Laravel & Lumen Guidelines

> *This is a living document and can be modified by anyone who wish to add or improve something. New ideas, improvements and suggestions are always welcome.*
>
> *The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.*

Directory Structure

In order to make our code more structured we split some sections into different folders.

```
app
 |- Clients
 |- Services
 |- Models
```

1. Clients: All external api integration should go in this folder. For instance, if it’s TMS integration we can have file called *TmsClient.php*. This serves same as service but only for external APIs.
2. Services: All business logic should be only in service.
3. Models: All connection/query to the database, entities, relationships should be in model and MUST NOT have other business logics.

PS: We are not using repository layer as majority of our devs were against it.

No Facades

We MUST use dependency injection everywhere.

One of the primary benefits of dependency injection is the ability to swap implementations of the injected class. This is useful during testing since you can inject a mock or stub and assert that various methods were called on the stub.

Another benefit of constructor injection is that it becomes obvious when your class is doing to much, because the constructor parameters grow too large. So we will know that we need to split up controllers that have too many responsibilities.

Explicit Checks

These type of checks are NOT allowed. You must throw and handle proper exception.

```
$user = User::find($request->input('user_id'));
if ($user === null) {
    return response()->json(['error' => self::ERROR_USER_NOT_EXISTS])
        ->setStatusCode(Response::HTTP_BAD_REQUEST);
}
```

For above case one line will be sufficient. The exception can be handled using Laravel/Lumen exception handler.

```
$user = $this->user->findOrFail($request->input('user_id'));
```

Configuration

Configuration files MUST use *kebab-case* and configuration keys MUST use *snake_case*.

```
// config/request-logger.php
return [
    'log_level' => 'warning',
];
```

Artisan Commands

The names given to artisan commands SHOULD all be *kebab-cased*.

```
// Good
php artisan delete-old-records

// Bad
php artisan deleteOldRecords
```

A command should always give some feedback on what the result is. Minimally you should let the handle method spit out a comment at the end indicating that all went well.

```
// in a Command
public function handle()
{
    // do some work
    $this->comment('All ok!');
}
```

If possible use a descriptive success message eg. *Old records deleted*.

Controllers

Controllers that control a resource MUST use the plural resource name.

```
class PostsController
{
    // ...
}
```

Try to keep controllers simple and stick to the default CRUD keywords (*index* , *create* , *store* , *show* , *edit* , *update* , *destroy*). Extract a new controller if you need other actions.

In the following example, we could have *PostsController@favorite*, and *PostsController@unfavorite*, or we could extract it to a seperate *FavoriteP ostsController.php*.

```
class PostsController
{
    public function create()
    {
        // ...
    }

    // ...

    public function favorite(Post $post)
    {
        request()->user()->favorites()->attach($post);
        return response(null, 200);
    }

    public function unfavorite(Post $post)
    {
        request()->user()->favorites()->detach($post);
        return response(null, 200);
    }
}
```

Here we fall back to default CRUD words, create and destroy .

```
class FavoritePostsController
{
    public function create(Post $post)
    {
        request()->user()->favorites()->attach($post);
        return response(null, 200);
    }

    public function destroy(Post $post)
    {
        request()->user()->favorites()->detach($post);
        return response(null, 200);
    }
}
```

Note that this is a loose guideline that doesn't need to be enforced.

Views

View files MUST use *snake_case*.

```
resources/
  views/
    user_report.blade.php

class ReportsController
{
    public function index()
    {
        return view('user_report');
    }
}
```

Validation

All user inputs MUST be validated and cleaned properly before using. Custom validation rules MUST use *snake_case*:

```
Validator::extend('organisation_type', function ($attribute, $value) {
    return $this->organisationType->isValid($value);
});
```

More details on validation implementation can be found in the doc below.

.

Routing

Public-facing urls MUST use *kebab-case*.

```
https://example.com/our-story
https://example.com/about-us/what-we-do
```

Use named routes everywhere. Route names MUST use *kebab-case*.

```
Route::get('our-story', 'AboutUsController@index')->name('our-story');

<a href="{{ route('our-story') }}">
    Our Story
</a>
```