**CODING GUIDELINES DOCUMENT**

1. INTRODUCTION
2. FILE FORMAT
3. PHP TAGS
4. COMMENTING
5. NAMING CONVENTIONS
6. STRINGS
7. LOGICAL OPERATORS
8. COMPARISONS & TYPE CASTING
9. CODE INDENTATION
10. ERROR HANDLING
11. SQL QUERIES
12. OTHERS

**INTRODUCTION**

This coding guidelines is based upon a Laravel Framework. This coding guidelines is truly specific for the development purpose. The purpose of this coding guidelines is to form the uniformity in the coding while is been reviewed by the different authors. It may have some change according to the project needs.

**FILE FORMAT**

* Files should be saved with Unicode(UTF-8) encoding.
* Css and js files should be saved inside the public folder.
* Controllers should be saved in the App\Http\Controllers folder.
* Models should be saved inside the App\Models folders.
* User defined middleware should be present in App\Http\Middleware.
* All the blade file should be present in resources/views folders.

**Sublime**

1. Preferences -> Settings – User, Add below line of code  
   "default\_encoding": "UTF-8",
2. View -> Line Endings, select **Unix** (for LF)
3. View -> Indentation, select **Tab Width: 4** and **Indent Using Spaces**

**PHP TAGS**

* File should use only <?php .
* No whitespace can precede the opening PHP tags or follow the closing tag.

|  |  |
| --- | --- |
| **INCORRECT: (Use of PHP tags)** | **CORRECT:** |
| **<? echo $foo; ?>**  **<?=$foo; ?>** | <?php echo $foo; ?> |

|  |  |
| --- | --- |
| **INCORRECT: (whitespace in file)** | **CORRECT:** |
| **<?php**  **// There is whitespace and a linebreak above the opening PHP tag**  **// As well as whitespace after the closing PHP tag**  **?>** | <?php  // No whitespace before or after the opening and  closing PHP tags  ?> |

**COMMENTING**

* Use **DocBlock style** coments preceding class and methods. (as it can be picked by IDE easily)
* Use single line comment within codes.
* Leave a blank line between large comment blocks and code.

|  |  |
| --- | --- |
| **Class/Files header comment:** | **Function header comment:** |
| **/\*\***  **\* Super Class**  **\***  **\* @package Package Name**  **\* @subpackage Subpackage**  **\* @category Category**  **\* @author Author Name**  **\* @link** [**http://example.com**](http://example.com/)  **\*/**  **class Super\_class {** | /\*\*  \* Encodes string for use in XML  \*  \* @access public function\_name  \* @param string foo  \* @return int bar  \*/  function xml\_encode($str) |

|  |
| --- |
| **Inline comments:** |
| **// Break up the string by newlines**  **$parts = explode("\n", $str);**    **// A longer comment that needs to give greater detail on what is**  **// occurring and why can use multiple single-line comments. Try to**  **// keep the width reasonable, around 70 characters is the easiest to**  **// read. Don't hesitate to link to permanent external resources**  **// that may provide greater detail:**  **//**  **//** [**http://example.com/information\_about\_something/in\_particular/**](http://example.com/information_about_something/in_particular/)  **$parts = $this->foo($parts);** |

**NAMING CONVENTIONS**

* File name should be in lower case inside the resource\view folder
* File name for controller and middleware should follow StudlyCaps follwed by 'Middleware or Controller' as per the laravel defined naming conventions
* Class names should in StudlyCaps.
* Method name should follow camelCased.
* Private Method names should be prefixed with underscore and should be
* Variables should always be in lowercase.
* Use underscore as separators for multiple words. Avoid overly long and verbose names.
* Very short, non-word variables should only be used as iterators in for() loops.
* **TRUE**, **FALSE**, and **NULL** keywords should be in uppercase.
* **CONSTANTS** are fully uppercase.

Example:

|  |  |
| --- | --- |
| **Incorrect (Class Names)** | **Correct(Class Names)** |
| **class class\_name** | class ClassName |

|  |  |
| --- | --- |
| **Incorrect(Method Name)** | **Correct(Method Name)** |
| **// not descriptive and no separator**  **function get\_info()**  **Function update\_info()** | //descriptive, CamelCase  function getInfo()  function updateInfo() |

|  |  |
| --- | --- |
| **Incorrect(variable name)** | **Correct(variable name)** |
| **$i = 'hello'; // single letter variable only for loops.**  **$Str // Uppercase letters**  **$bufferedText // CamelCased**  **$groupid // Multiple words, needs underscore**  **$name\_of\_last\_city\_used // Too long** | or ($j = 0; $j < 10; $j++)  $str  $buffer  $group\_id  $last\_city |

|  |  |
| --- | --- |
| **INCORRECT: (TRUE, FALSE, NULL)** | **CORRECT:** |
| if ($foo == **true**)  $bar = **false**;  function foo($bar=**null**) | if (**TRUE ===** $foo)  $bar = **FALSE**;  function foo($bar=**NULL**) |

|  |  |
| --- | --- |
| **INCORRECT: (CONSTANTS)** | **CORRECT:** |
| **MyConstant // Missing underscore and not uppercased**  **N // Single-letter constants**  **S\_C\_VER // Not descriptive**    **// Should use LD and RD constants**  **$str = str\_replace('{foo}', 'bar', $str);** | MY\_CONSTANT  NEWLINE  SUPER\_CLASS\_VERSION  $str = str\_replace(LD.'foo'.RD, 'bar', $str); |

|  |  |
| --- | --- |
| **INCORRECT: (Class having common words)** | **CORRECT:** |
| **class Email, class Xml, class Import** | class Pre\_email, class Pre\_xml, class Pre\_import |

|  |  |
| --- | --- |
| **INCORRECT: (File having common words)** | **CORRECT:** |
| **email.php, xml.php, import.php** | pre\_email.php, pre\_xml.php, pre\_import.php |

**STRINGS**

* Use single quoted strings unless you need variables parsed.
* Use braces when variables need to be parsed to prevent greedy token parsing. You may also use double-quoted strings if the string contains single quotes, so you do not have to use escape characters.

|  |  |
| --- | --- |
| **INCORRECT:** | **CORRECT:** |
| **// No variable parsing, so no use for double quotes**  **"My String"**    **"My string $foo" // Needs braces**  **'SELECT foo FROM bar WHERE baz = \'bag\'' // Ugly** | 'My String'  "My string {$foo}"  "SELECT foo FROM bar WHERE baz = 'bag' " |

**LOGICAL OPERATOR**

* Use **||**, **&&** and **!**

**COMPARISON & TYPE CASTING**

* Use **===** and **!==** as necessary.
* Use boolean/Constant on the left side and **variable on the right side**.(YODA conditions)
* Do typecasting wherever required. When casting a variable as a string, for instance, NULL and boolean FALSE variables become empty strings, 0 (and other numbers) become strings of digits, and boolean TRUE becomes "1".

|  |  |
| --- | --- |
| **INCORRECT:** | **CORRECT:** |
| **// If 'foo' is at the beginning of the string, strpos will return a 0, //resulting in this conditional evaluating as TRUE**  **if (strpos($str, 'foo') == FALSE)**    **function build\_string($str='')**  **{**  **// Oh! What if FALSE or the integer 0 is passed as an argument?**  **if ($str == '')**  **{**  **}**  **}** | if (FALSE === strpos($str, 'foo'))    function build\_string($str='')  {  if ('' === $str)  {  }  }    // Cast $str as a string  $str = (string) $str; |

**CODE INDENTATION**

* Line Indentation should be **4 spaces** and wrapped within a range of **75 -85 columns width**.
* **One space** should be given **between operand and operator**, this applies to all expressions.
* Always put **spaces after commas and on both sides** of logical, comparison, string and assignment operators.
* **No space** inside function parenthesis before and after assigment operators. **Space after comma is must.**
* **Logical grouping** of code should have **one new line before and after**.
* For **PHP**, **Opening braces** should start on the **next line** and indented at the same level as the control statement that "owns" them.(**Except Classes**)
* For **JavaScript**, **Opening braces** should start on the **same line.**
* Space should always follow PHP control structures that accept arguments with parenthesis (declare, do-while, elseif, for, foreach, if, switch, while), to help distinguish them from functions and increase readability.
* Other then above parenthesis and brackets, should not use any additional spaces.

|  |  |
| --- | --- |
| **INCORRECT: (PHP)** | **CORRECT: (PHP)** |
| **function foo($bar) {**  **// Your codes**  **}**    **foreach ($arr as $key => $val) {**  **// Your codes**  **}**    **if ($foo == $bar) {**  **// Your codes**  **} else {**  **// Your codes**  **}** | function foo($bar)  {  // Your codes  }    foreach ($arr as $key => $val)  {  // Your codes  }  if ($foo == $bar)  {  }  else  {  } |

|  |  |
| --- | --- |
| **INCORRECT: (JavaScript)** | **CORRECT: (JavaScript)** |
| **function foo($bar)**  **{**  **// Your codes**  **}**    **if(condition)**  **{**  **// Your codes**  **}**  **else**  **{**  **// Your codes**  **}** | function foo($bar) {  // Your codes  }    for (i = 0; i < count; i++) {  // Your codes  }    if (condition) {  // Your codes  } else {  // Your codes  } |

|  |  |
| --- | --- |
| **INCORRECT:** | **CORRECT:** |
| **if (!$foo)**  **if (! is\_array($foo))**  **if($foo || $bar)**  **if($foo && $bar)**  **$foo.='bar' array(4,2,2) $foo==9**  **$arr[ $foo ] = 'foo';**      **function foo ( $bar = 'value', $foo = TRUE)**  **{**  **}**    **foreach( $query->result() as $row )**    **$foo = array( 'foo\_one' => 'bar\_one','foo\_two' => 'bar\_two','foo\_three' => 'bar\_three');** | if ( ! $foo)  if ( ! is\_array($foo))  if ($foo || $bar)  if ($foo && $bar)  $foo .= 'bar' array(4, 2, 2) // Notice space after comma $foo == 9  $arr[$foo] = 'foo'; // No spaces around array keys      function foo($bar='value', $foo=TRUE)  {  }    foreach ($query->result() as $row)    $foo = array(  'foo\_one' => 'bar\_one',  'foo\_two' => 'bar\_two',  'foo\_three' => 'bar\_three'  ); |

**ERROR AND EXCEPTION HANDLING**

* Use of try catch should be there to handle errors and exception.
* Braces should be opened with single space between try or catch (try { ).
* Catching base exception should be the last step when handling nested exceptions.

|  |
| --- |
| **try {**  **// Your code**  **} catch (Exception $e) {**  **log\_message('error', "Exception: " . \_\_FUNCTION\_\_ . $e->getMessage());**  **}** |

**SQL QUERIES**

* SQL keywords should be in lowercase as per the Laravel framework specific.
* No need to write raw queries.
* Example:
* $a = " SELECT \* FROM table\_name";
* Use Laravel framework specific pre-defined function and Eloquent to perform db operation.

Example:

|  |
| --- |
| $users = DB::select('select \* from users where active = ?', [1]);  DB::table('users')->update(['votes' => 1]); |

**OTHERS**

* No php debugging code or laravel debugging code should be present unless required. (exit(), print\_r, var\_dump, dd() ).
* No javascript debuggling code should be present in the script file.
* All code should be compatible to the PHP 5.5.9 version.
* All production credential should present in the .env file.

**REASONS**

* **Files should be saved with Unicode (UTF-8) encoding. The BOM should not be used.**  
  (Unlike UTF-16 and UTF-32, there's no byte order to indicate in a UTF-8 encoded file, and the BOM can have a negative side effect in PHP of sending output, preventing the application from being able to set its own headers).
* **Files should be saved with Unix line breaks at the end.**  
  (This is more of an issue for developers who work in Windows, but in any case ensure that your text editor is setup to save files with Unix line breaks).
* **All PHP files should OMIT closing PHP tag '?>' and use a comment block to mark the end of file and it's location relative to the application root.**  
  (This allows you to identify a file as being complete and not truncated. The PHP closing tag on a PHP document ?> is optional to the PHP parser. However, if used, any whitespace following the closing tag, whether introduced by the developer, user, or an FTP application, can cause unwanted output, PHP errors, or if the latter are suppressed, blank pages).
* **No whitespace can precede the opening PHP tag or follow the closing PHP tag.**  
  (Output is buffered, so whitespace in your files can cause output to begin before Laravel outputs its content, leading to errors and an inability for Laravel to send proper headers).
* **Class and Filename having a common word likely to be identical named in another PHP script, provide a unique prefix to help prevent collision.**  
  (Always realize that your end users may be running other add-ons or third party PHP scripts. Choose a prefix that is unique to your identity as a developer or company.)
* **Use === and !== as necessary.**  
  (Some PHP functions return FALSE on failure, but may also have a valid return value of "" or 0, which would evaluate to FALSE in loose comparisons. Be explicit by comparing the variable type when using these return values in conditionals to ensure the return value is indeed what you expect, and not a value that has an equivalent loose-type evaluation.)
* **Use boolean/Constant on the left side and variable on the right side.**  
  (missing comparison operator like if ($foo = -1){} can lead to unexpected result)