



How I Structure Every Laravel Project

A practical map for working Laravel developers.
A starting point, not a standard.
Use what makes your next change clearer.

Small responsibilities.
Clear places to put them.

INPUT

Form Request



WORK

Action / Query



OUTPUT

Controller response

Make the responsibility easy to find.

Most of the structure is ordinary Laravel. The extra folders give recurring work a home.

```
app/  
├── Actions/  
├── DTOs/  
├── Enums/  
├── Http/  
│   ├── Controllers/  
│   ├── Middleware/  
│   └── Requests/  
├── Models/    Policies/    Providers/  
├── Services/   Queries/  
└── View/Components/
```

Actions

One named business operation.

Services

Integrations and supporting operations.

Queries

Reusable, nontrivial database reads.

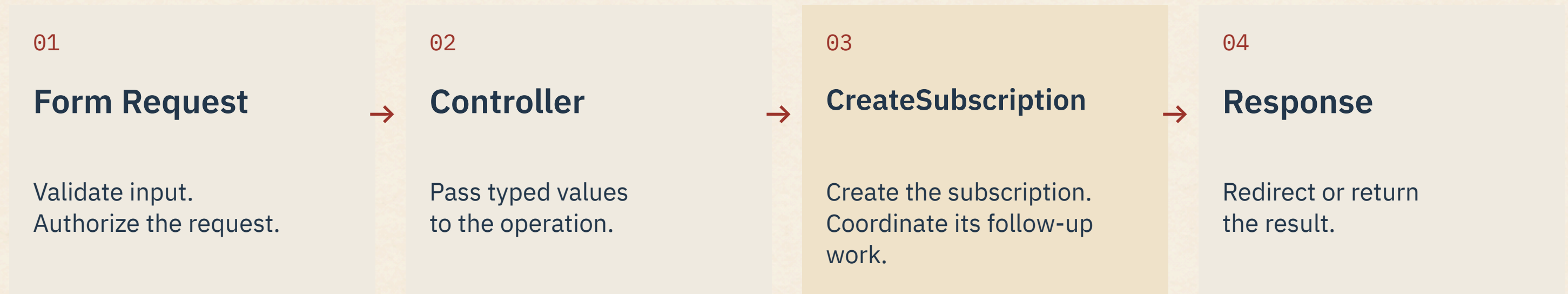
DTOs + enums

Typed inputs and a fixed set of choices.

Article: [DTOs/](#) • Current TLA: [Data/](#)

Keep the entry point thin.

A controller accepts validated input, calls the work, and returns a response.



One operation, several entry points

An HTTP controller, console command, queued job, or Filament action can call the same business operation.

Name the work, then keep it small.

The distinction is a useful default. It does not need to become a naming argument.

ACTION

CreateSubscription

One business operation.
Specific enough to describe in a sentence.
Callable from more than one entry point.

SERVICE

StripeService

An integration or supporting capability.
Wraps related operations behind one boundary.
Used by the application's business work.

Ask what the class does. "Creates a subscription" is clearer than "manages subscriptions."

Make the inputs explicit.

Use a DTO when several values travel together. Use an enum when a value has fixed choices.

DTO

SubscriptionData

Carries the plan, start date, and coupon.
A typed contract between layers.
Keep business rules in the operation.

ENUM

Plan

Monthly · Yearly · Lifetime
A named set of allowed values.
Replace scattered string comparisons.

Keep two simple parameters simple. Add a DTO when the grouped data makes the contract clearer.

Extract the query when it earns a name.

A Query class gives a complex read one place to change when several callers need it.

ActiveSubscriptionsQuery

Select active subscriptions whose end date is still in the future.

One definition

Keep shared filters together.
Avoid copying the same read
everywhere.

Several callers

Use from an action, command,
or page's data preparation.

One boundary

Return data or a query.
Keep HTTP responses outside the read.

A straightforward read can stay straightforward. Extract for complexity or reuse, not ceremony.

Test at the level where it can break.

Mirror the application’s responsibilities across Unit, Integration, and Feature suites.

Unit	Isolated logic	Enum choices, DTO values, pure calculations. No database, files, network, or queue.
Integration	Collaborating parts	An Action writes records; a Query filters them. Use the test database. Fake external boundaries.
Feature	An entry point	An HTTP request or console command. Check the response and the lasting effect.

Example: test Plan’s choices, CreateSubscription’s database effect, and the endpoint separately.

Start with the smallest useful structure.

Use the map to make the next change easier to understand, then stop adding layers.

01 Input rules

Form Request; policies for model permissions.

02 Business work

One Action, reusable by controllers and admin.

03 Integration or read

Service for a capability; Query for a reusable read.

04 Values and choices

DTO for grouped data; enum for fixed options.

05 Proof

Unit, Integration, or Feature at the right boundary.

BEFORE ADDING A LAYER

Can you name its responsibility?

Is there a real reuse or test boundary?

Would the existing Laravel pattern be enough?

[Read the full article: How I Structure Every Laravel Project](#)