# Code Testing 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 highlight the code testing plan and results that were implemented and received through the duration of the development cycle.

## Test Plan

Based on the separation of concerns we emphasized throughout the development process, we found the best way to test our code would be through the separation of testing for each concern. We did this by using different test plans for each type of concern.

**Server Side Testing Plan**

Server side testing will be done through the use of console logs, if there are any errors with the manipulation of our model, the console log will display that error to us and allow us to fix it.

**Client Side Testing Plan**

Client side testing will test all the features and functionality on our application. It will ensure that the features are working as expected and our client and server side are communicating correctly. This will be done using feature testing. It will also test that our style changes are correctly being shown.

### Database Testing Plan

Database testing will rely on the server side correctly manipulating the data, we will check our database to test if the correct inputs are being placed into our datatables. We also used MySQL Workbench to test our queries before implementing them to save us time and effort of testing them directly in our code.

### Responsiveness Testing Plan

Responsiveness testing will rely on the client side correctly displaying the style and changes of the code. We will test that the responsiveness adjusts the style of the application, the CSS, based on the device that is accessing the application.

## Test Results

This part of the report will go over how the testing for each concern was conducted and the results that were received.

### Server Side Testing Results

As mentioned, console logs were used to test that our server side was working as expected. If we notice that a page is not looking or working as expected, we can right click and click “Inspect” to view our browser’s tools (Figure 1). From the navigation bar at the top, we can click “Console” to see the logs and figure out where and what the error is (Figure 2). Throughout the development process, the use of console logs was extremely beneficial in helping us identify and solve errors. The logs help us realize where the error is occurring and allows us to find a fix faster and more efficiently. We also used status codes like 200 for OK, 401 for Not Authenticated, 403 for invalid token and 500 for server error. These codes really helped us in figuring out where the problem existed and made it much easier for us to test the API.

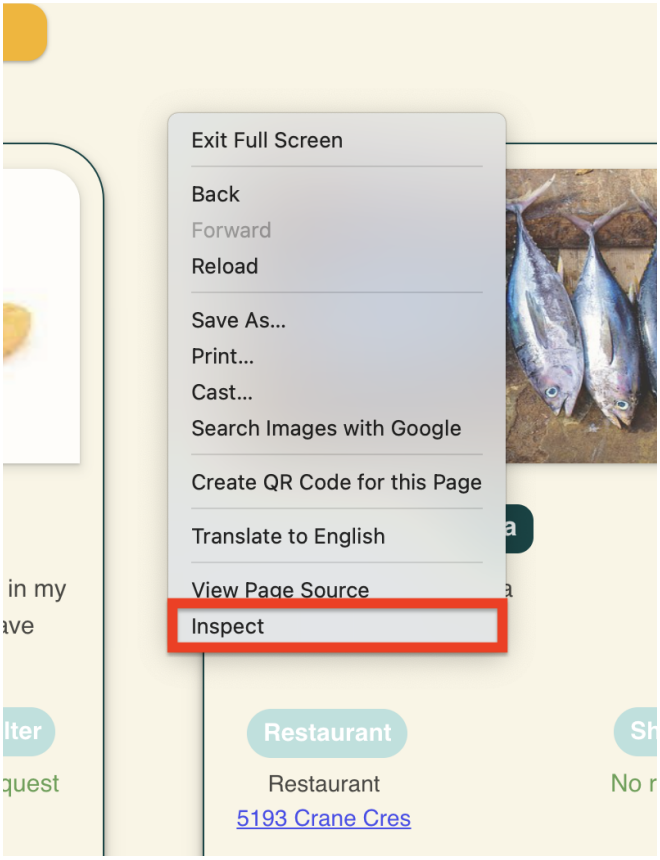


*Figure 1: Inspect browser tools*

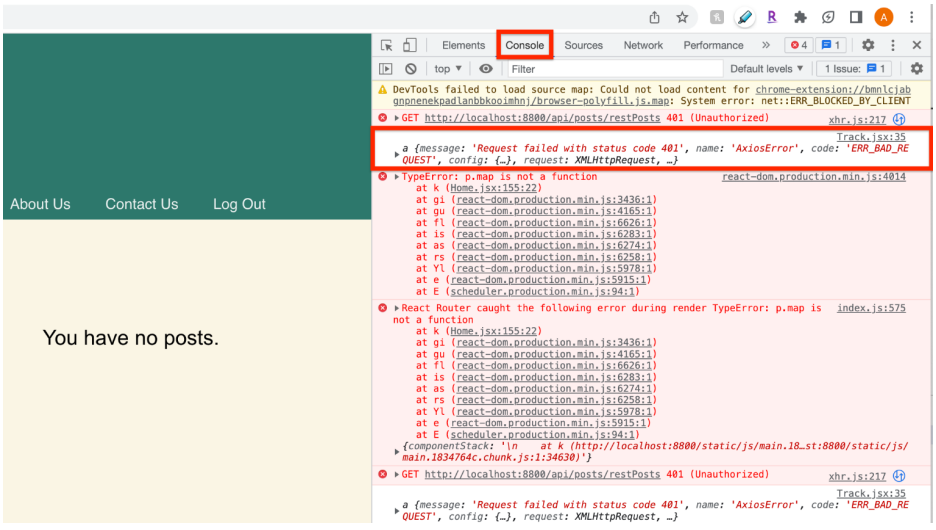


*Figure 2: View the console and error messages*

**Client Side Testing Results**

For the client side, we tested each feature on our application to ensure they were working as expected.

| Test Case | Objective | Precondition | Approach | Expected Result | Result |
|---|---|---|---|---|---|
| Signup | Ensure that the signup process correctly creates accounts with the correct roles. | User information must be entered into each field on the signup form. | Fill in the form and submit it to ensure that the user's data is correctly stored in the database. | Allow users who fill in all the fields to create an account. | Working as expected. |
| Login | Ensure that users with accounts and correct credentials can login. | Account must already exist to login successfully and the password entered must be correct. | Login with an existing account to ensure login works. Next, try to login with nonexistent accounts or incorrect passwords to ensure you cannot login. | Allow matching email and password combinations to login. | Working as expected. |
| Post creation | Ensure that posts are correctly being created, ensuring they show up on the dashboard as expected. | Must be logged into an account that is assigned the role of "Restaurant". | Create a post and check the dashboard or datatable to see if it was created successfully. | Allow restaurants to create posts of their excess food. | Working as expected. |
| Post pickup time input | Ensure that the post pickup time selected is followed | Must be logged into an account that is assigned the role of "Restaurant". | Select a pickup time during post creation and check the database to | Expected the pickup time to be displayed and used in the expiration of posts. | Did not have enough time to implement this. |

| | | | | | |
|---|---|---|---|---|---|
| | and that the post is no longer available past that time. | | ensure it is recorded and that the post expires as expected. | | |
| Post image input | Ensure that images are correctly displayed in posts on the dashboard. | Must be logged into an account that is assigned the role of “Restaurant”. | Select an image during post creation and check that it shows on the dashboard once the post is created. | The image selected is correctly shown on the dashboard to all users. | Ran into some trouble when working on localhost but after deployment, this works as expected. |
| Shelter post request | Ensure that once a shelter hits the “Request” button on a post, they are now the designated shelter for that post | Must be logged into an account that is assigned the role of “Shelter”, no shelter should be associated with this post yet. | Login as a shelter and hit the request button on any post. | The shelter's name and address should be displayed on the post now. | Working as expected. |
| Twilio phone notifications for status updates | Ensure that notifications are sent in accordance with our notification flow diagram. | Ensure that the correct buttons by the user’s is selected | Have a shelter request a post and check that the notification is sent to the number of the restaurant. | The correct notification is sent to the restaurant. | Working as expected. |
| Each user type being able to view their history | Ensure that volunteers can see all of their deliveries, that shelters can see their | Must have the correct role depending on the type of history being viewed. | Login as a shelter and confirm that the “My Requests” history is as expected. Login as a volunteer | See the correct history based on the user. | Working as expected. |

| | | | | | |
|---|---|---|---|---|---|
| | requests, and restaurants can see their posts. | | and confirm that the “My Deliveries” history is as expected. Login as a restaurant and confirm that the “My Posts’ history is as expected. | | |
| Volunteer accept request and status updates | Ensure that when a volunteer accepts a request and updates their status throughout, that that is correctly being reflected on the post and through the text notifications . | Must be logged into an account that is assigned the role of “Volunteer”, no volunteer should be associated with this request yet. Must be the volunteer associated with this request to update the status. | Login as a volunteer and accept a post, and then update the status of the post. | The volunteer’s name and number should be displayed on that post. When a volunteer updates their status, notifications should be sent to both the restaurant and shelter. | Working as expected. |

*Table 1: Feature testing*

**Database Testing Results**

Before we could have our client side functionality working, we needed to test post creation and user creation by checking that the values were put into our datatables successfully (Figure 3). After our dashboard and login started working, we no longer needed to test with the database, but if we ran into certain errors with the client side, we would refer back to the database to ensure that the values there were correct (check if the correct user role is assigned or check if a post has been requested correctly). We also used MySQL workbench to test our

queries beforehand to save us the hassle of testing in the development environment. If he had troubles in the development environment, we ran the queries in MySQL workbench to determine where the issue existed.

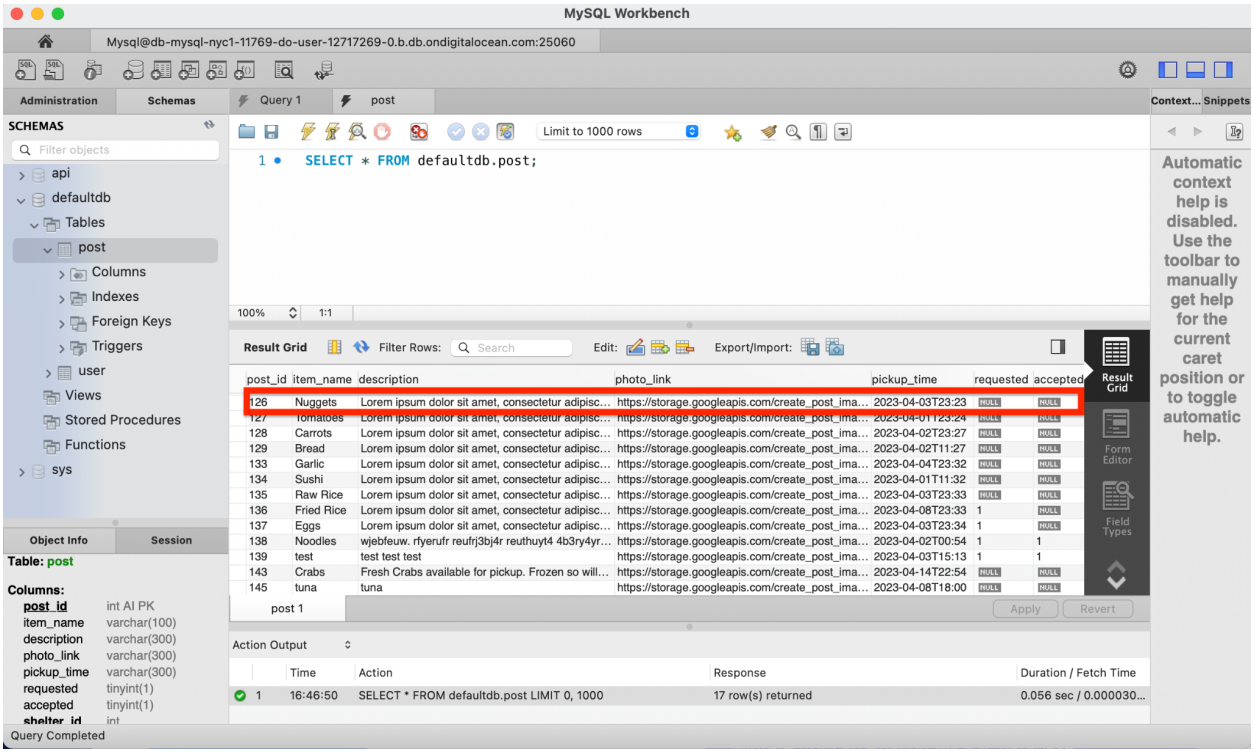


*Figure 3: Post creation in database*

**Responsiveness Testing Results**

To test responsiveness, we made use of our browser tools to select the dimensions of the device we want to imitate the application's view on. This way, anytime we added CSS code for the responsiveness, we could easily test it on our browser without needing to check on our mobile devices each time. To test the responsiveness, right click on the application page and select "Inspect" (as shown in Figure 1) and then select the tablet and phone icon from the top navigation bar. Once you select your desired dimensions based on the device, the responsiveness should be displayed differently (Figure 4)  than it does for the desktop view.

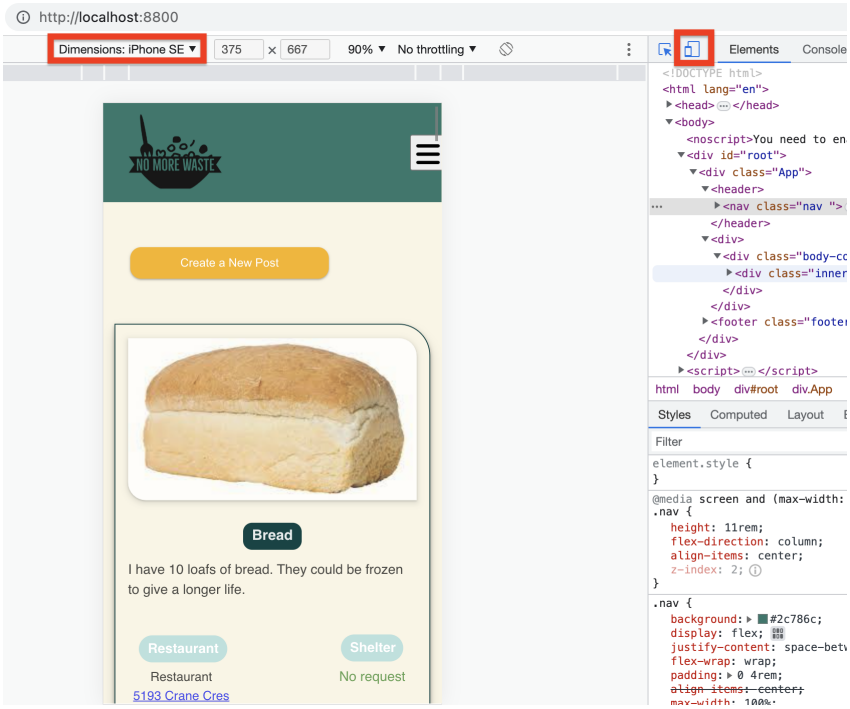


*Figure 4: iPhone SE view responsiveness*