Project Coding Standards

Table of Contents

Coding standards and best practices

1.1 Naming Conventions and Standards

> Note :
>
> The terms Pascal Casing and Camel Casing are used throughout this document.
>
> **Pascal Casing** - First character of all words are Upper Case and other characters are lower case.
>
> Example: BackColor
>
> **Camel Casing -** First character of all words, except the first word are Upper Case and other characters are lower case.
>
> Example: backColor

1. Use Pascal casing for Class names.

For Example: -

```
public class User
{
        ...
}
```

2. Use Camel casing for Method names.

For Example: -

```
public Products getProduct(int id)
{
        ...
}
```

3. Use Camel casing for variables and method parameters.

   For Example: -

```
int cartId;
public Products getProduct(int id)
{
        Loger.info("fetching product with ID" +id);
        ...
}
```

4. Use Meaningful, descriptive words to name variables. Do not use abbreviations.

For Example:-

Good:

```
string address
double price
```

5. No usage of single character variable names like i, n, s etc. Use names like index, temp

One exception in this case would be variables used for iterations in loops:

```
for ( int i = 0; i < count; i++ )
{
        ...
}
```

6. No usage of underscores (_) for local variable names.

7. Do not use variable names that resemble keywords.

8. File name should match with class name.

For example, for the class HelloWorld, the file name should be helloworld.cs (or, helloworld.vb)

9. Used Pascal Case for file names.

10. Import only necessary classes. Do not use wildcard imports (e.g., java.util.*) unless there are 4 or more classes from that package.

11. Do not use deprecated methods.

12. Code should only print to console when appropriate. Inside a library is not an appropriate place to print to the console. Use logger if need to output in such cases. If you are printing to the console, print to the correct stream.

13. Ensure the flow of the code is easily understandable.

1.2 Indentation and Spacing

1. Use TAB for indentation. No usage of SPACES.

2. Comments should be in the same level as the code (use the same level of indentation).

   For Example:-

```
// Details of the product by id.
@Override
public Products getProduct(int id) {
    logger.info("Fetching product with id " + id);
    Products product = productRepository.findById(id);
    logger.debug("Product with id = " + id + " is " + product);
    return product;
}
```

3. Curly braces ({}) should be in the same level as the code outside the braces.
   For Example:-

```
if ( ... )
{
        // Do something
        // ...

        return false;
}
```

4. Nesting the if-else blocks of code to have clear visibility of function.
   For Example:-

```
/*
* Once the delivery person login, It will navigate to the deliverhome.
*/
@GetMapping("/delivery/deliverhome/{id}")
    public String deliveryhome(@PathVariable("id") int id, ModelMap model) {
        model.addAttribute("user", service.getuser(id));
        List<Orders> orders = orderService.findAll();
        ArrayList<Integer> customerIds = new ArrayList<>();
        if (orders.size() > 0) {
            for (Orders order : orders) {
                if (customerIds.contains(order.getUser_id())) {
                }
                else {
                    customerIds.add(order.getUser_id());
                }
            }
        }
```

5. Kept private member variables, properties and methods in the top of the file and public members in the bottom.

1.3 Comments

Good and meaningful comments make code more maintainable. However,

1. Use **//** or **/* … */**for comments.

2. Writing comments wherever required. Good readable code required very less comments. All variables and method names were meaningful, that would make the code very readable and will not need many comments.

3. Obvious/obfuscated comments are useless. Do not use them.

4. Properly (but reasonably) comment your code. A developer should be able to get a general idea of what's going on by just reading comments (and no code).

5. Check your comments for spelling and grammatical errors.

1.4 Exception Handling

1. In case of exceptions, used a friendly message to display it to the user, but not the actual error with all possible details about the error, including the time it occurred, method and class name etc.

2. Always catch only the specific exception, but generic exception was in-use to know for the exposure to errors during the development-cycle.

```
@PostMapping("/delivery/getOrders/{userid}/{orderid}/{customerid}")
    public String DeleteProduct(Model model, @PathVariable("userid") int userid, @PathVariable("orderid") int id,
        @PathVariable("customerid") int customerid) throws AuthenticationException{
```

3. No need to catch the general exception in all the methods as stated with the above reason. This will help us find most of the errors during development cycle. We can have an application level (thread level) error handler where we can handle all general exceptions. In case of an 'unexpected general error', this error handler should catch the exception and should log the error in addition to giving a friendly message to the user before closing the application or allowing the user to 'ignore and proceed'.

1.5 Best Practices for front-end Coding

Some common basic rules for front-end coding.

- Create multiple files instead of writing a big file. (Componentization of code: fix to small functionality for each file)
- Place all your CSS files in one common folder.
- Avoid Inline CSS as and when possible (a CSS class should be created when there are more than 2 CSS attributes).
- Review your code before creating a pull request.
- Split your code into multiple smaller functions. Each with a single responsibility.
- Create many utility files that can help you remove duplicate code from multiple files.
- 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).
- Name your files logically according to the job that they perform.
- Clean code is self-commenting(using the right variable names and function names). Use comments only to explain complex functions.
- Always write test cases for your code. Keep tests files in sync with the files they are testing.
- Use useReducer when useState becomes complex.
- Putting imports in an order
  a. React import
  b. Library imports (Alphabetical order)
  c. Absolute imports from the project (Alphabetical order)
  d. Relative imports (Alphabetical order)
  e. Import * as
  f. Import './<some file>.<some extension>

1.React UI component's names should be PascalCase.
Example: LoginScreen.js

2. All other helper files should be camelCase. (non-component files)
Example: commonUtils.js

3. All the folder names should be camelCase.
Example: components

4. 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
Example: LoginScreen.css(for components), global.css(for global styles)

5. CSS class names should use a standard naming convention (personally use kebab-case because it's used by most of the CSS framework classes)
or any standard practice.
Document with several conventions: CSS naming conventions.

6. Test files should be named the same as the component or non-component file.
Example: LoginScreen.test.js
commonUtils.test.js