Author

`Md Niaz Rahaman`

Coding Standards and Best Practices

1. General Coding Guidelines

- **Indentation:** Use 2 spaces per indentation level. Consistency is crucial.
- **Semicolons:** Always use semicolons (;) to terminate statements.
- **File Naming:** Use camelCase for filenames (e.g., `userController.js`).
- **Line Length:** Keep line length to 80–100 characters. Use line breaks for readability.
- **Comments:**
  - Use single-line comments (`//`) for short explanations.
  - Use multi-line comments (`/* */`) for more detailed explanations or documentation.
  - Place comments above code blocks, not beside them.
- **Variables and Constants:**
  - Use `const` for variables that don't change and `let` for variables that can be reassigned.
  - Always use meaningful variable names.

  ```
  let totalCost = 0;    // Descriptive
  const MAX_USERS = 100;  // For constants
  ```

2. Modularization

- Split code into smaller, reusable modules.
- Each file should handle a specific task.
- Separate concerns into different files:
  - `routes`
  - `models`

  - `controllers`
  - `utilities` .

3. Error Handling

- Always handle asynchronous errors in promises or `async/await` using `try...catch` .
- Provide meaningful error messages.
- Log errors for debugging and troubleshooting.

```
try {
    const data = await someAsyncFunction();
} catch (error) {
    console.error('Error:', error.message);
    res.status(500).send('Internal server error');
}
```

4. Consistency

- Use consistent naming conventions throughout your codebase:
  - **Functions**: `camelCase` (`getUserData`)
  - **Classes**: `PascalCase` (`UserController`)
  - **Constants**: `UPPERCASE` (`MAX_USERS`)
- Organize your imports logically:
  - Import third-party modules first, followed by local imports.

5. Functions and Methods

- Keep functions short and focused.
- A function should do **one thing** and do it well.

6. Asynchronous Code

- Always use `async/await` over callbacks or `.then()` for asynchronous operations.
- This makes the code more readable and easier to maintain.

```
async function fetchData() {
    try {
        const result = await getDataFromAPI();
        return result;
    } catch (error) {
        console.error(error);
    }
}
```

7. Linting and Formatting

- Use a linter like **ESLint** to automatically enforce coding standards.

ESLint Setup Guide

- Install ESLint

If we want to add ESLint to a running project we follow those given command

```
npm install eslint --save-dev
```

Or, if using Yarn:

```
yarn add eslint --dev
```

- Initialize ESLint Configuration

We can create a configuration file (`.eslintrc.json`) using the following command:

```
npx eslint --init
```

This command will ask you a series of questions to set up your configuration. For example, it will ask if you want to check syntax, find problems, and enforce code style. You can also choose a popular style guide like Airbnb, Standard, or Google.

- Create/Modify ESLint Configuration File

We can manually create or edit the `.eslintrc.json` file to include specific rules. Here's a basic example:

```
{
  "env": {
    "browser": true,
    "es2021": true,
    "node": true
  },
  "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": 12,
    "sourceType": "module"
  },
  "rules": {
    "indent": ["error", 2],
    "quotes": ["error", "double"],
    "semi": ["error", "always"]
  }
}
```

- Run ESLint

To lint your files, use the following command:

```
npx eslint yourfile.js
```

You can also lint all JavaScript files in your project:

```
npx eslint .
```

- Integrate with IDE

Many IDEs and code editors like VSCode have ESLint plugins/extensions. We can install these to get real-time linting feedback as you write code.

- Fix Errors Automatically

ESLint can fix some errors automatically. To apply these fixes, use the `--fix` flag:

```
npx eslint . --fix
```

This will automatically correct fixable issues based on your configuration.

- Use ESLint with Prettier (Optional)

If you want to use ESLint in conjunction with Prettier, a popular code formatting tool, you can add the necessary plugins and configurations. This will ensure ESLint handles code quality and Prettier handles code style.

1. Install the Prettier and related plugins:

   ```
   npm install --save-dev prettier eslint-config-prettier eslint-plugin-prettier
   ```

2. Update your `.eslintrc.json`:

   ```
   {
     "extends": ["eslint:recommended", "plugin:prettier/recommended"],
     "rules": {
       "prettier/prettier": "error"
     }
   }
   ```

This setup ensures that Prettier rules are applied, and any code style issues are caught as ESLint errors.

- Set up **Prettier** for code formatting to maintain consistent styles across the codebase.

Comparison of Popular JavaScript Style Guides

Style Guide	Description	Installation Command
Airbnb	A comprehensive style guide that enforces a strict set of rules for writing clean and consistent JavaScript code. Recommended for projects prioritizing code consistency and readability.	`npx install-peerdeps --dev eslint-config-airbnb`
StandardJS	A minimalistic style guide with fewer configuration options and a no-nonsense approach. It aims to eliminate debates over stylistic decisions. Best for projects looking for simplicity and quick setup.	`npm install eslint-config-standard eslint-plugin-standard eslint-plugin-promise eslint-plugin-import eslint-plugin-node --save-dev`
Google	Emphasizes readability and consistency, commonly used in the Google ecosystem. Recommended for projects that follow Google’s coding standards or aim for strict guidelines.	`npm install eslint-config-google --save-dev`

Choosing the Right Style Guide

- **Airbnb:** Best for teams and projects that prioritize a well-defined and comprehensive set of rules.
- **StandardJS:** Best for those who prefer simplicity and a zero-config approach.
- **Google:** Best for projects aligned with Google's development practices or looking for a strong emphasis on code clarity.

You can customize these style guides further according to your project needs by overriding specific rules in your `.eslintrc.json` file.

8. Security Best Practices

- Sanitize inputs to avoid **SQL Injection** and **XSS** attacks.
- Use environment variables to store sensitive data (like API keys, DB credentials). Use a package like `dotenv` to load them.

```
require('dotenv').config();
const dbPassword = process.env.DB_PASSWORD;
```

- Use HTTPS wherever possible to protect data transmission.
- Handle authentication and authorization properly.

9. Testing

- Write **unit tests** and **integration tests**.
- Ensure code coverage for critical parts of the app.

10. Use Strict Mode

- Use `"use strict";` at the top of JavaScript files or functions to enforce secure coding practices.

11. Dependencies and Packages

- Use `package.json` wisely to manage dependencies.
- Keep track of package versions to avoid compatibility issues.
- Regularly run `npm audit` to check for vulnerabilities in dependencies.
- Remove unused packages to reduce app bloat and potential security risks.

12. Folder Structure

Organize code into logical folders for easy navigation and maintenance. Here's an example structure:

12. Folder Structure

Organize code into logical folders for easy navigation and maintenance. Here's an example structure:

Folder/File	Description
`nodejs-app`	Main application directory
`Configuration files`	db, environment settings
`controllers`	Controllers for handling requests and responses
`models`	Data models (MongoDB, Sequelize)
`routes`	API routes
`services`	Business logic, external API integrations
`middlewares`	Middleware functions (authentication, logging)
`utils`	Helper functions, utilities
`tests`	Unit and integration tests
`app.js`	Main app file
`package.json`	Dependencies and scripts

References

- Perfomatix - Node.js Coding Standards and Best Practices
- Stack Overflow - Coding Style Guide for Node.js Apps

- https://enlear.academy/5-best-javascript-style-guides-640485e7b630