Coding Standards for Our E-commerce Site Project

Project Name: Kontaniben? – an E-commerce site.

Programming Language: Javascript.
Framework Used for Backend: Express.js

For coding standards, there are some types but most of the developers used one coding standard. This coding standard is used all over the world. Best practice for writing node.js/express.js coding standards are --

Source : Node.js Coding Standards and Best Practices [Developer Guide] (perfomatix.com)

Project Structure Practices

Divide the project backend into several parts like

1. Routing folder for API's of the project
2. Source folder for source code
3. Model folder for database schema
4. Authentication folder for user authentication
5. Use environment-aware, secure, and hierarchical config. Like -
   i. PORT number,
   ii. Mongodb URL,
   iii. Secrect key

Error Handling Practices

Use Async-Await or promises for async error handling

Code Style Practices

1. **Use ESLint :**

ESLint is the de-facto standard for checking possible code errors and fixing code style, not only to identify nitty-gritty spacing issues but also to detect serious code anti-patterns like developers throwing errors without classification.

2. **Separate your statements properly :**

Use ESLint to gain awareness about separation concerns. Prettier or Standardjs can automatically resolve these issues.

3. **Name your functions :**

Name all functions, including closures and callbacks. Avoid anonymous functions.

4. **Use naming conventions for variables, constants, and functions :**

Use lowerCamelCase when naming constants, variables, and functions and upperCamelCase (capital first letter as well) when naming classes. This will help you to easily distinguish between plain variables/functions, and classes that require instantiation. Use descriptive names, but try to keep them short.

5. **Prefer const over let. Ditch the var:**

Using const means that once a variable is assigned, it cannot be reassigned. Preferring const will help you to not be tempted to use the same variable for different uses, and make your code clearer. If a variable needs to be reassigned, in a for loop, for example, use let to declare it. Another important aspect of let is that a variable declared using it is only available in the block scope in which it was defined.‘var’ is function scoped, not block scope, and shouldn’t be used in ES6 now that you have ‘const’ and let at your disposal.

6. **Require modules first, not inside functions:**

Require modules at the beginning of each file, before and outside of any functions.

7. **Use Async Await, avoid callbacks :**

Node 8 LTS now has full support for Async-await. This is a new way of dealing with asynchronous code which supersedes callbacks and promises. Async-await is non-blocking, and it makes asynchronous code look synchronous.

8. **Use arrow function expressions (=>)**

Though it's recommended to use async-await and avoid function parameters when dealing with older APIs that accept promises or callbacks – arrow functions make the code structure more compact and keep the lexical context of the root function (i.e. this).

Coding Standards for React

Coding standards help maintain consistency, readability, and best practices across a codebase.

Sources:

React coding Standards and Practices | by Navita Singhal | Medium

React code conventions and best practices | by Gaspar Nagy | Level Up Coding (gitconnected.com)

React Coding Standards and Practices To Level Up Your Code - Jon D Jones

Here are some coding standards for React that we will follow in our project:

1. **Naming Conventions:**

    i. Component's names should be written using pascal case. For Example: Header.js, LoginScreen.js.

ii. Non-components should be written using camel case. For Example: myUtilityFile.js, fetchApi.js.

iii. We will use camel case for JavaScript data types like variables, arrays, objects, functions, etc. For Example:
const variableName = ‘test’;
const userTypes = [...]
const getLastDigit = () => { ... }

iv. Unit test files should use the same name as its corresponding file. For Example: Header.test.js, LoginScreen.test.js, myUtilityFile.test.js.

v. CSS files should be named the same as the component PascalCase. Global CSS which applies to all components should be placed in global.css and should be named in camelCase. For Example: LoginScreen.css(for components), global.css(for global styles)

2. **Folderize the components:** We will keep all related files in one folder. Here is the folder structure with naming conventions to be used:

```
components/
  LoginScreen/
    LoginScreen.js
    LoginScreen.css
    LoginScreen.test.js
```

3. **Avoid default export:** Default exports don’t associate any name with the item being exported, meaning that any name can be applied during import. This may give us a flexibility, but if multiple developers imports the same module with different names or even non descriptive name, we are screwed. Named exports are explicit, forcing the consumer to import with the names the original author intended and removing any ambiguity.

   export default MyComponent; ✗

   export { MyComponent }; ✓

export const MyComponent = ...; ✓

export type MyComponentType = ...; ✓

4. **Brace Placement:** Place opening braces on the same line as the function or statement. Use a new line for the closing brace.

```
// Good

function myFunction() {

  // code

}

// Bad

function myFunction()

{

  // code

}
```

5. **Props:** We will destructure props in the function parameters for better readability.

```
// Good
function MyComponent({ prop1, prop2 }) {
  // code
}

// Bad
function MyComponent(props) {
  // code using props.prop1 and props.prop2
}
```

6. **Conditional Rendering:** We will use ternary operators or logical operators for concise conditional rendering.

```
// Good

{isLoggedIn ? <LogoutButton /> : <LoginButton />}

// Bad

{isLoggedIn && <LogoutButton />}

{isLoggedIn || <LoginButton />}
```

7. **We will avoid Inline CSS.**

8. Separate all your service calls into a separate file. If it's a big project try to split the services into multiple files. (name convention :module_name.service.js).

9. **We will use object destructuring:**

```
// Bad
return (
  <>
    <div> {user.name} </div>
    <div> {user.age} </div>
    <div> {user.profession} </div>
  </>
)
// Good
const { name, age, profession } = user;
return (
  <>
    <div> {name} </div>
    <div> {age} </div>
    <div> {profession} </div>
</>
)
```