**Coding standard to be followed**

# Environment

## Local Environment

There are multiple ways to run Laravel in local environment as outlined below:

* [Homestead](https://laravel.com/docs/5.8/homestead) - requires VirtualBox, Vagrant
* [Valet](https://laravel.com/docs/5.8/valet) - only runs in MacOS
* [Laradock](https://laradock.io/) - requires Docker Engine

You are free to choose which local environment that works for you as long as the following conditions are met:

* Your local environment MUST be able to duplicate the production server or close to its specifications such as PHP version, extensions installed and MySQL version
* .env file MUST NOT be committed into repository
* You SHOULD NOT be connecting to your any production server when you are debugging locally, to prevent accidental corruption of data, unintended API calls or similar incidents.
* You SHOULD NOT be using personally identifiable information (PII) of your end-users data or any data that could potentially identify a specific individual such as first/last name, address, medical condition so on and so forth, unless you are explicitly authorized by your company or client to do so.
* You MUST update the readme.md file for any special instruction on how to run the app in a local environment, so that other developers who will setup the app in their local machine can follow them properly.
* While it is possible to WAMP or XAMPP for Laravel, this is uncommon practice so you SHOULD try to familiarize yourself on how server components work and be comfortable in dealing with them.

## Staging Environment

Staging servers is a type of server that is used to test a software, website or service in a production-similar environment before being set live. It is the role of a staging environment or staging site, to serve as a temporary hosting and testing server for any new software or feature.

* It is RECOMMENDED to use [Continuous Integration](https://www.thoughtworks.com/continuous-integration) to automatically run your Tests and keep a record of the results. For example [Travis CI](https://travis-ci.org/) or [Jenkins](https://jenkins.io/)

## Production Environment

### You MUST regularly rotate your APP\_KEY

APP\_KEYS are set when you initialized a new Laravel application or executed the following command

php artisan key:generate

Laravel uses the key for all encrypted cookies, including the session cookie, before handing them off to the user’s browser, and it uses it to decrypt cookies read from the browser. This prevents the client from making changes to their cookies and granting themselves admin privileges or impersonating another user in your application. Encrypted cookies are an important security feature in Laravel

From [APP\_KEY And You](https://tighten.co/blog/app-key-and-you)

# Configuration

## Environment Variables

### You MUST put sensitive information into .env files

Use .env files to store any secure information and retrieve it via env function. There should be no instance on which you will put it inside models/controllers and commit it to Git.

Good

// .env

API\_HOST=https://example.com/api

API\_USERNAME=myuser

API\_PASSWORD=secret

// access the value from app/config.php file

return [

...

'api\_host' => env('API\_HOST', 'https://defaultdomain.com')

'api\_username' => env('API\_USER', 'defaultuser')

'api\_password' => env('API\_USER', 'defaultpassword')

...

]

Bad

define('API\_HOST', 'https://defaultdomain.com');

define('API\_USERNAME', 'defaultuser');

define('API\_PASSWORD', 'defaultpassword');

class DomainController extends Controller

{

public function index()

{

$api\_username

}

### your application key MUST be set. This is the APP\_KEY variable in your .env file. You can generate one via

php artisan key:generate

## Package Configuration

### Custom or Package configuration filename MUST be in snake\_case

Good

config/my\_config.php

Bad

config/MyConfig.php

### Config and language files indexes SHOULD be in snake\_case

Good

// config/myconfig.php

return [

'my\_api' => [

'domain' => env('API\_DOMAIN'),

'secret' => env('API\_SECRET'),

],

Bad

// config/myconfig.php

return [

'MyApi' => [

'DOMAIN' => env('API\_DOMAIN'),

'SECRET' => env('API\_SECRET'),

],

The best way to figure out if you have implemented best-practice in configuring your app, is if the codebase could be made open source at any moment without compromising any credentials

# 3. Naming Conventions

The following is the generally accepted naming conventions being used by Laravel Community:

## Controllers

### Controller name MUST start with a noun (in singular form) followed by the word “Controller”.

Good

class ArticleController extends Controller

{

Bad

class ArticlesController extends Controller

{

class wp\_articlesController extends Controller

{

class Article extends Controller

{

## You SHOULD Use Resource Controllers unless you have any particular reason not to do so

Good

class DomainController extends Controller

{

public function index(){} // list domains

public function create(){} // show create form

public function store(Request $request){ } // handle the form POST

public function show($id){} // show a single domain

public function edit($id){} // show edit page

public function update(Request $request, $id){} // handle show edit page POST

public function destroy($id){} // delete a domain

}

Bad

class DomainController extends Controller

{

public function list(){} // list domains

public function create\_or\_save(){} // show create form then handle save

public function show\_edit($id){} // show a single domain then show edit page

public function delete($id){} // delete a domain

}

## Models

### Model names MUST be in singular form with its first letter in uppercase

Good

class Flight extends Model

{

...

Bad

class Flights extends Model

{

...

class flight extends Model

{

...

### hasOne or belongsTo relationship methods MUST be in singular form

Good

class User extends Model

{

public function phone()

{

return $this->hasOne('App\Phone');

}

}

Bad

class User extends Model

{

public function phones()

{

return $this->hasOne('App\Phone');

}

}

### Any other relationships other than above MUST be in plural form

Good

class Post extends Model

{

public function comments()

{

return $this->hasMany('App\Comment');

}

}

Bad

class Post extends Model

{

public function comment()

{

return $this->hasMany('App\Comment');

}

}

### Model properties should be in snake\_case

Good

$user->created\_at

Bad

$user->createdAt

#### Methods should be in camelCase

Good

class User extends Model

{

public function scopePopular($query)

{

return $query->where('votes', '>', 100);

}

Bad

class User extends Model

{

public function scope\_popular($query)

{

return $query->where('votes', '>', 100);

}

## Functions

Laravel comes with a lot of useful helper functions, but you can also define your own helper functions, given the following conditions:

### You SHOULD place your custom helper functions by creating a file called helper.php

Good

project\_folder/app/helper.php

project\_folder/app/Http/helper.php

Bad

project\_folder/functions.php

### You MUST use Composer’s autoloading capability to load your functions

Good

// file composer.json

...

"autoload": {

"files": [

"app/helpers.php"

],

...

Bad

// file app/Http/Controllers/HomeController.php

class HomeController.php

{

function index(){

require\_once(app\_path("helpers.php"));

}

}

### You MUST check if the the function exists before defining it

Good

if (! function\_exists('my\_custom\_helper')) {

function my\_custom\_helper($key, $default = null) {

// ...

}

}

Bad

function my\_custom\_helper($key, $default = null) {

// ...

}

### Other General guides with functions

* If the function length exceeds 25 lines, you SHOULD break it down to multiple functions
* Each function SHOULD have a Unit Test associated with it

## Routes

### Routes should be in plural form of the resource it is trying to manipulate and SHOULD be all lower-case

Good

Route::get('/users', 'UserController@index');

Route::resource('photos', 'PhotoController');

Bad

Route::get('/user', 'UserController@index');

Route::get('/UsersList', 'UserController@index');

Route::resource('PHOTO', 'PhotoController');

### Named Routes SHOULD use snake\_case and dot notation

Good

Route::get('/user', 'UserController@active')->name('users.show\_active');

Bad

Route::get('/user', 'UserController@active')->name('users.show-active');

Route::get('/user', 'UserController@active')->name('show-active-users');

## Variables

### General rule for variable is it SHOULD be in camelCase

Good

$articlesWithAuthor

Bad

$articles\_with\_author

### Collection names SHOULD be descriptive and in plural form

Good

$activeUsers = User::active()->get()

Bad

$users = User::active()->get()

$user = User::active()->get()

$User = User::active()->get()

### Single Object SHOULD be descriptive and in singular form

Good

$activeUser = User::active()->first()

Bad

$users = User::active()->first()

## Views

### You SHOULD use snake\_case as file name of your Blade templates

Good

show\_filtered.blade.php

Bad

showFiltered.blade.php

show-filtered.blade.php

### You MUST not make non UI-related operations inside blade templates

Good

// $api\_results is passed by controller

<ul>

@foreach($api\_results as $result)

<li>{{ $result->name }}</li>

@endforeach

</ul>

Bad

@php

$api\_results = json\_decode(file\_get\_contents("https://api.example.com"));

@endphp

<ul>

@foreach($api\_results as $result)

<li>{{ $result->name }}</li>

@endforeach

</ul>

# 4. Database Conventions

## Table and Fields Naming

### Table names MUST be in plural form and MUST be all lower-case

Good

class CreateFlightsTable extends Migration

{

public function up()

{

Schema::create('flights', function (Blueprint $table) {

Bad

class CreateFlightsTable extends Migration

{

public function up()

{

Schema::create('flight', function (Blueprint $table) {

class CreateUsersTable extends Migration

{

public function up()

{

Schema::create('MyUsers', function (Blueprint $table) {

### Pivot table names MUST be in singular model names in alphabetical order

Good

post\_user

article\_user

photo\_post

Bad

posts\_users

user\_articles

post\_photos

### Table column names SHOULD be in snake\_case without the model name

Good

username

title

thumb\_url

Bad

UserName

\_title

ThumbUrl

post\_title

### Foreign keys MUST be singular model name with \_id suffix

Good

user\_id

Bad

userid

siteid

Memberid

TransactionID

### Primary Keys SHOULD be “id”

Good

id

Bad

ID

pkid

guid

## Database Alterations

### You MUST not be changing the database schema directly, use Database Migrations instead

Good

php artisan migrate

Bad

* use of PHPMyAdmin
* directly executing ALTER statement in mysql console / cli
* using sql file to change the db

### Migration filenames MUST follow to following pattern

creation of table

yyyy\_mm\_dd\_<timestamp>\_create\_<table name>\_table

Good

2019\_06\_06\_164210\_create\_domains\_table.php

Bad

2019\_06\_06\_164210\_domains.php

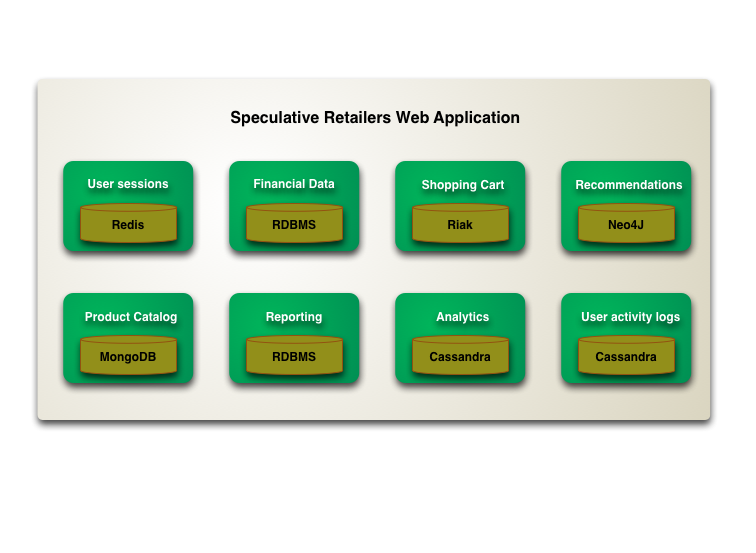
## Database Choice

### Polyglot Persistence

Is a practice of using different data storage technologies for different kinds of data. Eloquent ORM can support multiple databases for a reason, so don’t limit yourself to MySQL.

* It is RECOMMENDED to use [MongoDB](https://github.com/jenssegers/Laravel-MongoDB) for records that have attributes that vary a lot. For example, in an inventory system, an office supplies product might have a different set of fields compared to vehicle and auto supplies.
* It is RECOMMENDED to use [ElasticSearch](https://github.com/elasticquent/Elasticquent) for high volume data searching and indexing.
* It is RECOMMENDED to use [Neo4J](https://github.com/Vinelab/NeoEloquent) for applications that require complex relationships between models. For example a multi-level networking application, social network site and similar apps.

From this [article](https://martinfowler.com/bliki/PolyglotPersistence.html), here is a sample breakdown of different databases being used by a retailer company



# 5. Design Patterns

## SOLID

SOLID is five design principles intended to make software designs more understandable, flexible and maintainable.

### Single responsibility principle

A class should only have a single responsibility, that is, only changes to one part of the software’s specification should be able to affect the specification of the class.

### Open–closed principle

Software entities … should be open for extension, but closed for modification.

### Liskov substitution principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

### Interface segregation principle

Many client-specific interfaces are better than one general-purpose interface.

### Dependency inversion principle

One should depend upon abstractions, [not] concretions.

## Repository Pattern

The idea with this pattern is to have a generic abstract way for the app to work with the data layer without being bothered what storage technology is used when saving/retrieving the data.

We suggest first this [tutorial](https://medium.com/@jsdecena/refactor-the-simple-tdd-in-laravel-a92dd48f2cdd) for in-depth understanding about this design pattern.

### When reading/writing data, it is RECOMMENDED to wrap it into Repository Object

# 6. Testing

## Unit Testing

### Methods in test classes MUST start with “test” then a camelCased name of the test

Good

class ExampleTest extends TestCase

{

public function testBasicTest()

{

Bad

class ExampleTest extends TestCase

{

public function test\_basic\_test()

{

# 7. Extra general rules

* **Limited use of globals:**These rules tell about which types of data that can be declared global and the data that can’t be.
* **Standard headers for different modules:**  
  For better understanding and maintenance of the code, the header of different modules should follow some standard format and information. The header format must contain below things that is being used in various companies:
  + Name of the module
  + Date of module creation
  + Author of the module
  + Modification history
  + Synopsis of the module about what the module does
  + Different functions supported in the module along with their input output parameters
  + Global variables accessed or modified by the module
* **Naming conventions for local variables, global variables, constants and functions:**Some of the naming conventions are given below:
  + Meaningful and understandable variable name helps anyone to understand the reason for using it.
  + Local variables should be named using camel case lettering starting with a small letter (e.g. localData) whereas Global variables names should start with a capital letter (e.g. GlobalData). Constant names should be formed using capital letters only (e.g. CONSDATA).
  + It is better to avoid the use of digits in variable names.
  + The names of the function should be written in camel case starting with small letters.
  + The name of the function must describe the reason for using the function clearly and briefly.
* **Indentation:**Proper indentation is very important to increase the readability of the code. For making the code readable, programmers should use White spaces properly. Some of the spacing conventions are given below:
  + There must be a space after giving a comma between two function arguments.
  + Each nested block should be properly indented and spaced.
  + Proper Indentation should be there at the beginning and at the end of each block in the program.
  + All braces should start from a new line and the code following the end of braces also starts from a new line.
* **Error return values and exception handling conventions:**All functions that encounter an error condition should either return a 0 or 1 for simplifying the debugging. On the other hand, Coding guidelines give some general suggestions regarding the coding style that to be followed for the betterment of understandability and readability of the code.
* **Avoid using a coding style that is too difficult to understand:**Code should be easily understandable. The complex code makes maintenance and debugging difficult and expensive.
* **Avoid using an identifier for multiple purposes:**  
  Each variable should be given a descriptive and meaningful name indicating the reason behind using it. This is not possible if an identifier is used for multiple purposes and thus it can lead to confusion to the reader. Moreover, it leads to more difficulty during future enhancements.
* **Code should be well documented:**  
  The code should be properly commented for understanding easily. Comments regarding the statements increase the understandability of the code.
* **Length of functions should not be very large:**  
  Lengthy functions are very difficult to understand. That’s why functions should be small enough to carry out small work and lengthy functions should be broken into small ones for completing small tasks.
* **Try not to use GOTO statement:**  
  GOTO statement makes the program unstructured, thus it reduces the understandability of the program and also debugging becomes difficult.