Key Aspects of PHP Coding Standards

In PHP development, adhering to coding standards ensures consistency, readability, maintainability, and reliability across the codebase. These guidelines help developers work collaboratively and maintain high-quality software.

Below is an overview of PHP coding standards, along with examples to clarify the concepts for even non-programmers:

1. Naming Conventions
   * Use meaningful and descriptive names for variables, functions, and classes.
   * PHP conventions often prefer snake\_case for variable and function names and PascalCase for class names.

// Good Naming

$customer\_name = "John Doe"; // Descriptive variable

function calculate\_total\_price($price, $tax) {

return $price + $tax;

}

class CustomerDetails {

public $name;

public $email;

}

// Bad Naming

$a = "John"; // Ambiguous variable

function calc($x, $y) {

return $x + $y;

}

1. Formatting and Indentation
   * Use consistent indentation (e.g., 4 spaces).
   * Ensure braces and parentheses are consistently styled.
   * Limit line length to 80–120 characters.

// Good Formatting

if ($is\_logged\_in) {

echo "Welcome back!";

} else {

echo "Please log in.";

}

// Bad Formatting

if($is\_logged\_in){echo"Welcome back!";}else{echo"Please log in.";}

1. Comments and Documentation
   * Use comments to explain the purpose of code but avoid unnecessary comments.
   * PHPDoc is a popular standard for writing documentation in PHP.

/\*\*

\* Calculates the area of a rectangle.

\*

\* @param float $length Length of the rectangle.

\* @param float $width Width of the rectangle.

\* @return float The calculated area.

\*/

function calculate\_area($length, $width) {

return $length \* $width;

}

1. Organize Code
   * Group related code into classes or functions.
   * Separate code blocks with whitespace and meaningful comments.

// Grouped code for handling users

class UserManager {

public function create\_user($name, $email) {

// Code to create a user

}

public function delete\_user($user\_id) {

// Code to delete a user

}

}

1. DRY Principle (Don't Repeat Yourself)
   * Avoid repeating similar code by using reusable functions.

// Avoid duplicate logic

function display\_message($message) {

echo "<p>$message</p>";

}

display\_message("Welcome to the site!");

display\_message("Thank you for visiting!");

1. Code Reusability
   * Encapsulate reusable code in functions or utility classes.

class MathUtilities {

public static function add($a, $b) {

return $a + $b;

}

public static function subtract($a, $b) {

return $a - $b;

}

}

echo MathUtilities::add(5, 3);

echo MathUtilities::subtract(10, 4);

1. Avoid Deep Nesting Structures
   * Reduce complexity by breaking down deeply nested logic into smaller functions.

// Deep nesting (Bad)

if ($is\_logged\_in) {

if ($has\_permission) {

if ($item\_in\_stock) {

echo "You can purchase this item.";

}

}

}

// Improved version (Good)

function can\_purchase($is\_logged\_in, $has\_permission, $item\_in\_stock) {

return $is\_logged\_in && $has\_permission && $item\_in\_stock;

}

if (can\_purchase($is\_logged\_in, $has\_permission, $item\_in\_stock)) {

echo "You can purchase this item.";

}

1. Short Line Length
   * Keep lines shorter than 80 characters to enhance readability.

// Good

$message = "This is a short and concise message.";

echo $message;

// Bad

echo "This is a very long line of code that stretches beyond the standard line length and becomes hard to read.";

1. OOP vs. Procedural Programming
   * Use Object-Oriented Programming (OOP) for complex applications requiring reusable components.
   * Use Procedural Programming for simpler scripts.

class Product {

private $name;

private $price;

public function \_\_construct($name, $price) {

$this->name = $name;

$this->price = $price;

}

public function display\_product() {

echo "Product: $this->name, Price: $this->price";

}

}

$product = new Product("Laptop", 1200);

$product->display\_product();

$product\_name = "Laptop";

$product\_price = 1200;

function display\_product($name, $price) {

echo "Product: $name, Price: $price";

}

display\_product($product\_name, $product\_price);

By following these PHP coding standards, developers can write cleaner, more maintainable code, making collaboration and debugging significantly easier.

1. Always Use <?php Tags

* Always start your PHP code with the <?php tag instead of the short <? tag. The short tags may not be enabled on all servers.

<?php

echo "Hello, world!";

1. Close All Open Tags

* If you use PHP inside an HTML file, ensure all PHP blocks are closed properly.

<!DOCTYPE html>

<html>

<head>

<title>Welcome</title>

</head>

<body>

<h1><?php echo "Hello, world!"; ?></h1>

</body>

</html>

1. Avoid Mixing Logic and HTML
   * Separate the PHP logic from the HTML structure as much as possible to keep code clean and easy to maintain.

Bad Practice (Mixed):

<!DOCTYPE html>

<html>

<body>

<?php

if ($user\_logged\_in) {

echo "<p>Welcome back, user!</p>";

} else {

echo "<p>Please log in.</p>";

}

?>

</body>

</html>

Good Practice (Separated):

<?php

$message = $user\_logged\_in ? "Welcome back, user!" : "Please log in.";

?>

<!DOCTYPE html>

<html>

<body>

<p><?php echo $message; ?></p>

</body>

</html>

1. Use === Instead of ==

* Use strict comparison (===) to avoid unexpected results, as it checks both the value and the data type.

// Bad: Loose comparison

$value = "5"; // String

if ($value == 5) {

echo "This will run, but it's not ideal.";

}

// Good: Strict comparison

if ($value === 5) {

echo "This will not run because the types do not match.";

}

1. Use Constants for Fixed Values
   * Define constants for values that do not change, such as database credentials or application settings.

define("SITE\_NAME", "My Website");

echo "Welcome to " . SITE\_NAME;

1. Validate and Sanitize User Input
   * Always validate and sanitize user input to prevent security issues such as SQL injection and XSS.

// Bad: Directly using user input

$username = $\_POST['username'];

// Good: Sanitizing user input

$username = htmlspecialchars($\_POST['username'], ENT\_QUOTES, 'UTF-8');

1. Use Include or Require for Reusable Code

* Use include or require to reuse code from other files, such as a common header or footer.

header.php:

<!DOCTYPE html>

<html>

<head>

<title>My Website</title>

</head>

<body>

<header>

<h1>Welcome to My Website</h1>

</header>

index.php:

<?php include 'header.php'; ?>

<p>This is the homepage content.</p>

</body>

</html>

1. Use Arrays for Organized Data
   * Arrays help group related information together, making code cleaner.

// Array of user details

$user = [

'name' => 'John Doe',

'email' => 'john.doe@example.com',

'age' => 30

];

// Access array values

echo "Name: " . $user['name'];

1. Avoid Hardcoding

* Avoid writing fixed values (like database credentials or URLs) directly in your code. Use configuration files instead.

config.php

define("DB\_HOST", "localhost");

define("DB\_USER", "root");

define("DB\_PASS", "password");

define("DB\_NAME", "example\_db");

db\_connection.php:

<?php

$conn = mysqli\_connect(DB\_HOST, DB\_USER, DB\_PASS, DB\_NAME);

1. Use Meaningful Defaults

* When defining a function, provide default values for optional parameters.

function greet($name = "Guest") {

echo "Hello, $name!";

}

greet(); // Outputs: Hello, Guest!

greet("John"); // Outputs: Hello, John!

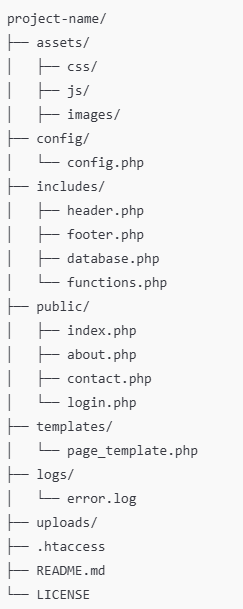
1. Always Close Database Connections
   * When working with databases, close connections after completing your operations to save resources.

$conn = mysqli\_connect("localhost", "root", "password", "example\_db");

// Perform database operations

mysqli\_close($conn);

**Core PHP Project Structure**



Folder Descriptions

**assets/**

This folder contains all static files like CSS, JavaScript, and images.

* **css/**: Contains all CSS files.
* **js/**: Contains JavaScript files.
* **images/**: Stores images for the project.

**config/**

Contains configuration files such as database credentials or global settings.

config.php: A file to store all essential configuration variables.

<?php config.php

define('DB\_HOST', 'localhost');

define('DB\_USER', 'root');

define('DB\_PASS', 'password');

define('DB\_NAME', 'project\_db');

define('BASE\_URL', 'http://localhost/project-name/');

?>

**includes/**

Contains reusable PHP files for modular code.

* **header.php**: Code for the common header of your pages.
* **footer.php**: Code for the common footer of your pages.
* **database.php**: Code for database connection.
* **functions.php**: Utility functions that are used across the project.

**public/**

Contains all publicly accessible PHP files, such as the main pages of your website or app.

* **index.php**: The homepage.
* **about.php**: An example About page.
* **contact.php**: A contact form page.

<?php

include '../config/config.php';

include '../includes/header.php';

?>

<h1>Welcome to My Website</h1>

<?php

include '../includes/footer.php';

?>

**templates/**

Stores reusable HTML or PHP templates for specific pages or components.

<div class="content">

<h2><?php echo $page\_title; ?></h2>

<p><?php echo $page\_content; ?></p>

</div>

**logs/**

Stores log files for errors or debugging purposes.

Used to log errors using PHP’s error logging system:

error\_log("Error message here", 3, "../logs/error.log");

**uploads/**

A directory for storing user-uploaded files, such as images or documents.

**.htaccess**

A file to handle URL rewriting, security configurations, and access rules.

# Prevent direct access to config and includes files

<FilesMatch "\.(php|ini)$">

Deny from all

</FilesMatch>

# Enable URL rewriting

RewriteEngine On

RewriteRule ^$ public/index.php [L]

RewriteRule (.\*) public/$1 [L]

**README.md**

A file containing project documentation for other developers or users.

**LICENSE**

Specifies the terms under which the project can be used or distributed.

**Benefits of This Structure:**

* **Separation of Concerns**: Static files, configuration, reusable code, and logic are separated.
* **Reusability**: Shared components like headers and footers are reusable.
* **Scalability**: The project can grow without becoming disorganized.
* **Maintainability**: Code is easier to navigate, update, and debug.