
}
```

Compare this to Django's `post_save` receiver, which gets loose `sender`/`instance`/`created` keyword arguments — Laravel's event is a genuine typed object, with exactly the data it declares and nothing implicit:

```php
// app/Listeners/SendOrderConfirmation.php
namespace App\Listeners;

use App\Events\OrderPlaced;
use Illuminate\Support\Facades\Mail;

class SendOrderConfirmation
{
    public function handle(OrderPlaced $event): void
    {
        Mail::to($event->order->customer_email)->send(
            new \App\Mail\OrderConfirmation($event->order)
        );
    }
}
```

🚀 Dispatching: Application-Triggered, Not Automatic

Here's a real difference from Django's `post_save`: a Laravel event fires only where *you* explicitly dispatch it — not automatically on every model save:

```
// app/Http/Controllers/OrderController.php
use App\Events\OrderPlaced;

public function store(StoreOrderRequest $request)
{
    $order = Order::create($request->validated());

    OrderPlaced::dispatch($order);  // fires exactly here, and nowhere else
return redirect()->route('orders.show', $order);
}
```

Django's Automatic Signal vs Laravel's Manual Dispatch

Django

```
@receiver(post_save, sender=Order)
def on_order_saved(sender, instance, created, **kwargs):
    if created:
        # fires automatically, EVERY time an Order is saved —
# including from the admin, a script, a migration...
```

Laravel

```
OrderPlaced::dispatch($order);
// fires ONLY at this exact line — a script or Tinker
// session creating an Order directly never triggers it
```

Laravel does have *model events* (`creating`, `created`, `saving`, `saved`) that fire automatically on Eloquent lifecycle actions — those are the closer parallel to `post_save`, and carry the exact same "fires on every save, no exceptions" trade-off Django's signals do (see this chapter's gotcha).

Registering Listeners: Auto-Discovery

Modern Laravel auto-discovers listeners in `app/Listeners` — no manual registration step, unlike Django's `apps.py` `ready()` import requirement from the Django course:

Auto-Discovery (Laravel 11+)

A listener's `handle()` method type-hint tells Laravel which event it's for — placing the class in `app/Listeners` is enough; nothing extra to wire up.

Manual Registration (Older Laravel)

Pre-11 Laravel required listing the event/listener pair in `EventServiceProvider`'s `$listen` array — the same "forgot to register it" trap as Django's `apps.py` gotcha.

⏳ Queued Listeners

The cleanest built-in integration point in this course so far — implementing one interface moves a listener onto the queue automatically:

Making a Listener Non-Blocking

```
namespace App\Listeners;

use App\Events\OrderPlaced;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Support\Facades\Mail;

class SendOrderConfirmation implements ShouldQueue
{
    public function handle(OrderPlaced $event): void
    {
        Mail::to($event->order->customer_email)->send(
            new \App\Mail\OrderConfirmation($event->order)
        );
    }
}
// implements ShouldQueue — that's the ENTIRE change. No .delay() call,
// no separate task file — the SAME class runs on the queue instead of
// synchronously, unlike the Django course's Celery chapter, which required
// the signal receiver to explicitly call a separate task's .delay() method.
```

💻 Coding Challenges

Challenge 1: Create an Event and Listener Pair

Generate a `UserRegistered` event carrying a `User`, and a `SendWelcomeEmail` listener that sends a welcome email, then dispatch it from a registration controller.

Goal: Practice the full generate-event, generate-listener, dispatch workflow.

→ Solution

Challenge 2: Make a Listener Queued

Take the `SendWelcomeEmail` listener from Challenge 1 and make it non-blocking by implementing `ShouldQueue`, and explain in one sentence what changes about when the email actually gets sent.

Goal: Practice the one-interface change that moves a listener onto the queue.

→ Solution

Challenge 3: Spot the Model Event Trap

Given an Eloquent model with a `static::created()` model event hook that sends a real welcome email, explain what happens when a test suite or a database seeder creates instances of that model, and propose a fix.

Goal: Practice recognizing the gotcha around automatic Eloquent model events firing in unintended contexts.

→ Solution

⚠️ Gotcha: Eloquent Model Events Fire on Every Save — Seeders Included

A model event hook (`static::created()`, or a `creating`/`saving` listener) fires on **every** save that model ever goes through — a factory in a test, a `php artisan db:seed` run populating fake data, a Tinker session, all trigger it exactly the same as a real user action would. This is the same trade-off Django's `post_save` signal carries, and it's a real risk: a model event that sends an actual email or charges a payment can fire unexpectedly while seeding a development database. A manually-dispatched custom event (`OrderPlaced::dispatch($order)`) doesn't have this problem — it only ever fires exactly where you call it — which is one real argument for preferring explicit dispatch over an automatic model event for anything with a real-world side effect.

🎯 What's Next

Queued listeners got a preview of running work outside the request cycle — the next chapter goes deeper: **Queues**, Laravel's queue system in full, compared to the Django course's Celery chapter.

⏳ Queues

Same architecture goal as Celery — hand slow work to a separate worker, return to the user immediately — but Laravel's queue system is built into the framework itself, no separate package required the way Django needs Celery bolted on. This chapter covers `Job` classes directly, going further than Chapter 4's brief `ShouldQueue` preview.

Queue Drivers

```
// .env
QUEUE_CONNECTION=database  // or redis, sqs, sync
```

`database` / `redis`

Real queue backends — jobs sit in a table or Redis list until a worker process picks them up, the same broker role Redis/RabbitMQ play for Celery.

`sync`: No Worker Needed

Runs jobs immediately, in the same process, no queue or worker at all — genuinely convenient for local development. Celery has an "eager mode" for similar behavior, but it's a less first-class, more manual configuration.

🛠️ Creating a Job

```
php artisan make:job GenerateBookReport


// app/Jobs/GenerateBookReport.php
namespace App\Jobs;

use App\Models\Book;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class GenerateBookReport implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    public function __construct(
        public Book $book // the ACTUAL model — see below
    ) {}

    public function handle(): void
    {
        logger()->info("Report: {$this->book->title}, \${$this->book->price}");
    }
}
```

A Genuinely Better Answer to the Django Course's Gotcha

The Django project's Celery chapter had a whole gotcha dedicated to this: *never pass a model instance to a task — pass its ID and re-fetch inside the task*, because the instance gets serialized as a stale snapshot. Laravel's `SerializesModels` trait solves this automatically:

Celery's Manual Discipline vs Laravel's Automatic Handling

Celery (Django Course)

```
# Passing the model directly is a documented anti-pattern —
# the developer MUST remember to pass an ID instead:
@shared_task
def generate_report(book_id):  # NOT book
    book = Book.objects.get(pk=book_id)  # re-fetch by hand
```

Laravel: `SerializesModels`

```
public function __construct(public Book $book) {}
// The trait stores only the model's class + ID when queuing,
// then automatically re-fetches a FRESH instance from the database
// the moment the worker actually runs the job. Nothing to remember.
```

This is a genuine ergonomic win: the exact discipline the Django course had to teach as a rule to memorize is handled automatically by a trait every generated Job already includes.

Dispatching a Job

```
use App\Jobs\GenerateBookReport;

GenerateBookReport::dispatch($book);  // queues it, returns immediately
```

🔁 Retries and Failure Handling

```
class GenerateBookReport implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    public int $tries = 3;
    public int $backoff = 10;  // seconds between retries
public function __construct(public Book $book) {}

    public function handle(): void { /* ... */ }

    public function failed(\Throwable $exception): void
    {
        logger()->error("Report generation failed for book {$this->book->id}: {$exception->getMessage()}");
    }
}
```

Directly parallel to Celery's `autoretry_for`/`retry_backoff`/`max_retries` — same idea, class properties instead of decorator arguments.

Running the Queue Worker

```
php artisan queue:work
# Directly analogous to: celery -A mysite worker
```

💻 Coding Challenges

Challenge 1: Create and Dispatch a Job

Generate a `ProcessOrder` job that takes an `Order` model in its constructor and logs a summary line, then dispatch it from a controller after creating an order.

Goal: Practice the generate-job, dispatch workflow, passing the model directly.

→ Solution

Challenge 2: Add Retry Logic

Extend the `ProcessOrder` job with `$tries = 3` and a `failed()` method that logs the failure, and explain what happens differently compared to a job with no retry configuration at all.

Goal: Practice configuring retries and handling permanent failure.

→ Solution

Challenge 3: Diagnose a Silently Stuck Queue

Given a controller that calls `ProcessOrder::dispatch($order)` but the order confirmation never seems to actually process, list two possible causes covered in this chapter and how to check for each.

Goal: Practice the same "queued but nothing consuming it" debugging instinct from the Django course's Celery chapter.

→ Solution

⚠️ Gotcha: A Missing Worker, or an Accidental `sync` Driver

Two distinct traps, both worth knowing: first, exactly like Celery, `dispatch()` queues a job silently even with zero workers running — if jobs never seem to execute, check that `php artisan queue:work` is actually running before assuming the job code is broken. Second, a Laravel-specific one: local `.env` commonly defaults `QUEUE_CONNECTION` to `sync` for easy local development (no worker needed) — it's easy to forget to switch this to `database` or `redis` before deploying, which means every "queued" job actually runs synchronously in production, silently defeating the entire point and blocking real requests on slow work.

🎯 What's Next

Background work is now handled correctly — the next chapter speeds up repeated work instead: **Caching**, Laravel's cache facade compared to Django's cache framework.

⚡ Caching

Same goal as Django's cache framework — expensive or rarely-changing data shouldn't be recomputed every request. Laravel's `Cache` facade is the closest 1:1 parallel this whole comparison has found: `Cache::remember()` and Django's `cache.get_or_set()` are nearly identical in shape. But this chapter also has real gaps worth being honest about — Django's core ships more caching machinery than Laravel does.

Cache Stores

```
// config/cache.php
'default' => env('CACHE_STORE', 'database'),
```

`redis` / `memcached`

Real production stores — the same role Redis/Memcached play for Django's `CACHES` setting.

`array`: Testing Only

Stores cached values only in memory for the current process — never persists across separate requests, directly parallel to Django's `DummyCache` in intent, though `array` actually caches within one process rather than always missing.

🔑 The `Cache` Facade

```
use Illuminate\Support\Facades\Cache;

Cache::put('featured_book_count', 42, 300);  // cache for 300 seconds
Cache::get('featured_book_count');        // 42 — or null if expired/missing
Cache::forget('featured_book_count');     // explicit invalidation
```

`Cache::remember()`: The Closest 1:1 Parallel Yet

Django's `get_or_set()` vs Laravel's `remember()`

Django

```
stats = cache.get_or_set(
    "book_stats",
    compute_stats,
    timeout=600
)
```

Laravel

```
$stats = Cache::remember(
    'book_stats',
    600,
    fn() => Book::query()->selectRaw('AVG(price) as avg_price, COUNT(*) as total')->first()
);
```

Same three arguments in almost the same order — key, timeout (position differs), and a callback that only runs on a cache miss. Of every comparison in this whole project, this pairing is about as close as two frameworks get.

Where Django's Core Actually Has More

Two honest gaps, not advantages — worth stating plainly rather than always finding a Laravel win:

No Built-in `@cache_page` Equivalent

Django's `@cache_page` decorator caches an entire view's response in one line. Laravel core has no direct equivalent — full-response caching typically needs a package (e.g. `spatie/laravel-responsecache`) or manually wrapping a controller method's return value in `Cache::remember()` yourself.

No Built-in `{% cache %}` Equivalent

Django's `{% cache %}` template tag caches a template fragment directly. Blade has no core directive for this — the same pattern is achieved by wrapping the relevant data (not the rendered HTML) in `Cache::remember()` before passing it to the view.

```
// The Laravel-native way to approximate Django's @cache_page for one controller method:
public function index()
{
    $books = Cache::remember('books.index', 900, fn() => Book::all());
    return view('books.index', ['books' => $books]);
}
// Caches the DATA, not the rendered response — close, but not identical to
// Django's whole-response caching, which also skips re-rendering the template.
```

Cache Invalidation via Model Events

The exact same signal-based invalidation pattern from Django's caching chapter, using the Eloquent model events from Chapter 4:

Invalidating on Save/Delete

```
// app/Models/Book.php
use Illuminate\Support\Facades\Cache;

protected static function booted(): void
{
    static::saved(fn() => Cache::forget('book_stats'));
    static::deleted(fn() => Cache::forget('book_stats'));
}
```

Remember Chapter 4's gotcha: these model events fire on **every** save, including factories and seeders — for cache invalidation specifically, that's actually fine (clearing a cache entry too often is harmless, unlike sending an unwanted email), so model events are a reasonable choice here even though they weren't for side effects like emails.

💻 Coding Challenges

Challenge 1: Cache an Expensive Query

Write a controller method that caches the count and average price of all books under the key `book_stats` for 10 minutes using `Cache::remember()`.

Goal: Practice the `Cache::remember()` pattern, the closest parallel to Django's `get_or_set()` in this course.

→ Solution

Challenge 2: Approximate Full-Response Caching

Write a controller method that caches the entire book list data for 15 minutes, and explain in one sentence how this differs from Django's `@cache_page`, which caches the full rendered response instead of just the data.

Goal: Practice recognizing the gap between Laravel's data-level caching and Django's response-level caching.

→ Solution

Challenge 3: Invalidate the Cache on Save

Add a `booted()` method to the `Book` model that clears the `book_stats` cache key whenever a book is saved or deleted, and explain why this is a safe use of an automatic model event, unlike Chapter 4's welcome-email trap.

Goal: Practice cache invalidation via model events, and articulate why this particular side effect is safe to run on every save.

→ Solution

⚠️ Gotcha: Invalidation Is Still the Hard Part

The same warning from Django's caching chapter applies word-for-word here: forgetting to call `Cache::forget()` (or not tying it to the right model event) means stale data persists until the timeout expires, with no error indicating anything is wrong — it just quietly shows outdated numbers. Separately, a Laravel-specific trap: the `array` cache driver only persists within a single process/request lifecycle — using it in a test to verify caching behavior across what should be two separate requests will misleadingly appear to "not cache at all," when really the driver was never meant to persist across process boundaries in the first place.

🎯 What's Next

With expensive data cached correctly, the next chapter covers a different kind of protection: **Authorization** — Laravel's Policies and Gates, compared to Django's built-in permission system.

🛡️ Authorization

Chapter 1 answered "who are you." This chapter answers "what are you allowed to do" — and here's a genuine Laravel advantage worth stating plainly: Policies are **object-level** by default. Django's built-in permission system is model-level by default ("can change any Book"); checking "can THIS user edit THIS specific book" normally needs a third-party package like `django-guardian` bolted on.

Gates: Simple, Closure-Based Checks

For an authorization rule not tied to a specific model — "can this user view the admin dashboard at all" — a Gate is a plain closure:

```
// app/Providers/AppServiceProvider.php
use Illuminate\Support\Facades\Gate;

public function boot(): void
{
    Gate::define('view-admin-dashboard', function (User $user) {
        return $user->is_admin;
    });
}


if (Gate::allows('view-admin-dashboard')) { /* ... */ }
if ($request->user()->can('view-admin-dashboard')) { /* ... */ }
```

📋 Policies: Object-Level Authorization, By Default

```
php artisan make:policy BookPolicy --model=Book


// app/Policies/BookPolicy.php
namespace App\Policies;

use App\Models\Book;
use App\Models\User;

class BookPolicy
{
    public function update(User $user, Book $book): bool
    {
        return $user->id === $book->author_id;  // THIS user, THIS book — genuinely object-level
    }

    public function delete(User $user, Book $book): bool
    {
        return $user->id === $book->author_id || $user->is_admin;
    }
}
```

Auto-discovered by naming convention — `BookPolicy` pairs with `Book` automatically, no manual registration needed (in older Laravel versions, or if the naming doesn't match, it's registered explicitly in `AppServiceProvider`).

Django's Model-Level Default vs Laravel's Object-Level Default

Django (Built-in)

`user.has_perm("catalog.change_book")` — a coarse, model-wide permission. It answers "can this user edit *some* book," not "can they edit *this specific* book." Per-object checks need `django-guardian`.

Laravel (Built-in)

`BookPolicy::update(User $user, Book $book)` receives the actual `Book` instance — object-level authorization is the default shape, no additional package required.

Using Policies

```
// In a controller
public function update(UpdateBookRequest $request, Book $book)
{
    $this->authorize('update', $book);  // throws 403 automatically if denied
$book->update($request->validated());
    return redirect()->route('books.show', $book);
}
```

Notice `authorize()` — the exact same method name as `FormRequest`'s `authorize(): bool` hook from Chapter 1's Course 1. That hook existed for exactly this purpose all along; it's the natural place to call a Policy check instead of always returning `true`:

```
// app/Http/Requests/UpdateBookRequest.php
public function authorize(): bool
{
    return $this->user()->can('update', $this->route('book'));
}
```

In Blade, `@can`/`@cannot` check the same policy for conditional rendering:

```
@can('update', $book)
  <a href="{{ route('books.edit', $book) }}">Edit</a>
@endcan
```

`viewAny/view/create/update/delete`

The conventional method names Laravel checks for automatically when a Policy is registered — matching, not coincidentally, the same actions `Route::resource()` generates.

Policies Compose With `Gate::before()`

A `Gate::before()` closure can grant a superuser blanket access before any individual Policy method even runs — a common pattern for an `is_admin` bypass.

💻 Coding Challenges

Challenge 1: Write a Policy

Generate an `AuthorPolicy` for the `Author` model with an `update()` method that only allows a user to edit an author record they created (assume an `owner_id` column), and use `$this->authorize()` in a controller.

Goal: Practice writing and applying an object-level authorization check.

→ Solution

Challenge 2: Use a Policy in a Blade View

Write a Blade template that shows an "Edit" link for a book only if the current user is authorized to update it, using the `@can` directive.

Goal: Practice conditional UI rendering based on the same Policy check used server-side.

→ Solution

Challenge 3: Add an Admin Bypass With `Gate::before()`

Add a `Gate::before()` closure in `AppServiceProvider` that automatically grants every permission to a user with `is_admin = true`, and explain why this runs before individual Policy methods rather than needing to be added to each one.

Goal: Practice a global authorization override that composes with per-model Policies.

→ Solution

⚠️ Gotcha: `authorize()` Throws — It Doesn't Return a Boolean

`$this->authorize('update', $book)` throws an `AuthorizationException` (automatically converted to a 403 response) when denied — it does **not** return `false` the way `Gate::allows()` or `$user->can()` do. Writing `if ($this->authorize(...))` is a mistake: the code inside that `if` never runs on denial anyway, since the exception is thrown before `authorize()` would ever return. Use `authorize()` as a standalone statement when you want a hard stop, and `can()`/`Gate::allows()` when you need an actual boolean to branch on. Separately: if a Policy's class name doesn't match its model by convention (a custom name, or a model in a non-standard namespace), it needs explicit registration in `AppServiceProvider`'s policy mapping — the same "forgot to wire it up" trap this course has hit repeatedly.

🎯 What's Next

The final chapter of this course covers getting everything built across both courses onto a real server: **Deployment** — Laravel's production configuration compared to the Django course's deployment chapter.

🚀 Deployment

Everything built across both Laravel courses needs to run somewhere real. This final chapter covers the traditional PHP deployment model, Artisan's production commands, and a full circle back to `.env` from Course 1's very first chapter — the same closing structure the Django course used.

Nginx + PHP-FPM: A Different Shape From Gunicorn

Django's Course 2 covered Gunicorn as a long-running WSGI application server with worker *processes*. PHP's traditional model is genuinely different: **PHP-FPM** (FastCGI Process Manager) spins up a fresh PHP process (or reuses one from a pool) to handle each individual request, then discards its state — PHP has traditionally never been a long-running application process the way a Gunicorn worker is:

Traditional Model: Nginx + PHP-FPM

Nginx serves static files and proxies dynamic requests to PHP-FPM, which boots the Laravel application fresh (or from a pool) for each request — no in-memory state persists between requests by default.

Laravel Octane: The Newer Alternative

Keeps the application booted in memory across requests (via Swoole or RoadRunner), closer to how Gunicorn or Node actually works — a real performance option, but not the traditional default.

⚙️ Artisan's Production Commands

Chapter 8 of Course 1 previewed `config:cache` and `route:cache` — here's the full picture, plus one command that bundles them:

```
php artisan config:cache   # combine config/*.php into one cached file
php artisan route:cache    # cache the compiled route table
php artisan view:cache     # pre-compile every Blade template to plain PHP
# Or, all at once:
php artisan optimize
```

Django's Checklist vs Laravel's Action Command

> **Django:** `check --deploy`
>
> A **diagnostic** tool — it reports warnings about misconfiguration (`DEBUG` left on, missing security settings) but doesn't change anything itself.

> **Laravel:** `optimize`
>
> An **action** tool — it directly performs the caching optimizations (config, routes, views) rather than just warning you they're missing.

Genuinely different philosophies solving adjacent problems: Django tells you what's wrong and lets you fix it; Laravel's `optimize` just does the optimization work directly, no separate "now go fix it" step.

Full Circle: `.env` Revisited

Back to Course 1, Chapter 1 — the same two settings, the same risk, in production this time:

```
# .env — production values
APP_ENV=production
APP_DEBUG=false       # same risk as Django's DEBUG — see this chapter's gotcha
APP_KEY=base64:...    # must be set — Chapter 1's key:generate gotcha, still true in production

DB_CONNECTION=mysql
DB_HOST=your-production-db-host
QUEUE_CONNECTION=redis # NOT sync — Chapter 5's gotcha, still true here
```

👷 Queue Workers Need a Process Supervisor

`php artisan queue:work` from Chapter 5 is a long-running process — the same requirement Django's Gunicorn setup has, just for a different piece:

```
# /etc/supervisor/conf.d/laravel-worker.conf
[program:laravel-worker]
command=php /var/www/example-app/artisan queue:work --sleep=3 --tries=3
autostart=true
autorestart=true
numprocs=2
```

Supervisor (or systemd) keeps the worker running and restarts it automatically if it crashes — directly parallel to how Django's Gunicorn process needs the same kind of process supervision to survive a server reboot or a crash.

Maintenance Mode: `artisan down/up`

A genuinely nice built-in feature worth introducing fresh here — putting the whole site into a maintenance page is one command:

```
php artisan down --render="errors::503"
# serves a maintenance page immediately, for every visitor
# ... run migrations, deploy new code ...

php artisan up
# site is live again
```

A Full Deployment Sequence

Putting It All Together

```
php artisan down --render="errors::503"

git pull origin main
composer install --no-dev --optimize-autoloader
php artisan migrate --force
php artisan optimize

php artisan queue:restart   # see this chapter's gotcha — critical, easy to forget

php artisan up
```

💻 Coding Challenges

Challenge 1: Write Production `.env` Values

Given a `.env` with `APP_DEBUG=true`, `QUEUE_CONNECTION=sync`, and no `APP_KEY`, write the corrected production values and the command needed to generate a missing key.

Goal: Practice recognizing every environment-config risk covered across both courses in one place.

→ Solution

Challenge 2: Write a Deployment Script

Write the ordered shell commands for deploying an update: enabling maintenance mode, pulling code, installing dependencies, migrating, optimizing, restarting queue workers, then disabling maintenance mode.

Goal: Practice the full deployment sequence and reasoning about the correct order.

→ Solution

Challenge 3: Diagnose Stale Queue Worker Code

A deploy changes a Job's `handle()` method, but jobs processed after the deploy still run the OLD logic. Explain why, and what command fixes it.

Goal: Practice recognizing a deployment gotcha specific to long-running queue workers.

→ Solution

⚠️ Gotcha: Forgetting `queue:restart` After a Deploy

This one is specific to queue systems and has no real Django parallel, since Django doesn't run persistent worker processes tied to a single deploy command the way Laravel's queue workers do. A `queue:work` process loads your application code (including every Job class) into memory **once**, when it starts, and keeps running indefinitely — deploying new code to disk does **not** make a running worker pick it up. A worker that's been running since before the deploy keeps executing the *old* version of `handle()` for every job it processes, silently, with no error indicating the code is stale. `php artisan queue:restart` signals every worker to finish its current job and exit gracefully — your process supervisor (Supervisor/systemd) then restarts them fresh, picking up the new code. This is exactly as easy to forget as `APP_DEBUG=false`, just far less obvious when it happens, since everything appears to work — it's just running last week's logic.

🎓 Course Complete

That closes **Laravel Intermediate/Advanced** — and with it, the full Laravel project (16 chapters across both courses). Course 1 built the batteries-included fundamentals against FastAPI, Express, and Django; Course 2 layered on everything a real application needs beyond CRUD: authentication built on ownable, editable scaffolding rather than a closed framework box, a real API layer split cleanly between input validation and output shaping, a test suite whose safety net you have to opt into rather than inherit for free, decoupled events with a genuinely elegant path to background processing, a built-in queue system that solved the Django course's Celery gotcha automatically, object-level authorization by default, and finally a production deployment with its own genuinely PHP-specific traps. Across both courses, Laravel's own personality came through clearly: batteries included, but handed to you as code you're expected to read, own, and edit — not a black box.