General Tips

Keep it simple. Stick with our standard Laravel-esque structure (Form Requests, Controllers, Services, Resources, Enums, etc.). This is the best approach; sometimes, it makes sense to use other design patterns, such as Actions, Repositories, etc. However, these often lead to more complexity and make our code less standardized (this is already the case).

More abstractions & complexity aren't what makes good code. Write code with others in mind; how hard will it be for someone else to dive in and understand? Adding in 10 levels of abstractions & magic probably isn't adding much value for most features, only headaches for anyone trying to figure out how your code works. Focus on making complex code seem simple.

Consistency! Following our naming conventions, design patterns, database design, etc., makes our codebase scalable and easy to maintain. Aim to follow Laravel closely as well, and if you aren't sure about something, ask someone or find an existing example in our code.

Write test Cases! You must add Feature and unit tests to both Front-End and Back-End code. Tests will help you find bugs you'll never notice when testing manually. Keeping things running without proper tests is almost impossible as our codebase evolves and undergoes feature changes, additions, library upgrades, etc.

Lean towards official Laravel packages (Horizon, Passport, etc.) and those from well-known contributors (Spatie, BeyondCode, etc.). This obviously won't always be possible. We already use a few packages from lesser-known developers, but use them cautiously, as they can be poorly maintained and cause massive headaches when upgrading to a new Laravel or PHP version.

N+1 Queries

Please, let's avoid database queries in loops! Use eager loading/preload data outside the loop.

The same applies to other external operations (redis, API calls, elastic search, etc.).

Check your code in Telescope to find performance issues like this!

Bad:

foreach (Car::all() as $car) {

echo $car->engine->name;

}

Good:

foreach (Car::with('engine')->get() as $car) {

echo $car->engine->name;

}

// Also, for scenarios when you can't use Laravel eager loading

$engines = Engine::all()->indexBy('id');

foreach (Car::all() as $car) {

echo $engines->get($car->engine\_id)?->name;

}

N+1 Inserts, Updates, Deletes

In most cases, starting with the default Eloquent methods for inserting, updating, or deleting records makes sense. These methods will be the easiest to maintain and allow us to use any required model events we have configured (updating search index, cascading soft deletes, etc.). However, when dealing with a large number of records, performing single query operations in a loop is not feasible and should be changed to batch queries.

Model events can also be fired manually for records after they've been inserted, etc. For example, records are batch-inserted but must be saved to Elasticsearch with our existing model events.

Sometimes Bad:

foreach ($cars as $car) {

Car::create($car);

}

Sometimes Good:

$records = collect();

foreach ($cars as $car) {

$records->push([

'id' => Uuid::uuid4()->toString(),

'name' => $car['name'],

'user\_id' => $car['user\_id'],

]);

}

foreach ($records->chunk(500) as $record\_chunk) {

// can also use the insertOrIgnore method or some of the other batch methods included in Laravel

Car::insert($record\_chunk->toArray());

}

Database Transactions

Always use transactions when doing multiple related queries to maintain data integrity

Be careful when using many queries in a single transaction; for example, inserting millions of rows in one transaction will take forever to roll back and typically leave records locked.

The second parameter of the Transaction method is the number of retries where it will auto-retry on failure (transaction lock, etc.).

DB::transaction(function(){

$user = User::create([...]);

// be careful with this, set $afterCommit to true in the job class

SendWelcomeEmail::dispatch($user);

Team::create([

'owner\_id' => $user->id,

...

]);

}, 3);

Type Hinting

There are multiple reasons why this is important, but it is critical for readability alone! Aim always to type hints unless there's a reason not to (which should be rare). Strict types are optional; use them at your discretion.

class CoolService

{

private Collection $numbers;

// can also use property promotion, but I like it this way :)

public function \_\_construct(Collection $numbers)

{

$this->numbers = $numbers;

}

public function doSomeMath(int $first, int $second): Collection

{

return $this->numbers->map(fn ($n) => $n \* ($first + $second + 1));

}

}

Validation

Move validation from controllers to Request classes. Use $request->validated() instead of $request->all() when retrieving data

Bad:

public function store(Request $request)

{

$request->validate([

'title' => 'required|unique:posts|max:255',

'body' => 'required',

'publish\_at' => 'nullable|date',

]);

....

$data = $request->all()

}

Good:

public function store(PostRequest $request)

{

$data = $request->validated()

}

class PostRequest extends Request

{

public function rules(): array

{

return [

'title' => 'required|unique:posts|max:255',

'body' => 'required',

'publish\_at' => 'nullable|date',

];

}

}

Response Formatting

Use API Resources for response transformation!

Relations should also use API Resources here & whenLoaded.

Good:

class UserResource extends JsonResource

{

/\*\*

\* Transform the resource into an array.

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'id' => $this->id,

'name' => $this->name,

'email' => $this->email,

'created\_at' => $this->created\_at,

'updated\_at' => $this->updated\_at,

'role' => new RoleResource($this->whenLoaded('role')),

'teams' => TeamResource::collection($this->whenLoaded('teams')),

];

}

}

Business logic should be in a Service class

A controller must have only one responsibility, so move business logic from controllers to service classes.

Ideally, most database queries / complex logic should be in a Service

Bad:

public function store(Request $request)

{

if ($request->hasFile('image')) {

$request->file('image')->move(public\_path('images'));

}

....

}

Good:

public function store(Request $request)

{

$this->article\_service->handleUploadedImage($request->file('image'));

....

}

class ArticleService

{

public function handleUploadedImage($image): void

{

if (!is\_null($image)) {

$image->move(public\_path('images'));

}

}

}

Do not get data from the .env file directly

Pass the data to config files instead and then use the config() helper function to use the data in an application.

Bad:

$api\_key = env('API\_KEY');

Good:

// config/api.php

'key' => env('API\_KEY'),

// Use the data

$api\_key = config('api.key');

Prefer descriptive method and variable names over comments

Bad:

// Determine if there are any joins

if (count((array) $builder->getQuery()->joins) > 0)

Good:

if ($this->hasJoins())

Use config and language files, and constants instead of text in the code

Bad:

public function isNormal(): bool

{

return $article->type === 'normal';

}

return back()->with('message', 'Your article has been added!');

Good:

public function isNormal()

{

return $article->type === Article::TYPE\_NORMAL;

}

return back()->with('message', \_\_('app.article\_added'));

Use IoC container or facades instead of new Class

new Class syntax creates tight coupling between classes and complicates testing. Use IoC containers or facades instead.

Bad:

$user\_service = new UserService;

$user\_service->create($request->validated());

Good:

public function \_\_construct(UserService $user\_service)

{

$this->user\_service = $user\_service;

}

....

$this->user\_service->create($request->validated());

// or use the app or resolve helpers to instantiate

$this->user\_service = app(UserService::class);

Chunk data for data-heavy tasks

Bad:

$flights = Flight::all()

foreach ($flights as $flight) {

...

}

Good:

https://laravel.com/docs/master/eloquent#chunking-results

Flight::chunk(200, function ($flights) {

foreach ($flights as $flight) {

//

}

});

Try, Catch carefully

Don't be afraid to throw Exceptions, and don't blindly enclose everything in Try Catch blocks! A lot of the time, you're better off not catching exceptions and letting the Laravel exception handler do its thing. If you are catching exceptions, at least add logging and meaningful messaging!

Bad:

try {

$url = $this->getClientUrl(135135135135);

} catch (Throwable $e) {

// do nothing and fail silently, so this is impossible to debug

}

Good:

try {

$url = $this->getClientUrl(135135135135);

} catch (MyAPICallFailedException $e) {

// use native laravel file logging, these go to Cloudwatch as well :)

Log::error('My API Call failed', ['client\_id' => 135135135135]);

// can also include some meaningful user messaging with a relevant status\_code

abort(500, "Hey this one thing is broken, we'll fix it");

}

Prefer to use Eloquent over using Query Builder and raw SQL queries. Prefer collections over arrays

Eloquent allows you to write readable and maintainable code. It also has excellent built-in tools, such as soft deletes, events, scopes, etc.

Sometimes, the QueryBuilder is preferred over Eloquent (complex reports, dashboards, etc.) But lean towards Eloquent.

Bad:

SELECT \*

FROM `articles`

WHERE EXISTS (SELECT \*

FROM `users`

WHERE `articles`.`user\_id` = `users`.`id`

AND EXISTS (SELECT \*

FROM `profiles`

WHERE `profiles`.`user\_id` = `users`.`id`)

AND `users`.`deleted\_at` IS NULL)

AND `verified` = '1'

AND `active` = '1'

ORDER BY `created\_at` DESC

Good:

Article::has('user.profile')->verified()->latest()->get();