PHP Style Guide

> *This document is a draft should not be considered as a final document. This is a living document and can be modified by anyone who wish to add something to it. 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.*

All code in any code-base should look like a single person typed it, and should follow same code structures everywhere possible, no matter how many people contributed. To ensure this, we stick with following coding practices for PHP.

Basic Coding Styles

Here at Lazada,

- We strictly follow PSR-2: Coding Style Guide for all of our php projects.
- Since we use PHP 7.1+, we MUST also follow new features of PHP7 like *Type Hinting, Return type declarations, Null Coalesce* etc.
- We MUST follow clean coding practice and SOLID Principles wherever possible.
- We MUST avoid code duplication (DRY).
- We MUST keep our code as simple as possible (KISS).

Variables

We MUST always use meaningful and semantically correct variable names.

Bad:

```
$ymd = date('Y-m-d');

$black = '#fff'; // white is not black
```

Good:

```
$currentDate = date('Y-m-d');

$black = '#000';
```

Alignment of Assignments

To support readability, the equal signs *SHOULD* be aligned in block-related assignments:

Bad:

```
$short = foo($bar);
$longVariable = foo($baz);
```

Good:

```
$short        = foo($bar);
$longVariable = foo($baz);
```

The rule SHOULD be broken when the length of the variable name is at least eight to ten characters longer or shorter than the surrounding ones. If the variable name is very long, you SHOULD place it in a new line.

Bad:

```
$short                      = foo($bar);
$longVariable               = foo($baz);
$thisIsaVeryLongVariableName = foo($bat);
```

Good:

```
$short        = foo($bar);
$longVariable = foo($baz);

$thisIsaVeryLongVariableName = foo($bat);
```

Array

You MUST follow following rule to format arrays.

Bad:

```
$options = [
              'headers' => ['Content-Type' => 'application/json'],
              'json' => [
                          'username' => config('services.username'),
                          'password' => config('services.password')
                        ]
           ];
```

Good:

```
$options  = [
    'headers' => ['Content-Type' => 'application/json'],
    'json' => [
        'username' => config('services.username'),
        'password' => config('services.password'),
    ],
];
```

Variables and Code Block Grouping

There is no strict rule for this but its best practice to group code blocks with similar logical behavior.

Bad:

```
function something()
{
 $user = getUser();
 $articles = getUserArticles($user);
 $user->setName('Lazada');
 $articles->getMetaData();
 $user->setEmail('email@lazada.com');
 $user->sendEmail();
}

function forceDelete()
{
    $this->forceDeleting = true;
    $deleted = $this->delete();
    $this->forceDeleting = false;
    return $deleted;
}
```

Good:

```
function something()
{
    $user     = getUser();
    $articles = getUserArticles($user);

    $user->setName('Lazada');
    $user->setEmail('email@lazada.com');
    $user->sendEmail();

    $articles->getMetaData();
}

function forceDelete()
{
    $this->forceDeleting = true;

    $deleted = $this->delete();

    $this->forceDeleting = false;

    return $deleted;
}
```

Early Return

Early return refers to exiting a function early. In most cases, returning early reduces the complexity and makes the code more readable. We MUST always try to handle invalid or negative cases first, either simply exiting or raising exceptions as appropriate.

Bad:

```
function foo($user)
{
    if ($user->isAuthorized()) {
        // ...statements
    } else {
        // 403
    }
}
```

Good:

```
function foo($user)
{
    if (!$user->isAuthorized()) {
        // 403
    }

    // ...statements
}
```

Avoid Unnecessary else

An `if` expression with an `else` branch is usually not necessary. We MUST use early returns to rewrite the conditions in a way that the else is not necessary and the code becomes simpler to read. For very simple statements we SHOULD use ternary operations. For complex logic we MUST split the code into several smaller methods.

Bad:

```
function bar($flag)
{
    if ($flag) {
        return 'Hello';
    } else {
        return 'World';
    }
}
```

Good:

```
function bar($flag)
{
    return $flag ? 'Hello' : 'World';
}

// or

function bar($flag)
{
    if ($flag) {
        return 'Hello';
    }

    return 'World';
}
```

Strict Type Checkings

You MUST use strict type checkings with `===` wherever possible. Strict type checks are always safe than weak type checks.

Bad:

```
function isEqual(int $a, int $b) : bool
{
    return $a == $b;
}

function searchInArray(string $needle, array $haystack) : bool
{
    return in_array($needle, $haystack);
}
```

Good:

```
function isEqual(int $a, int $b) : bool
{
    return $a === $b;
}

function searchInArray(string $needle, array $haystack) : bool
{
    return in_array($needle, $haystack, true); // Third parameter ensures strict type checking
}
```

Type Hinting

Use type hinting and return types wherever possible.

Bad:

```
function add($a, $b)
{
    return $a + $b;
}
```

Good:

```
function add(int $a, int $b) : int
{
    return $a + $b;
}
```

Nullable Return Types

Do not misuse nullable return types.

Bad:

```
public function create(array $data) : ?bool
{
    // ...some code

    if (!$someCheck) {
        // ...some error handlings
        return null;
    }

    // ...some more codes

    return $status; // true or false
}

public function getWithStatus(?string $status = null)
{
 // ...some code
}
```

Good:

```
public function create(array $data) : bool
{
    // ...some code

    if (!$someCheck) {
        throw new SomeException('This was not supposed to happen');
    }

    // ...some more codes

    return $status; // true or false
}

public function getName() : ?string
{
    // name can be null
    return $this->name;
}

// since we are assigning null as default value we don't need `?`
public function getWithStatus(string $status = null)
{
 // ...some code
}
```

Super Globals

Accessing a super-global variable directly is considered a bad practice. These variables should be encapsulated in objects that are provided by a framework, for instance.

Bad:

```
$pathInfo = $_SERVER['PATH_INFO'];

$file = $_FILES['upload'];
```

Good:

```
// Laravel
$pathInfo = $request->server('PATH_INFO');

$file = $request->file('upload');
```

Doc Blocks

All of the methods and properties MUST have proper doc blocks associated to it. You can learn more about php documentor here.

- *description*, *@param* and *@return* attributes are required.
- *description* and *@param* comment MUST end with a full stop.
- If the method is only used in ajax call, you MUST use custom attribute *@ajax* to denote that it's only used in ajax request.
- Proper grouping of parameters is required as shown in the example.

Bad:

```
/**
 * Get package
 * @param int    $id           Id
 * @param string $status       Status
 * @return Response
 */
public function getPackage(int $id, string $status) : Response
{
 // ...statements
}
```

Good:

```
/**
 * Get package based on package ID and given status.
 *
 * @param int    $id     Package ID.
 * @param string $status Package status to fetch.
 *
 * @return Response
 */
public function getPackage(int $id, string $status) : Response
{
 // ...statements
}
```

Further Readings

1. Clean Code concepts adapted for PHP
2. PHP: The Right Way
3. Object Calisthenics

Bonus

1. Get Basic Knowledge of PHP Internals, see PHP Internals Book
2. Read about interesting internal concepts like When does foreach copy? or How big are PHP arrays (and values) really?
3. Keep track of what's coming next in PHP, see PHP RFCs