CSS Coding Standards

1. Naming Conventions

- **Class names** should use **lowercase letters** with hyphens (-) as separators (BEM methodology is recommended).
  ✅ **Good:** .profile-card, .stat-container
  ❌ **Avoid:** .peregraph, .card-h3 (use .card__heading instead)

- **ID names** should follow the same convention but should be **used sparingly** for unique elements.
  ✅ **Good:** #navigation, #profile-card
  ❌ **Avoid:** #search-&bg (Hyphens should be used instead of &, e.g., #search-bg)

2. Formatting and Structure

- Always maintain **proper indentation** (2 or 4 spaces per level).

Use **consistent spacing**:
✅ **Good:**

```
.highlight:hover {
  transform: scale(1.05);
}
```

❌ **Avoid:**

```
.highlight:hover{transform:scale(1.05);}
```

-
- Group related styles together using **comments** (/* Section Name */).

- Keep **one selector per line** for better readability.

3. Color and Styling Best Practices

Use **CSS variables** or a **color palette** for better maintainability.

✅ **Good:**

```
:root {
  --primary-color: rgb(20, 93, 93);
  --background-dark: black;
  --text-light: white;
}

.mark {
  background-color: var(--primary-color);
}
```

❌ **Avoid:**

```
.mark {
  background-color: rgb(20, 93, 93);
}
```

-

4. Consistent Units

Use **relative units** like em, rem, % for font sizes and spacing instead of fixed px.

✅ **Good:**

```
.stat-card-title {
  font-size: 2rem;
}
```

❌ **Avoid:**

```
.stat-card-title {
  font-size: 32px;
}
```

-

5. Hover and Interactive States

- Always define :hover, :focus, and :active states for interactive elements.

Ensure **hover effects** are smooth using transition.
✅ **Good:**

```
.highlight:hover {
  transform: scale(1.05);
  transition: transform 0.3s ease-in-out;
}
```

-

6. Responsive Design

- Use **flexbox** or **grid** for layout structure instead of float.

Keep **media queries organized** at the bottom of the file or inside a dedicated section.
✅ **Good:**

```
@media (max-width: 600px) {
  .stat-container {
    flex-direction: column;
    align-items: center;
  }
}
```

-

7. Avoid Repetition

Merge repeated styles into a common class to **reduce redundancy**.
❌ **Avoid (repeated code):**

```
#navigation {
  background-color: black;
  color: white;
}

.dropdown {
  background-color: black;
  color: white;
```

}
✅ **Good:**
```
.dark-theme {
  background-color: black;
  color: white;
}
```

-

8. Accessibility (A11Y)

- Ensure text has **sufficient contrast** against the background.

Use focus styles for **keyboard accessibility**.
✅ **Good:**
```
button:focus {
  outline: 2px solid #3498db;
}
```

JavaScript Coding Standards

1. Naming Conventions

- **Variables & Functions** should use **camelCase**.
  ✅ **Good:** profileBtn, dropdownMenu, deleteCourse
  ❌ **Avoid:** profile_btn, delete_course

- **Boolean Variables** should start with is, has, or should.
  ✅ **Good:** isVisible, hasError
  ❌ **Avoid:** visible, errorStatus

- **Constants** should be in uppercase with underscores.
  ✅ **Good:** const DELETE_CONFIRMATION_MESSAGE = "Are you sure you want to delete this course?";

❌ **Avoid:** const deleteConfirmationMessage = "Are you sure...";

2. Event Handling Best Practices

Use **named functions** instead of inline arrow functions for better readability and debugging.

❌ **Avoid:**

```
 profileBtn.addEventListener("click", () => {
   dropdownMenu.classList.toggle("hidden");
});
```

✅ **Good:**

```
 function toggleDropdown() {
   dropdownMenu.classList.toggle("hidden");
}
profileBtn.addEventListener("click", toggleDropdown);
```

-
- **Use document.addEventListener** for events affecting the entire page instead of window.addEventListener.

3. Code Structure & Formatting

- Maintain **consistent indentation** (2 or 4 spaces).
- Keep **one statement per line** for readability.
- Use **early return** in functions for better clarity.

✅ **Good:**

```
function deleteCourse(courseId) {
   const DELETE_CONFIRMATION_MESSAGE = "Are you sure you want to
delete this course?";
   if (!confirm(DELETE_CONFIRMATION_MESSAGE)) return;
```

```
    window.location.href =
`delete_course.php?id=${encodeURIComponent(courseId)}`;
}
```

---

4. Security Best Practices

Escape user inputs before inserting into URLs to prevent XSS attacks.
✅ **Good:**

```
 window.location.href =
`delete_course.php?id=${encodeURIComponent(courseId)}`;
```

-

Prevent Global Scope Pollution by wrapping code inside an IIFE (Immediately Invoked Function Expression).
✅ **Good:**

```
 (function () {
    const profileBtn = document.getElementById("profile-btn");
    const dropdownMenu = document.getElementById("dropdown-menu");

    function toggleDropdown() {
        dropdownMenu.classList.toggle("hidden");
    }

    function closeDropdown(event) {
        if (!profileBtn.contains(event.target) &&
!dropdownMenu.contains(event.target)) {
            dropdownMenu.classList.add("hidden");
        }
    }

    profileBtn.addEventListener("click", toggleDropdown);
    document.addEventListener("click", closeDropdown);
})();
```

5. Avoid Hardcoded Strings

- Store confirmation messages in **constants** for better maintainability.

✅ **Good:**

```
const DELETE_CONFIRMATION_MESSAGE = "Are you sure you want to
delete this course?";

function deleteCourse(courseId) {
    if (!confirm(DELETE_CONFIRMATION_MESSAGE)) return;
    window.location.href =
`delete_course.php?id=${encodeURIComponent(courseId)}`;
}
```

6. Avoid Repetitive Code

- Use **utility functions** for common operations like toggling visibility.

✅ **Good:**

```
function toggleVisibility(element) {
    element.classList.toggle("hidden");
}

profileBtn.addEventListener("click", () => toggleVisibility(dropdownMenu));
```

PHP Coding Standards

1. General Guidelines

- **File Naming**: Use lowercase letters and hyphens for file names (my-class.php, user-profile.php). Avoid underscores.
- **File Extensions**: Always use .php for PHP files.
- **Encoding**: Use UTF-8 without BOM (Byte Order Mark).
- **Line Length**: Limit lines to 80-120 characters, especially for readability.
- **Line Endings**: Use Unix-style line endings (LF), not Windows-style (CRLF).
- **Whitespace**:
  - Always include one blank line after the namespace and use statements.
  - No extra spaces at the end of a line.
  - Indentation should be done using **spaces**, not tabs. Typically, 4 spaces per indentation level.

2. PHP Tags

- Always use the short PHP tag <?php instead of <? for compatibility.

```
<?php
// Good
```

3. Naming Conventions

- **Variables**: Use **camelCase** for variables (e.g., $userId, $userName).
- **Functions**: Functions should also use **camelCase** (e.g., getUserInfo(), fetchDataFromDb()).
- **Classes**: Classes should follow **PascalCase** (e.g., UserController, ProductService).
- **Constants**: Use **UPPER_SNAKE_CASE** for constants (e.g., MAX_FILE_SIZE).
- **Methods & Functions**: Methods should be named with a verb and describe an action (e.g., getUser(), deletePost()).

- **Class Names**: Class names should be singular (e.g., User, Post).

4. Variables

- Always initialize variables.
- Avoid using single-letter variables except for loop counters (e.g., $i, $j).
- Use descriptive variable names to make your code self-explanatory (e.g., $userId vs. $x).

5. Arrays

- Use **short array syntax**: [] instead of array().
- Avoid array shorthand if the array has associative keys (e.g., ['key' => 'value']).

```
// Good
$arr = [1, 2, 3];

// Bad
$arr = array(1, 2, 3);
```

6. Functions and Methods

- Functions should have clear, descriptive names and should do one thing only.
- Avoid functions that do multiple things, split them into smaller methods.

Return early to make your code more readable:

```
 function getUserInfo($id) {
    if (!$id) {
        return null; // Early exit if no id
    }

    // Fetch user info here
}
```

-
- Use null for no value and false for failure where appropriate.

7. Control Structures

- **Always use braces** ({}) for control structures (if, while, foreach, etc.) even if the block contains only one line of code. This improves readability and avoids errors.

```
// Good
if ($user) {
    echo "User exists";
}

// Bad
if ($user)
    echo "User exists";
```

Single-line conditionals should only be used for short, simple statements:

```
 // Good
if ($condition) echo "Success";
```

-

8. Commenting

- **Block comments**: Use for functions, complex logic, or code that isn't self-explanatory.

```
/**
 * Retrieves a user by their ID.
 *
 * @param int $id The user ID.
 * @return User|null The user object or null if not found.
 */
function getUserById($id) {
    // function logic
}
```

- **Inline comments**: Use sparingly and only for clarification where necessary.

```
$result = $database->query("SELECT * FROM users"); // Get all users
```

9. Error Handling

- Use **exceptions** for error handling where possible.
- Avoid die() and exit() for normal control flow. Use them only for emergencies.
- Wrap code that might throw exceptions inside try-catch blocks.

```
try {
   // Risky code
} catch (Exception $e) {
   echo $e->getMessage();
}
```

10. SQL Queries

- **Prepared Statements**: Always use prepared statements to protect against SQL injection.

```
$stmt = $conn->prepare("SELECT * FROM users WHERE id = ?");
$stmt->bind_param("i", $userId);
$stmt->execute();
```

- Use **bind_param** for variables passed to SQL queries.
- Always escape user inputs before using them in a query.

11. Autoloading

- Follow PSR-4 autoloading standards. Use **Composer** to manage autoloading of classes.
- Organize files by namespace and class name. For example, src/Controller/UserController.php should define namespace Controller; class UserController.

12. Security Practices

- Always **sanitize** user input (e.g., using filter_var(), htmlspecialchars(), etc.).
- **Escape outputs** when displaying user data to prevent XSS attacks.
- **Hash passwords** using a secure method (password_hash() and password_verify()).
- Use **CSRF tokens** for form submissions to prevent cross-site request forgery.

13. File Organization

- Separate logic into well-defined files or classes.
- Place classes in their own file (User.php for the User class).
- Group files logically into directories (e.g., controllers/, models/, views/).

14. Database Interactions

- Use **PDO** or **MySQLi** for database interactions.

For **PDO**:

```
 $pdo = new PDO($dsn, $username, $password);
$stmt = $pdo->prepare("SELECT * FROM users WHERE id = :id");
$stmt->execute(['id' => $userId]);
$user = $stmt->fetch();
```

-

15. PSR Recommendations

- **PSR-1**: Basic Coding Standard (e.g., class names and methods must follow the naming conventions).

- **PSR-2**: Coding Style Guide (indenting with 4 spaces, method and function declarations).
- **PSR-4**: Autoloading standard for classes.

Summary Checklist:

- **Naming**: CamelCase for variables and methods, PascalCase for classes, UPPER_SNAKE_CASE for constants.
- **Control Structures**: Always use braces {}, no single-line conditionals.
- **Indentation**: Use 4 spaces per indentation level.
- **Comments**: Block comments for complex logic, inline comments for clarification.
- **Security**: Sanitize and validate input, use prepared statements, escape output.

HTML Coding Standards

1. General Best Practices

- Use semantic HTML elements to improve accessibility and SEO.
- Maintain a clean and well-structured document layout.
- Use lowercase for element names, attributes, and filenames.
- Properly close all tags, including self-closing ones.
- Indent nested elements consistently using either 2 or 4 spaces.

Example:

```
<!-- Good Practice -->

<section>

    <h2>Our Services</h2>
    <p>We provide web development solutions.</p>
</section>


<!-- Bad Practice -->
<div>
    <b>Our Services</b>
    <br>We provide web development solutions.
</div>
```

2. File Structure & Naming

- Use meaningful and descriptive filenames.
- Use lowercase letters with hyphens for file names (e.g., `contact-form.html`).
- Organize files in structured directories (e.g., `/pages`, `/components`).

3. Document Structure

- Always include the `<!DOCTYPE html>` declaration.
- Define the `lang` attribute in the `<html>` tag for accessibility.
- Include `<meta charset="UTF-8">` for proper character encoding.
- Use `<meta name="viewport">` to ensure mobile responsiveness.
- Keep `<title>` short, descriptive, and relevant.

Example:

```
<!DOCTYPE html>
```

```
<html lang="en">

<head>

   <meta charset="UTF-8">

   <meta name="viewport" content="width=device-width, initial-scale=1.0">

   <title>My Website</title>

</head>

<body>

</body>

</html>
```

4. Headings & Text

- Follow a logical heading hierarchy (<h1> to <h6>) without skipping levels.
- Use a single <h1> per page, representing the main topic.
- Avoid excessive use of <br> for spacing; use CSS instead.
- Provide meaningful alternative text (alt) for images.

Good Example:

```
<h1>About Us</h1>  <!-- Main heading -->

<h2>Our Mission</h2>  <!-- Subheading -->

<p>We aim to deliver the best services.</p>
```

Bad Example

```
<h1>About Us</h1>

<h3>Our Mission</h3> <!-- Skipped h2 -->
```

5. Links & Buttons

- Use descriptive anchor text instead of generic phrases like "click here."
- Include alt attributes in images within links for accessibility.
- External links should open in a new tab using target="_blank" and include rel="noopener noreferrer".
- Buttons should be used for actions, while links should be used for navigation.

Example:

```
<a href="https://example.com" target="_blank" rel="noopener noreferrer">Visit Our Website</a>

<button type="button">Learn More</button>
```

6. Forms & Inputs

- Associate input fields with labels for accessibility.
- Use the appropriate input type (email, password, number, etc.) for better validation.
- Group related inputs inside a <fieldset> for better organization.
- Use required and other validation attributes when applicable.

Example:

```
<form action="/submit" method="POST">

   <label for="email">Email:</label>

   <input type="email" id="email" name="email" required>

   <button type="submit">Submit</button>

</form>
```

7. Accessibility (a11y)

- Provide meaningful alt attributes for images.
- Use aria-label attributes when necessary to improve screen reader support.

- Ensure all interactive elements are keyboard accessible.

Example:

```
<img src="logo.png" alt="Company Logo">

<button aria-label="Close Menu">X</button>
```

8. Performance & SEO

- Use external stylesheets instead of inline styles.
- Optimize images to reduce loading times.
- Use semantic HTML elements to improve search engine visibility.
- Defer or asynchronously load JavaScript files to prevent render blocking.

Example:

```
<link rel="stylesheet" href="styles.css">

<script src="script.js" defer></script>
```

Here are the best practices and coding standards for PHP development to ensure clean, maintainable, and efficient code.

PHP Coding Standards and Best Practices

1. General Best Practices

- Use the **latest PHP version** whenever possible.
- Follow **PSR (PHP Standard Recommendations)**, mainly:
    - **PSR-1** (Basic coding standards)
    - **PSR-2** (Coding style guide)

    - **PSR-4** (Autoloading standard)
- Keep **code DRY (Don't Repeat Yourself)** and follow **SOLID** principles.
- Avoid using **global variables**; instead, use dependency injection.
- Use **composer** to manage dependencies.

2. File & Directory Structure

Keep a **logical structure**, separating concerns:
/app
/config
/public
/routes
/resources
/views
Naming Conventions:

- Files should be in snake_case.php (user_controller.php).
- Class names should follow StudlyCaps (UserController).
- Variables and function names should be camelCase (getUserName).
- Constants should be uppercase with underscores (MAX_UPLOAD_SIZE).

3. Coding Style & Formatting

Indentation & Spacing

- Use **4 spaces per indentation**, no tabs.
- Add **spaces around operators** ($sum = $a + $b;).
- No trailing spaces at the end of lines.

Braces & Control Structures

Opening braces { should go **on the same line**.

Use **consistent spacing** for control structures:

```
if ($condition) {
    // Code
} elseif ($otherCondition) {
    // Code
} else {
    // Code
}
```

4. Variables & Data Handling

Variable Naming

Use **descriptive names**, and avoid single-letter variables.

Use **camelCase** for variable names:

```
$userName = "John Doe";
```

String Handling

Prefer **single quotes (')** over double quotes (") unless interpolation is needed:

```
$name = 'John';
echo "Hello, $name"; // Good
echo 'Hello, ' . $name; // Also good
```

Array Handling

Use **short array syntax ([])**:

```
$users = ['John', 'Alice', 'Bob'];
```

Associative arrays should have **aligned keys** for readability:

```
$user = [
    'name'  => 'John',
    'email' => 'john@example.com',
    'age'   => 30
];
```

5. Functions & Methods

Function Naming

Use camelCase for function names:

```
 function getUserData($id) {
    return "User: " . $id;
}
```

-
- Functions should be **self-explanatory and short**.

Parameter & Return Type Hints (Strong Typing)

Use **type hints** for parameters and return values:

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

Default Parameter Values

Use default values for optional parameters:

```
 function greet($name = 'Guest') {
    return "Hello, $name!";
}
```

6. Classes & Object-Oriented Programming

- Use **PSR-4 Autoloading** for class files.

- **Class Naming**:

  - Use StudlyCaps for class names (UserController).
  - Use camelCase for method names (getUserData).

Use **proper access modifiers** (public, private, protected):

```
 class User {
    private string $name;

    public function setName(string $name) {
        $this->name = $name;
    }

    public function getName(): string {
        return $this->name;
    }
}
```
Use **constructor dependency injection** instead of global variables:

```
class UserController {
    private Database $db;

    public function __construct(Database $db) {
        $this->db = $db;
    }
}
```

-
- Use **interfaces & abstract classes** where needed.

- Use **traits** for reusable methods instead of multiple inheritance.

7. Security Best Practices

Sanitize & validate user input using filter_var() or htmlspecialchars():

```
$email = filter_var($_POST['email'], FILTER_SANITIZE_EMAIL);
```

Use prepared statements for database queries to prevent SQL injection:

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE email = ?");
$stmt->execute([$email]);
```

- **Hash passwords** using password_hash():
  ```
  $hashedPassword = password_hash($password, PASSWORD_BCRYPT);
  ```
-
- **Avoid storing sensitive data in plain text**.
- **Use HTTPS** for secure data transmission.

8. Error Handling & Logging

Use try-catch blocks for exception handling:

```
 try {
   $db = new PDO($dsn, $user, $password);
} catch (PDOException $e) {
   error_log($e->getMessage());
   die("Database connection failed.");
}
```

Use **error logging** instead of displaying errors in production:

```
 ini_set('display_errors', 0);
ini_set('log_errors', 1);
error_reporting(E_ALL);
```

9. Performance Optimization

- Use **OPcache** to cache compiled PHP scripts.
- Avoid **unnecessary database queries**; use caching if possible.
- Minimize **loops and function calls** inside loops.
- Use **JSON instead of XML** for data exchange when possible.
- Optimize database queries by using **indexes**.
- **Avoid excessive session usage**; clear old sessions.

10. Commenting & Documentation

Use **PHPDoc comments** for functions and classes:

```
 /**
 * Get user data by ID.
 *
 * @param int $id User ID.
 * @return array User data.
 */
function getUser(int $id): array {
   return [];
}
```

-

- Keep comments **short and meaningful**.
- Avoid **redundant comments** that simply repeat the code.