**HUrricane**

**Coding Standards**

1. **Introduction**

This document defines suitable coding standards and evaluates the project based on them. Coding standards ensure that the codebase is readable, maintainable, and scalable. Following these standards will support an efficient and improved working environment in terms of collaboration, testing/debugging, and better connections between MVC layers.

1. **Description**

When defining and applying coding standards, the principles should prioritize efficiency, objectivity, and ease of use. These standards encompass naming conventions, commenting practices, proper file organization and layering, and overall readability. Additionally, coding standards ensure that the project can be easily integrated and maintained across different countries, companies, and environments, thanks to standardization. The primary goal of our project’s code standardization is to enable seamless collaboration and development with minimal effort while maximizing efficiency.

1. **Coding Standards Specifications**

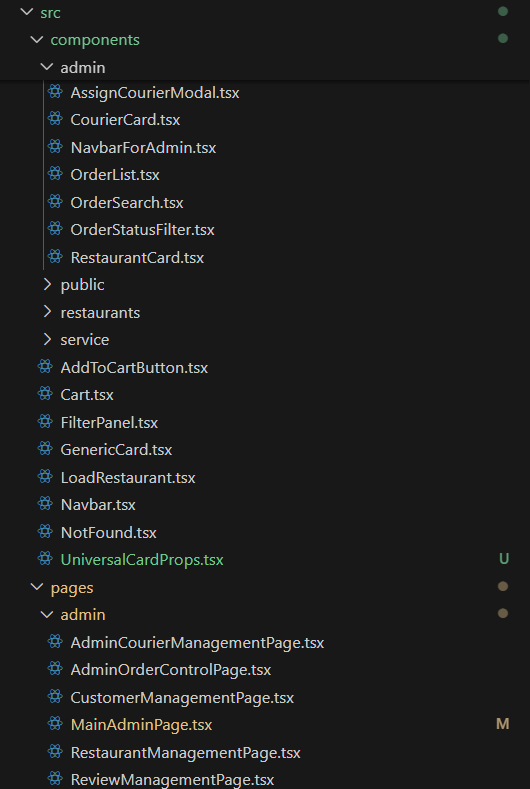
We utilized a combination of CamelCase and PascalCase conventions to ensure consistency and readability across our codebase. For instance, in the CustomerController class, method names like createOrder and updateOrder follow the CamelCase convention, which is typical for methods in Java. Similarly, classes such as OrderDTO and OrderService use PascalCase, as is customary for class names. This consistent application of naming conventions ensures that we maintain a clear and standardized approach, which leads to a high degree of compatibility with common Java and React practices. In this case, the compatibility percentage can be considered high, as we adhere to widely accepted standards for naming conventions in both backend (Java) and frontend (React) components, leading to better maintainability and collaboration across the project.

* + In the backend system, our files are organised in a stable template, as seen in the picture 1.

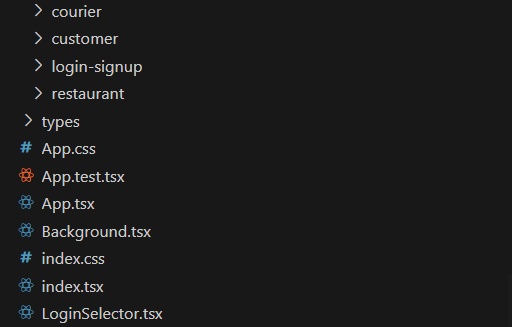


picture-1

For each component of the system—admin, user, and order—we maintain a dedicated folder. Inside each component folder, the structure is consistently divided into subfolders such as controller, service, dto, entity, mapper, and repository. This modular organization ensures that all files related to a specific component are grouped together, improving maintainability and scalability. File names follow a consistent lowercase-with-hyphens (kebab-case) convention (e.g., order-controller.java, user-service.java), which aligns with standard naming practices for files in many Java and front-end projects. Class names inside the files follow the PascalCase convention (e.g., OrderService, AdminController), while method and variable names use camelCase (e.g., getUserById, createOrder). This well-defined and uniform file and naming structure enhances readability, eases collaboration across teams, and increases the overall compatibility and professionalism of the codebase.



picture-2



picture-3

In the frontend system, we followed similar naming conventions to maintain consistency with the backend. As seen in Picture-2 and Picture-3, we used specialized and descriptive file names to organize our file hierarchy clearly and intuitively. The main folder structure under the src directory includes folders such as pages, components, and types, alongside global files like App.tsx and App.css. The pages folder contains route-based components, components holds reusable UI elements, and types stores TypeScript interfaces and type definitions for better type safety and maintainability. Since we primarily used Tailwind CSS for styling, only the App.css file is actively used to define global styles and Tailwind configuration imports. File and folder names follow the kebab-case convention (e.g., user-card.tsx, login-form.tsx), while React components use the PascalCase convention (e.g., LoginForm, UserCard). This structured approach ensures a clean, scalable, and collaborative frontend architecture.

We followed a balanced and intentional approach to commenting throughout the project. While we aimed to write clean, self-explanatory code that minimizes the need for excessive comments, we still included comments where they added meaningful clarity. Most comments serve as concise explanations of function and class purposes, helping readers quickly understand the role and logic behind key components. In some cases, we also used comments as internal reminders for future improvements or to highlight edge cases. This approach supports both readability and maintainability without cluttering the codebase, and aligns with best practices that prioritize code clarity while still enabling documentation generation when needed.

Throughout the project, we adhered to consistent coding conventions to ensure code quality, readability, and maintainability. In both the backend and frontend, we followed clear structural patterns: each system component (admin, user, order) was organized into folders with standardized subdirectories such as controller, service, dto, entity, mapper, and repository. All files followed a lowercase naming convention with descriptive names relevant to their function. In the frontend, the file hierarchy under src included clearly separated folders like pages, components, and types, alongside shared files like App.tsx and App.css. We also used meaningful names for files and components, favoring clarity over brevity. Poor practices such as inconsistent naming, tightly coupled logic, or bloated files were consciously avoided. This consistent application of coding conventions helped maintain a clean and scalable codebase.

We maintained consistent use of white space throughout the project to enhance code readability. Proper indentation, spacing around operators, and blank lines between logical blocks made the code easier to navigate and understand.

In our project, we established clear coding standards tailored to Java (Spring Boot) for the backend and React with TypeScript for the frontend. We used Maven for builds and worked in VSCode with relevant Java tools. In the frontend, we followed consistent JSX and TypeScript patterns, using Tailwind CSS extensively and plain CSS minimally. While some choices—like indentation—were arbitrary, all conventions aimed to improve clarity and team communication. These standards evolved with the team and were applied consistently to maintain a clean and maintainable codebase.

1. **References**

ChatGPT: <https://chatgpt.com/share/6817bb07-7ad0-800f-adc9-75e3d8d23275>

ChatGPT: <https://chatgpt.com/share/6817bbc9-9a4c-800f-9ad1-4c6c8e392af4>