# Backend Guidelines

Welcome to the comprehensive guide on Laravel PHP coding standards tailored specifically for the backend developers in our tech team. Consistency and clarity in code are paramount for efficient collaboration and seamless maintenance of projects. This wiki aims to provide clear guidelines and best practices to ensure that our Laravel PHP projects adhere to industry standards and maintain high quality.

## 1. Naming Standard

a) **Controllers/Model**: Controllers or models should adhere to PascalCase/CapitalCase and be in singular form, suffixed with "Controller". Each word should be capitalized, with no spacing between them. (This format also includes Seeder, FormRequest, Enum, Trait(adjective), Interface)

*Example:* `BlogController` , `AuthController` , `User` , `UserInvoice`

b) **Variables/Function**: All of this should generally be in camelCase, with the first character lowercase. Plural form is preferred for arrays or collections. (This format also models relationships)

*Example:* `$users` , `$bannedUsers`

c) **Database Tables** : Database tables should be in lowercase with underscores to separate words (snake_case) and should be in plural form.

*Example:* `posts` , `project_tasks` , `uploaded_images`

d) **Pivot Tables**: Pivot tables should be in lowercase, with models in alphabetical order, separated by underscores (snake_case).

*Example:* `post_user` , `task_user`

e) **Table Columns/Configs**: Table column names should be in snake_case, in lowercase, without referencing the table name.

*Example:* `post_body`, `id`, `created_at`, `google_calendar.php`

## 2. FormRequest Class

Request classes in Laravel serve as intermediaries between incoming HTTP requests and controller methods. These classes extend Laravel's base request class and allow developers to define validation rules, authorize the request, and perform additional processing before passing control to the corresponding controller method.

When a request is received, Laravel automatically resolves the appropriate request class based on route parameters and invokes its validation process. This process involves checking incoming data against defined validation rules and handling any validation errors before executing the controller method.

- can create it using command

`php artisan make:request StoreUserRequest`

- Example :

```php
<?

namespace App\Http\Requests;

use Illuminate\Support\Str;
use Illuminate\Foundation\Http\FormRequest;

class StoreUserRequest extends FormRequest
{
    // Determine if the user is authorized to make this request
    public function authorize()
    {
        return true;
    }

    // Validation rules can be defined here
    public function rules()
```

```php
    {
        return [
            'name' => 'required|string|max:255',
            'email' => 'required|email|unique:users|max:255',
            'password' => 'required|string|min:8|confirmed',
        ];
    }

    // Custom error messages can be defined here
    public function messages()
    {
        return [
            'name.required' => 'The name field is required.',
            'email.required' => 'The email field is required.',
        ];
    }

    // Custom attribute names can be defined here
    public function attributes()
    {
        return [
            'email' => 'E-mail address',
            'password' => 'Password',
        ];
    }

    // Manipulate the request data before validation occurs
    protected function prepareForValidation()
    {
        $this->merge([
            'phone_number' => preg_replace('/[^0-9]/', '', $this->phone_number),
        ]);
    }

    // Perform additional processing on the validated data before passed to the controller method
    public function validated($keys = null, $default = null)
    {
        $validatedData = parent::validated($keys, $default); // Retrieve the validated data

        $steps = $validatedData['steps'] ?? [];

        foreach ($steps as $key => $step) {
            if (isset($step['condition'])) {
                $steps[$key]['condition'] = json_encode($step['condition']);
```

```
62              }
63          }
64
65          $validatedData['steps'] = $steps;
66
67          return $validatedData;
68      }
69
70      // Modify the format of validation error messages before they are
    returned to the client.
71      protected function failedValidation(Validator $validator)
72      {
73          $errors = $validator->errors()->toArray();
74
75          throw new HttpResponseException(
76              response()->json([
77                  'status' => 'error',
78                  'message' => 'The given data was invalid.',
79                  'errors' => $errors
80              ], JsonResponse::HTTP_UNPROCESSABLE_ENTITY)
81          );
82      }
83  }
```

## 3. No Class property in controller

It's considered a best practice in Laravel PHP development to avoid declaring class properties directly within controllers. Instead, utilize methods to handle data and operations within the controller. This promotes encapsulation and improves code maintainability.

## 4. Single Responsibility Principle

The Single Responsibility Principle (SRP) states that a class or method should have only one reason to change, meaning it should have only one responsibility or job. This principle promotes modular, maintainable, and reusable code by ensuring that each component of your application is focused on a specific task.

Example :

- Bad Example

```php
public function getFullNameAttribute(): string
{
    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 Example

```php
public function getFullNameAttribute(): string
{
    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;
}
```

## 5. Model Fat, Controller Thin

The Model Fat, Controller Thin principle advocates for moving all database-related logic into the Eloquent models or repository classes if you're using Query Builder or raw SQL queries. This

helps keep controllers lightweight and focused on handling HTTP requests and responses, while models handle data manipulation and interactions with the database.

Example :

- Bad Example

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

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

- Good Example

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

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

## 6. Business Logic in Service Classes

The principle of keeping business logic within service classes advocates for moving complex logic out of controllers and into dedicated service classes. Controllers should have a single responsibility, typically handling HTTP requests and responses, while service classes encapsulate the application's core business logic.

Example :

- Bad Example

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

    ...
}
```

- Good Example

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

    ...
}

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

## 7. Don't Repeat Yourself (DRY)

The Don't Repeat Yourself (DRY) principle encourages reusing code whenever possible to avoid duplication and promote maintainability. This principle extends to Laravel development, where PSR standards help developers avoid redundancy. Additionally, reusing Blade templates, Eloquent scopes, and other Laravel features can further streamline development and reduce code duplication.

Example :

- Bad Example

```php
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 Example

```php
public function scopeActive($q)
{
    return $q->where('verified', 1)->whereNotNull('deleted_at');
}

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

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

## 8. Mass Assignment

Mass assignment allows you to assign an array of attributes to a model all at once, instead of setting each attribute individually. This principle promotes cleaner and more concise code, reducing redundancy and improving readability. Laravel provides convenient methods for mass assignment, making it easy to assign multiple attributes with a single line of code.

Example :

- Bad Example

```
$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 Example

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

## 9. Avoid Directly Accessing .env Data

Directly accessing data from the .env file can lead to security risks and inflexibility. Instead, it's recommended to redirect this data to configuration files and then utilize the `config()` function to access it within your application. This approach improves security by centralizing sensitive information and provides flexibility in managing configuration settings.

Example :

- Bad Example

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

- Good Example

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

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

## 10. Reference

PHP-FIG PSR-12 Coding Style Guide: https://www.php-fig.org/psr/psr-12/

The PHP-FIG PSR-12 Coding Style Guide provides recommendations for the coding style and structure of PHP code. It covers aspects such as file formatting, indentation, namespaces, classes, methods, properties, control structures, and more. Adhering to PSR-12 can help maintain consistency across PHP projects and improve code readability and maintainability.