Coding Standard

1. Consistency

- Maintain consistent coding style across the entire project.
- Use uniform **naming conventions**, **indentation**, and **formatting** throughout all files to enhance readability and maintainability.

---

2. Naming Conventions

2.1 Variables

- Use **camelCase** for variable names.
- Variable names should be descriptive and meaningful.
- Prefix internal variables with _ if necessary.

Examples:

```
let cartItems = [];
let totalPrice = 499;
let _internalFlag = true;
```

2.2 Constants

- Use **UPPERCASE_SNAKE_CASE** for constants.
- Declare constants with `const`.

Examples:

```
const MAX_CART_SIZE = 50;
const DEFAULT_SORT_ORDER = "asc";
```

2.3 Functions

- Use **camelCase** for function names.
- Function names should describe the action they perform.

Examples:

```
function addToCart(bookId) {
    // logic
}

function sortBooksByDate(order) {
    // logic
}
```

2.4 Classes

- Use **PascalCase**.
- Class names should be nouns and describe what the class represents.

Examples:

```
class CartManager {
    constructor() {
        this.items = [];
    }
}
```

2.5 Packages and Modules

- Use **short, lowercase** names for files/modules.
- Avoid using underscores in file names.

Example:

```
/utils/cart.js
/controllers/sort.js
```

---

3. Comments and Documentation

- Use `/** ... */` for function and class documentation.
- Use `//` for inline or single-line comments.

Example:

```
/**
 * Calculates the total price of the cart.
 * @param {Array} items - List of book objects.
 * @returns {number}
 */
function calculateTotal(items) {
    return items.reduce((sum, item) => sum + item.price, 0);
}
```

---

4. Formatting and Indentation

- Use **2 spaces** or **4 spaces** consistently (decide as a team — 2 is more common in JS).
- Limit line length to **80–100 characters**.
- Use proper indentation for blocks and functions.

---

5. Error Handling

- Use `try...catch` for error-prone code.
- Log or rethrow errors with meaningful messages.

Example:

```
try {
    processOrder(cart);
} catch (error) {
    console.error("Order processing failed:", error);
}
```

---

6. Import Formatting

- Use **ES6 import/export** syntax.
- Group imports:
    - Built-in modules
    - Third-party packages
    - Local modules

Example:

```
// Standard library (if using Node.js)
import fs from 'fs';

// Third-party
import express from 'express';

// Local
import { addToCart } from './cartUtils.js';
```

---

7. URL Formatting (if applicable for APIs)

- Use **lowercase** with hyphens or underscores to separate words.

Example:

```
/api/sort-by-price
/api/add-to-cart
```

---

8. Template Style (HTML/JSX)

- Use clean, properly indented HTML.
- Use semantic tags and descriptive class names.

Example:

```
<div class="book-card">
  <h2>{{ book.title }}</h2>
  <button onclick="addToCart(book.id)">Add to Cart</button>
</div>
```

---

9. Code Readability

- Break large logic into smaller functions.
- Use descriptive variable and function names.

---

10. Code Reusability

- Extract repeated logic into utility functions or reusable components.

Example:

```
function isEmptyCart(cart) {
    return cart.length === 0;
}
```

---

11. Testing and Quality Assurance

- Use `Jest`, `Mocha`, or `Vitest` for unit testing.
- Write clear, independent tests.

Example with Jest:

```
test('adds book to cart', () => {
    const cart = [];
    addToCart(cart, { id: 1, price: 100 });
    expect(cart.length).toBe(1);
});
```

---

12. Security

- Always validate and sanitize user inputs (especially if accepting form data or working with databases).
- Prevent client-side injection or manipulation in sort/cart logic.