Clean Code in Laravel

A Practical Guide to Laravel Best Practices

Ahmad Mayahi

Clean Code in Laravel

A Practical Guide to Laravel Best Practices

Ahmad Mayahi

Version 2026-02-28

Published February 28, 2026

www.mayahi.net

Table of Contents

Part 5: Bonus

The Philosophy of Simplicity

Think Clean

One of the most consistent messages from Taylor Otwell is a warning against being too "clever." In a 2025 episode of the Maintainable podcast, he described the pattern clearly:

> *We as developers sometimes tend to try to build these beautiful cathedrals of software... but we can sometimes create very complex cathedrals of complexity that aren't so easy to change. Software should be a little bit more simple and disposable and easy to change.*

This is not a throwaway quote. It is the single most important idea in this entire book. The Laravel framework was not built to impress computer science professors. It was built so that working developers could ship real applications without drowning in abstraction.

Clever code is code that makes the author feel smart. Simple code is code that makes the reader feel smart. There is a world of difference between the two, and Taylor has spent over fourteen years choosing the latter.

Consider this example. A clever and compact ternary chain:

```
$status = $user->isAdmin() ? ($order->isPaid() ? 'approved' : 'pending') : 'rejected';
```

Now look at the same logic written simply:

```php
public function determineOrderStatus(Order $order): string
{
    /** @var \App\Models\User $user */
    $user = $order->user;

    if ($user->isAdmin()) {
        if ($order->isPaid()) {
            return 'approved';
        }

        return 'pending';
    }


    return 'rejected';
}
```

The second version is longer, but any developer on your team can read it and understand what it does - including the one who joins six months from now. That is the trade Taylor makes every single time: more lines for more clarity.

Convention Over Configuration

Laravel is an opinionated framework. It has opinions about where your controllers go, how your models are named, what your migration files look like, and how your routes are structured. These opinions are not arbitrary - they are conventions that eliminate thousands of small decisions from your daily work.

When you follow Laravel's conventions, every Laravel developer who reads your code already knows where to find things. The framework's tooling - Artisan commands, route model binding, automatic dependency injection - works seamlessly because it expects things to be in certain places. And you stop wasting mental energy on decisions that do not matter.

Here is a practical example. Laravel expects a `User` model to map to a `users` table. You do not need to configure this. You do not need a mapping file. You do not need an annotation. It just works:

```
// Laravel knows this maps to the "users" table
class User extends Model
{
    // No $table property needed
}
```

The moment you fight this convention - naming your model `UserAccount` but keeping the `users` table, or naming your table `tbl_users` - you create friction. Now you need explicit configuration:

```
class UserAccount extends Model
{
    protected $table = 'users'; // Fighting the convention
}
```

Every piece of explicit configuration is a small tax on every developer who reads your code. Sometimes that tax is worth paying. Most of the time, it is not.

Code That Reads Like English

One of Laravel's most distinctive qualities is how its code reads. Taylor designs APIs so that method chains feel like sentences rather than instructions. Look at this Eloquent query:

```php
$users = User::where('active', true)
    ->whereHas('subscription', function (Builder $query): void {
        $query->where('plan', 'premium');
    })
    ->orderBy('name')
    ->get();
```

You can read that aloud: "Get users where active is true, where they have a subscription with a premium plan, ordered by name." Now compare the same thing in raw SQL:

```sql
SELECT *
FROM users
WHERE active = 1
AND EXISTS (
    SELECT 1
    FROM subscriptions
    WHERE subscriptions.user_id = users.id
      AND subscriptions.plan = 'premium'
)
ORDER BY name ASC;
```

SQL tells the database how to fetch the data. Eloquent tells the reader what you want. Both work, but one is easier to come back to six months later.

This pattern runs through the entire framework. The query builder, the collection methods, the validation rules - they are all designed to chain into readable sentences. When you write `$collection->where('active', true)->sortBy('name')->values()`, even someone unfamiliar with PHP can roughly follow what it does.

When you write your own code, ask yourself: can I read this line aloud and have it make sense?

Embrace the Framework

One of the most common mistakes in the Laravel community is fighting the framework. Developers coming from other ecosystems often try to impose patterns that do not fit - custom validation layers, hand-rolled authentication systems, or abstraction layers on top of things Laravel already handles.

Here is an example. A developer coming from a framework without built-in validation might build something like this:

```
class UserValidator
{
    public function validate(array $data): array
    {
        $errors = [];

        if (empty($data['email'])) {
            $errors[] = 'Email is required.';
        }

        if (! filter_var($data['email'], FILTER_VALIDATE_EMAIL)) {
            $errors[] = 'Email is not valid.';
        }

        if (User::where('email', $data['email'])->exists()) {
            $errors[] = 'Email is already taken.';
        }

        return $errors;
    }
}
```

Laravel already does all of this in one line:

```
$validated = $request->validate(['email' => ['required', 'email', 'unique:users']]);
```

And that is just validation. The same pattern applies across the framework:

```
// Model not found? 404 automatically.
$user = User::findOrFail($id);

// Caching? One line.
$users = Cache::remember('users', 3600, fn () => User::all());

// File storage? One line.
$path = $request->file('avatar')->store('avatars');
```

This does not mean you should never build custom solutions. It means you should reach for Laravel's built-in tools first. The framework is remarkably complete, and the code you do not write is the code that never has bugs.

PHP Built-in Features Are Not Always the Answer

There is a subtler version of fighting the framework that does not involve building custom solutions from scratch. Instead, developers see a shiny new PHP feature and reach for it as a drop-in replacement for something the framework already handles.

A common example is PHP 8.4 property hooks. You might be tempted to use them inside Eloquent models:

```
class User extends Model
{
    public string $first_name {
        set(string $value) => $this->attributes['first_name'] = ucfirst($value);
        get => ucfirst($this->attributes['first_name']);
    }
}
```

It looks clean. It uses a modern PHP feature. But it quietly breaks how Eloquent works.

The Laravel way to handle this is with the attribute system:

```php
use Illuminate\Database\Eloquent\Casts\Attribute;

class User extends Model
{
    protected function firstName(): Attribute
    {
        return Attribute::make(
            get: fn (string $value) => ucfirst($value),
            set: fn (string $value) => ucfirst($value),
        );
    }
}
```

The difference is not cosmetic. Eloquent's attribute system is wired into the model's lifecycle. Serialization methods like `toArray()`, `toJson()`, and `$appends` all depend on it. Property hooks bypass serialization entirely - `$post->title` and `$post->toArray()['title']` will return different values. Your model might look like it works when you access a property directly, but it will behave unexpectedly the moment you serialize it for an API response.

Beyond the technical issues, framework conventions are where your team and the ecosystem will look. Nova, Filament, API Resources, and every Laravel package that touches models understands the attribute system. New developers joining your project will check `casts()` and look for `Attribute` methods - not property hooks.

This is not a criticism of property hooks themselves. They are a great PHP feature - in plain PHP classes, DTOs, value objects, and anywhere you are not inside Eloquent's system. But inside a model, Eloquent's tools exist for a reason. Use them.

You Are Not Going to Need It

Developers love to plan for the future. "What if we need to switch databases?" "What if we need to support multiple payment providers?" "What if this needs to scale to millions of users?" These are valid questions - for later. Right now, they lead to code that solves problems you do not have.

This idea has a name: YAGNI - You Aren't Gonna Need It. It means do not build something until you actually need it.

Here is what premature abstraction looks like:

```
// You have one payment provider: Stripe.
// But "what if we add PayPal later?"

interface PaymentGatewayInterface
{
    public function charge(int $amount, string $currency): PaymentResult;
    public function refund(string $transactionId): RefundResult;
}

class StripePaymentGateway implements PaymentGatewayInterface { /* ... */ }
class PaymentGatewayFactory { /* ... */ }
class PaymentGatewayManager { /* ... */ }
```

You now have four files to maintain instead of one, and you still only use Stripe. If PayPal comes along next year, you can refactor then - and you will have a much better understanding of what the abstraction should look like because you will have a real second use case, not an imaginary one.

The simple version:

```php
class StripeService
{
    public function charge(int $amount, string $currency): PaymentIntent
    {
        return Stripe::paymentIntents()->create([
            'amount' => $amount,
            'currency' => $currency,
        ]);
    }
}
```

One class, no interface, no factory. When you need a second payment provider, that is the time to extract an interface - not before.

The Repository Pattern in Laravel

The most common example of YAGNI in the Laravel world is the Repository Pattern. The idea is to put a layer between your application and the database so that you can "swap the database later." It looks something like this:

```php
interface UserRepositoryInterface
{
    public function findById(int $id): ?User;
    public function findActiveUsers(): Collection;
    public function create(array $data): User;
}

class EloquentUserRepository implements UserRepositoryInterface
{
    public function findById(int $id): ?User
    {
        return User::find($id);
    }

    public function findActiveUsers(): Collection
    {
        return User::where('active', true)->get();
    }

    public function create(array $data): User
    {
        return User::create($data);
    }
}
```

Every method in this class is a thin wrapper around something Eloquent already does. The interface adds nothing - it just mirrors the methods on the other side. And the justification is always the same: "What if we need to switch from MySQL to MongoDB?"

Be honest with yourself: how often does that happen? In years of building Laravel applications, I have never seen a project that actually needed to swap its database. And even if it did, the repository would not save you - switching databases means changing queries, relationships, indexes, migrations, and data types. A thin wrapper around Eloquent does not protect you from any of that.

Laravel already gives you two powerful layers for querying data: Eloquent for working with models and relationships, and the Query Builder for more complex queries. Between the two, you can handle virtually anything. If your query logic gets complex, use Eloquent scopes - not a repository that repeats what Eloquent already does.

```
// Instead of a repository, use scopes
use Illuminate\Database\Eloquent\Attributes\Scope;

class User extends Model
{
    #[Scope]
    protected function active(Builder $query): void
    {
        $query->where('active', true);
    }
}

// Clean, simple, and uses what Laravel gives you
$activeUsers = User::active()->get();
```

The Repository Pattern has its place in frameworks that do not have a built-in ORM, or in applications that genuinely talk to multiple data sources. In a typical Laravel application, it is extra code that solves a problem you do not have.

> *The best abstraction is the one you write after you have two concrete implementations, not before you have one.*

Code Is Read More Than It Is Written

You write a line of code once. Your team reads it dozens of times - in code reviews, while debugging, while adding features nearby, while onboarding new developers. Every decision you make while writing should favor the reader, not the writer.

This means:

- Choose clarity over brevity. A longer, obvious name beats a short, cryptic one. `$activeSubscriptions` is better than `$subs`.
- Choose boring over clever. A simple `if` statement that anyone can follow beats a one-liner that requires a second look.
- Choose explicit over implicit. If something is not obvious from the code, make it obvious - even if it takes an extra line.

A good rule of thumb: if you need to add a comment to explain what a piece of code does, the code itself is probably too complex. Rewrite it until the comment is unnecessary.

```
// Bad: needs a comment to explain
// Get users who registered in the last 30 days and have verified their email
$users = User::where('created_at', '>', now()->subDays(30))
    ->whereNotNull('email_verified_at')
    ->get();

// Good: the code explains itself
$users = User::query()
    ->registeredInLast(days: 30)
    ->verified()
    ->get();
```

The second version uses Eloquent scopes to give the query conditions meaningful names. The logic is the same, but the reader does not need a comment to understand it.

The Three Questions

Before writing any piece of code in a Laravel application, ask yourself three questions:

1. Is this the simplest way to solve this problem?

If you are reaching for a design pattern, a third-party package, or a custom abstraction, pause. Is there a simpler way? Could a plain PHP class do the job? Could a built-in Laravel feature handle it?

2. Will a new team member understand this in thirty seconds?

If your code requires a README, a diagram, or a verbal explanation to understand, it is too complex. The best code explains itself.

3. Am I following Laravel's conventions?

If you are naming things differently, putting files in unusual places, or structuring your application in a non-standard way, you should have a very good reason. Convention is a gift - accept it.

These three questions will guide every decision in this book. When we discuss Actions, Services, DTOs, or Domain-Driven Design in later chapters, we will always come back to these fundamentals: simplicity, clarity, and convention.

What This Book Is (and What It Is Not)

This book is not a Laravel tutorial. It assumes you already know how to build Laravel applications - you understand routing, controllers, Eloquent, Blade, and the basics of the framework.

This book is about how to build Laravel applications well. It is a style guide, an architecture guide, and a philosophy guide - all grounded in practical, real-world code examples using Laravel 12. Every pattern we discuss, every convention we recommend, and every package we introduce serves one purpose: making your code simpler, clearer, and more maintainable.

We start with the small things - naming, code style, controller structure - and gradually build toward larger architectural patterns like Domain-Driven Design and Event Sourcing. Each chapter builds on the previous one. No concept is used before it is explained.

By the end of this book, you will not just write Laravel code. You will write Laravel code that any Laravel developer can read, understand, and maintain.

Summary

- Simple beats clever. Write code that makes the reader feel smart, not the author. More lines for more clarity is always a good trade.
- Follow Laravel's conventions. The framework expects things in certain places. When you follow those expectations, the tooling works for you and every Laravel developer already knows their way around your code.
- Let code read like English. Eloquent, collections, and validation are designed to chain into readable sentences. Write your own code the same way.
- Use what Laravel gives you. Before building a custom solution, check if the framework already solves the problem. It usually does.
- Do not build for imaginary requirements. Solve the problem in front of you. Abstractions, interfaces, and patterns like the Repository Pattern can wait until you have a real reason to reach for them.
- Favor the reader. You write code once. Your team reads it dozens of times. Choose clarity over brevity, boring over clever, and explicit over implicit.
- Ask three questions before writing code: Is this the simplest way? Will a new team member understand it quickly? Am I following Laravel's conventions?

Test Your Understanding

Think you've mastered this chapter? Take the quiz and find out.

http://mayahi.test/quiz/book/clean-code-in-laravel/the-philosophy-of-simplicity

Naming Conventions

Think Clean

Naming is the most underrated skill in software development. A good name eliminates the need for a comment. A bad name creates confusion that ripples through your entire codebase. In Laravel, naming is especially important because the framework uses names to make automatic connections - a `User` model maps to a `users` table, a `PostController` handles `Post` routes, and a `SendWelcomeEmail` job does exactly what its name says.

Taylor Otwell's code is remarkably consistent in its naming. If you read through the Laravel framework source code, you will notice that every class, method, and variable follows predictable patterns. This chapter documents those patterns so you can follow them in your own applications.

Models

Models are always singular nouns in PascalCase. A model represents a single record in a database table, so it makes sense to name it in the singular:

Good	Bad	Why It's Bad
`User`	`Users`	Plural - a model is one record
`Invoice`	`InvoiceModel`	Redundant suffix
`OrderItem`	`Order_Item`	Snake_case in a class name
`FlightBooking`	`Flightbooking`	Missing PascalCase word boundary

Laravel automatically maps model names to table names by converting PascalCase to snake_case and pluralizing:

```
class OrderItem extends Model
{
    // Automatically maps to "order_items" table
    // No $table property needed
}
```

When your model name has multiple words, Laravel handles it correctly. `OrderItem` becomes `order_items`. `FlightBooking` becomes `flight_bookings`. Trust the convention.

Controllers

Controllers follow the pattern `{SingularResource}Controller`. They are always singular because they handle operations on a type of resource, not on multiple resources. We cover controller structure in detail in the Thin Controllers chapter.

Good	Bad	Why It's Bad
`UserController`	`UsersController`	Plural
`InvoiceController`	`InvoiceCtrl`	Abbreviated
`OrderItemController`	`OrderItemsController`	Plural

For controllers that do not map to a single resource, use a descriptive name that explains what the controller does:

```
class DashboardController extends Controller { }
class SettingsController extends Controller { }
class SearchController extends Controller { }
```

For invokable controllers - controllers with a single `__invoke` method - name them after the action they perform. We explain when and why to use invokable controllers in the Thin Controllers chapter.

```
class ExportOrdersController extends Controller
{
    public function __invoke(Request $request): Response
    {
        // Single action: export orders
    }
}
```

Methods

Methods in Laravel always use camelCase and typically start with a verb that describes what they do:

```
// Good: verb-first, descriptive
public function calculateTotal(): Money { }
public function sendNotification(): void { }
public function findByEmail(string $email): ?User { }
public function markAsPaid(): void { }
public function isActive(): bool { }
public function hasSubscription(): bool { }

// Bad: vague, noun-first, or unclear
public function total(): Money { }        // Is this getting or calculating?
public function notification(): void { }   // Is this sending or creating?
public function email(string $email) { }   // What about the email?
```

Notice the naming patterns for boolean methods: `is` prefix for state checks (`isActive`, `isPaid`, `isAdmin`) and `has` prefix for relationship/ownership checks (`hasSubscription`, `hasPermission`, `hasVerifiedEmail`).

Variables

Variable naming is arguably the most important naming decision you make, because variables are everywhere. A method gets named once. A variable gets read dozens of times - in conditions, loops, function arguments, and return values. If the name is unclear, every line that uses it becomes harder to understand.

The rule is simple: a variable name should tell you what it holds, not just that it exists. Use camelCase and be specific.

Say What It Is

The biggest mistake is using short, vague names that force the reader to look elsewhere to understand what the variable contains:

```
// Bad: what is $u? What is $data? What is $inv?
$u = User::where('active', true)->get();
$data = $orders->sum('total');
$inv = Invoice::where('status', 'pending')->first();
$temp = $user->created_at->diffInDays(now());

// Good: the name tells you exactly what it holds
$activeUsers = User::where('active', true)->get();
$monthlyRevenue = $orders->sum('total');
$pendingInvoice = Invoice::where('status', 'pending')->first();
$daysSinceRegistration = $user->created_at->diffInDays(now());
```

When you read `$monthlyRevenue`, you know what it is. When you read `$data`, you have to trace back to where it was assigned. That tracing adds up across an entire codebase.

Avoid Abbreviations

It is tempting to shorten names to save keystrokes, but abbreviated names cost more time in reading than they save in typing. Your editor has autocomplete - use it:

Bad	Good	Why It's Bad
`$usr`	`$user`	Saves two characters, loses clarity
`$addr`	`$shippingAddress`	Which address? Billing? Shipping?
`$qty`	`$quantity`	Not everyone reads `qty` as "quantity"
`$calc`	`$calculatedDiscount`	Calculated what?
`$res`	`$response`	`$res` could mean result, resource, or response
`$e`	`$exception`	Single letters hide meaning

The only place single-letter variables are acceptable is in short loops where the scope is tiny and the meaning is obvious:

```
for ($i = 0; $i < $retryLimit; $i++) {
    // $i is fine here - small scope, obvious meaning
}
```

Singular vs. Plural

Use plural names for collections and singular names for single items. This small habit tells the reader whether they are looking at one thing or many things without checking the type:

```
$users = User::all();              // Collection - plural
$user = User::findOrFail($id);    // Single model - singular

foreach ($users as $user) {        // Iterating: plural → singular
    $user->notify(new WelcomeNotification());
}
```

This also protects you from mistakes. If you see `$user->sum('balance')`, something is wrong - a single model does not have a `sum` method. The plural name `$users->sum('balance')` immediately makes sense.

Booleans Should Read Like Questions

A boolean variable should read naturally in an `if` statement. Prefix it with `is`, `has`, `can`, `should`, or `was`:

```
// Good: reads like English
$isActive = $user->active;
$hasSubscription = $user->subscription !== null;
$canEditPost = $user->id === $post->user_id;
$shouldSendReminder = $invoice->due_at->isToday();

// Bad: what does "true" or "false" mean here?
$active = $user->active;           // Is this a boolean or the active record?
$subscription = true;              // Is this a boolean or a Subscription object?
$edit = $user->id === $post->user_id;  // Edit what?
```

When you read `if ($isActive)`, it flows like a sentence: "if is active." When you read `if ($active)`, it could mean anything.

Avoid Generic Names

Names like `$data`, `$result`, `$info`, `$temp`, and `$value` tell you nothing. They are placeholders that never got replaced with a real name:

```php
// Bad: generic names that could mean anything
$data = $request->validated();
$result = $paymentGateway->charge($amount);
$info = $user->profile;
$items = $request->input('products');

// Good: specific names that describe the content
$validatedInput = $request->validated();
$chargeResult = $paymentGateway->charge($amount);
$userProfile = $user->profile;
$selectedProducts = $request->input('products');
```

Sometimes `$result` seems fine because the context is obvious. But code gets moved, refactored, and read out of context. A specific name survives those changes; a generic one does not.

Match the Domain Language

Use the same words your team and your business use. If the business calls them "invoices," do not call them "bills" in code. If the product team says "subscription," do not use "plan":

```php
// If the business says "coupon"
$appliedCoupon = Coupon::where('code', $code)->first();

// Don't call it a "discount code" or "promo" in code
$promo = Coupon::where('code', $code)->first(); // Confusing
```

This idea comes from Domain-Driven Design - using a shared language between developers and the business. When everyone uses the same words, there is less room for misunderstanding.

Database

Tables

Database tables are always plural and use snake_case. This is the convention that Eloquent relies on for automatic model-to-table mapping:

Model	Table	Pivot Table
`User`	`users`	-
`Post`	`posts`	-
`OrderItem`	`order_items`	-
`User` + `Role`	-	`role_user` (alphabetical)

Pivot tables combine the two model names in singular snake_case, sorted alphabetically, joined by an underscore. So `User` and `Role` become `role_user`, not `user_role`.

Columns

Columns use snake_case and should describe what they store. See migrations for all available column types:

```
Schema::create('orders', function (Blueprint $table): void {
    $table->id();
    $table->foreignId('user_id')->constrained();     // Foreign key: model_id
    $table->string('shipping_address');                // Descriptive
    $table->decimal('total_amount', 10, 2);           // Clear unit
    $table->timestamp('paid_at')->nullable();          // Timestamp: verb_at
    $table->boolean('is_refundable')->default(true);   // Boolean: is_ prefix
    $table->timestamps();                              // created_at, updated_at
});
```

Notice the patterns: foreign keys use `{model}_id`, timestamps use `{verb}_at` (like `paid_at`, `shipped_at`, `verified_at`), and booleans use `is_` or `has_` prefixes.

Routes

Routes use plural nouns in kebab-case for the URI, and follow RESTful conventions:

```
// Good: RESTful, plural, kebab-case
Route::get('/users', [UserController::class, 'index']);
Route::get('/users/{user}', [UserController::class, 'show']);
Route::get('/order-items', [OrderItemController::class, 'index']);
Route::post('/order-items', [OrderItemController::class, 'store']);

// Bad: singular, camelCase, or verb-based
Route::get('/user', [UserController::class, 'index']);       // Singular
Route::get('/orderItems', [OrderItemController::class, 'index']); // camelCase
Route::get('/getUsers', [UserController::class, 'index']);   // Verb in URL
```

Dashes vs. Underscores in URLs

When a route has multiple words, use dashes (kebab-case), not underscores. This is a web-wide convention - Google recommends dashes over underscores in URLs, and it is what most REST APIs use:

```
// Good: dashes in the URL
Route::get('/order-items', [OrderItemController::class, 'index']);
Route::get('/payment-methods', [PaymentMethodController::class, 'index']);
Route::post('/forgot-password', ForgotPasswordController::class);

// Bad: underscores or camelCase in the URL
Route::get('/order_items', [OrderItemController::class, 'index']);
Route::get('/paymentMethods', [PaymentMethodController::class, 'index']);
```

There is one exception: route parameters must use camelCase or snake_case because they map to PHP variables. Dashes are not allowed in PHP variable names, so `{order-item}` would not work:

```
// Route parameters use camelCase -dashes don't work here
Route::get('/users/{user}', [UserController::class, 'show']);
Route::get('/order-items/{orderItem}', [OrderItemController::class, 'show']);
```

Named Routes

Named routes use dot notation with the resource name:

```
Route::resource('users', UserController::class);
// Generates: users.index, users.create, users.store, users.show, etc.

// Custom named routes follow the same pattern
Route::get('/dashboard', DashboardController::class)->name('dashboard');
Route::get('/users/{user}/orders', [UserOrderController::class, 'index'])
    ->name('users.orders.index');
```

When a route name has multiple words, there is no single official convention. Some teams use camelCase, others use kebab-case. Spatie's guidelines recommend camelCase:

```
// camelCase (Spatie convention)
Route::get('/account/billing-history', [BillingController::class, 'history'])
    ->name('account.billingHistory');

// kebab-case (matches what Laravel generates for resource routes)
Route::get('/account/billing-history', [BillingController::class, 'history'])
    ->name('account.billing-history');
```

Both work fine. What matters is that your team picks one and sticks with it. Avoid snake_case - `billing_history` -since neither the framework nor the community uses it for route names.

Dots separate resources. The style you choose separates words within a segment. If you stay consistent, you can always guess the route name without running `php artisan route:list`.

Views

View files use kebab-case and are organized in directories matching their resource:

```
resources/views/
├── users/
│   ├── index.blade.php
│   ├── show.blade.php
│   ├── create.blade.php
│   └── edit.blade.php
├── order-items/
│   ├── index.blade.php
│   └── show.blade.php
└── components/
    ├── alert.blade.php
    └── user-card.blade.php
```

Reference them with dot notation:

```
return view('users.index', compact('users'));
return view('order-items.show', compact('orderItem'));
```

Config Files

Configuration files and their keys use snake_case. The filename itself is also snake_case:

```
// config/payment.php
return [
    'stripe_key' => env('STRIPE_KEY'),
    'default_currency' => 'usd',
    'tax_rate' => 0.21,
];

// Accessing config values with dot notation
$key = config('payment.stripe_key');
$currency = config('payment.default_currency');
```

Keep config keys flat and descriptive. If you find yourself nesting deeply, it is usually a sign that the config file is doing too much and should be split into separate files.

Language Files

Laravel supports two approaches for localization: PHP files with short keys, and JSON files with full translation strings as keys.

PHP Language Files

PHP language files live under `lang/{locale}/` and use snake_case for both the filename and the keys:

```
lang/
├── en/
│   ├── messages.php
│   ├── validation.php
│   └── auth.php
└── fr/
    ├── messages.php
    ├── validation.php
    └── auth.php
```

```php
// lang/en/messages.php
return [
    'welcome' => 'Welcome to our application!',
    'order_placed' => 'Your order has been placed.',
    'greeting' => 'Hello, :name!',
];

// Accessing translations
echo __('messages.welcome');
echo __('messages.greeting', ['name' => 'Ahmad']);
```

JSON Language Files

For applications with many translatable strings, JSON files are simpler. Each language gets a single file named after its locale, and the keys are the full English strings:

```json
// lang/es.json
{
    "Welcome to our application!": "¡Bienvenido a nuestra aplicación!",
    "Your order has been placed.": "Tu pedido ha sido realizado."
}
```

The advantage of JSON files is that you do not need to invent short keys -the English text is the key. The trade-off is that they can get large in applications with hundreds of strings.

For territory-specific languages, use the ISO 15897 format: `en_GB` for British English, `pt_BR` for Brazilian Portuguese -not `en-gb` or `pt-br`.

Form Requests

Form requests describe the action they validate:

Good	Bad
`StoreUserRequest`	`UserRequest`
`UpdateOrderRequest`	`OrderValidation`
`CreateInvoiceRequest`	`InvoiceFormRequest`

The `Store` and `Update` prefixes match the resource controller methods they serve, making the connection obvious:

```
class UserController extends Controller
{
    public function store(StoreUserRequest $request): RedirectResponse { }
    public function update(UpdateUserRequest $request, User $user): RedirectResponse { }
}
```

Events, Listeners, Jobs, and Notifications

These classes follow a consistent pattern: they describe what happened (events), what to do about it (listeners/jobs), or what to tell someone (notifications):

Type	Naming Pattern	Example
Event	Past tense verb	`OrderPlaced`, `UserRegistered`, `PaymentFailed`
Listener	Action to take	`SendOrderConfirmation`, `CreateUserProfile`
Job	Action to perform	`ProcessPayment`, `GenerateInvoicePdf`
Notification	What is being communicated	`OrderShippedNotification`, `WelcomeNotification`
Mail	What is being sent	`OrderConfirmationMail`, `PasswordResetMail`

Events use past tense because they describe something that already happened. Listeners, jobs, and notifications describe what will happen in response.

Enums

Enums use singular PascalCase, and their cases use PascalCase as well. Laravel has built-in support for casting model attributes to enums:

```
enum OrderStatus: string
{
    case Pending = 'pending';
    case Processing = 'processing';
    case Shipped = 'shipped';
    case Delivered = 'delivered';
    case Cancelled = 'cancelled';
}
```

The enum name describes what it represents (`OrderStatus`, `PaymentMethod`, `UserRole`), and each case reads naturally: `OrderStatus::Shipped`, `PaymentMethod::CreditCard`.

Middleware

Middleware names describe what they check or enforce. Use PascalCase and keep the name short:

Good	Bad	Why It's Bad
`EnsureEmailIsVerified`	`CheckEmail`	Too vague -check what?
`HandleLocale`	`LocaleMiddleware`	Redundant suffix
`Authenticate`	`Auth`	Abbreviated

Laravel's own middleware follow this pattern: `EnsureFrontendRequestsAreStateful`, `TrustProxies`, `HandleCors`. The name should tell you what the middleware does without opening the file.

Policies

Policies map directly to models. The naming is simple: `{Model}Policy`:

Model	Policy
`User`	`UserPolicy`
`Invoice`	`InvoicePolicy`
`OrderItem`	`OrderItemPolicy`

Policy methods match the action they authorize: `view`, `create`, `update`, `delete`, `restore`, `forceDelete`. These names come from Laravel's default resource methods, and the framework auto-discovers them when you follow this convention.

Traits

Traits use PascalCase and describe the capability they provide. Laravel's built-in traits follow a clear pattern -they start with a verb or `Has` prefix that reads naturally when you see them in a `use` statement:

```
use HasFactory;
use Notifiable;
use SoftDeletes;
use HasApiTokens;
```

Follow the same pattern in your own traits. The name should describe what the trait gives the class, not what the class is:

Good	Bad	Why It's Bad
`HasSubscription`	`SubscriptionTrait`	Redundant suffix
`Searchable`	`SearchTrait`	Redundant suffix
`TracksActivity`	`Activity`	Sounds like a model, not a behavior
`HandlesPayments`	`PaymentFunctions`	Vague and unusual

When you read `class User extends Model` followed by `use HasSubscription, Searchable`, it reads like a description: "User has subscription, is searchable."

Scopes

Eloquent scopes define reusable query constraints on a model. In Laravel 12, scopes use the `#[Scope]` attribute on a `protected` method:

```php
use Illuminate\Database\Eloquent\Attributes\Scope;
use Illuminate\Database\Eloquent\Builder;

class Order extends Model
{
    #[Scope]
    protected function paid(Builder $query): void
    {
        $query->whereNotNull('paid_at');
    }

    #[Scope]
    protected function recent(Builder $query): void
    {
        $query->where('created_at', '>=', now()->subDays(30));
    }

    #[Scope]
    protected function forUser(Builder $query, User $user): void
    {
        $query->where('user_id', $user->id);
    }
}

// Reads naturally when chained
$orders = Order::paid()->recent()->forUser($user)->get();
```

The method name is what you call when chaining - `paid()`, `recent()`, `forUser()`. Pick names that are adjectives or past-tense verbs so they read well in a chain. Notice how `Order::paid()->recent()->forUser($user)` reads almost like English. That is the goal.

Avoid scope names that start with verbs like `get` or `find` -those suggest a method that returns a result, not a scope that filters a query:

Good	Bad	Why It's Bad
`active`	`getActive`	Scopes don't "get" -they filter
`published`	`findPublished`	Same issue -sounds like it returns a result
`forUser`	`byUser`	Less clear, but acceptable

Blade Components

Blade components use kebab-case in their tag names and PascalCase for the class. Laravel handles the conversion automatically:

```
// Class: app/View/Components/UserCard.php
class UserCard extends Component { }

// Usage in Blade
<x-user-card :user="$user" />
```

For nested components, the directory structure maps to dot notation in the tag:

```
app/View/Components/
├── Form/
│   ├── Input.php          → <x-form.input />
│   └── Select.php         → <x-form.select />
└── Navigation/
    └── Breadcrumb.php     → <x-navigation.breadcrumb />
```

Anonymous components -Blade files without a class -follow the same kebab-case pattern:

```
resources/views/components/
├── alert.blade.php        → <x-alert />
├── user-card.blade.php    → <x-user-card />
└── form/
    └── input.blade.php    → <x-form.input />
```

Constants

Class constants and global constants use `SCREAMING_SNAKE_CASE`:

```php
class Invoice
{
    public const MAX_LINE_ITEMS = 50;
    public const DEFAULT_CURRENCY = 'usd';
    public const TAX_RATE = 0.21;
}
```

This is a PHP-wide convention, not specific to Laravel, but it is worth mentioning because mixing constant naming styles in a codebase is a common mistake.

If a class exists only to hold a group of related constants, consider using an enum instead. Enums give you type safety, autocompletion, and a single place to manage allowed values -which plain constants do not:

```php
// Instead of a class full of constants
class Currency
{
    public const USD = 'usd';
    public const EUR = 'eur';
    public const GBP = 'gbp';
}

// Use an enum
enum Currency: string
{
    case Usd = 'usd';
    case Eur = 'eur';
    case Gbp = 'gbp';
}
```

The Complete Naming Reference

What	Convention	Example
Model	Singular, PascalCase	`OrderItem`
Controller	Singular + Controller	`OrderItemController`
Migration	snake_case, descriptive	`create_order_items_table`
Seeder	Singular + Seeder	`OrderItemSeeder`
Factory	Singular + Factory	`OrderItemFactory`
Table	Plural, snake_case	`order_items`
Pivot table	Alphabetical, singular	`item_order`
Column	snake_case	`total_amount`
Foreign key	Singular model + _id	`order_id`
Route	Plural, kebab-case	`/order-items`
Named route	Dot notation	`order-items.index`
View	kebab-case directory	`order-items/index.blade.php`
Config	snake_case	`config('payment.stripe_key')`
Method	camelCase, verb-first	`calculateTotal()`
Variable	camelCase	`$orderItem`
Property	camelCase	`$this->shippingAddress`
Constant	SCREAMING_SNAKE_CASE	`MAX_LINE_ITEMS`
Form Request	Verb + Resource + Request	`StoreOrderItemRequest`

What	Convention	Example
Event	Past tense	`OrderItemAdded`
Listener	Action phrase	`UpdateInventoryCount`
Job	Action phrase	`ProcessOrderPayment`
Notification	Descriptive + Notification	`OrderShippedNotification`
Enum	Singular, PascalCase	`OrderStatus`
Middleware	Descriptive, PascalCase	`EnsureEmailIsVerified`
Policy	Model + Policy	`OrderItemPolicy`
Trait	PascalCase, describes capability	`HasSubscription`
Scope	`#[Scope]` + adjective/verb	`paid()`, `active()`
Blade component	kebab-case tag, PascalCase class	`<x-user-card>` / `UserCard`

Follow these conventions consistently, and any Laravel developer can navigate your codebase without extra documentation.

Summary

- Models are singular PascalCase (`OrderItem`). Laravel maps them to plural snake_case tables (`order_items`) automatically.
- Controllers are singular with a `Controller` suffix (`OrderItemController`). Invokable controllers are named after the action they perform.
- Methods use camelCase and start with a verb (`calculateTotal`, `sendNotification`). Boolean methods use `is` / `has` / `can` prefixes.
- Variables should say what they hold (`$activeUsers`, not `$data`). Use plural for collections, singular for single items, and boolean prefixes for true/false values.

- Database tables are plural snake_case. Columns are snake_case. Foreign keys use `{model}_id`. Timestamps use `{verb}_at`. Booleans use `is_` / `has_` prefixes.
- Routes use plural kebab-case URLs (`/order-items`). Named routes use dot notation (`users.orders.index`).
- Views use kebab-case directories and filenames (`order-items/index.blade.php`).
- Events use past tense (`OrderPlaced`). Listeners, jobs, and notifications describe the action (`SendOrderConfirmation`).
- Traits describe a capability (`HasSubscription`, `Searchable`). Scopes use the `#[Scope]` attribute with adjectives or past-tense verbs that chain naturally (`Order::paid()->recent()`).
- Blade components use kebab-case tags (`<x-user-card>`) and PascalCase classes (`UserCard`).
- When in doubt, check how Laravel names the same thing in its own source code - then do that.

Test Your Understanding

Think you've mastered this chapter? Take the quiz and find out.

http://mayahi.test/quiz/book/clean-code-in-laravel/naming-conventions

Code Quality and Automation

Think Clean

Most books put code style at the end, as an afterthought. We put it here, in Chapter 3, because code style is not a finishing touch - it is a foundation. If your team cannot agree on how code looks, every pull request becomes a formatting debate instead of a logic review.

Taylor Otwell settled this for the Laravel community. Laravel follows PSR-12 with a few opinionated additions, and the official tool for enforcing it is Laravel Pint.

Laravel Pint

Laravel Pint is a zero-configuration code style fixer built on top of PHP-CS-Fixer. It ships with every new Laravel application. Out of the box, Pint uses the `laravel` preset, which is Taylor's preferred style. You do not need a configuration file. You do not need to debate tabs versus spaces. You run Pint, and your code looks like Laravel code.

To fix the style of every PHP file in your project, run Pint with no arguments:

```
./vendor/bin/pint
```

If you want to see what Pint would change without actually modifying any files, use the `--test` flag. This is useful in CI pipelines where you want to fail a build on style violations rather than auto-fix them:

```
./vendor/bin/pint --test
```

You can also target specific files or directories instead of scanning the entire project:

```
./vendor/bin/pint app/Models
./vendor/bin/pint app/Http/Controllers/UserController.php
```

If you need to customize rules, create a `pint.json` file in your project root:

```
{
    "preset": "laravel",
    "rules": {
        "concat_space": {
            "spacing": "one"
        },
        "ordered_imports": {
            "sort_algorithm": "length"
        }
    }
}
```

The best practice is to run Pint automatically. Add it to your CI pipeline, your pre-commit hooks, or your editor's save action. Code style should never be a manual task.

Type Hints and Return Types

Type hints and return types serve as documentation, enable static analysis, and catch bugs before they reach production. Use them everywhere:

```php
// Good: fully typed
public function findActiveUsers(int $limit = 10): Collection
{
    return User::where('active', true)
        ->limit($limit)
        ->get();
}

public function calculateDiscount(Order $order, float $percentage): Money
{
    return $order->total->multiply($percentage / 100);
}

public function markAsPaid(Invoice $invoice): void
{
    $invoice->update(['paid_at' => now()]);
}

// Bad: no types — what does this accept? What does it return?
public function findActiveUsers($limit = 10)
{
    return User::where('active', true)->limit($limit)->get();
}
```

Use nullable types when a value might not exist:

```php
public function findByEmail(string $email): ?User
{
    return User::where('email', $email)->first();
}
```

Use union types when a method can return different types:

```php
public function resolve(string $key): string|int|null
{
    return config($key);
}
```

The same applies to class properties. Use typed properties instead of relying on docblocks or hoping the right value is assigned. The `readonly` keyword is especially useful for injected dependencies — it prevents reassignment after construction:

```
// Good: typed properties
class OrderService
{
    public function __construct(
        private readonly OrderRepository $repository,
        private readonly int $maxRetries = 3,
    ) {}
}

// Bad: untyped — anything could end up in these
class OrderService
{
    private $repository;
    private $maxRetries;
}
```

Typed properties make your code predictable. If someone tries to assign a string to `$maxRetries`, PHP catches it immediately instead of letting it silently break something downstream.

DocBlocks

A docblock is a special comment that starts with `/**` and sits above a class, method, or property. Tools like PHPDoc and IDEs read these comments to understand your code — what a method accepts, what it returns, and what it does. Before PHP had type hints, docblocks were the only way to document types.

Now that PHP has typed properties, return types, and parameter types, most of your code is already self-documenting. That means most docblocks are just noise — they repeat what the code already says:

```php
// Bad: the docblock adds nothing
/**
 * Get the user's full name.
 *
 * @return string
 */
public function fullName(): string
{
    return "{$this->first_name} {$this->last_name}";
}

// Good: the method signature says it all
public function fullName(): string
{
    return "{$this->first_name} {$this->last_name}";
}
```

Only add a docblock when it provides information that the type system cannot express. The most common case is describing the shape of arrays and collections:

```php
/**
 * @param  array<string, mixed>  $attributes
 * @return Collection<int, User>
 */
public function findMatching(array $attributes): Collection
{
    return User::where($attributes)->get();
}
```

Here the docblock is useful — `array` and `Collection` tell you nothing about what is inside them. `array<string, mixed>` and `Collection<int, User>` do.

Another good use is documenting exceptions or non-obvious side effects:

```php
/**
 * @throws PaymentFailedException
 */
public function charge(Order $order): PaymentIntent
{
    return $this->gateway->charge($order->total);
}
```

The rule is simple: if the docblock repeats the signature, delete it. If it adds information the signature cannot express, keep it.

Static Analysis

Static analysis means checking your code for bugs without running it. Instead of waiting for something to break in production, a static analysis tool reads your files, follows how data moves through your methods, and tells you about problems before they happen. Think of it as a spell-checker for your logic.

Type hints are good - they tell PHP what types to expect. Static analysis takes that further. PHPStan reads your entire codebase and catches bugs that tests might miss - calling a method that does not exist, passing the wrong type to a function, or using a variable that might be null. Larastan is a PHPStan extension that understands Laravel's magic - Eloquent models, facades, collections, and more.

Unlike Pint, Laravel does not ship with PHPStan or any configuration for it. You need to install it yourself:

```
composer require --dev larastan/larastan
```

Then create a `phpstan.neon.dist` configuration file in your project root:

```
includes:
    - vendor/larastan/larastan/extension.neon

parameters:
    paths:
        - app/
    level: 6
    checkMissingIterableValueType: false
```

You will notice PHPStan uses two possible config filenames: `phpstan.neon.dist` and `phpstan.neon`. The `.dist` file is the one you commit to your repository — it holds the shared configuration that everyone on your team uses. The `phpstan.neon` file (without `.dist`) is for local overrides. If PHPStan finds both, it uses `phpstan.neon` and ignores the `.dist` file. This way, a developer can temporarily lower the level or ignore a path on their machine without changing the committed config. Add `phpstan.neon` to your `.gitignore` so local overrides stay local.

PHPStan has levels from 0 (loose) to 9 (strictest). Start at level 5 or 6 and work your way up:

Level	What It Checks
0	Basic checks: unknown classes, functions, methods
1	Possibly undefined variables
2	Unknown methods on `mixed` types
3	Return types
4	Dead code, unreachable statements
5	Argument types passed to methods
6	Missing type hints on properties
7	Union types handled correctly
8	Nullable types handled correctly
9	Mixed type is never used

Run the analysis:

```
./vendor/bin/phpstan analyse
```

Larastan understands Laravel-specific patterns that would confuse plain PHPStan:

```
// Larastan knows User::where() returns Builder<User>
// Larastan knows $user->posts returns HasMany<Post>
// Larastan knows config('app.name') returns string|null
// Larastan knows Route::get() accepts a closure or controller array
```

> *A word of caution: Static analysis can be annoying if it is not set up well. If you crank the level too high on a codebase that is not ready, you will get hundreds of errors and your team will just start ignoring them. If you are adding PHPStan to an existing project, start at a low level, fix what you can, and use a baseline to track the rest. Do not add it to your CI/CD pipeline until your codebase passes cleanly - a pipeline that always fails is worse than no pipeline at all. It is also worth spending some time reading the PHPStan documentation to understand how rules and levels work before you roll it out.*

Strict Types Declaration

By default, PHP silently coerces types. If a function expects an `int` and you pass the string `"3"`, PHP quietly converts it and moves on. This can mask real bugs:

```
function calculateTotal(int $quantity, float $price): float
{
    return $quantity * $price;
}

// PHP silently converts "3" to 3 — no error, no warning
calculateTotal("3", 19.99); // Returns 59.97
```

Adding `declare(strict_types=1)` at the top of a file changes this behavior. PHP will no longer coerce types for you — if the types do not match, it throws a `TypeError` immediately:

```php
<?php

declare(strict_types=1);

function calculateTotal(int $quantity, float $price): float
{
    return $quantity * $price;
}

// TypeError: Argument #1 ($quantity) must be of type int, string given
calculateTotal("3", 19.99);
```

Laravel itself does not use `declare(strict_types=1)` — the framework is designed to be flexible with type coercion. However, in your own application code, enabling strict types is worth considering. It forces you and your team to be explicit about types, which pairs well with PHPStan to catch problems early.

Whether you adopt strict types is a team decision. If you do, be consistent — apply it across all your application files rather than mixing strict and non-strict files, which can lead to confusing behavior at the boundaries.

> *A word of caution: Adding `strict_types` to an existing codebase - especially a large one - can break code that has been working fine for years. Every place where a string quietly became an integer will now throw an error. If you are starting a new project, strict types are easy to adopt from day one. If you are working on an existing codebase, introduce them gradually — file by file, with tests covering each change — rather than turning it on everywhere at once.*

Automated Refactoring with Rector

Pint fixes how your code looks. PHPStan tells you what is wrong. Rector goes a step further - it rewrites your code for you. Rector reads your PHP files, applies a set of rules, and changes the code automatically. It can rename deprecated method calls, add type declarations, replace old patterns with modern ones, and even upgrade your code from one Laravel version to the next.

Install Rector with the Laravel extension:

```
composer require --dev driftingly/rector-laravel
```

Then create a `rector.php` configuration file in your project root. The simplest setup uses Rector's `LaravelSetProvider`, which automatically detects your Laravel version from `composer.json` and applies the matching rules:

```
<?php

declare(strict_types=1);

use Rector\Config\RectorConfig;
use RectorLaravel\Set\LaravelSetProvider;

return RectorConfig::configure()
    ->withSetProviders(LaravelSetProvider::class)
    ->withComposerBased(laravel: true);
```

Run it in dry-run mode first to preview what would change:

```
./vendor/bin/rector --dry-run
```

When you are happy with the proposed changes, run it without the flag to apply them:

```
./vendor/bin/rector
```

Targeted Rule Sets

Instead of applying everything at once, you can pick specific rule sets for the improvements you care about. For example, the `LARAVEL_IF_HELPERS` set replaces verbose conditional patterns with Laravel's concise helpers:

```
// Before: manual if + abort
if (! $user->isAdmin()) {
    abort(403);
}

// After: Rector rewrites it to
abort_if(! $user->isAdmin(), 403);
```

The `LARAVEL_TYPE_DECLARATIONS` set adds return types and parameter types to your code based on how methods are actually used — a great way to prepare your codebase for higher PHPStan levels.

To use specific sets, reference them in your configuration:

```
use RectorLaravel\Set\LaravelSetList;

return RectorConfig::configure()
    ->withSets([
        LaravelSetList::LARAVEL_IF_HELPERS,
        LaravelSetList::LARAVEL_TYPE_DECLARATIONS,
    ]);
```

If you are upgrading between Laravel versions, you can target a specific version level. For example, to apply all rules up to Laravel 12:

```
use RectorLaravel\Set\LaravelLevelSetList;

return RectorConfig::configure()
    ->withSets([
        LaravelLevelSetList::UP_TO_LARAVEL_120,
    ]);
```

Rector is powerful, but it is also opinionated — always review its changes before committing. Run your tests after every Rector pass. Treat it as a very fast junior developer: it does the tedious work, but you still review the pull request.

Laravel Shift

If you want to take automated upgrades even further, Laravel Shift is a paid service built specifically for upgrading Laravel applications between major versions. While Rector applies individual code transformations, Shift handles the full upgrade - configuration changes, dependency updates, namespace adjustments, removed features, and everything documented in Laravel's upgrade guide.

The workflow is simple: you sign in with your GitHub, Bitbucket, or GitLab account, point Shift at your repository, and it opens a pull request with atomic commits and detailed comments explaining every change it made. You review the PR, run your tests, and merge.

Shift supports upgrades all the way from Laravel 4.2 to the latest release. Individual shifts cost between $9 and $39, or you can subscribe to a Shifty Plan starting at $99 per year for unlimited runs and automatic PRs whenever Laravel tags a new release.

> *Shift works best on projects that follow Laravel conventions closely. If your codebase is heavily customized or has drifted far from the standard Laravel structure, you may need to fix some things by hand after the automated run. That said, even in those cases, Shift handles most of the repetitive work and lets you focus on the parts that actually need your attention.*

Laravel Coding Style

Here is a summary of the key style rules that Taylor follows in the Laravel framework source code:

Braces and spacing:

```
// Opening brace on the same line for classes and methods
class UserController extends Controller
{
    public function index(): View
    {
        // Code here
    }
}

// Single blank line between methods
public function index(): View
{
    return view('users.index');
}

public function show(User $user): View
{
    return view('users.show', compact('user'));
}
```

Early returns to reduce nesting:

```php
// Good: early return
public function update(UpdateUserRequest $request, User $user): RedirectResponse
{
    if ($user->isLocked()) {
        return back()->with('error', 'Account is locked.');
    }

    $user->update($request->validated());

    return redirect()->route('users.show', $user);
}

// Bad: deeply nested
public function update(UpdateUserRequest $request, User $user): RedirectResponse
{
    if (! $user->isLocked()) {
        $user->update($request->validated());
        return redirect()->route('users.show', $user);
    } else {
        return back()->with('error', 'Account is locked.');
    }
}
```

Trailing commas in multi-line arrays and arguments:

```php
// Good: trailing comma makes diffs cleaner
$user = User::create([
    'name' => $request->name,
    'email' => $request->email,
    'password' => Hash::make($request->password),
]);
```

Use `!` instead of `not` and prefer `blank()` / `filled()` helpers:

```php
// Laravel style
if (! $user->isActive()) { }
if (blank($value)) { }
if (filled($value)) { }

// Not Laravel style
if ($user->isActive() === false) { }
if (empty($value) || $value === '') { }
```

Prefer string interpolation over concatenation:

```php
// Good: easy to read
$greeting = "Hello, {$user->name}. You have {$count} notifications.";

// Bad: hard to follow with dots everywhere
$greeting = 'Hello, ' . $user->name . '. You have ' . $count . ' notifications.';
```

Use curly braces `{$var}` inside double-quoted strings. For simple variables you can skip the braces, but using them consistently makes the code easier to scan — especially when accessing properties or array keys.

Ternary operators - keep them short or do not use them:

A ternary is fine for simple assignments:

```php
$status = $user->isAdmin() ? 'admin' : 'member';
```

When it gets longer, break it into multiple lines:

```php
$label = $order->isPaid()
    ? 'Completed'
    : 'Pending payment';
```

If you find yourself nesting ternaries, stop and use an `if` statement instead. Nested ternaries are hard to read and easy to get wrong:

```
// Bad: nested ternary — what does this even return?
$role = $user->isAdmin() ? 'admin' : ($user->isEditor() ? 'editor' : 'viewer');

// Good: just use if statements
if ($user->isAdmin()) {
    $role = 'admin';
} elseif ($user->isEditor()) {
    $role = 'editor';
} else {
    $role = 'viewer';
}
```

For simple null checks, use the null coalescing operator instead of a ternary:

> *Read more about ternary operator here.*

```
// Good: null coalescing
$name = $user->nickname ?? $user->name;

// Unnecessary: ternary for null check
$name = $user->nickname !== null ? $user->nickname : $user->name;
```

Use array syntax for validation rules:

When writing validation rules, use arrays instead of pipe-delimited strings. Arrays are easier to read, easier to diff in pull requests, and let you mix string rules with Rule objects without awkward syntax:

```php
// Good: array syntax
$request->validate([
    'email' => ['required', 'email', 'unique:users'],
    'name' => ['required', 'string', 'max:255'],
    'role' => ['required', Rule::in(['admin', 'editor', 'viewer'])],
]);

// Bad: pipe syntax — harder to read and cannot mix with Rule objects
$request->validate([
    'email' => 'required|email|unique:users',
    'name' => 'required|string|max:255',
]);
```

CI/CD Pipeline

Running these tools locally is great, but the real power comes when you enforce them automatically on every push and pull request. GitHub Actions makes this straightforward.

Before setting up CI, consider adding Composer scripts so your team can run the same checks locally with a single command:

```json
{
    "scripts": {
        "lint": "./vendor/bin/pint --test",
        "fix": "./vendor/bin/pint",
        "analyse": "./vendor/bin/phpstan analyse",
        "refactor": "./vendor/bin/rector --dry-run",
        "check": [
            "@lint",
            "@analyse",
            "@refactor"
        ]
    }
}
```

Now `composer check` runs Pint, PHPStan, and Rector in one go — the same checks your CI pipeline will enforce.

Automated Code Style Fixing with Pint

This workflow runs Pint on every push and pull request that touches PHP files, and automatically commits the fixes back to the branch:

```yaml
name: Fix PHP code style issues

on:
  push:
    branches: [main]
    paths:
      - '**.php'
  pull_request:
    branches: [main]
    paths:
      - '**.php'

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true

permissions:
  contents: write

jobs:
  php-code-styling:
    runs-on: ubuntu-latest
    timeout-minutes: 5

    steps:
      - name: Checkout code
        uses: actions/checkout@v6
        with:
          ref: ${{ github.head_ref }}

      - name: Fix PHP code style issues
        uses: aglipanci/laravel-pint-action@2.6

      - name: Commit changes
        uses: stefanzweifel/git-auto-commit-action@v7
        with:
          commit_message: Fix styling
```

Let's break down how this works:

- `paths: ['**.php']` - The workflow only triggers when PHP files change. There is no reason to run a PHP style fixer when you edit a README or a JavaScript file.
- `concurrency` - If you push twice in quick succession, the first run is cancelled. This saves runner minutes and prevents two runs from trying to commit style fixes at the same time.
- `permissions: contents: write` - The workflow needs write access to push the auto-fix commit back to the branch.
- `ref: ${{ github.head_ref }}` - On pull requests, this checks out the actual feature branch (not the merge commit), so the style fix commit lands on the correct branch.
- `aglipanci/laravel-pint-action@2.6` - A community action that sets up PHP and runs Pint. It handles the PHP setup so you don't have to.
- `stefanzweifel/git-auto-commit-action@v7` - After Pint runs, this action checks if any files were modified. If Pint changed something, it commits and pushes the fixes automatically. If nothing changed, it does nothing.

The result: developers never need to argue about code style in reviews. Push your code, and CI formats it for you.

Static Analysis with PHPStan

This workflow runs PHPStan to catch bugs and type errors:

```yaml
name: PHPStan

on:
  push:
    branches: [main]
    paths:
      - '**.php'
      - 'phpstan.neon.dist'
      - 'phpstan-baseline.neon'
      - '.github/workflows/phpstan.yml'
  pull_request:
    branches: [main]
    paths:
      - '**.php'
      - 'phpstan.neon.dist'
      - 'phpstan-baseline.neon'
      - '.github/workflows/phpstan.yml'

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true

jobs:
  phpstan:
    name: PHPStan
    runs-on: ubuntu-latest
    timeout-minutes: 5
    steps:
      - uses: actions/checkout@v6

      - name: Setup PHP
        uses: shivammathur/setup-php@v2
        with:
          php-version: '8.4'
          coverage: none

      - name: Install composer dependencies
        uses: ramsey/composer-install@v3
```

```
      - name: Run PHPStan
        run: ./vendor/bin/phpstan --error-format=github
```

Here is what each piece does:

- `paths` - Notice this workflow triggers on more than just PHP files. Changes to `phpstan.neon.dist` (the configuration), `phpstan-baseline.neon` (the baseline of ignored errors), and the workflow file itself all trigger a re-run. If you tighten your PHPStan rules, you want to know immediately if the codebase still passes.
- `shivammathur/setup-php@v2` - Sets up the exact PHP version your project uses. The `coverage: none` option skips installing Xdebug, which speeds up the job since PHPStan does not need code coverage.
- `ramsey/composer-install@v3` - Installs your Composer dependencies with built-in caching. On subsequent runs, it restores the `vendor/` directory from cache, making the install step near-instant.
- `--error-format=github` - This is the key flag. Instead of printing errors to the console, PHPStan outputs them in GitHub's annotation format. This means errors appear as inline annotations directly on the pull request diff, right next to the line that caused the problem.

Unlike the Pint workflow, PHPStan does not auto-fix anything - it fails the build. This is intentional. A type mismatch might mean the code is wrong, or it might mean the type hint is wrong. Only the developer can decide which one to fix.

Automated Refactoring with Rector

This workflow runs Rector in dry-run mode. It checks whether there are any refactoring rules that have not been applied yet. If there are, the build fails. The idea is that developers should run Rector locally, review what it changed, and commit on purpose — not have CI silently rewrite their code:

```yaml
name: Rector

on:
  push:
    branches: [main]
    paths:
      - '**.php'
      - 'rector.php'
      - '.github/workflows/rector.yml'
  pull_request:
    branches: [main]
    paths:
      - '**.php'
      - 'rector.php'
      - '.github/workflows/rector.yml'

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true

jobs:
  rector:
    name: Rector
    runs-on: ubuntu-latest
    timeout-minutes: 5
    steps:
      - uses: actions/checkout@v6

      - name: Setup PHP
        uses: shivammathur/setup-php@v2
        with:
          php-version: '8.4'
          coverage: none

      - name: Install composer dependencies
        uses: ramsey/composer-install@v3

      - name: Run Rector
        run: ./vendor/bin/rector --dry-run
```

Here is what each piece does:

- `paths` — Like the PHPStan workflow, this triggers on PHP files, the Rector configuration (`rector.php`), and the workflow file itself. Changing a rule set should immediately verify the codebase still passes.
- `--dry-run` — Rector checks the code and reports what it would change, but does not modify any files. If there are pending changes, the build fails.

The workflow uses the same `shivammathur/setup-php` and `ramsey/composer-install` actions as the PHPStan workflow, keeping the setup consistent across all your CI jobs.

Why Separate Workflows?

You might wonder why we don't combine Pint, PHPStan, and Rector into a single workflow. There are a few reasons:

1. **Different behaviors** — Pint auto-fixes and commits. PHPStan and Rector fail the build. Mixing these in one workflow creates confusing logic where part of the job succeeds and part fails.
2. **Independent triggers** — PHPStan triggers on `phpstan.neon.dist` changes, Rector triggers on `rector.php` changes, and Pint only needs PHP files. Keeping them separate means each workflow has clean, focused trigger rules.
3. **Clear feedback** — When a workflow fails, you immediately know what failed: style, analysis, or refactoring. A single combined workflow forces you to dig through logs to figure out which tool broke.

All three workflows use `timeout-minutes: 5` as a safety net. If something hangs, the job stops after five minutes instead of burning through your Actions quota.

With these tools in place, every chapter that follows will produce code that is not only well-structured but also clean, checked, and ready to ship. This is the foundation that makes everything else possible.

Summary

- **Code style is a foundation, not a finishing touch.** Settle formatting debates once with Laravel Pint, then automate it so nobody thinks about it again.
- **Use type hints everywhere.** Parameters, return types, and typed properties serve as documentation, enable static analysis, and catch bugs before they reach production.
- **Only write docblocks that add information.** If a docblock repeats what the type signature already says, delete it. Keep docblocks for array shapes, collection types, and thrown exceptions.
- **Static analysis catches bugs without running code.** PHPStan with Larastan understands Laravel's magic and finds problems your tests might miss. Start at a low level and work your way up.
- **Strict types are a team decision.** Laravel does not use them, but enabling `declare(strict_types=1)` in your own code forces explicit types. Add them gradually to existing projects.
- **Automate refactoring with Rector.** It rewrites your code to use modern PHP and Laravel patterns. Use Laravel Shift for full version upgrades.
- **Prefer string interpolation** over concatenation, **array syntax** over pipe-delimited validation rules, and **simple ternaries** over nested ones.
- **Keep ternaries short.** If you are nesting them, use `if` statements instead. Use the null coalescing operator (`??`) for null checks.
- **Enforce everything in CI.** Separate workflows for Pint, PHPStan, and Rector give clear feedback, independent triggers, and clean separation of concerns.

- **The code you do not write is the code that never has bugs.** Let tools handle style, types, and refactoring so you can focus on logic.

Test Your Understanding

Think you've mastered this chapter? Take the quiz and find out.

http://mayahi.test/quiz/book/clean-code-in-laravel/code-quality-and-automation

Dependency Injection

Think Clean

Dependency injection is a simple idea with a fancy name. Instead of a class creating the things it needs, you pass them in from the outside. That is it. The class says "I need this" and someone else provides it.

Here is a controller that creates its own dependency:

```
class OrderController extends Controller
{
    public function store(StoreOrderRequest $request): RedirectResponse
    {
        // The controller creates its own dependency
        $service = new OrderService();
        $order = $service->placeOrder($request->toDto());

        return redirect()->route('orders.show', $order);
    }
}
```

And here is the same controller with the dependency injected:

```
class OrderController extends Controller
{
    public function store(
        StoreOrderRequest $request,
        OrderService $service,
    ): RedirectResponse {
        $order = $service->placeOrder($request->toDto());

        return redirect()->route('orders.show', $order);
    }
}
```

The difference is one line - `new OrderService()` is gone. But that one line changes everything about how testable, flexible, and maintainable your code is.

Why `new` Is a Problem

When you write `new OrderService()` inside a class, you are making a hard decision that cannot be changed from outside. The class is permanently tied to that specific implementation. This creates three problems:

1. You cannot swap it in tests.

If `OrderService` talks to Stripe, sends emails, or queries the database, your test has no way to replace it with a fake. You are stuck testing the real thing:

```php
// This controller ALWAYS uses the real OrderService
// You cannot replace it with a mock or fake
class OrderController extends Controller
{
    public function store(StoreOrderRequest $request): RedirectResponse
    {
        $service = new OrderService(); // Hardcoded - no way to swap
        $order = $service->placeOrder($request->toDto());

        return redirect()->route('orders.show', $order);
    }
}
```

2. You cannot change behavior without editing the class.

If you need a different `OrderService` in a different context - maybe a `DiscountedOrderService` during a sale - you have to open the controller and change the code. With injection, you just bind a different class in the container.

3. You hide what the class depends on.

When dependencies are created with `new` inside methods, you have to read the entire class to figure out what it needs. When they are listed in the constructor, you can see all dependencies at a glance:

```php
// Clear: you can see all dependencies without reading the method bodies
class PlaceOrderAction
{
    public function __construct(
        private readonly OrderService $orderService,
        private readonly PaymentGateway $paymentGateway,
        private readonly InventoryService $inventoryService,
    ) {}
}
```

Dependency Injection in Plain PHP

Before looking at Laravel's tools, it helps to understand that dependency injection is not a framework feature - it is a plain PHP pattern. You can use it without any framework at all.

Imagine a `UserService` that sends welcome emails. Without injection, the class creates its own mailer:

```
class UserService
{
    public function register(string $email, string $name): void
    {
        $user = new User($email, $name);
        $user->save();

        // Hardcoded - this class decides HOW to send email
        $mailer = new SmtpMailer('smtp.example.com', 587);
        $mailer->send($email, 'Welcome!', "Hello {$name}");
    }
}
```

This works, but `UserService` is now permanently tied to `SmtpMailer`. You cannot test it without an SMTP server. You cannot switch to a different mailer without editing this class. The dependency is hidden inside the method.

With injection, you pass the mailer in from outside:

```php
class UserService
{
    public function __construct(
        private readonly MailerInterface $mailer,
    ) {}

    public function register(string $email, string $name): void
    {
        $user = new User($email, $name);
        $user->save();

        // The class does not know or care HOW the email is sent
        $this->mailer->send($email, 'Welcome!', "Hello {$name}");
    }
}

// In production
$service = new UserService(new SmtpMailer('smtp.example.com', 587));

// In tests
$service = new UserService(new FakeMailer());
```

The `UserService` no longer knows what kind of mailer it is using. It just knows it has something that can send emails. That is dependency injection - the caller decides what to provide, not the class itself.

This pattern works in any PHP project. But creating objects by hand gets tedious fast. In a real application, classes depend on other classes, which depend on other classes. Building the full chain manually is painful:

```php
// Building dependencies by hand - this gets old quickly
$httpClient = new HttpClient();
$paymentGateway = new StripePaymentGateway($httpClient);
$inventoryService = new InventoryService();
$orderService = new OrderService($paymentGateway, $inventoryService);
$controller = new OrderController($orderService);
```

That is where a container comes in.

What Is a Container?

A container is an object that knows how to build other objects. Instead of manually creating dependencies and wiring them together, you tell the container "I need an `OrderService`" and it figures out the rest - what `OrderService` needs, what those dependencies need, and so on, all the way down.

Think of it like a factory that reads blueprints. You ask for the finished product, and it assembles all the parts for you.

Here is a simplified version of how a container works under the hood:

```php
class Container
{
    private array $bindings = [];

    public function bind(string $abstract, string $concrete): void
    {
        $this->bindings[$abstract] = $concrete;
    }

    public function make(string $class): object
    {
        // If there is a binding, use it
        if (isset($this->bindings[$class])) {
            $class = $this->bindings[$class];
        }

        // Read the constructor to find out what it needs
        $reflection = new ReflectionClass($class);
        $constructor = $reflection->getConstructor();

        if ($constructor === null) {
            return new $class();
        }

        // Resolve each parameter recursively
        $dependencies = [];
        foreach ($constructor->getParameters() as $parameter) {
            $type = $parameter->getType()->getName();
            $dependencies[] = $this->make($type); // Recursive!
        }

        return $reflection->newInstanceArgs($dependencies);
    }
}
```

You would use it like this:

```php
$container = new Container();
$container->bind(PaymentGateway::class, StripePaymentGateway::class);

// The container builds StripePaymentGateway, HttpClient, and everything else
$service = $container->make(OrderService::class);
```

You do not need to build your own container - this is just to show that there is no magic involved. It is PHP reflection and recursion. Laravel's service container does the same thing, with a lot more features.

Laravel's Service Container

Laravel's service container is the engine behind the entire framework. Every time Laravel creates a controller, resolves a Form Request, runs a job, or boots a service provider, it uses the container.

The good news is that most of the time you do not need to configure anything. If `OrderService` has no special requirements, Laravel just creates it:

```php
// You never write this - the container does it automatically
$service = new OrderService();

// You just type-hint it and Laravel handles the rest
public function __construct(private readonly OrderService $service) {}
```

This is called automatic resolution. The container reads the constructor, sees what type each parameter expects, and creates the entire chain of dependencies recursively. If `OrderService` depends on `PaymentGateway`, which depends on `HttpClient`, the container builds all three - no configuration needed.

Constructor Injection

Constructor injection is the most common pattern. You declare your dependencies as constructor parameters, and Laravel injects them when the class is created:

```
class PlaceOrderAction
{
    public function __construct(
        private readonly OrderService $orderService,
        private readonly PaymentGateway $paymentGateway,
    ) {}

    public function handle(OrderData $data): Order
    {
        $order = $this->orderService->create($data);
        $this->paymentGateway->charge($order);

        return $order;
    }
}
```

Use constructor injection when the dependency is used across multiple methods, or when the class cannot function without it. The `readonly` keyword prevents the dependency from being reassigned after construction — a good safety net. We use this pattern extensively in the Actions chapter.

Method Injection

In controllers, you can also inject dependencies directly into a method. Laravel resolves them the same way:

```
class OrderController extends Controller
{
    public function store(
        StoreOrderRequest $request,
        PlaceOrderAction $action,
    ): RedirectResponse {
        $order = $action->handle($request->toDto());

        return redirect()->route('orders.show', $order);
    }

    public function index(OrderService $service): View
    {
        return view('orders.index', [
            'orders' => $service->getOrdersForUser(auth()->user()),
        ]);
    }
}
```

Method injection is useful when a dependency is only needed for one specific action. If `PlaceOrderAction` is only used in the `store` method, there is no reason to inject it into the constructor and make it available to every method. This is exactly how we inject Actions into thin controllers.

A practical guideline: if a dependency is used in two or more methods, put it in the constructor. If it is used in one method, inject it into that method.

Binding Interfaces to Implementations

Sometimes you want to type-hint an interface instead of a concrete class. This is called binding, and it is useful when you have different implementations for different contexts - or when you want to swap an implementation in tests.

Tell the container which class to use for a given interface in a service provider:

```php
// app/Providers/AppServiceProvider.php
use App\Contracts\PaymentGateway;
use App\Services\StripePaymentGateway;

public function register(): void
{
    $this->app->bind(PaymentGateway::class, StripePaymentGateway::class);
}
```

Now anywhere you type-hint `PaymentGateway`, the container gives you `StripePaymentGateway`:

```php
class PlaceOrderAction
{
    public function __construct(
        private readonly PaymentGateway $gateway, // Resolves to StripePaymentGateway
    ) {}
}
```

In tests, you can swap it out:

```php
$this->app->bind(PaymentGateway::class, FakePaymentGateway::class);
```

Do not reach for interfaces by default. As we discussed in the YAGNI chapter, only create an interface when you have a real reason — multiple implementations, or a clear testing need. A concrete class works fine until then.

Using `app()` to Resolve

Sometimes you need to resolve a class outside of a constructor or controller method - for example, in a helper function or a place where Laravel does not automatically inject dependencies. Use the `app()` helper:

```
$service = app(OrderService::class);
```

This asks the container to build `OrderService` the same way it would for constructor injection. It respects all your bindings and resolves the full dependency chain.

You can also use `app()->make()`, which does the same thing:

```
$service = app()->make(OrderService::class);
```

Use `app()` sparingly. It is a service locator - instead of declaring dependencies upfront, you reach into the container at runtime. This hides dependencies and makes code harder to follow. Prefer constructor or method injection wherever possible. Reserve `app()` for places where injection is not available.

That said, not everything needs to come from the container. Simple value objects, DTOs, and data structures are perfectly fine to create with `new`:

```
// These are fine — they are plain data, not services
$dto = new OrderData(
    items: $request->items,
    shippingAddressId: $request->shipping_address_id,
);

$money = new Money(amount: 2500, currency: 'usd');

$period = new DateRange(
    from: now()->subDays(30),
    to: now(),
);
```

The rule of thumb: if the object does something (calls an API, queries a database, sends an email), inject it. If the object holds data, create it with `new`.

Testing with Dependency Injection

The biggest practical benefit of dependency injection is testability. When dependencies are injected, you can replace them with fakes or mocks in your tests:

```
use App\Contracts\PaymentGateway;

it('places an order', function (): void {
    // Swap the real payment gateway with a fake
    $this->app->bind(PaymentGateway::class, FakePaymentGateway::class);

    $response = $this->post('/orders', [
        'items' => [
            ['product_id' => 1, 'quantity' => 2],
        ],
        'shipping_address_id' => 1,
    ]);

    $response->assertRedirect();
    expect(Order::count())->toBe(1);
});
```

Without injection, you would have to hit the real Stripe API in every test. With injection, the test is fast, isolated, and reliable.

You can also use Mockery to create a mock on the fly:

```
use App\Services\OrderService;

it('delegates to the order service', function (): void {
    $mock = Mockery::mock(OrderService::class);
    $mock->shouldReceive('placeOrder')->once()->andReturn(new Order());

    $this->app->instance(OrderService::class, $mock);

    $this->post('/orders', [/* ... */]);
});
```

The container makes all of this possible. Because the controller asks for `OrderService` instead of creating it, you can give it whatever you want in your tests.

Summary

- Dependency injection means passing dependencies in instead of creating them. A class declares what it needs, and something else provides it.
- Avoid `new` for services. When you write `new OrderService()`, you hardcode that dependency. You cannot swap it in tests, cannot change it from outside, and cannot see it in the constructor signature.
- Laravel's service container resolves dependencies automatically. Type-hint a class in a constructor or method, and the container builds it - including all of its own dependencies.
- Use constructor injection for shared dependencies and method injection for action-specific ones. If a dependency is used in two or more methods, put it in the constructor.
- Bind interfaces to implementations in a service provider when you need to swap classes - but do not create interfaces until you have a real reason.
- Use `app()` sparingly. It works, but it hides dependencies. Prefer constructor or method injection wherever possible.
- `new` is fine for data. DTOs, value objects, and plain data structures do not need the container. If it holds data, use `new`. If it does something, inject it.
- The biggest win is testability. When dependencies are injected, you can replace them with fakes in tests. Fast, isolated, reliable.

Test Your Understanding

Think you've mastered this chapter? Take the quiz and find out.

http://mayahi.test/quiz/book/clean-code-in-laravel/dependency-injection