LARAVEL

Fundamentals

Getting Started · Routing · Controllers

Blade Templates · Eloquent ORM

Eloquent Relationships · Form Requests & Validation

Middleware & Artisan

Philip Osztromok

Generated with Claude

Table of Contents

Laravel Fundamentals — 8 Chapters

🎼 Getting Started

Laravel occupies roughly the same "batteries-included" niche as Django — a full ORM, templating engine, routing, and CLI tooling all included from the first command — but built on PHP with its own ecosystem and conventions. This chapter gets a project running and maps Laravel's structure onto what you already know from Django.

Composer, Not Just PHP

Where a plain PHP script needs nothing more than a web server, Laravel is installed and managed through **Composer**, PHP's dependency manager — the same role `pip` plays for Django:

```
composer create-project laravel/laravel example-app
cd example-app
php artisan serve
# INFO  Server running on [http://127.0.0.1:8000].
```

Django's `startproject` vs Laravel's `create-project`

Django

```
pip install django
django-admin startproject mysite
cd mysite
python manage.py runserver
```

Laravel

```
composer create-project laravel/laravel example-app
cd example-app
php artisan serve
```

📁 Project Structure

Laravel's folder layout maps closely onto Django's app structure, even though the names differ:

```
example-app/
├── artisan            # the CLI entry point — Laravel's manage.py
├── .env               # environment config (APP_KEY, DB credentials, ...)
├── routes/
│   └── web.php        # URL routing table — Laravel's urls.py
├── app/
│   ├── Http/
│   │   └── Controllers/   # request-handling logic — Laravel's views.py
│   └── Models/        # Eloquent models — Laravel's models.py
├── resources/
│   └── views/         # Blade templates — Laravel's templates/ directory
├── database/
│   └── migrations/    # schema change history — same concept as Django's migrations/
└── config/            # split-by-topic settings — Laravel's version of settings.py
```

One Project, Not Project + Apps

Laravel doesn't split "project" from "app" the way Django does — everything lives under one `app/` directory, organized by role (Controllers, Models) rather than by feature area.

`config/` Instead of One `settings.py`

Configuration is split across multiple files by topic (`config/database.php`, `config/app.php`, ...) rather than one monolithic settings file — each reads its actual values from `.env`.

Artisan: Laravel's `manage.py`

Every Laravel-specific command runs through `artisan` — migrations, code generation, the dev server, all in one place:

Common Commands, Side by Side

Django (`manage.py`)

```
python manage.py runserver
python manage.py migrate
python manage.py startapp blog
python manage.py shell
```

Laravel (`artisan`)

```
php artisan serve
php artisan migrate
php artisan make:controller BlogController
php artisan tinker
```

`php artisan make:*`

A whole family of code-generation commands — `make:controller`, `make:model`, `make:migration` — scaffolding boilerplate the way Django's `startapp` does, just at a finer grain.

`php artisan tinker`

An interactive REPL with the full application already loaded — directly equivalent to Django's `manage.py shell`.

The `.env` File

Laravel bakes environment-based configuration in from the very first command — no separate setup step needed the way Django's Course 2 Deployment chapter had to introduce it:

```
# .env
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:GENERATED_ON_INSTALL
APP_DEBUG=true

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_DATABASE=example_app
DB_USERNAME=root
DB_PASSWORD=
```

`APP_KEY` is generated automatically as part of `create-project` — it's used to encrypt sessions and other sensitive data, and Laravel refuses to run correctly without one (see this chapter's gotcha).

💻 Coding Challenges

Challenge 1: Create Your First Project

Run `composer create-project laravel/laravel library`, then start the dev server and confirm the welcome page loads. List the top-level folders it generated and, in one sentence each, what each one is for.

Goal: Get comfortable with the project structure before writing any actual code.

→ Solution

Challenge 2: Explore Artisan

Run `php artisan list` and pick three commands you haven't used yet. For each, write one sentence describing what it does and when you'd reach for it.

Goal: Build familiarity with Artisan as the single entry point for nearly every Laravel task — the same habit Django's Course 1 built around `manage.py help`.

→ Solution

Challenge 3: Inspect the `.env` File

Open a newly created project's `.env` file and identify: the database connection settings, the debug flag, and the app key. Explain in one sentence what would break if `APP_KEY` were left empty.

Goal: Practice reading environment-based configuration before it becomes relevant to real settings decisions later in the course.

→ Solution

⚠️ Gotcha: A Missing `APP_KEY`

`composer create-project` generates `APP_KEY` automatically — but if a project is ever cloned from a repository where `.env` wasn't committed (correctly — `.env` should never be committed, same as any language's secrets file), the new copy has no key at all. Running the app in that state produces a very literal `RuntimeException: No application encryption key has been specified` the moment anything tries to use sessions or encrypted data. The fix is `php artisan key:generate`, which creates a fresh key and writes it into `.env` — a one-time step worth knowing about the first time it happens, since the error message doesn't obviously point at ".env is missing a value."

🎯 What's Next

With a project running, the next chapter covers how a request finds its way to your code: **Routing** — `routes/web.php`, `Route::get()`, route parameters, and how Laravel's routing compares to Django's `urls.py`.

🧭 Routing

`routes/web.php` is Laravel's single routing table — the same concept as Django's `urls.py`, expressed as fluent, chainable method calls instead of a list of `path()` entries. This chapter covers basic routes, parameters, named routes, groups, and the single line that replaces an entire CRUD route set.

Basic Routes

A route maps an HTTP verb and URI to a closure or, more commonly, a controller method (covered next chapter):

```
// routes/web.php
use Illuminate\Support\Facades\Route;

Route::get('/books', function () {
    return 'All books';
});
```

🔢 Route Parameters

Curly braces capture a URL segment — directly analogous to Django's angle-bracket converters, just without a built-in type conversion syntax of its own:

```
Route::get('/books/{id}', function (string $id) {
    return "Showing book #{$id}";
});

// Optional parameter — note the ?, and a default value for the closure argument
Route::get('/books/{id?}', function (string $id = 'latest') {
    return "Showing book: {$id}";
});

// Constrain the parameter to digits only, using ->where()
Route::get('/books/{id}', function (string $id) {
    return "Showing book #{$id}";
})->where('id', '[0-9]+');
```

Django's Converters vs Laravel's `where()`

Django

```
path("<int:book_id>/", views.book_detail)
# the <int:...> converter validates AND type-converts
```

Laravel

```
Route::get('/{id}', ...)->where('id', '[0-9]+');
// a regex constraint validates the FORMAT — the value still
// arrives as a string; there's no built-in type conversion
```

Named Routes

The exact same idea as Django's `name=` + `reverse()` — never hardcode a URL that a named route already describes:

```
Route::get('/books/{id}', [BookController::class, 'show'])->name('books.show');


// In a Blade template (Chapter 4) or PHP code:
<a href="{{ route('books.show', ['id' => 42]) }}">View Book</a>
// -> "/books/42" — built from the route definition, not a hand-typed string
```

📦 Route Groups

Laravel doesn't split routes across separate per-app files the way Django's `include()` does — instead, `Route::prefix()` and `->group()` apply a shared prefix (and, commonly, shared middleware) to a block of routes within the same file:

```
Route::prefix('admin')->group(function () {
    Route::get('/books', [AdminBookController::class, 'index']);
    Route::get('/orders', [AdminOrderController::class, 'index']);
});
// Both routes are now reachable under /admin/books and /admin/orders
```

Grouping, Not Separate Files

A large Laravel app *can* split routes across multiple files (`require`'d into `web.php`), but there's no built-in per-app routing file the way every Django app gets its own `urls.py`.

Groups Also Share Middleware

`Route::middleware('auth')->group(...)` applies an auth check to every route inside — Course 2 covers middleware and auth in depth.

Resource Routes: One Line, Seven Routes

This is Laravel's version of a pattern this project has seen twice already — Rails' `resources :books` and Django REST Framework's `router.register()` — one line generating a full set of RESTful routes:

```
Route::resource('books', BookController::class);
// Generates all seven RESTful routes at once:
//   GET    /books          -> index
//   GET    /books/create   -> create
//   POST   /books          -> store
//   GET    /books/{book}   -> show
//   GET    /books/{book}/edit -> edit
//   PUT    /books/{book}   -> update
//   DELETE /books/{book}   -> destroy
```

Run `php artisan route:list` (from the last chapter's challenge) after adding a resource route to see exactly what it expanded into.

💻 Coding Challenges

Challenge 1: Write Basic Routes With Parameters

Write three routes in `routes/web.php`: a book list at `/books`, a book detail at `/books/{id}` constrained to digits only, and a category listing at `/books/category/{slug}` — each with a sensible `name()`.

Goal: Practice route parameters, `where()` constraints, and naming routes from the start.

→ Solution

Challenge 2: Group Routes Under a Prefix

Group the three routes from Challenge 1 under an `/api` prefix using `Route::prefix()->group()`, and state the final full path for each.

Goal: Practice route grouping and reasoning about the combined prefix + pattern.

→ Solution

Challenge 3: Replace Manual Routes With `Route::resource()`

Given a hand-written set of five separate `Route::get()`/`post()`/`put()`/`delete()` calls for a `Product` controller, replace them with a single `Route::resource()` call and list the exact route names it generates.

Goal: Practice recognizing when a hand-written route set is really just a resource route in disguise.

→ Solution

⚠️ Gotcha: Route Order Matters

Laravel matches routes top-to-bottom, same as Django — the first pattern that matches wins, even if a later one would have been a better fit. Defining `Route::get('/books/{id}', ...)` **before** `Route::get('/books/create', ...)` means visiting `/books/create` actually matches the `{id}` route first, with `$id` literally set to the string `"create"` — the more specific static route never gets a chance to run. Always define specific, static routes (like `/books/create`) *before* parameterized ones that could shadow them (like `/books/{id}`).

🎯 What's Next

Routes now point somewhere — the next chapter covers what they point *to*: **Controllers**, `php artisan make:controller`, resource controllers, and how Laravel's controllers compare to Django's views.

🎮 Controllers

The closures in `routes/web.php` from the last chapter work fine for a one-line example — real request-handling logic belongs in a controller, Laravel's direct equivalent of Django's `views.py`. This chapter covers generating controllers, resource controllers matching last chapter's `Route::resource()`, and route model binding — a genuinely distinctive Laravel convenience.

Generating a Controller

```
php artisan make:controller BookController
# Controller created successfully.
# app/Http/Controllers/BookController.php

// app/Http/Controllers/BookController.php
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class BookController extends Controller
{
    public function index()
    {
        return 'All books';
    }
}


// routes/web.php — wire the controller into a route
Route::get('/books', [BookController::class, 'index']);
```

Controller Methods vs Django Views

Django Function-Based View vs Laravel Controller Method

Django

```
def book_detail(request, book_id):
    query = request.GET.get("q", "")
    return HttpResponse(...)
```

`request` always arrives as the first positional argument.

Laravel

```
public function show(Request $request, string $id)
{
    $query = $request->query('q', '');
    return ...;
}
```

`Request $request` is *type-hinted* — Laravel's service container injects it automatically based on the type, not its position.

This type-hint-based injection is a real PHP/Laravel-specific idea worth noticing early: any parameter Laravel recognizes by its type (a `Request`, a model class, a custom service) gets resolved and passed in automatically — order relative to route parameters doesn't matter the way Python's positional arguments require.

📤 Returning Responses

```
public function index()
{
    return view('books.index', ['books' => Book::all()]);  // Blade — Chapter 4
}

public function apiIndex()
{
    return response()->json(['books' => Book::all()]);
}

public function store(Request $request)
{
    // ... save ...
return redirect()->route('books.index');  // by named route, per Chapter 2
}
```

`view()`

Combines a Blade template + data array + response into one call — the direct parallel to Django's `render()`.

`response()->json()`

The equivalent of Django's `JsonResponse` — wraps a PHP array as a proper JSON response with the correct content type.

Resource Controllers

`--resource` generates method stubs for exactly the seven actions `Route::resource()` expects — the controller-side half of last chapter's route generation:

```
php artisan make:controller BookController --resource
```

```
class BookController extends Controller
{
    public function index() { /* GET /books */ }
    public function create() { /* GET /books/create */ }
    public function store(Request $request) { /* POST /books */ }
    public function show(Book $book) { /* GET /books/{book} */ }
    public function edit(Book $book) { /* GET /books/{book}/edit */ }
    public function update(Request $request, Book $book) { /* PUT /books/{book} */ }
    public function destroy(Book $book) { /* DELETE /books/{book} */ }
}
```

Wiring it up needs only one route line, exactly matching Chapter 2:

```
Route::resource('books', BookController::class);
```

🔗 Route Model Binding

Notice `show(Book $book)` above, not `show(string $id)` — this is Laravel's standout convenience. Type-hint a model class as a parameter whose name matches the route segment, and Laravel fetches the row for you automatically:

Manual Lookup vs Route Model Binding

Django: Manual Lookup

```
def book_detail(request, book_id):
    book = get_object_or_404(Book, pk=book_id)
    return render(request, "book_detail.html", {"book": book})
```

Laravel: Automatic Binding

```
public function show(Book $book)
{
    return view('books.show', ['book' => $book]);
}
// $book is ALREADY the fetched model — no findOrFail() call written by hand
```

Laravel matches the route parameter `{book}` to the type-hinted `Book $book` by name, runs `Book::findOrFail($id)` internally, and injects the actual model instance — a missing ID produces an automatic 404, with zero lookup code in the controller at all.

💻 Coding Challenges

Challenge 1: Generate and Wire a Controller

Run `php artisan make:controller AuthorController`, add an `index()` method returning a plain string, and write the `routes/web.php` line needed to actually reach it at `/authors`.

Goal: Practice the generate-then-wire-up sequence, including the step that's easy to forget.

→ Solution

Challenge 2: Write a Resource Controller

Generate a resource controller for a `Product` model with `--resource`, fill in `index()` and `show(Product $product)` to return simple placeholder responses, and wire it up with `Route::resource()`.

Goal: Practice pairing a resource controller with its matching resource route from Chapter 2.

→ Solution

Challenge 3: Convert Manual Lookup to Route Model Binding

Given a controller method `show(string $id)` that manually calls `Book::findOrFail($id)`, rewrite it to use route model binding instead — `show(Book $book)` — and explain what happens differently if the ID doesn't exist.

Goal: Practice recognizing when manual lookup code can be replaced by Laravel's automatic binding.

→ Solution

⚠️ Gotcha: Generating a Controller Doesn't Wire It Up

`php artisan make:controller` only creates the file — nothing routes to it until you add a matching entry in `routes/web.php`. This is the same two-step trap already seen twice in the Django courses (`INSTALLED_APPS`, `include()`): the code exists on disk, but the framework doesn't know about it yet. Separately, route model binding's automatic 404 is usually exactly what you want — but it means a `Book $book` parameter can never actually be `null` inside the method; if you need to handle "not found" yourself instead of letting Laravel's default 404 page render, look up the model manually with `Book::find($id)` instead of relying on binding.

🎯 What's Next

Controllers can now return a view — the next chapter covers what's actually inside one: **Blade Templates**, Laravel's templating syntax, control structures, and template inheritance compared to the Django Template Language.

⚡ Blade Templates

`view()` from the last chapter needs an actual template file. Blade is Laravel's templating engine — same job as the Django Template Language, different punctuation: `@if` instead of `{% if %}`, and directives compiled straight down to plain PHP rather than DTL's intentionally restricted mini-language.

Where Blade Views Live

Every Blade file lives in `resources/views/`, with dot notation mapping to nested folders — a flatter convention than Django's app-namespaced `templates/appname/` directories:

```
resources/views/
├── layouts/
│   └── app.blade.php
├── books/
│   ├── index.blade.php
│   └── show.blade.php


// In a controller (Chapter 3):
return view('books.index', ['books' => Book::all()]);
// 'books.index' -> resources/views/books/index.blade.php
```

🔤 Variables & Echoing

`{{ }}` auto-escapes by default — the same XSS defense Django's `{{ }}` provides, just with curly braces instead of DTL's own:

```
<h1>{{ $book->title }}</h1>

// If $book->title contains <script>, this renders as inert text,
// not an executable script tag — identical protection to Django's auto-escaping.
```

`{!! !!}` is Blade's equivalent of Django's `|safe` filter — it opts *out* of escaping entirely:

```
{!! $book->trusted_html_description !!}
// ⚠️ NEVER on unsanitized user input — same warning as Django's |safe,
// direct callback to the XSS course
```

Control Structures

```
@if ($books->isEmpty())
  <p>No books yet.</p>
@else
  <ul>
  @foreach ($books as $book)
    <li>{{ $loop->iteration }}. {{ $book->title }}</li>
  @endforeach
  </ul>
@endif
```

`$loop` is Blade's automatic loop variable — `$loop->iteration`, `$loop->first`, `$loop->last` — the direct equivalent of Django's `forloop.counter` and friends, available inside any `@foreach` without being passed in from the controller.

Django Template Language vs Blade — Same Job, Different Punctuation

Django Template Language

```
{% if books %}
 {% for book in books %}
  {{ book.title|upper }}
 {% endfor %}
{% else %}
 No books.
{% endif %}
```

Blade

```
@if($books->isNotEmpty())
 @foreach($books as $book)
  {{ strtoupper($book->title) }}
 @endforeach
@else
 No books.
@endif
```

🧩 Template Inheritance

Blade's version of `extends`/`block` uses `@extends`/`@section`/`@yield` instead:

A Layout and a Child View

```
<!-- resources/views/layouts/app.blade.php -->
<!DOCTYPE html>
<html>
<head>
 <title>@yield('title', 'My Library')</title>
</head>
<body>
 <nav>Home | Books</nav>
 @yield('content')
 <footer>&copy; 2026</footer>
</body>
</html>

<!-- resources/views/books/index.blade.php -->
@extends('layouts.app')

@section('title', 'All Books')

@section('content')
 <ul>
 @foreach ($books as $book)
  <li>{{ $book->title }}</li>
 @endforeach
 </ul>
@endsection
```

`@yield` vs `@section`/`@endsection`

The layout declares a *placeholder* with `@yield`; a child view *fills* it with `@section`/`@endsection` — Django's `{% block %}` plays both roles in one tag, Blade splits it into two.

`@include`

Pulls in a reusable partial inline — `@include('books.book-card')` — the same job as Django's `{% include %}`.

A Real Philosophical Difference

Course 1's Django Templates chapter emphasized DTL's deliberate restriction — no method calls with arguments, no arbitrary Python. Blade takes the opposite stance: it compiles directly to plain PHP, so `@php` blocks (and even calling PHP functions inline, as `strtoupper($book->title)` did above) are fully available:

```
@php
    $discountedPrice = $book->price * 0.9;
@endphp

<p>Sale price: {{ number_format($discountedPrice, 2) }}</p>
```

This is genuinely more powerful and genuinely easier to misuse — DTL's restriction exists specifically to keep business logic out of templates; Blade trusts the developer to maintain that discipline manually.

💻 Coding Challenges

Challenge 1: Render a List With Blade

Write a Blade view that loops over a `$books` variable, printing each title using `$loop->iteration` for numbering, with a fallback message if the collection is empty.

Goal: Practice `@if`/`@foreach` together with the `$loop` variable.

→ Solution

Challenge 2: Build a Layout + Two Child Views

Write a `layouts/app.blade.php` with `@yield('title')` and `@yield('content')`, then two child views (`books/index.blade.php` and `books/show.blade.php`) that each `@extends` it with different content.

Goal: Practice the `@extends`/`@section`/`@yield` inheritance pattern across more than one page.

→ Solution

Challenge 3: Demonstrate Escaping

Pass a string containing `<script>` tags into a Blade view, render it two ways — once as `{{ $value }}` and once as `{!! $value !!}` — and explain what each one actually sends to the browser.

Goal: Practice recognizing Blade's auto-escaping in action, the same way Django's Templates chapter did for DTL.

→ Solution

⚠️ Gotcha: `{!! !!}` on User Input

The exact same warning from Django's `|safe` filter applies here, just with Blade's syntax: `{!! $comment->text !!}` on anything that ultimately traces back to user input is a direct XSS vulnerability, since it disables the one built-in protection Blade provides by default. Reserve `{!! !!}` for content that's already been through a real sanitization step — never as a quick fix for an escaping-related rendering issue.

🎯 What's Next

Views can now display data — the next chapter covers where that data actually comes from: **Eloquent ORM**, defining models, migrations, and querying with Eloquent's query builder compared to Django's ORM.

🗂️ Eloquent ORM

Eloquent is Laravel's ORM, included the same "batteries first" way Django's ORM is — but one real difference stands out immediately: Django's migrations are auto-generated by diffing your models, while Laravel's are hand-written PHP files, both the "up" and the "down" direction, from the start.

Defining a Model

```
php artisan make:model Book -m
# Model created successfully.
# Created Migration: 2026_07_05_120000_create_books_table.php
```

The `-m` flag generates a matching migration in the same step — Django keeps model definition and `makemigrations` as two separate commands; Laravel bundles them:

```
// app/Models/Book.php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    protected $fillable = ['title', 'description', 'price'];
}
```

Convention Over Configuration

Eloquent infers the table name (`books`, pluralized snake_case of `Book`) and primary key (`id`) automatically — no explicit declaration needed unless you deviate from convention.

`-m`: One Command, Two Files

A convenience Django doesn't offer directly — creating a model and its first migration together, rather than running `make:model` then a separate migration step.

🔀 Migrations: Hand-Written, Not Auto-Generated

This is the single biggest ORM difference from Django worth calling out explicitly. Django's Course 1 `makemigrations` diffs your models against migration history and writes the file for you. Laravel migrations are plain PHP files *you* write, defining both directions yourself:

```php
// database/migrations/2026_07_05_120000_create_books_table.php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    public function up(): void
    {
        Schema::create('books', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('description')->nullable();
            $table->decimal('price', 6, 2);
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('books');
    }
};
```

Django's Auto-Generated Migration vs Laravel's Hand-Written One

Django

```
python manage.py makemigrations
# diffs models.py against migration history,
# WRITES the migration file for you —
# including the reverse operation automatically.
```

Laravel

```
// you write up() (create the table)
// AND down() (drop it) yourself —
// nothing is inferred from a model diff.
```

`up()`/`down()` mirror Django's own up/down migration concept exactly — the difference is authorship, not the underlying idea.

```
php artisan migrate
# Migrating: 2026_07_05_120000_create_books_table
# Migrated:  2026_07_05_120000_create_books_table (12.34ms)
```

The Eloquent Query Builder

Once past the migration difference, Eloquent's query API reads remarkably close to Django's `QuerySet`:

```
Book::all();                          // all rows
Book::where('price', '<', 15)->get();       // filtered
Book::where('in_stock', true)->orderBy('title')->get();  // chained
Book::find(1);                        // by primary key — returns null if missing
Book::findOrFail(1);                  // throws ModelNotFoundException if missing
Book::where('price', 0)->count();
```

🛡️ Mass Assignment Protection

This exact concept has now appeared in every backend framework covered so far — Rails' strong parameters, Django's `ModelForm.Meta.fields`, DRF's serializer `fields` — and Eloquent's version is `$fillable`:

```
class Book extends Model
{
    protected $fillable = ['title', 'description', 'price'];
    // Only these fields can be set via mass-assignment methods like create()/update()
}

// Safe — only fillable fields are actually written
Book::create($request->only(['title', 'description', 'price']));
```

Every ORM/framework combination this project has covered enforces the same rule with different syntax: explicitly state which fields are writable, never assume "all of them" is safe.

Automatic SQL Injection Protection

The same callback as Django's ORM chapter — Eloquent parameterizes every value passed through the query builder automatically:

```
// Completely safe — Eloquent parameterizes $userSuppliedTitle regardless of content
Book::where('title', $userSuppliedTitle)->get();

// ❌ Still dangerous — raw string interpolation into a raw query bypasses protection,
// the exact same SQLi course warning, just in PHP:
// DB::select("SELECT * FROM books WHERE title = '$userSuppliedTitle'")
```

💻 Coding Challenges

Challenge 1: Create a Model and Migration

Run `php artisan make:model Author -m`, write the migration's `up()`/`down()` methods for a table with `name` (string) and `bio` (nullable text), and add a proper `$fillable` array to the model.

Goal: Practice writing both migration directions by hand and setting up mass-assignment protection from the start.

→ Solution

Challenge 2: Write Five Query Builder Queries

Against the `Book` model, write Eloquent queries for: all in-stock books ordered by price ascending, books under $15, the single book with `id=3` (throwing if missing), all books excluding out-of-stock ones, and the count of all books.

Goal: Build fluency with `where`, `orderBy`, `findOrFail`, and `count`.

→ Solution

Challenge 3: Fix an Open `$fillable`

Given a `User`-like model with `protected $guarded = [];` (meaning nothing is protected) and a controller calling `User::create($request->all())`, rewrite both to safely allow only `name` and `email`, and explain what a malicious request could have done to the original version.

Goal: Practice recognizing and fixing the sharpest version of the mass-assignment vulnerability seen yet.

→ Solution

⚠️ Gotcha: An Open Model + `$request->all()`

This is the sharpest version of the mass-assignment warning this project has covered. If a model sets `$guarded = []` (meaning "nothing is protected") or omits `$fillable` entirely on an older Laravel setup, calling `User::create($request->all())` blindly writes **every field present in the request** — including one an attacker added that was never part of the intended form, like `is_admin` or `role`. Always define `$fillable` explicitly, and never pass `$request->all()` directly into a mass-assignment method without validating and allowlisting the fields first (Course 2's Validation-adjacent chapters build on this further). Separately: writing a migration file doesn't apply it — `php artisan migrate` still has to be run, the same "code exists but the framework doesn't know about it yet" gap seen in earlier chapters' controller/route wiring.

🎯 What's Next

A single model can now be created, queried, and migrated — the next chapter covers what happens when models need to reference each other: **Eloquent Relationships**, `hasMany`/`belongsTo`/`belongsToMany`, and eager loading with `with()` to solve the N+1 problem this project keeps running into.

🔗 Eloquent Relationships

A real difference from Django worth noticing before anything else: Django declares a relationship as a *field* (`ForeignKey`). Eloquent declares one as a *method* that returns a relationship object — then lets you access it like a property anyway, through PHP "magic" that calls and caches the method the first time it's touched.

`belongsTo` and `hasMany`

```
// app/Models/Book.php
class Book extends Model
{
    protected $fillable = ['title', 'price', 'author_id'];

    public function author()
    {
        return $this->belongsTo(Author::class);  // assumes an author_id foreign key
    }
}

// app/Models/Author.php
class Author extends Model
{
    protected $fillable = ['name'];

    public function books()
    {
        return $this->hasMany(Book::class);
    }
}
```

Django's Declarative Field vs Eloquent's Relationship Method

Django: A Field Declaration

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

The relationship IS the field — both directions come from this one declaration.

Laravel: A Method Per Direction

```
// on Book:
public function author() { return $this->belongsTo(Author::class); }
// on Author:
public function books() { return $this->hasMany(Book::class); }
```

Each direction is its own explicit method — more to write, but each side reads clearly on its own model.

```
// Access "like a property" — Eloquent calls author() once, then caches the result
$book->author->name;

// Author -> Books — a Collection, since an author can have many
$author->books;
```

`belongsToMany`: Many-to-Many

```php
// app/Models/Tag.php
class Tag extends Model
{
    protected $fillable = ['name'];

    public function books()
    {
        return $this->belongsToMany(Book::class);  // assumes a "book_tag" pivot table
    }
}

// app/Models/Book.php — add the inverse
public function tags()
{
    return $this->belongsToMany(Tag::class);
}



$book->tags()->attach($sciFiTag->id);   // add a tag
$book->tags()->detach($sciFiTag->id);   // remove a tag
$book->tags()->sync([1, 2, 3]);       // set the EXACT tag list, replacing whatever was there
$book->tags;                        // every tag on this book
$sciFiTag->books;                     // every book with this tag
```

Pivot Table Naming Convention

Eloquent assumes a table named from the two model names, alphabetized and singular, joined by an underscore — `book_tag` for `Book`/`Tag` — configurable if you need a different name.

`attach` / `detach` / `sync`

Three distinct pivot-management methods — add one, remove one, or replace the entire set — with no single direct Django ORM equivalent (Django manages M2M through the field's own `.add()`/`.remove()`/`.set()`, which map closely to these three).

⚠️ The N+1 Problem, a Third Time

This is the third framework in this project to run into exactly this trap — Rails, Django, and now Eloquent all share the identical root cause: a relationship access inside a loop triggers one query per row.

N+1 Loop vs Eager Loading

N+1: One Query Per Book

```
$books = Book::all();          // 1 query
foreach ($books as $book) {
    echo $book->author->name;    // 1 MORE query, per book
}
```

Fixed: `with()`

```
$books = Book::with('author')->get();  // 2 queries, total
foreach ($books as $book) {
    echo $book->author->name;    // already loaded — no extra query
}
```

One genuine simplification over Django here: `with()` is the **single** mechanism for eager loading regardless of relationship type — Django needs `select_related()` for `ForeignKey`/`OneToOne` and a separate `prefetch_related()` for `ManyToMany`/reverse FK. Eloquent doesn't split this into two methods:

One Method, Any Relationship Type

```
$books = Book::with(['author', 'tags'])->get();
// author is belongsTo, tags is belongsToMany — BOTH eager-loaded via the same with() call.
// Exactly 3 queries total: books, then authors, then tags — regardless of row count.
foreach ($books as $book) {
    echo $book->title . ': ' . $book->author->name;
    foreach ($book->tags as $tag) {
        echo $tag->name;
    }
}
```

Filtering by Relationship: `whereHas()`

```
// Authors who have written at least one book under $15
Author::whereHas('books', function ($query) {
    $query->where('price', '<', 15);
})->get();
```

💻 Coding Challenges

Challenge 1: Add a `belongsTo`/`hasMany` Pair

Add a `Publisher` model with a `hasMany` relationship to `Book`, and add the matching `belongsTo` relationship method (plus a `publisher_id` foreign key) on `Book`.

Goal: Practice writing both directions of a one-to-many relationship as separate methods.

→ Solution

Challenge 2: Manage a Many-to-Many Relationship

Using the `Book`/`Tag` models from this chapter, write the code to attach two tags to a book, then remove one of them, then finally replace the whole tag list with `sync()`.

Goal: Practice `attach()`, `detach()`, and `sync()` and understand how each behaves differently.

→ Solution

Challenge 3: Fix an N+1 Across Two Relationship Types

Given a loop over `Book::all()` that, for each book, prints its author's name AND every tag's name, rewrite the query to eliminate the N+1 problem for both relationships using a single `with()` call.

Goal: Practice using Eloquent's unified `with()` for both a `belongsTo` and a `belongsToMany` relationship at once.

→ Solution

⚠️ Gotcha: Eager Loading Has to Happen Before the Query Runs

`with('author')` only helps if it's part of the query *before* `get()` executes — calling it inside the loop, or trying to "add" eager loading after `Book::all()` has already run, does nothing; the N+1 queries have already happened by then. Also worth knowing: `whereHas()` runs an additional subquery to check the relationship condition — convenient for filtering, but it's a real performance cost of its own on a large table, not a free operation; reach for a plain join instead if the same filter needs to run on a hot, high-traffic path.

🎯 What's Next

Relationships between models are now readable and efficient — the next chapter turns to getting new data *in*: **Form Requests & Validation**, Laravel's `FormRequest` classes, validation rules, and Blade's `@csrf` directive compared to Django's `{% csrf_token %}`.

📝 Form Requests & Validation

This chapter mirrors Django's Forms & Validation chapter closely — validating submitted data and protecting every form against CSRF are both handled here without a separate package, the same "included from day one" story. The shape differs: Laravel's validation runs through a dedicated `FormRequest` class injected by type-hint, running *before* the controller method body even executes.

Inline Validation

For a simple case, `$request->validate()` runs right inside a controller method — rules expressed as pipe-separated strings:

```
public function store(Request $request)
{
    $validated = $request->validate([
        'title' => 'required|string|max:255',
        'price' => 'required|numeric|min:0',
        'author_id' => 'required|exists:authors,id',
    ]);

    Book::create($validated);
    return redirect()->route('books.index');
}
```

A failed validation automatically redirects back to the previous page with the errors and old input flashed to the session — no manual `if`/`else` branch needed the way Django's `is_valid()` requires.

📋 `FormRequest` Classes

For anything beyond the simplest case, a dedicated request class keeps validation rules out of the controller entirely:

```
php artisan make:request StoreBookRequest


// app/Http/Requests/StoreBookRequest.php
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class StoreBookRequest extends FormRequest
{
    public function authorize(): bool
    {
        return true;  // or a real permission check
    }

    public function rules(): array
    {
        return [
            'title' => 'required|string|max:255',
            'price' => 'required|numeric|min:0',
            'author_id' => 'required|exists:authors,id',
        ];
    }
}


// app/Http/Controllers/BookController.php
use App\Http\Requests\StoreBookRequest;

public function store(StoreBookRequest $request)
{
    // By the time this method body runs, validation has ALREADY passed —
// an invalid request never reaches this line at all.
    Book::create($request->validated());
    return redirect()->route('books.index');
}
```

Django's `is_valid()` vs Laravel's Type-Hinted `FormRequest`

Django

```
form = BookForm(request.POST)
if form.is_valid():
    form.save()
    return redirect("book-list")
# explicit check — invalid path handled by falling through
```

Laravel

```
public function store(StoreBookRequest $request)
{
    Book::create($request->validated());
    return redirect()->route('books.index');
}
// no explicit check — invalid requests never reach the method body
```

🛡️ Built-in CSRF Protection

The exact same "already on by default" story as Django's CSRF chapter — `VerifyCsrfToken` middleware is already active, and Blade's `@csrf` directive is the one-line template requirement:

```
<form method="POST" action="{{ route('books.store') }}">
 @csrf
 <input type="text" name="title">
 <button type="submit">Save</button>
</form>
```

`@csrf` compiles to a hidden `_token` input field, matched against a token stored in the session — the same synchronizer token pattern from the dedicated CSRF course, just expressed as one Blade directive instead of Django's `{% csrf_token %}`.

Displaying Validation Errors

An `$errors` variable is automatically shared with every view after a failed validation redirect — no controller code needed to pass it in:

```
<form method="POST" action="{{ route('books.store') }}">
 @csrf
 <input type="text" name="title" value="{{ old('title') }}">
 @error('title')
  <p class="error">{{ $message }}</p>
 @enderror
 <button type="submit">Save</button>
</form>
```

`old('title')`

Refills a field with the previously submitted value after a failed validation — so the user doesn't have to retype everything just because one field was wrong.

`@error` / `@enderror`

Renders its content only if that specific field failed validation, with `$message` holding the actual error text — no manual `if 'title' in errors` check needed.

Two Layers of the Same Defense: `validated()` + `$fillable`

`$request->validated()` returns **only** the fields that were actually declared in `rules()` — paired with Chapter 5's `$fillable`, this is the same two-layer mass-assignment defense Django's Forms chapter built with `ModelForm.Meta.fields`:

```
// Even if an attacker adds an extra "is_admin" field to the request body:
Book::create($request->validated());
// validated() only returns keys declared in rules() — "is_admin" was never
// declared there, so it's dropped before it ever reaches create(). And even
// if it somehow got through, Book's own $fillable would block it too.
```

💻 Coding Challenges

Challenge 1: Write Inline Validation

Write a controller method that validates a submitted `name` (required, string, max 100 chars) and `bio` (optional string) using `$request->validate()`, then creates an `Author` from the validated data.

Goal: Practice the simplest validation form before introducing a dedicated request class.

→ Solution

Challenge 2: Create a `FormRequest`

Generate an `UpdateBookRequest` with `php artisan make:request`, define `rules()` for `title`/`price`, and update a controller's `update()` method to use it via type-hint injection instead of inline validation.

Goal: Practice the full generate-rules-inject workflow for a dedicated request class.

→ Solution

Challenge 3: Fix a Form Missing `@csrf`

Given a Blade `<form method="POST">` that's missing `@csrf` and failing on every submission, fix it, and explain the exact HTTP status code Laravel returns for a CSRF failure (and how it differs from Django's).

Goal: Practice recognizing this chapter's most common first-timer mistake by its actual symptom.

→ Solution

⚠️ Gotcha: A Missing `@csrf` Gives a 419, Not a 403

Omit `@csrf` from a POST form and every submission fails — but Laravel's specific error is **419 Page Expired**, not the 403 Django returns for the identical mistake. Same root cause (a missing/invalid CSRF token), different framework, different status code — worth knowing the number specifically so a 419 in the browser's network tab immediately points at this, not a generic auth or permission problem. Separately: inside a `FormRequest`-validated controller method, always call `$request->validated()`, never `$request->all()` — validation still ran either way, but `all()` throws away the allowlist benefit entirely, passing through any extra field an attacker included regardless of whether it was declared in `rules()`.

🎯 What's Next

The final chapter of this course covers the two pieces holding everything together behind the scenes: **Middleware & Artisan** — Laravel's middleware pipeline, more Artisan commands, and how `.env` configuration ties back to everything covered so far.

🪚 Middleware & Artisan

`VerifyCsrfToken` from the last chapter is itself a piece of middleware — this closing chapter looks at the mechanism directly: Laravel's middleware pipeline, how it differs from Django's in a genuinely meaningful way, and a full circle back to Artisan and `.env` from Chapter 1.

The Middleware Pipeline

Same "onion" model as Django's middleware chain — a request passes through each layer before reaching the route, and the response passes back through in reverse. Laravel expresses each layer as a class with a `handle()` method, rather than Django's function-returning-a-function pattern:

```
php artisan make:middleware LogRequests

// app/Http/Middleware/LogRequests.php
namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;

class LogRequests
{
    public function handle(Request $request, Closure $next)
    {
        $start = microtime(true);

        $response = $next($request);  // calls the next layer — the route, or the next middleware
$duration = (microtime(true) - $start) * 1000;
        logger()->info("{$request->path()} took {$duration}ms");

        return $response;
    }
}
```

Django's Closure vs Laravel's Middleware Class

Django

```
def request_timing_middleware(get_response):
    def middleware(request):
        # before
        response = get_response(request)
        # after
return response
    return middleware
```

Laravel

```
class LogRequests
{
    public function handle($request, Closure $next)
    {
        // before
$response = $next($request);
        // after
return $response;
    }
}
```

📌 Registering Middleware: A Real Default Difference

Here's a genuine behavioral difference, not just syntax: Django's `MIDDLEWARE` list applies to **every** request by default. Laravel middleware is **opt-in per route** unless explicitly added to a shared group:

```
// bootstrap/app.php
->withMiddleware(function (Middleware $middleware) {
    $middleware->alias([
        'log.requests' => \App\Http\Middleware\LogRequests::class,
    ]);
})


// routes/web.php — applied to ONE route
Route::get('/books', [BookController::class, 'index'])
    ->middleware('log.requests');

// or applied to a whole group (Chapter 2's Route::prefix pattern)
Route::middleware('auth')->group(function () {
    Route::get('/dashboard', [DashboardController::class, 'index']);
});
```

Middleware Groups (`web` / `api`)

Laravel does apply a *group* of middleware globally to all `web` or `api` routes (session handling, CSRF, etc.) — but anything beyond that default group is opt-in per route or per group, not automatically project-wide.

Django's Model: Opt-Out, Not Opt-In

A new entry in Django's `MIDDLEWARE` list runs on every request immediately — there's no per-view "don't apply this one" short of writing custom logic inside the middleware itself.

Revisiting Artisan

Chapter 1 introduced Artisan as Laravel's `manage.py` — a few more commands worth knowing before wrapping up this course:

```
php artisan config:cache   # combine all config/*.php files into one cached file — production speed boost
php artisan route:cache    # cache the compiled route table — skips re-parsing routes/web.php on every request
php artisan make:middleware LogRequests
php artisan make:request StoreBookRequest
```

`config:cache` and `route:cache` are genuinely Laravel-specific — Django has nothing quite equivalent for its `urls.py`/`settings.py`, since those are re-evaluated per-process-start rather than needing an explicit cache-compilation step. Course 2's Deployment chapter builds on this.

⚙️ Tying It Together: `.env` and `config/`

Full circle back to Chapter 1 — every setting referenced loosely across this course traces back to one of these two places:

How `.env` Feeds `config/`

```
// .env
APP_DEBUG=true
DB_CONNECTION=mysql
DB_DATABASE=example_app

// config/app.php — reads .env via env()
return [
    'debug' => env('APP_DEBUG', false),
];

// config/database.php — same pattern
return [
    'connections' => [
        'mysql' => [
            'database' => env('DB_DATABASE', 'laravel'),
        ],
    ],
];

// Application code never reads .env directly — it reads config():
config('app.debug');       // true
config('database.connections.mysql.database');
```

This is Laravel's equivalent of Django's `settings.py` tour from Course 1's final chapter — except split across many small `config/*.php` files instead of one monolithic file, each reading its real values from `.env` via the `env()` helper.

💻 Coding Challenges

Challenge 1: Write and Apply Custom Middleware

Generate a middleware called `EnsureBookstoreIsOpen` that returns a 503 response outside of 9am–5pm (simulate with a fixed hour check), register it as an alias, and apply it only to routes under an `/orders` prefix — not globally.

Goal: Practice writing middleware and deliberately scoping it to specific routes rather than the whole app.

→ Solution

Challenge 2: Compare Middleware Application Models

Explain, in your own words, what would happen differently if Laravel applied every registered middleware globally by default (the way Django does) — specifically for a middleware that adds a 200ms artificial delay for logging purposes.

Goal: Practice reasoning about why Laravel's opt-in model is a deliberate design choice, not an oversight.

→ Solution

Challenge 3: Fix Middleware That Never Calls `$next()`

Given a custom middleware whose `handle()` method does some logging but never calls or returns `$next($request)`, explain what happens when a request passes through it, and fix the bug.

Goal: Practice recognizing and fixing the single most common custom-middleware mistake.

→ Solution

⚠️ Gotcha: Forgetting to Call `$next($request)`

A middleware's `handle()` method **must** call `$next($request)` (and return its result) to pass control forward — skip it, and the request simply never reaches the route, the controller, or any middleware after it in the pipeline. There's no error message pointing at the cause; the symptom is just "this route hangs or returns nothing," which can be confusing to trace back to one missing line in a middleware class that otherwise looks correct. Equally worth remembering as this course closes: middleware applied globally (or to the broad `web` group) runs on *every* matching request — an expensive middleware meant for one specific feature should be scoped to that feature's routes specifically, not left in the global stack "just in case."

🎓 Course Complete

That closes **Laravel Fundamentals**. Across eight chapters, Laravel occupied a similar "batteries-included" niche to Django — an ORM, templating, routing, and CLI tooling all included from the first command — while making its own distinct choices along the way: hand-written migrations instead of auto-diffed ones, relationships as explicit methods rather than declarative fields, route model binding eliminating manual lookups, and an opt-in middleware model rather than Django's global-by-default one. The mass-assignment defense ($fillable/validated()) that's now appeared in Rails, Django, DRF, and Laravel underscores how consistently the same real-world problems recur across frameworks, even when the syntax differs completely. **Course 2 (Intermediate/Advanced)** — authentication, API Resources, testing with PHPUnit, events/listeners, queues, caching, authorization, and deployment — is sketched and ready in the course bucket list whenever you want to continue.