Code Quality Review Report

No More Waste

Bulbul Arora
Faizah Kolapo
Arika Pasha

Table of Contents

Introduction

No More Waste's source code was built with a separation of concerns in mind, there is client side and a server side. The client side was created with React and the server side was built with a Node JS REST API. Node JS is also used to access the MySQL database, which is hosted on DigitalOcean. The server side was then deployed with Google App Engine. Based on the research conducted pre-development, we found that restaurants and shelters would most benefit from a responsive web application to make desktop and mobile use easily accessible, thus we used CSS to implement this responsiveness. This report will review the quality of the code used in the creation of No More Waste.

Code Formatting

The formatting of No More Waste's codes ensures that it is readable with minimal blockers based on the usage of proper left margin alignment (Figure 1), white space (Figure 1), and identifiable start and end points of code blocks (Figure 1). Proper naming conventions were also followed to ensure that the code is well formatted. Pascal Case was used for the React components, Camel Case was used for functions, variables and methods, and Snake Case was used for database columns, this ensured that there was a clear distinction in the code and allowed us to easily identify what each code segment is used for (Figure 2). The code also fits the standard 14 inch laptop screen with no horizontal scrolling required to view it, the one exception being post.js.

Figure 1: Code sample

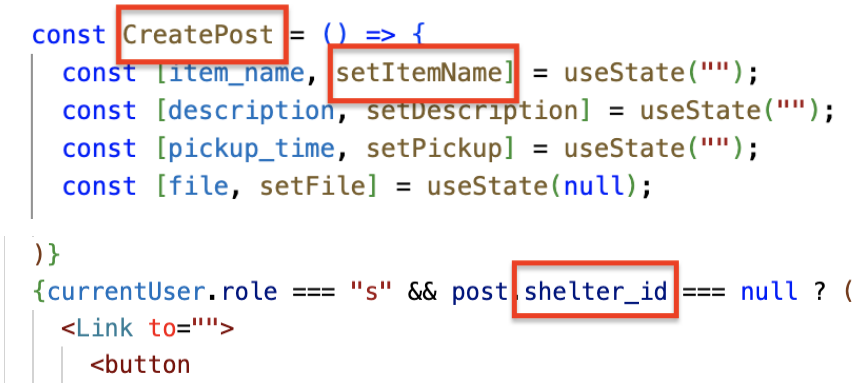

Figure 2: Code sample with Pascal, Camel and Snake Case

Code Architecture

To ensure that there was a clear separation of concerns, No More Waste used the Model View Controller pattern. The model consists of the MySQL database that was deployed using DigitalOcean, the view consists of the User Interface (UI) that was implemented using React Native and the controller, which receives the input from the user in the UI and manipulates the model, was implemented with Node JS. The view's responsiveness was implemented using CSS.

This separation of concerns allowed us to structure our code into separate folders that handled these respective concerns. This made the writing and editing of the code smoother and easier to identify. The client side folder includes the CSS and React pages, whereas the server side handles all the API calls, database manipulation and routing (Figure 3).

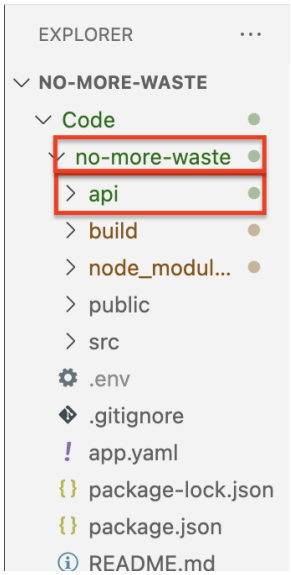

Figure 3: Folder architecture, server side (api) is inside our client (no-more-waste)

Coding Best Practices

We ensured that we used the best possible coding practices. There was no hard-coding in our code. Instead of repeatedly coding our navigation and footer on every page, with the help of react-router-dom library, we made them components that we could call on. We wrote descriptive comments throughout the coding process that indicate why we wrote code a specific way, when we found better solutions to our previous code, we commented out the previous code and coded the better alternative. This ensures that each of us knew what code was being replaced and why, we kept the previous code to ensure we could revert back to it if we ran into a problem with the new code. We also added comments through the code of pending tasks to let each other know

what has yet to be completed. The use of writing components, styling components, routing, properties and states are all framework features we used in our code. For the final submission, we have removed unnecessary commented console.log statements and any commented testing statements to make the code look cleaner and easier to understand.

Non Functional Requirements

To ensure maintainability of the code we wanted to make the application as easy to support as possible. The code is quite readable and descriptive making it self-explanatory for viewers to understand, this was done through the use of naming conventions for function, variables and components. The code is also very testable, we ensured that our functions are used for one responsibility that we can easily confirm that it works. We also used the interface for simple testing of account creation, account signup, post creation, post request, post acceptance and driver status update. For debuggability we made use of console logs to detect errors and this immensely helped us throughout the coding process when we ran into trouble with our code. We have also maintained the configurability by ensuring that our database table files are kept in place, and our env file was sent out to each member and updated for each additional feature (Google Cloud Storage, Twilio, Database). We used the Do not Repeat Yourself (DRY) principle when making use of our components and API routes. Our code is also easily extendable, allowing us to add features without having to change the existing code (when adding the Google Map links, when adding the Twilio notifications). The security of our code was maintained with our env files, ensuring no outside users have access to our sensitive information. We tested usability with our test users, peers and the individuals who dropped by our mock tradeshow. We listened to their feedback to make our application more usable.