Coding Standard

Naming Conventions:

1. Variables:

Use camelCase for variable and function names. Also use verbs for function names to clearly define actions.

Example:

```
// Good Practice
let userName = 'JohnDoe';
function getUserData() {
  // function body
}

// Bad Practice
let UserName = 'JohnDoe'; // Starts with uppercase
function get_user_data() {
  // Uses underscores
}
```

Eslint Rule: `camelCase`

2. Constants:

Use UPPERCASE with underscores for constants.

Example:

```
// Good Practice
const MAX_LIMIT = 100;
const API_URL = 'https://api.example.com';

//Bad Practice
const MaxLimit = 100; // Not all uppercase
const apiUrl = 'https://api.example.com'; // Not all uppercase
```

Eslint Rule: `no-underscore-dangle`

3. Classes:

Use PascalCase for class names.

Example:

```
// Good Practice
class UserProfile {
  // class body
}

// Bad Practice
class userProfile {
  // Starts with lowercase
}
```

Eslint Rule: `new-cap`

4. Private Members:

Prefix private variables or methods with an underscore (_).

Example:

```
// Good Practice
class User {
  constructor() {
    this._password = 'secret';
  }

  _validatePassword() {
    // method body
  }
}

// Bad Practice
class User {
  constructor() {
    this.password = 'secret'; // Missing underscore
  }

  validatePassword() {
    // Missing underscore
  }
}
```

Eslint Rule: `no-underscore-dangle`

5. **Booleans:**

Start boolean variables with is, has, or should for clarity.

Example:

```
// Good Practice
let isActive = true;

//Bad Practice
let a = true; // Does not start with is or has, and has no clarity
```

Layout Conventions:

1. **Indentation:**

Use 2 spaces for indentation.

Example:

```
// Good Practice
function fetchData() {
  if (true) {
    console.log('Indented with 2 spaces');
  }
}

// Bad Practice
function fetchData() {
  if (true) {
      console.log('Incorrect indentation'); // Indented with 4 spaces
  }
}
```

Eslint Rule: `indent`

2. **Line Length:**

Limit lines to 80-100 characters.

Example:

```
// Good Practice
const userDataGood = fetchUserData(userId, includeAddress,
includePreferences);

// Bad Practice
const userDataBad = fetchUserData(userId, includeAddress, includePreferences,
additionalParam, anotherParam); // Line too long
```

Eslint Rule: `max-len`

3. **Semicolons:**

Always end statements with semicolons.

Example:

```
// Good Practice
let totalGood = 0;
function calculate() {
  // function body
}

// Bad Practice
let totalBad = 0
function calculate() {
  // Missing semicolons
}
```

Eslint Rule: `semi`

4. **Curly Braces for Loops, Functions, and Conditions:**

Place the opening brace on the same line as the loop, function, or condition. Closing braces should be on a new line after the block. Always use braces even for single-line statements.

Example:

```
// Good Practice
// For loops
for (let i = 0; i < 5; i++) {
  console.log(i);
}

// Functions
function sayHello(name) {
  console.log(`Hello, ${name}`);
}

// Conditions
if (isValid) {
  proceed();
} else {
  halt();
}

// Bad Practice
// Opening Brace on newline
function sayHello(name)
{
  console.log(`Hello, ${name}`);
}
```

```
// Missing braces
if (isValid) proceed();
```

Eslint Rule: `brace-style, curly`

5. Spaces:

Place spaces around operators and after commas.

Example:

```
// Good Practice
let sumGood = a + b;
function multiply(x, y) {
  return x * y;
}

// Bad Practice
let sumBad=a+b; // No spaces around operators
function multiply(x,y){
  return x*y;
}
```

Eslint Rule: `space-infix-ops, comma-spacing`

6. Newlines:

Enforce a newline between code blocks and at the end of files.

Example:

```
// Good Practice
function init() {
  // initialization code
}

function start() {
  // start code
}

// Bad Practice
function init() {
  // initialization code
}
function start() {
  // Missing newline between functions
  // start code
}
```

Eslint Rule: `newline-after-var, eol-last`

Member Order:

1. Static properties
2. Instance properties
3. Constructor
4. Static methods
5. Public methods
6. Private methods

Example:

```
// Good Practice
class User {
  // Static properties
  static maxUsers = 100;

  // Instance properties
  name;
  age;

  // Constructor
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  // Static methods
  static getMaxUsers() {
    return User.maxUsers;
  }

  // Public methods
  getName() {
    return this.name;
  }

  // Private methods
  _calculateScore() {
    // private method logic
  }
}

// Bad Practice
class User {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
```

```
  static maxUsers = 100;

  _calculateScore() {
    // Private method placed before public methods
  }

  getName() {
    return this.name;
  }

  static getMaxUsers() {
    return User.maxUsers;
  }
}
```

Comments:

1. **Single Line Comments:**

   Use for short, descriptive notes within the code. Place above the code if the comment applies to the next line.

   Example:

```
// Good Practice
// Initialize the app
initApp();

// Bad Practice
//Initialize the app (No space after //)
initApp();
```

   Eslint Rule: `spaced-comment`

2. **Multi-line Comments:**

   Use for detailed explanations or documentation of complex logic.

   Example:

```
// Good Practice
/*
 * This function processes user input and returns the result.
 * It handles various edge cases and ensures data integrity.
 */
function processInput(input) {
  // function body
}

// Bad Practice
/*
This comment doesn't follow the correct formatting.
```

```
Missing asterisk alignment.
*/
function processInput(input) {
  // function body
}
```

Eslint Rule: `spaced-comment`

3. **JSDoc Comments:**

Use JSDoc comments to document functions, methods, parameters, and return values.

Example:

```
// Good Practice
/**
 * Calculates the area of a rectangle.
 *
 * @param {number} width - The width of the rectangle.
 * @param {number} height - The height of the rectangle.
 * @returns {number} The area of the rectangle.
 */
function calculateArea(width, height) {
  return width * height;
}

// Bad Practice
/**
 * Calculates the area.
 * Missing parameter and return descriptions.
 */
function calculateArea(width, height) {
  return width * height;
}
```

Eslint Rule: `eslint-plugin-jsdoc`

Comparison of Different Coding Standards:

Coding Standard	Pros	Cons
JavaScript Standard Style (Standard)	• Simplicity: Provides a clear, opinionated set of rules, making it straightforward to follow. • Easy Adoption: Minimal setup required, allowing for quick implementation.	• Limited Flexibility: May not fit all project needs due to its rigid style guidelines. • Steeper Learning Curve: Can be challenging for those used to more customizable standards.
ESLint	• Customizable: Offers extensive customization with built-in and user-defined rules. • Flexible: Integrates well with a variety of tools and environments. • Supports Modern JavaScript: Compatible with ES6+ features like async/await and modules. • Rule Enforcement: Promotes consistent coding style, enhancing readability and maintainability. • Strong Community Support: Backed by a large, active community providing a wealth of plugins and resources.	• Learning Curve: Beginners might find it complex, especially when setting up custom rules.
Google JavaScript Style Guide	• Readability: Focuses on writing clear and maintainable code. • Consistency: Ensures a uniform coding style within Google's projects. • Proven Track Record: Established and effective within Google's extensive codebase.	• Less Widely Adopted: Not as commonly used outside of Google's ecosystem. • Learning Curve: Requires familiarity with Google's specific conventions.
jQuery	• Simple and Readable Guidelines: Emphasizes easy-to-read, clean code, making it suitable for any project using jQuery. • Large Library: Offers a vast array of functions and utilities, which reduces the need to write custom code for common tasks. Google's projects.	• Outdated for Modern JavaScript: Does not incorporate ES6 syntax or features, making it less relevant for modern JavaScript development. • Limited Scope: These standards are specifically tailored for jQuery projects and are not applicable to the

Coding Standard	Pros	Cons
	• Consistency Across jQuery Projects: Following these standards ensures uniformity across jQuery-based projects, which is important for smooth collaboration within teams. • Well documented: Well-documented and familiar to developers working with legacy systems or older applications, making it easier to adopt in such environments.	broader JavaScript ecosystem. • Not Suitable for Modern Frameworks: Does not align with modern frameworks like React, Vue, or Angular, limiting its usability in contemporary development environments.

Why Use ESLint:

1. Highly Customizable: ESLint allows for extensive customization of rules and plugins, making it adaptable to various project needs.
2. Supports Modern JavaScript: Fully compatible with ES6+ features, ensuring it remains relevant with current JavaScript advancements.
3. Automatic Error Fixing: ESLint can automatically correct many issues, such as formatting errors, using the --fix command.
4. Strong Ecosystem: Boasts a vast array of plugins for frameworks like React, Vue, and TypeScript, and integrates seamlessly with tools like Prettier for comprehensive code quality and formatting solutions.
5. Widely Adopted: Preferred by many industry leaders, ensuring robust support, frequent updates, and a broad user base.

Integrating Eslint to Project:

Follow these steps to integrate ESLint into your project and enforce the above coding standards.

Step 1: Install ESLint and Required Plugins

Use Yarn to install ESLint and any required plugins as development dependencies.

```
yarn add eslint eslint-plugin-sort-class-members eslint-plugin-jsdoc --dev
```

Step 2: Initialize ESLint
Initialize ESLint in your project:

```
yarn run eslint --init
```

During the setup, select the following options:

- How would you like to use ESLint? To check syntax and find problems.
- What type of modules does your project use? JavaScript modules (import/export).
- Which framework does your project use? None of these.
- Does your project use TypeScript? No.
- Where does your code run? Browser.
- eslint, globals, @eslint/js Would you like to install them now? Yes.
- Which package manager do you want to use? Yarn

This will create a eslint.config.mjs file in your project.

Step 3: Configure ESLint Rules in eslint.config.mjs
Replace or update the content of eslint.config.mjs with the following configuration:

```
// eslint.config.mjs

import sortClassMembers from 'eslint-plugin-sort-class-members';
import jsdoc from 'eslint-plugin-jsdoc';

export default [
  {
    ignores: ['node_modules/**'], // Optional: Add ignored directories or
files here.
  },
  {
    languageOptions: {
      ecmaVersion: 2021,
      sourceType: 'module',
    },
    plugins: {
      'sort-class-members': sortClassMembers,
      jsdoc: jsdoc,
    },
    rules: {
      // Naming Conventions
      'camelcase': ['error', { properties: 'always' }],
      'no-underscore-dangle': ['error', { allow: ['_password',
'_validatePassword'] }],
      'new-cap': ['error', { newIsCap: true }],

      // Layout Conventions
```

```
    'indent': ['error', 2],
    'max-len': ['error', { code: 100 }],
    'semi': ['error', 'always'],
    'brace-style': ['error', '1tbs', { allowSingleLine: false }],
    'curly': ['error', 'all'],
    'space-infix-ops': ['error'],
    'comma-spacing': ['error', { before: false, after: true }],
    'newline-after-var': 'error',
    'eol-last': ['error', 'always'],

    // Member Order
    'sort-class-members/sort-class-members': [
      'error',
      {
        order: [
          '[static-properties]',
          '[properties]',
          'constructor',
          '[static-methods]',
          '[methods]',
          '[private-methods]',
        ],
        groups: {
          'private-methods': [
            {
              name: '/^_/',
              type: 'method',
            },
          ],
        },
      },
    ],

    // Code Comments
    'spaced-comment': ['error', 'always', { exceptions: ['-', '+'] }],
    'multiline-comment-style': ['error', 'starred-block'],

    // JSDoc Rules
    'jsdoc/check-param-names': 'error',
    'jsdoc/check-tag-names': 'error',
    'jsdoc/check-types': 'error',
    'jsdoc/require-param': 'error',
    'jsdoc/require-param-description': 'error',
    'jsdoc/require-param-type': 'error',
    'jsdoc/require-returns': 'error',
    'jsdoc/require-returns-description': 'error',
    'jsdoc/require-returns-type': 'error',
  },
```

```
  },
];
```

Step 4: Add ESLint Script to package.json
In your package.json file, add the following script:

```
{
  "scripts": {
    "lint": "eslint ."
  }
}
```

Step 5: Run ESLint
To check your code for linting errors, run:

```
yarn lint
```

Step 6: Auto-fix Linting Errors
To automatically fix fixable errors, run:

```
yarn lint -fix
```

Step 7: Install ESLint Extension in Visual Studio Code (VS Code)

- Open VS Code.
- Go to the Extensions view (Ctrl+Shift+X on Windows/Linux or Cmd+Shift+X on macOS).
- Search for "ESLint" and install the official ESLint extension by Microsoft.

Step 8: Configure VS Code to use ESLint

- Once the extension is installed, VS Code will automatically lint your code in real-time. If there are ESLint rule violations, the problematic lines will be underlined with red or yellow squiggly lines.
- You can also add this configuration to your workspace settings to ensure ESLint runs correctly:

```
// .vscode/settings.json
{
  "eslint.validate": [
    "javascript",
    "javascriptreact",
    "typescript",
    "typescriptreact"
  ],
  "editor.codeActionsOnSave": {
```

```
      "source.fixAll.eslint": true
    }
  }

```

Step 9: Save and Lint

When you write code that violates ESLint rules, VS Code will display underlines and highlight errors in the Problems tab (Ctrl+Shift+M).

Example:

Some code following bad practice:

```
let UserName = 'JohnDoe'; // Starts with uppercase
function get_user_data() {
  // Uses underscores
}

const MaxLimit = 100; // Not all uppercase
const apiUrl = 'https://api.example.com'; // Not all uppercase

function fetchData() {
  if (true) {
      console.log('Incorrect indentation'); // Indented with 4 spaces
  }
}

function sayHello(name)
{
  console.log(`Hello, ${name}`);
}

/*
This comment doesn't follow the correct formatting.
Missing asterisk alignment.
*/
function processInput(input) {
  // function body
}
```

Here, VS Code is already showing errors for violating the ESLint rules.

Running yarn lint:

```
11:19:57 mahiyat@LAPTOP-CTU89NUP example → yarn lint
yarn run v1.22.22
$ eslint .

/home/mahiyat/example/index.js
   1:1   error  Expected blank line after variable declarations                                newline-after-var
   2:10  error  Identifier 'get_user_data' is not in camel case                               camelcase
  11:1   error  Expected indentation of 4 spaces but found 6                                  indent
  16:1   error  Opening curly brace does not appear on the same line as controlling statement brace-style
  21:1   error  Expected a '*' at the start of this line                                      multiline-comment-style
  22:1   error  Expected a '*' at the start of this line                                      multiline-comment-style
  23:1   error  Expected this line to be aligned with the start of the comment                multiline-comment-style
  26:2   error  Newline required at end of file but not found                                 eol-last

✖ 8 problems (8 errors, 0 warnings)
  7 errors and 0 warnings potentially fixable with the `--fix` option.

error Command failed with exit code 1.
info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command.
```

Running yarn lint --fix and correcting variable names manually:

```
1   let userName = 'JohnDoe'; // Starts with uppercase
2
3   function getUserData() {
4     // Uses underscores
5   }
6
7   const MAXLIMIT = 100; // Not all uppercase
8   const APIURL = 'https://api.example.com'; // Not all uppercase
9
10  function fetchData() {
11    if (true) {
12      console.log('Incorrect indentation'); // Indented with 4 spaces
13    }
14  }
15
16  function sayHello(name) {
17    console.log(`Hello, ${name}`);
18  }
19
20  /*
21   *This comment doesn't follow the correct formatting.
22   *Missing asterisk alignment.
23   */
24  function processInput(input) {
25    // function body
26  }
27
```

References:

1. https://standardjs.com/rules.html
2. https://developer.wordpress.org/coding-standards/wordpress-coding-standards/javascript/
3. https://github.com/rwaldron/idiomatic.js
4. https://github.com/airbnb/javascript?tab=readme-ov-file
5. https://chatgpt.com/share/66eae320-a6a0-8001-ad4e-f7cc634e9c18
6. https://www.jscripters.com/jquery-disadvantages-and-advantages/
7. https://www.franciscomoretti.com/blog/the-pros-and-cons-of-using-eslint#pros-of-using-eslint-