# Coding Guidelines | PHP

https://guidelines.spatie.be/code-style/laravel-php

Software And Tools Version

Use the current stable version of PHP and WebServer (Apache). Use the current stable version of Laravel framework.

**Design Pattern And Framework**

Use MVC design pattern and Laravel framework (<http://laravel.com/)> Use pure Object Oriented PHP.

Follow single responsibility principle, i.e classes should only have one responsibility.

Don't create html strings in models or controllers. Pass data to view and put html code there.

**File Header Template**



/\*\*

* Copyright 2017-2018 Appster LLP
* All rights reserved.
* File: filename.h
* CodeLibrary/Project: CodeLibraryName/ProjectName
* Author: Author Name
* CreatedOn: date (dd/mm/yyyy)

\*/

**Naming**

Use meaningful, descriptive, consistent naming conventions for classes, interfaces, variables and methods. Use Pascal case for class (Model, Controller, Request, Helper, Service etc.) and interface names.

Use lower camel case for methods. Use snake case for variables.

Use Pascal case for file names.

Use Upper snake case for constants. Check spellings properly.

**Code Readability And Documentation**

Indent code and use white space for readability. The function code should not exceed 200 lines. The class code should not exceed NNN lines.

Comment your code to define the purpose of every class, method and parameter. Comment your code to explain logic behind a complex implementation.

Use tools (e.g. PHPDoc) to maintain a proper documentation of your code.

**Password**

Use the built-in password hashing functions to hash and compare passwords.

**Query**

Use Eloquent and it's query builder to prepare and execute SQL queries. Use transactions while executing multiple queries at a time.

The mysql user account should have only insert, update and delete (if record is being deleted) permissions.

Do not send sensitive data e.g. password, accessToken, facebookId, confirmationCode etc. response to a request.

**Character set**

Set the character set to utf8mb4 in the connection string to store Unicode data correctly in your database. Make sure that your actual database tables are in the utf8mb4 character set.

**Performance and Cache**

Use the Memcached client library Minify css, js and php files enable gzip

Only for production/staging run below commands

* 1. php artisan optimize
  2. php artisan optimize — force
  3. php artisan config:cache

**Sanitizing and Validation**

Validate every input parameter, even if it is not required.

Sanitizing HTML input and output. Use the htmlentities() function for simple sanitization and any library (e.g. the HTML Purifier library) for complex sanitization.

Validate cookie data before processing.

**Date and time**

Use the DateTime class.

Always specify timezone (UTC) while creating a DateTime object.

**Errors And Exceptions**

Use try...catch statements to handle statements that may raise exceptions.

**Data security**

Before performing any operation check the role of current user.

Before performing any operation check if the user is permitted to work on selected data.

Prevent Cross Site Scripting (XSS). If you are taking HTML as input. Remove dangerous tags like <script> etc.

For facebook sign-up and sign-in, ask for facebook access token and facebook id. Check them at server end. And then save into database.

Do not expose pages or controller methods to users, who are not permitted to view those pages. Keep CSRF protection active.

Don't Put phpinfo() in your Webroot. Don't allow directory listing.

Don't allow search engine crawling on dev and qa and staging server. Application don't allow loading of an iframe.

Disable production webserver fingerprinting.

**Note:** Few Functional security to be fixed:-

E-mail bombing and spamming. An attacker can use the mail server to bomb and spam user’s inbox by brute forcing.

Weak password policy. A weak password policy has been enforced, exposing the application to sniffing and brute-forcing attacks. Account lockout policy. The Account is should locked after a specific number of wrong attempts.

Proper session termination. The application should terminate sessions properly when the user resets password.

**File uploads**

Check the file types before saving it on server.

If the image is being fetched from a URL (e.g. facebook profile image), check validity of the URL and type of the file.

**Debugging and Logging**

Turn on Error Reporting

Use MonoLog (https://github.com/Seldaek/monolog) logging framework. Add logs in code at different places to help in debugging.

Use at-least 3 level of logging – INFO (general informational messages), ERROR (business error cases) & FATAL (System failures). Write error/fatal logs in production mode while info logs for development.

Send email in case of FATAL errors and needed business errors to support group of person responsible for handling errors. Print full stack-trace in log and in email.

The thumb rule is: No webpage or API should receive a technical error. All errors should be caught and presented with appropriate error messages.

**Good coding practice**

Don't Repeat Yourself.

If you have multiple similar code blocks, put them in a function.

If you have multiple similar methods in different classes, put them in a super class. Do not define helper methods in a model or controller. Put them in a helper class. Keep Functions Outside of Loops.

Remove extra use statements, variables, methods, classes etc. Try to Prevent Deep Nesting.

Put model related constants in the model class, not in config. e.g. User model can have constants/enum related to user status. Don't hard code any values. e.g. error messages etc. Move them to related config or resource files.

**Code Versioning**

Controllers and APIs should be versioned properly to support backward compatibility.

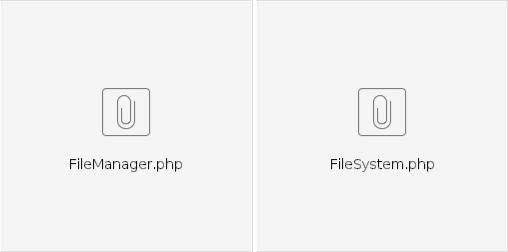
**Unit Testing:**

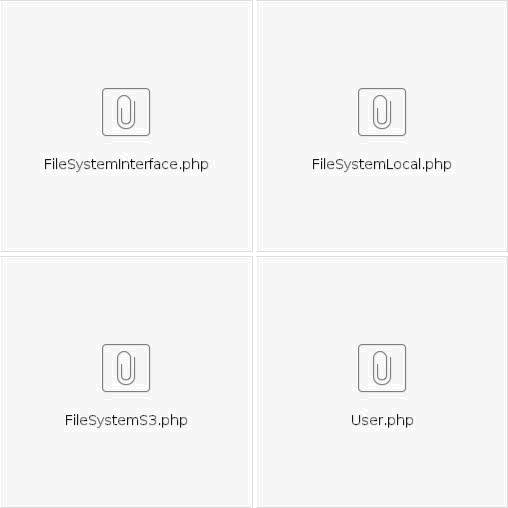
Use PHPUnit framework for unit testing.

Make sure that you cover 70-80% of code using test cases.

This will reduce bugs in QA and reduce the over all effort in project development.

**File handling Laravel**





**Moving file handling code from controllers and models to oops based structure**

In most of our laravel projects we are handling file uploads by users and saving the uploading files to either local file system of the S3 server.

In many cases we may need to move from local file system to S3 server as the userbase grows for that you may need to do many changes in your controllers that handle file upload or models .

If you are using Laravel ORM model there is a better oops based way to handle it Step 1

Create you ORM Model User

*<?php namespace App;*

*use Illuminate\Database\Eloquent\Model;*

*class User extends Model { use FileManager;*

*protected $table = 'mp\_user'; protected $primaryKey = 'user\_id';*

*}*

Here we are using a trait FileManger that will handle all get and set requests

Same trait can handle getter setter of multiple ORM models so you don't need to copy paste getter meathod if you are refrencing other model in join

<?php namespace App;

trait FileManager

{

public function getUserImageAttribute()

{

$file\_system='local';

if(isset($this->attributes['user\_image']))

{

$file\_manager=new FileSystem($file\_system);

return $this->attributes['user\_image'] = $file\_manager->read\_file($this->attributes['user\_image']);

}

}

public function setUserImageAttribute($user\_image)

{

$file\_system='local'; // Read this from config if(is\_object($user\_image)&& $user\_image->isValid())

{

//Get your path from config

$file\_manager=new FileSystem($file\_system);

$this->attributes['user\_image'] =

$file\_manager->save\_file($user\_image,public\_path().'/uploads/users/','venue');

}else

{

$this->attributes['user\_image'] = $user\_image;

}

}

}

Controller

In our controller we can directly save the file to orm model

$user=User::find($user\_id); if(is\_object($user))

{

$user->first\_name=$request->input('first\_name');

$user->user\_image=$request->input('user\_image');

$user->save();

}

Demo Controller Code

In our controller we can directly save the file to orm model

$user=User::find($user\_id); if(is\_object($user))

{

$user->first\_name=$request->input('first\_name');

$user->user\_image=$request->input('user\_image');

/ We are directly setting the orm property to the uploaded file . Also do file validation before saving it to server $user->save();

}

For saving files we are using Strategy Pattern class FileSystems that can handle file save to local file system,S3 file system or any other way

Check the code for more details

Demo Controller Code

In our controller we can directly save the file to orm model

$user=User::find($user\_id); if(is\_object($user))

{

$user->first\_name=$request->input('first\_name');

$user->user\_image=$request->input('user\_image');

/ We are directly setting the orm property to the uploaded file . Also do file validation before saving it to server $user->save();

}

For saving files we are using Strategy Pattern class FileSystems that can handle file save to local file system,S3 file system or any other way

Check the attached file for info on how it is implemented