Web App: Coding Standards and Guidelines

1. Purpose

This document defines coding standards, best practices, and development guidelines for building maintainable, scalable, secure, and high-performance web applications using:

- HTML5
- CSS3
- React.js
- Node.js

The objectives are:

- Improve code readability and consistency
- Facilitate collaboration among developers
- Reduce technical debt
- Enhance application security and performance
- Simplify maintenance and future enhancements

2. General Development Principles

2.1 Follow SOLID Principles

Single Responsibility Principle (SRP)

Each module, component, and function should have only one responsibility.

Good

```
function calculateTax(amount) {
  return amount * 0.18;
}
```

Avoid

```
function processInvoice(invoice) {
  calculateTax();
  sendEmail();
  saveDatabase();
}
```

2.2 DRY (Don't Repeat Yourself)

Avoid duplicated logic.

Good

```
export const formatCurrency = amount =>
  new Intl.NumberFormat('en-US').format(amount);
```

3. Project Structure

React Frontend

```
src/
├── assets/
├── components/
│   ├── common/
│   └── feature/
├── hooks/
├── pages/
├── services/
├── store/
├── utils/
├── routes/
├── styles/
├── constants/
└── App.jsx
```

Node.js Backend

```
src/
├── config/
├── controllers/
├── middleware/
├── models/
├── routes/
├── services/
├── repositories/
├── utils/
├── validations/
├── app.js
└── server.js
```

4. HTML Standards

4.1 Use Semantic HTML

Good

```
<header>
<nav>
<main>
<section>
<article>
<footer>
```

Avoid

```
<div class="header">
<div class="navigation">
```

4.2 Accessibility

Every page must comply with WCAG (Web Content Accessibility Guidelines: technical standards developed by the W3C) .

Images

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

Form Inputs

```
<label for="email">Email</label>
<input id="email" type="email">
```

Buttons

```
<button type="submit">
  Save
</button>
```

Never use clickable divs.

4.3 Document Structure

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Application</title>
 </head>
 <body>
 </body>
</html>
```

5. CSS Standards

5.1 Naming Convention

Use BEM (Block, Element, Modifier that is .block__element--modifier) naming convention.

```
.card {}
.card__title {}
.card__description {}
.card--active {}
```

A child part of a Block that has no standalone meaning and is semantically tied to its parent is appended to the Block name using a double underscore (__). Modifier which is a flag on a Block or Element used to change its appearance, state, or behaviour is ppended to the Block or Element using a double hyphen (--).

5.2 Avoid Inline Styles

Avoid

```
<div style={{ color: 'red' }}>
```

Preferred

```
<div className="error-text">
```

5.3 CSS Organization

Order properties consistently.

```
.card {
  display: flex;

  width: 100%;
  height: auto;

  margin: 1rem;
  padding: 1rem;

  background: white;
  border-radius: 8px;

  font-size: 1rem;
}
```

5.4 Responsive Design

Use relative units.

```
.container {
  width: 100%;
  max-width: 1200px;
}
```

Prefer:

- rem
- em
- %
- vw
- vh

Avoid fixed pixel layouts.

5.5 CSS Variables

Declaring a variable requires prefixing the name with two dashes (`--`). To retrieve and apply the value, wrap the variable name inside the **`var()`.**

```
:root {
  --primary-color: #2563eb;
  --secondary-color: #64748b;
  --spacing-md: 16px;
  --main-padding: 10px
}

.button {
  background-color: var(--primary-color);
  padding: var(--main-padding)
}
```

Use **variables** throughout the application.

6. JavaScript Standards

6.1 Use ES6+

Use:

- let
- const
- arrow functions
- template literals
- destructuring
- optional chaining

```
const userName = user?.profile?.name;
```

6.2 Variable Naming

Variables

```
const customerName;
const totalAmount;
```

Constants

```
const MAX_RETRY_COUNT = 3;
```

Boolean Variables

```
isActive;
hasPermission;
canEdit;
```

6.3 Function Naming

Use **verbs**.

```
fetchUsers();
saveOrder();
calculateTotal();
```

6.4 Function Length

- Maximum 40 lines preferred
- One responsibility per function

6.5 Avoid Hard Coded Numbers

Bad

```
if (age > 18)
```

Good

```
const LEGAL_AGE = 18;

if (age > LEGAL_AGE)
```

6.6 Error Handling

```
try {
  await saveUser(user);
} catch (error) {
  logger.error(error);
  throw error;
}
```

Never silently ignore errors.

7. React Standards

7.1 Component Naming

Use PascalCase.

```
UserProfile.jsx
OrderSummary.jsx
```

7.2 Functional Components Only

```
function UserCard() {
  return <div>User</div>;
}
```

Avoid class components.

7.3 Component Size

A component should ideally be under 200 lines.

Split large components into smaller reusable pieces.

7.4 Props Destructuring

```
function UserCard({ name, email }) {
  return (
    <>
      <h2>{name}</h2>
      <p>{email}</p>
    </>
  );
}
```

7.5 State Management

Local State

```
const [loading, setLoading] = useState(false);
```

Global State

Use:

- Redux Toolkit
- Context API

Avoid excessive global state.

7.6 Custom Hooks

Extract reusable logic.

```
function useUsers() {
  const [users, setUsers] = useState([]);

  return { users };
}
```

7.7 API Calls

Never call APIs directly inside UI components.

Good

```
services/userService.js

export const getUsers = () =>
  api.get('/users');
```

7.8 Keys in Lists

```
users.map(user => (
  <UserCard
    key={user.id}
    user={user}
  />
))
```

Never use array indexes as keys.

7.9 React Performance

Use:

```
useMemo()
useCallback()
React.memo()
```

Only when necessary.

Avoid premature optimization.

8. Node.js Standards

8.1 Layered Architecture

```
Route
 → Controller
   → Service
      → Repository
         → Database
```

8.2 Controllers

Controllers should only:

- Validate requests
- Call services
- Return responses

```
async function getUsers(req, res) {
  const users = await userService.getUsers();

  res.json(users);
}
```

8.3 Business Logic

Business logic belongs in services.

```
async function createOrder(order) {
  validateOrder(order);

  return repository.save(order);
}
```

8.4 Async/Await

Always use async/await.

```
const users = await userRepository.findAll();
```

Avoid nested promise chains.

8.5 Environment Variables

Store configuration in environment files.

```
PORT=3000
DATABASE_URL=
JWT_SECRET=
```

Never hardcode secrets.

8.6 Logging

Use structured logging.

```
logger.info('User created', {
  userId
});
```

Do not use console.log in production.

9. API Standards

9.1 REST Naming

Good

```
GET    /users
```

```
GET    /users/123
POST   /users
PUT    /users/123
DELETE /users/123
```

Avoid

```
GET   /getUsers
POST  /createUser
```

9.2 Response Format

Success

```
{
  "success": true,
  "data": {}
}
```

Error

```
{
  "success": false,
  "message": "Invalid request"
}
```

9.3 HTTP Status Codes

Code	Meaning
200	Success
201	Created
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
409	Conflict
500	Internal Server Error

10. QA and Testing

Test coverage is a software quality metric that measures how much of your application's code or functionality is exercised by automated tests.

1. **Line Coverage**: Percentage of code lines executed during tests
2. **Statement Coverage**: Percentage of executable statements tested
3. **Branch Coverage**: Percentage of decision paths (if/else, switch cases) tested
4. **Function Coverage**: Percentage of functions/methods called by tests.

10.1 Recommended Coverage Targets for Web Applications

Coverage targets should balance quality, effort, and maintenance costs.

Application Type	Recommended Coverage
Internal tools / prototypes	60–70%
Standard business web applications	75–85%
Enterprise applications	80–90%
Financial, healthcare, safety-critical systems	90–100% (for critical modules)

10.2 Recommended Target for a React + Node.js Web Application

A practical industry target is:

Layer	Coverage Target
React Components	> 80%
Business Logic	> 90%
API Services	> 85%
Utility Functions	> 90%
Critical Workflows	> 95%
Overall Project	80–85%

10.3 Test Pyramid for Web Applications

Unit Tests (70%)

- React components, Hooks, Utility functions, Node.js services

Integration Tests (20%)

- API endpoints, Database interactions, Authentication flows

End-to-End Tests (10%)

- Login, Create project, Upload file, Dashboard navigation, User workflows