Coding Standard By Labonno

Course Code: ICE-4102
Course Title: Software Testing and Maintenance Lab

Submitted by

Group-04

Submitted to

Dr. Md. Musfique Anwar, Professor, CSE, JU

Md. Sazzadul Islam Prottasha, Lecturer, ICT, BUP

Information and Communication Technology

Bangladesh University of Professionals

Dhaka, Bangladesh

Contents

1. Abstract

This report presents the coding standards applied in the development of the project, covering PHP, HTML, CSS, JavaScript, and SQL. The focus of these standards is to ensure that the code is readable, maintainable, consistent, and secure.

The document explains the conventions followed for variable naming, functions, classes, files, HTML classes and IDs, CSS styling, JavaScript practices, and database structure. It also covers commenting practices, formatting, indentation, error handling, security measures such as input validation, SQL injection prevention, password hashing, session management, and access control.

By adhering to these standards, the project ensures high-quality, maintainable code, promotes collaboration among developers, and reduces potential security risks. Examples of properly structured and documented code are included to demonstrate compliance with the standards.

2. OBJECTIVE

The main objectives of this task are:

- To understand and apply coding standards for PHP, HTML, CSS, JavaScript, and SQL.
- To ensure code readability, maintainability, and consistency across the project.
- To practice proper naming conventions for variables, functions, classes, files, and database objects.
- To implement security best practices in coding, including input validation, SQL injection prevention, and secure session handling.
- To improve collaboration among developers through consistent code structure and documentation.

Expected Outcomes:

By the end of this task, participants are expected to:

- Write clean, readable, and maintainable code following the defined coding standards.
- Apply consistent naming conventions across all code components and database objects.
- Ensure secure coding practices are implemented in PHP, HTML, JavaScript, and SQL.
- Produce well-documented code with comments, structured formatting, and modular design.
- Collaborate effectively on the project with reduced errors and easier debugging.

3. CODING STANDARDS

Introduction

Coding standards are a set of guidelines that ensure code is readable, maintainable, and consistent across a project. These standards improve collaboration, reduce errors, enhance security, and simplify maintenance. This project applies standards to PHP, HTML, CSS, JavaScript, SQL, and overall project structure.

1. HTML Standards

- **Semantic Structure:** Use semantic tags like `<header>`, `<section>`, `<footer>` for better readability and accessibility.
- **Class Naming:** Use `kebab-case` for CSS classes.
- **ID Naming:** Use `camelCase` for unique element identifiers (usually JS hooks).
- **Attributes:** Include `alt` for images and descriptive `name` for form inputs.
- **Accessibility:** Use ARIA attributes like `role` and `aria-label` where necessary.
- **Forms:** Use proper input types, `required`, and `pattern` for client-side validation.
- **Meta tags:** Proper charset and viewport in `<head>` for responsive design.

Example:

```
<div id="adBanner" class="ad-banner" role="region" aria-label="Adver
  <span class="ad-close" onclick="closeAd()">&times;</span>
  <a href="handmade/index.html" target="_blank" class="ad-link">
    <img id="adImage" src="images/destination/ad.jpg" alt="Advertise
  </a>
</div>
```

```
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0
```

2. CSS Standards

- Classes: `kebab-case` (e.g., `info-box`, `read-more-btn`).
- IDs: `camelCase` for unique elements.
- Indentation: 2 spaces.
- Avoid inline styles; use classes.
- Use CSS variables for colors, fonts, and spacing.
- Responsive design with mobile-first media queries.
- Comment sections for clarity and maintainability.

Example:

```
:root {
  --primary-color: #44f19a;
  --secondary-color: #fff;
  --font-size-base: 16px;
}

.signup-op a {
  text-decoration: none;
  border: 2px solid #000;
  background-color: var(--primary-color);
  padding: 8px 20px;
  color: black;
  transition: 0.8s ease;
}

.signup-op a:hover {
  background-color: var(--secondary-color);
}
```

```
@media (max-width: 768px) {
  .ad-banner { display: none; }
}
```

3. JavaScript Standards

- Variables and functions: `camelCase`.
- Constants: UPPER$_{C}ASE$.
- Use ES6+ features: let, const, arrow functions, template literals.
- Encapsulate logic in window.onload or IIFE to avoid global scope pollution.
- Event delegation preferred for multiple elements.
- Error handling using try-catch.
- Modular JS: separate files per component.

Example:

```
window.onload = function () {
  const adImages = ["images/ad1.jpg","images/ad2.jpg"];
  let currentAdIndex = 0;
  const adImageElement = document.getElementById("adImage");

  const changeAd = () => {
    currentAdIndex = (currentAdIndex + 1) % adImages.length;
    adImageElement.src = adImages[currentAdIndex];
  }

  const adInterval = setInterval(changeAd, 3000);

  window.closeAd = () => {
    document.getElementById("adBanner").style.display = "none";
    clearInterval(adInterval);
  };
};
```

4. PHP Standards

- Variables: snake$_{c}ase(\$user_{i}d, \$total_{p}roducts)$.
- Functions: camelCase (loginUser(), addToCart()).
- Classes: PascalCase (UserController).
- Files: lowercase with underscores (connect.php, user$_{h}eader.php$).

- Strict typing enabled (declare(strict$_{t}ypes = 1);).
- Use require_once for critical includes.
- Input validation with filter_input() and sanitization.
- Error reporting set differently for development vs production.

Example:

```
declare(strict_types=1);

// Password hashing and prepared statement
$passwordHash = password_hash($password, PASSWORD_BCRYPT);

$stmt = $conn->prepare("INSERT INTO users (name,email,password) VALU
$stmt->execute([$name, $email, $passwordHash]);

// Input sanitization
$name = filter_input(INPUT_POST, 'name', FILTER_SANITIZE_STRING);
```

5. Database Standards

- Table and column names: snake$_{c}ase$.
- Primary keys: auto-increment integers.

- SQL keywords uppercase, columns/tables lowercase.
- Foreign keys for table relationships.
- Indexing for performance optimization.
- Character set: utf8mb4.
- Auto timestamps: created_at, updated_at.
- Use prepared statements for all queries.

Example:

```
CREATE TABLE `products` (
  `id` int(100) NOT NULL AUTO_INCREMENT,
  `name` varchar(100) NOT NULL,
  `details` varchar(500) NOT NULL,
  `price` int(10) NOT NULL,
  `image_01` varchar(100) NOT NULL,
  `created_at` TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `orders` (
  order_id INT AUTO_INCREMENT PRIMARY KEY,
  user_id INT NOT NULL,
  total DECIMAL(10,2) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

6. Security Practices

- **Input Validation:** Sanitize and validate using htmlspecialchars(), strip$_tags(), and$fi
- **SQL Injection Prevention:** Prepared statements and parameterized queries for all database interactions.
- **XSS Prevention:** Escape dynamic content output with htmlspecialc

- **CSRF Protection:** Include CSRF tokens in all critical forms and verify them server-side.
- **Password Security:** Hash passwords using password_hash() (BCRYPT) and verify using password_verify().
- **Session Management:** Secure session initialization with session_start(), regenerate session ID on login, and destroy sessions on logout.
- **Access Control:** Role-based access restrictions for sensitive pages (admin dashboard, user management).
- **HTTPS:** Use HTTPS protocol to encrypt data in transit.
- **Error Handling:** Avoid exposing internal error messages; log errors securely.

Example:

```
// Input sanitization
$comment = htmlspecialchars(strip_tags($_POST['comment']));

// SQL Injection Prevention
$stmt = $conn->prepare("SELECT * FROM users WHERE email=?");
$stmt->execute([$email]);

// CSRF token check
if ($_POST['csrf_token'] !== $_SESSION['csrf_token']) {
    die("Invalid CSRF token.");
}

// Password hash verification
if (password_verify($password, $storedHash)) { ... }

// Secure session
session_start();
session_regenerate_id(true);
```

7. Project Structure and Version Control

- **Folder structure:**

```
/assets
  /css
  /js
  /images
/components
  header.php
  footer.php
  user_header.php
  connect.php
/pages
  home.php
  products.php
  cart.php
  login.php
/config
  config.php
/vendor
index.php
README.md
```

- Git branches: main (production), dev (development), feature/xyz.
- Commit messages: `feat`, `fix`, `docs`, `refactor`, `chore`.
- Follow GitHub workflow for pull requests and code reviews.

References

1. PSR-12: Extended Coding Style Guide. `https://www.php-fig.org/psr/psr-12/`
2. W3C HTML & CSS Guidelines. `https://www.w3.org/standards/webdesign/htmlcss`

3. JavaScript Style Guide (Airbnb). `https://github.com/airbnb/javascript`

4. MySQL Naming Conventions. `https://dev.mysql.com/doc/`

4. CONCLUSIONS

Adhering to coding standards ensures our project is readable, maintainable, and secure. Consistent naming, modular code, and proper project structure make collaboration easier, reduce errors, and simplify future updates. Security best practices protect user data, while standardized workflows support scalability and high-quality development. Following these guidelines lays a strong foundation for clean, professional code.