You might also want to check out the real-world Laravel example application

Contents

Single responsibility principle

A class and a method should have only one responsibility.

Bad:

```
public function getFullNameAttribute()
{
    if (auth()->user() && auth()->user()->hasRole('client') && auth()->user()->isVerified()) {
        return 'Mr. ' . $this->first_name . ' ' . $this->middle_name . ' ' . $this->last_name;
    } else {
        return $this->first_name[0] . '. ' . $this->last_name;
    }
}
```

Good:

```
public function getFullNameAttribute(): bool
{
    return $this->isVerifiedClient() ? $this->getFullNameLong() : $this->getFullNameShort();
}

public function isVerifiedClient(): bool
{
    return auth()->user() && auth()->user()->hasRole('client') && auth()->user()->isVerified();
}

public function getFullNameLong(): string
{
    return 'Mr. ' . $this->first_name . ' ' . $this->middle_name . ' ' . $this->last_name;
}

public function getFullNameShort(): string
{
    return $this->first_name[0] . '. ' . $this->last_name;
}
```

Back to contents

Fat models, skinny controllers

Put all DB related logic into Eloquent models.

Bad:

```
public function index()
{
    $clients = Client::verified()
        ->with(['orders' => function ($q) {
            $q->where('created_at', '>', Carbon::today()->subWeek());
        }])
        ->get();

    return view('index', ['clients' => $clients]);
}
```

Good:

```
public function index()
{
    return view('index', ['clients' => $this->client->getWithNewOrders()]);
}

class Client extends Model
{
    public function getWithNewOrders(): Collection
    {
        return $this->verified()
            ->with(['orders' => function ($q) {
                $q->where('created_at', '>', Carbon::today()->subWeek());
            }])
            ->get();
    }
}
```

Back to contents

Validation

Move validation from controllers to Request classes.

Bad:

```
public function store(Request $request)
{
    $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
        'publish_at' => 'nullable|date',
    ]);

    ....
}
```

Good:

```
public function store(PostRequest $request)
{
    ....
}

class PostRequest extends Request
{
    public function rules(): array
    {
        return [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
            'publish_at' => 'nullable|date',
        ];
    }
}
```

Back to contents

Business logic should be in service class

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

Bad:

```
public function store(Request $request)
{
    if ($request->hasFile('image')) {
        $request->file('image')->move(public_path('images') . 'temp');
    }

    ....
}
```

Good:

```
public function store(Request $request)
{
    $this->articleService->handleUploadedImage($request->file('image'));

    ....
}

class ArticleService
{
    public function handleUploadedImage($image): void
    {
        if (!is_null($image)) {
            $image->move(public_path('images') . 'temp');
        }
    }
}
```

Back to contents

Don't repeat yourself (DRY)

Reuse code when you can. SRP is helping you to avoid duplication. Also, reuse Blade templates, use Eloquent scopes etc.

Bad:

```
public function getActive()
{
    return $this->where('verified', 1)->whereNotNull('deleted_at')->get();
}

public function getArticles()
{
    return $this->whereHas('user', function ($q) {
            $q->where('verified', 1)->whereNotNull('deleted_at');
        })->get();
}
```

Good:

```
public function scopeActive($q)
{
    return $q->where('verified', true)->whereNotNull('deleted_at');
}

public function getActive(): Collection
{
    return $this->active()->get();
}

public function getArticles(): Collection
{
    return $this->whereHas('user', function ($q) {
            $q->active();
        })->get();
}
```

Back to contents

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. Also, Eloquent has great built-in tools like soft deletes, events, scopes etc.

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();
```

Back to contents

Mass assignment

Bad:

```
$article = new Article;
$article->title = $request->title;
$article->content = $request->content;
$article->verified = $request->verified;
// Add category to article
$article->category_id = $category->id;
$article->save();
```

Good:

```
$category->article()->create($request->validated());
```

Back to contents

Do not execute queries in Blade templates and use eager loading (N + 1 problem)

Bad (for 100 users, 101 DB queries will be executed):

```
@foreach (User::all() as $user)
    {{ $user->profile->name }}
@endforeach
```

Good (for 100 users, 2 DB queries will be executed):

```
$users = User::with('profile')->get();

...

@foreach ($users as $user)
    {{ $user->profile->name }}
@endforeach
```

Back to contents

Chunk data for data-heavy tasks

Bad ():

```
$users = $this->get();

foreach ($users as $user) {
    ...
}
```

Good:

```
$this->chunk(500, function ($users) {
    foreach ($users as $user) {
        ...
    }
});
```

Back to contents

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())
```

Back to contents

Do not put JS and CSS in Blade templates and do not put any HTML in PHP classes

Bad:

```
let article = `{{ json_encode($article) }}`;
```

Better:

```
<input id="article" type="hidden" value='@json($article)'>

Or

<button class="js-fav-article" data-article='@json($article)'>{{ $article->name }}<button>
```

In a Javascript file:

```
let article = $('#article').val();
```

The best way is to use specialized PHP to JS package to transfer the data.

Back to contents

Use config and language files, 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'));
```

Back to contents

Use standard Laravel tools accepted by community

Prefer to use built-in Laravel functionality and community packages instead of using 3rd party packages and tools. Any developer who will work with your app in the future will need to learn new tools. Also, chances to get help from the Laravel community are significantly lower when you're using a 3rd party package or tool. Do not make your client pay for that.

Task	Standard tools	3rd party tools
Authorization	Policies	Entrust, Sentinel and other packages
Compiling assets	Laravel Mix	Grunt, Gulp, 3rd party packages
Development Environment	Laravel Sail, Homestead	Docker
Deployment	Laravel Forge	Deployer and other solutions
Unit testing	PHPUnit, Mockery	Phpspec

Task	Standard tools	3rd party tools
Browser testing	Laravel Dusk	Codeception
DB	Eloquent	SQL, Doctrine
Templates	Blade	Twig
Working with data	Laravel collections	Arrays
Form validation	Request classes	3rd party packages, validation in controller
Authentication	Built-in	3rd party packages, your own solution
API authentication	Laravel Passport, Laravel Sanctum	3rd party JWT and OAuth packages
Creating API	Built-in	Dingo API and similar packages
Working with DB structure	Migrations	Working with DB structure directly
Localization	Built-in	3rd party packages
Realtime user interfaces	Laravel Echo, Pusher	3rd party packages and working with WebSockets directly
Generating testing data	Seeder classes, Model Factories, Faker	Creating testing data manually
Task scheduling	Laravel Task Scheduler	Scripts and 3rd party packages
DB	MySQL, PostgreSQL, SQLite, SQL Server	MongoDB

Back to contents

Follow Laravel naming conventions

Follow PSR standards.

Also, follow naming conventions accepted by Laravel community:

What	How	Good	Bad
Controller	singular	ArticleController	~~ArticlesController~~
Route	plural	articles/1	~~article/1~~
Named route	snake_case with dot notation	users.show_active	~~users.show-active, show-active-users~~
Model	singular	User	~~Users~~
hasOne or belongsTo relationship	singular	articleComment	~~articleComments, article_comment~~
All other			~~articleComment,

relationships What	plural How	articleComments Good	article_comments~~ Bad
Table	plural	article_comments	~~article_comment, articleComments~~
Pivot table	singular model names in alphabetical order	article_user	~~user_article, articles_users~~
Table column	snake_case without model name	meta_title	~~MetaTitle; article_meta_title~~
Model property	snake_case	$model->created_at	~~$model->createdAt~~
Foreign key	singular model name with _id suffix	article_id	~~ArticleId, id_article, articles_id~~
Primary key	-	id	~~custom_id~~
Migration	-	2017_01_01_000000_create_articles_table	~~2017_01_01_000000_articles~~
Method	camelCase	getAll	~~get_all~~
Method in resource controller	table	store	~~saveArticle~~
Method in test class	camelCase	testGuestCannotSeeArticle	~~test_guest_cannot_see_article~~
Variable	camelCase	$articlesWithAuthor	~~$articles_with_author~~
Collection	descriptive, plural	$activeUsers = User::active()->get()	~~$active, $data~~
Object	descriptive, singular	$activeUser = User::active()->first()	~~$users, $obj~~
Config and language files index	snake_case	articles_enabled	~~ArticlesEnabled; articles-enabled~~
View	kebab-case	show-filtered.blade.php	~~showFiltered.blade.php, show_filtered.blade.php~~
Config	snake_case	google_calendar.php	~~googleCalendar.php, google-calendar.php~~
Contract (interface)	adjective or noun	AuthenticationInterface	~~Authenticatable, IAuthentication~~

Trait	adjective	Notifiable	~~NotificationTrait~~
What	**How**	**Good**	**Bad**

Back to contents

Use shorter and more readable syntax where possible

Bad:

```
$request->session()->get('cart');
$request->input('name');
```

Good:

```
session('cart');
$request->name;
```

More examples:

Common syntax	Shorter and more readable syntax
`Session::get('cart')`	`session('cart')`
`$request->session()->get('cart')`	`session('cart')`
`Session::put('cart', $data)`	`session(['cart' => $data])`
`$request->input('name'), Request::get('name')`	`$request->name, request('name')`
`return Redirect::back()`	`return back()`
`is_null($object->relation) ? null : $object->relation->id`	`optional($object->relation)->id` (in PHP 8: `$object->relation?->id`)
`return view('index')->with('title', $title)->with('client', $client)`	`return view('index', compact('title', 'client'))`
`$request->has('value') ? $request->value : 'default';`	`$request->get('value', 'default')`
`Carbon::now(), Carbon::today()`	`now(), today()`
`App::make('Class')`	`app('Class')`
`->where('column', '=', 1)`	`->where('column', 1)`
`->orderBy('created_at', 'desc')`	`->latest()`
`->orderBy('age', 'desc')`	`->latest('age')`
`->orderBy('created_at', 'asc')`	`->oldest()`
`->select('id', 'name')->get()`	`->get(['id', 'name'])`
`->first()->name`	`->value('name')`

Back to contents

Use IoC container or facades instead of new Class

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

Bad:

```
$user = new User;
$user->create($request->validated());
```

Good:

```
public function __construct(User $user)
{
    $this->user = $user;
}

....

$this->user->create($request->validated());
```

Back to contents

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:

```
$apiKey = env('API_KEY');
```

Good:

```
// config/api.php
'key' => env('API_KEY'),

// Use the data
$apiKey = config('api.key');
```

Back to contents

Store dates in the standard format. Use accessors and mutators to modify date format

Bad:

```
{{ Carbon::createFromFormat('Y-d-m H-i', $object->ordered_at)->toDateString() }}
{{ Carbon::createFromFormat('Y-d-m H-i', $object->ordered_at)->format('m-d') }}
```

Good:

```
// Model
protected $dates = ['ordered_at', 'created_at', 'updated_at'];
public function getSomeDateAttribute($date)
{
    return $date->format('m-d');
}

// View
{{ $object->ordered_at->toDateString() }}
{{ $object->ordered_at->some_date }}
```

Back to contents

Other good practices

Avoid using patterns and tools that are alien to Laravel and similar frameworks (i.e. RoR, Django). If you like Symfony (or Spring) approach for building apps, it's a good idea to use these frameworks instead.

Never put any logic in routes files.

Minimize usage of vanilla PHP in Blade templates.

Use in-memory DB for testing.

Do not override standard framework features to avoid problems related to updating the framework version and many other issues.

Use modern PHP syntax where possible, but don't forget about readability.

Avoid using View Composers and similar tools unless you really know what you're doing. In most cases, there is a better way to solve the problem.

Back to contents