TurboTrucks: Delivery System Project

SOEN 343 – Software Architecture and Design

Sprint 4 submission : Refinement and Refactoring

Dr. Joumana Dargham

Team Code Ninja

Jonathan Della Penta – 40128210

Julie Makary – 40243160

George Ezzat – 40245502

Monday, 2nd December 2024

Table of contents:

Diagram Refinement / Addition

Class Diagram:

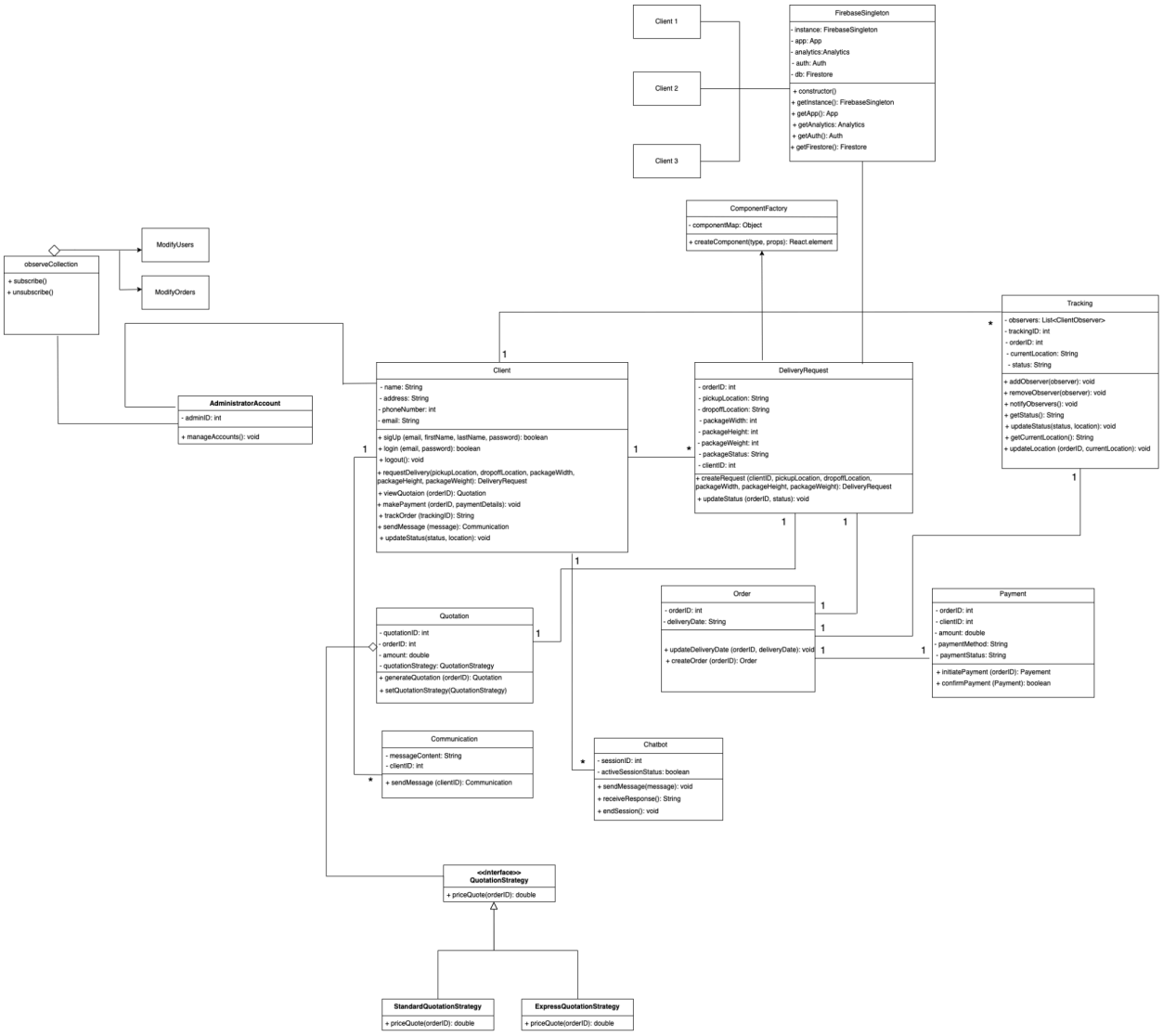


Figure 1: Class diagram connected with design patterns

The class diagram now utilizes specific design patterns to address certain design challenges effectively. The GOF design patterns chosen stayed the same (Factory, Observer, Singleton, Strategy), but they were implemented differently.

Instead of the singleton pattern being applied to the chatbot, it is now applied to Firebase. FirebaseSingleton class acts as a centralized interface for the database. This ensures that only one instance is created which handles all communication so that all clients communicate only with that one instance.
Instead of the factory pattern being applied to the accounts of clients, it is now applied to the Google Maps API. This pattern is used to create input fields that implement the Google Maps API. Since many components use this feature (ex: home page, delivery request, etc.) it ensures that new instances are created flexibly and decoupled from the code.

The observer pattern was applied to the tracking class, but now it is applied to the administrators. The administrators have the ability to modify the users or the orders; this leads to the database changing constantly. The clients subscribe to the observer, and they automatically get updates when there has been a change made to their order or account.

Lastly, the strategy pattern remained the same where it is applied to the delivery request. Because there are 2 types of shipping methods (standard & express), each one is implemented using a different strategy to accommodate the different behaviours.

These changes were essential for several reasons, as they enhanced the clarity, scalability and maintainability of the system's design. It improved the clarity of the code. By applying the Singleton pattern to the firebase centralized database interactions, ambiguity was reduced and consistent communication was ensured. These changes addressed new requirements too because the factory method now handles Google Maps API input fields, enabling reuse across multiple components. The observer pattern now ensures that clients receive updates about their orders.

Overall, these changes improved modularity, scalability and maintainability, ensuring that the system is well prepared for current and future needs.

Sequence Diagrams:

A significant change was the process that orders were created. At first, orders were created separately from the payment process. In this sprint, we decided that orders (or deliveries) should only be uploaded and saved to the database once payment is received, otherwise it may store lots of junk delivery objects that were never processed. Package information is still saved locally during the “Start Delivery” process, but is only saved by the system once payment is confirmed. This resulted in more efficient database storage, since it only stores deliveries that are confirmed. The costCalculator class was made to store the functions used to calculate the total costs of a delivery.

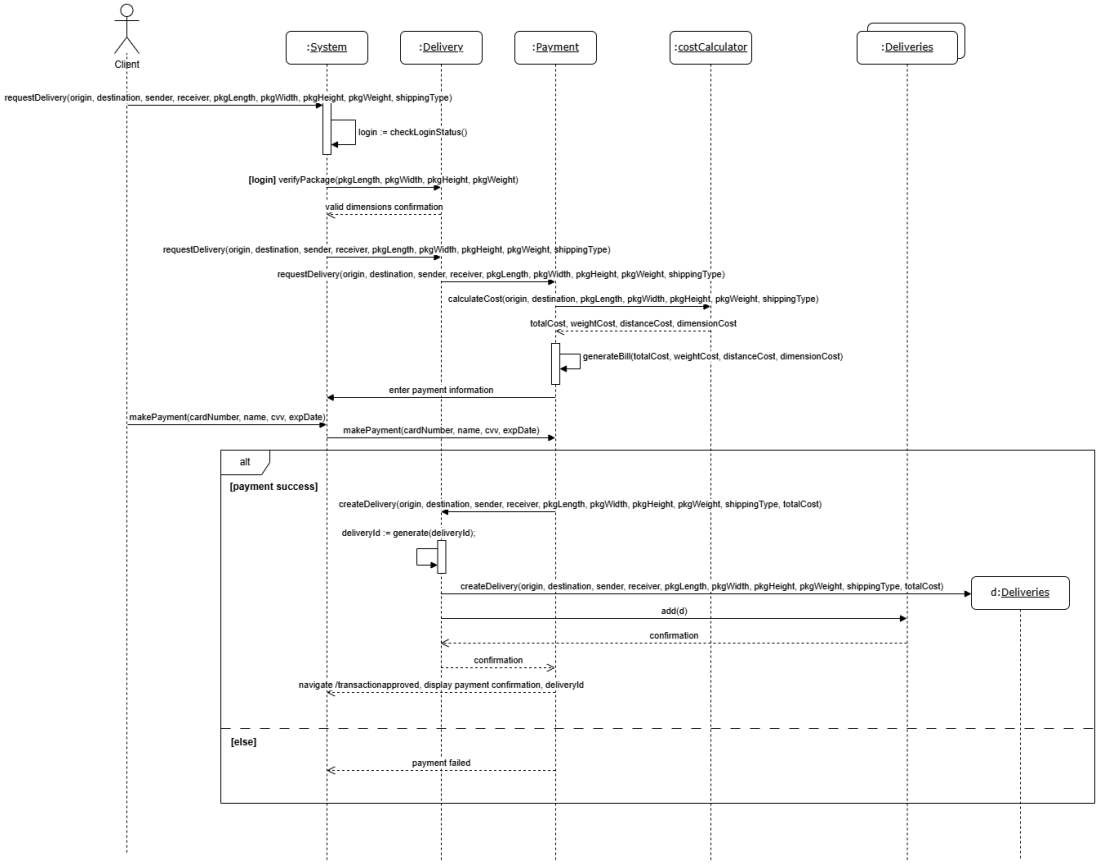


Figure 2. Creating a Delivery and Processing Payment Sequence Diagram

Minor improvements were made to the “Quote Service” and “Track Delivery” sequence diagrams to include more parameters that were added in sprint 4, as well as the costCalculator class.

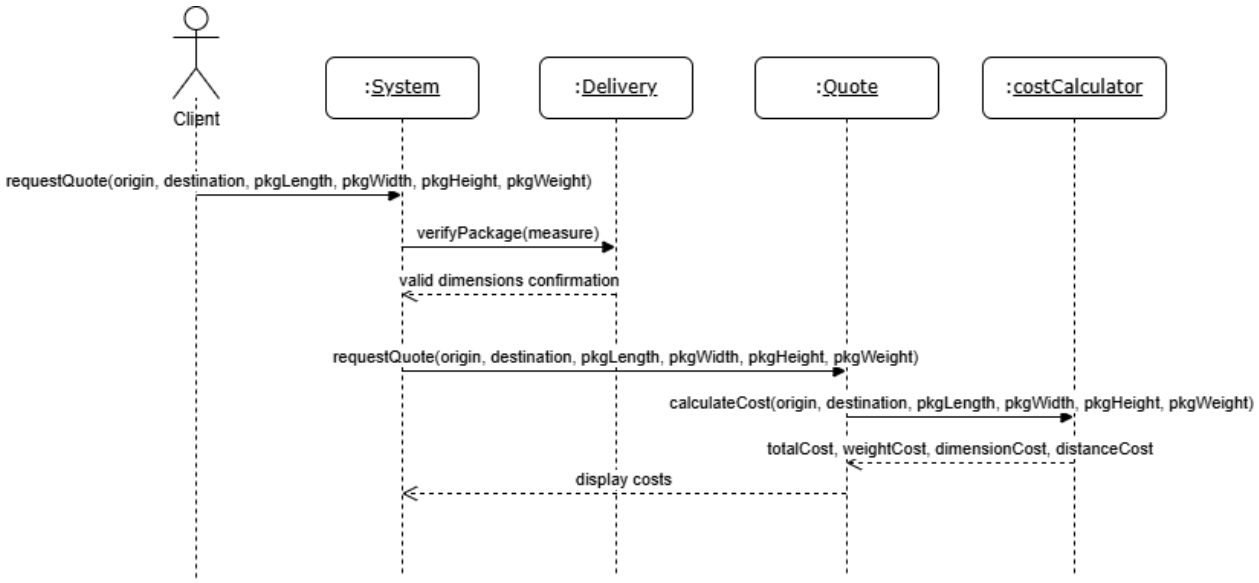


Figure 3. Requesting a Quote Sequence Diagram

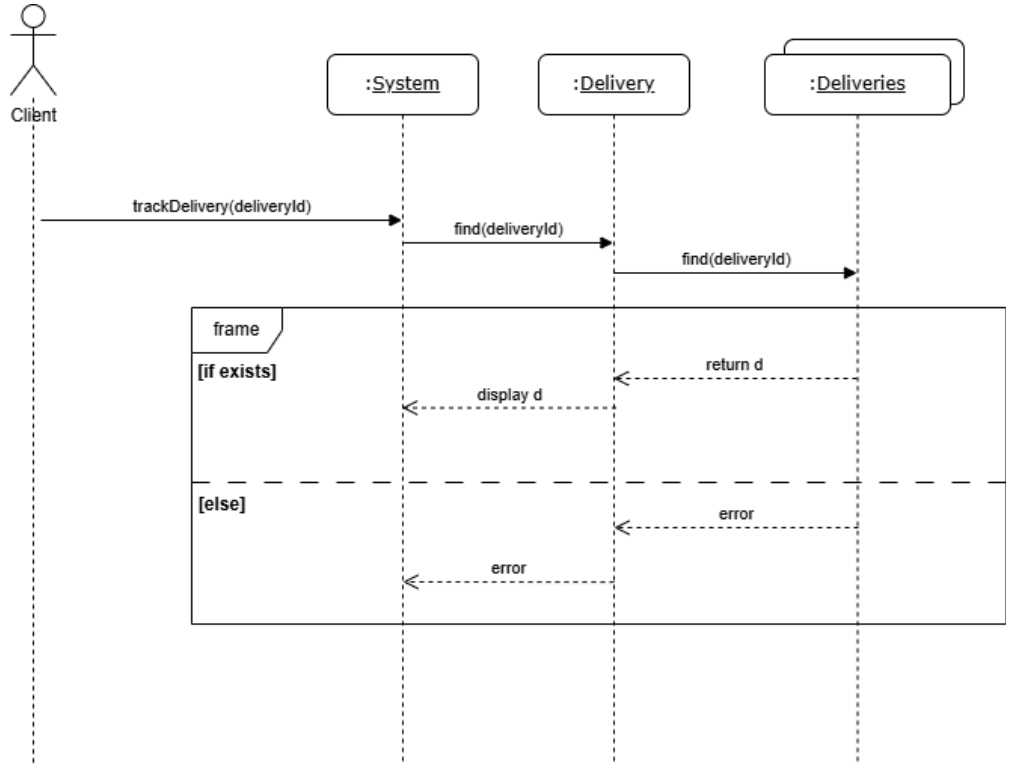


Figure 4. Tracking Delivery Sequence Diagram

Combining both the delivery and payment process into one diagram made the entire process easier to understand, since they work together. It displays the cohesion between the two processes. The benefit of the addition of the costCalculator class shows how the entire process interacts with separate classes that have specific roles. Instead of calculating the cost in each component where its needed, instead to reduce coupling, we create a separate class whose sole purpose is to be a cost calculator.

Code Refactoring

Design patterns were used in this sprint to help structure the existing code more effectively, such as the Strategy pattern, Singleton pattern, Factory pattern, and Observer pattern. Changes in the codebase were made to utilise these design patterns.

Many of the refactoring actions involved decoupling, increasing cohesion and consolidating duplicate code. Firstly, the Singleton pattern was used to handle all interactions with our database and authentication service through Firebase. The initial code for firebase was refactored to include a constructor and methods so that it may be used as a single instance in multiple components.

Figure 5. Singleton implementation for database interactions

Secondly, we used the Observer pattern to modify components that were required to generate and display lists of collections from our database. Instead of implementing the same methods in multiple classes, a listener class was created, and the existing code was modified. The listener class is used to subscribe to any changes that are made to the database, and may update simultaneously.

```
import { onSnapshot, collection, doc } from "firebase/firestore";
import FirebaseSingleton from './FirebaseSingleton';

const db = FirebaseSingleton.getFirestore();

export const observeCollection = (collectionName, callback) => {
  const collectionRef = collection(db, collectionName);
  return onSnapshot(collectionRef, (snapshot) => {
    const data = snapshot.docs.map(doc => ({ id: doc.id, ...doc.data() }));
    callback(data);
  });
};

export const observeDocument = (collectionName, docId, callback) => {
  const docRef = doc(db, collectionName, docId);
  return onSnapshot(docRef, (doc) => {
    if (doc.exists()) {
      callback({ id: doc.id, ...doc.data() });
    } else {
      callback(null);
    }
  });
};
```

Figure 6. Listener class for Observer method

Lastly, the factory pattern was used to create a factory class that creates components that are to be used often. It was designed to help implement input fields that used the Google Maps Autocomplete API in many different components. Using this pattern made the process of creating input fields simpler, as it needed less code than before.

code-ninjas-app > frontend > src > components > JS ComponentFactory.js > ...

```
1   import React from 'react';
2   import AutocompleteInput from './AutocompleteInput';
3
4   const componentMap = {
5     AutocompleteInput,
6   };
7
8   const ComponentFactory = {
9     createComponent: (type, props) => {
10      const Component = componentMap[type];
11      if (!Component) {
12        throw new Error(`Component type "${type}" is not recognized.`);
13      }
14      return <Component {...props} />;
15    },
16  };
17
18  export default ComponentFactory;
```

Figure 7. Component Factory used to create Autocomplete components

Features Implementation

All the system features have been finished and put together, such as creating user accounts, handling delivery requests, processing payments, and sending updates. Unit tests were used to check that each feature works correctly on its own. All tests were successful, and everything is working as it should. With the tests passed and no major problems, the system is now ready to be launched.

Feature implementation

1. Creating an account: Users can sign up to create an account to use the delivery service. Users can also log in to get access to the delivery service if the account is already created

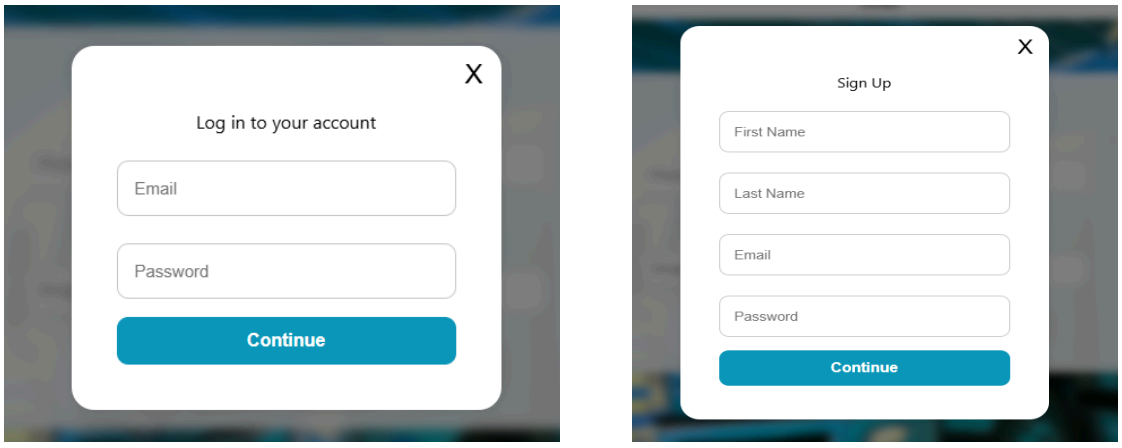
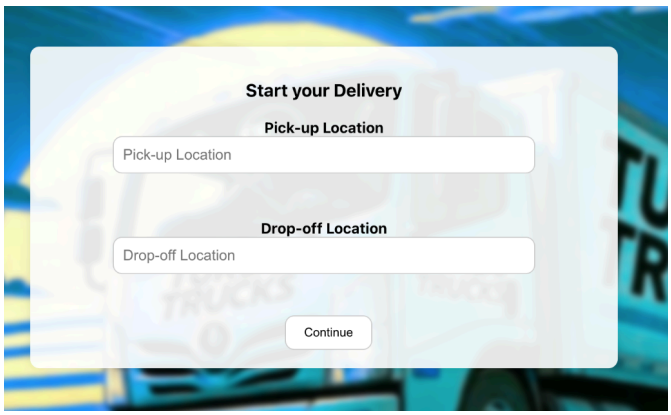


Figure 8: Log in & sign up for creating an account

2. Request for delivery: Users can fill in details about where to pick up and deliver their items, as well as dimensions and weight. Users can choose the type of shipping: free or express.



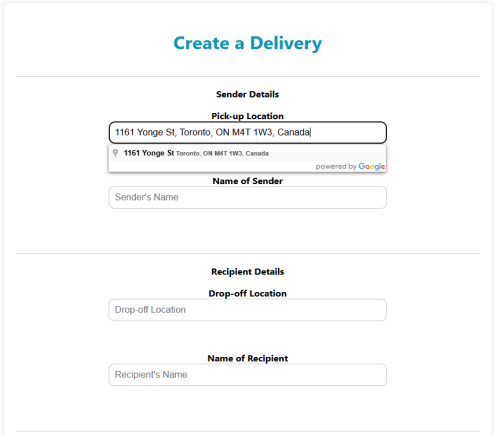


Figure 9: Create a delivery

3. Proposal of a quotation for the service: The system shows how much the delivery will cost based on the details provided.

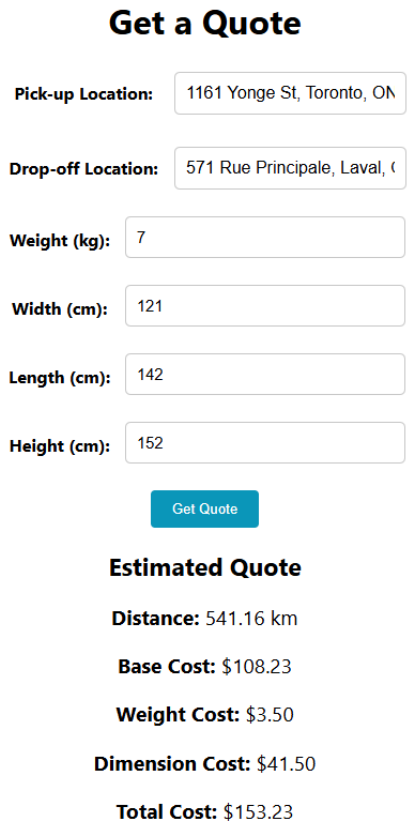


Figure 10: Get a Quote

4. Communication about the service: Communication about the service involves sending emails to users after payment is completed, notifying them of any package delays, and confirming when the package has arrived at its destination.

5. Tracking the order: Users can see the status of the package and when it will arrive.

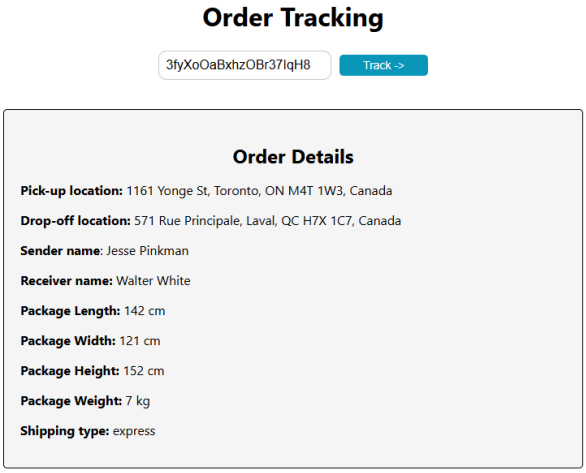


Figure 11: Tracking order page

6. Payment: Users can pay for the delivery safely using different payment methods.

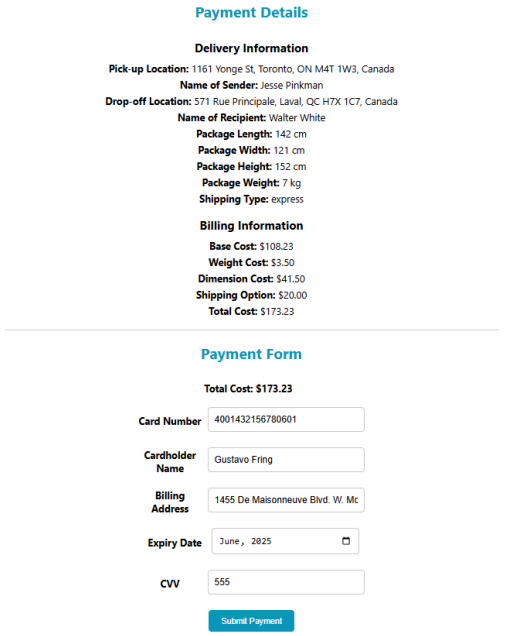


Figure 12: Payment of the package

7. Help assistance by using Chatbot: A chatbot is an AI that is Operational at all times. It helps users with questions or problems quickly and easily.

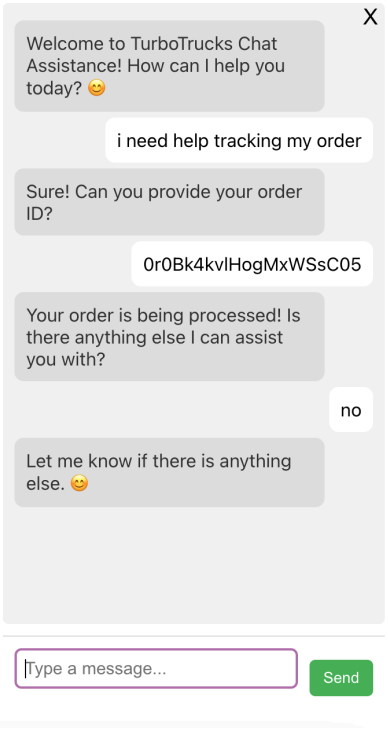


Figure 13: Asking chatbot on how to track the order

8. Admin management: Admins can manage accounts, deliveries, and other parts of the service to keep everything running smoothly.

Figure 14: admin main page

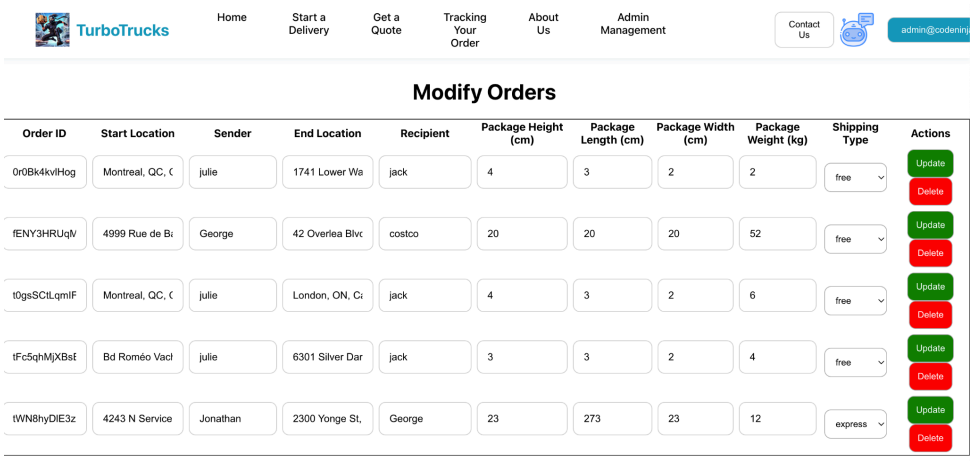


Figure 15: orders database

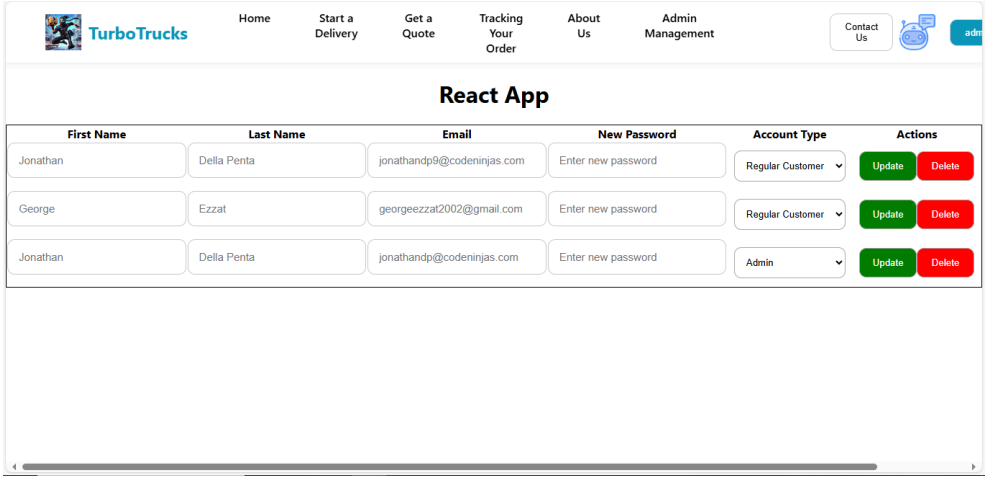


Figure 16: Users database

9. Firebase: Firebase is used to handle user accounts, store data, and make the app work well and fast.

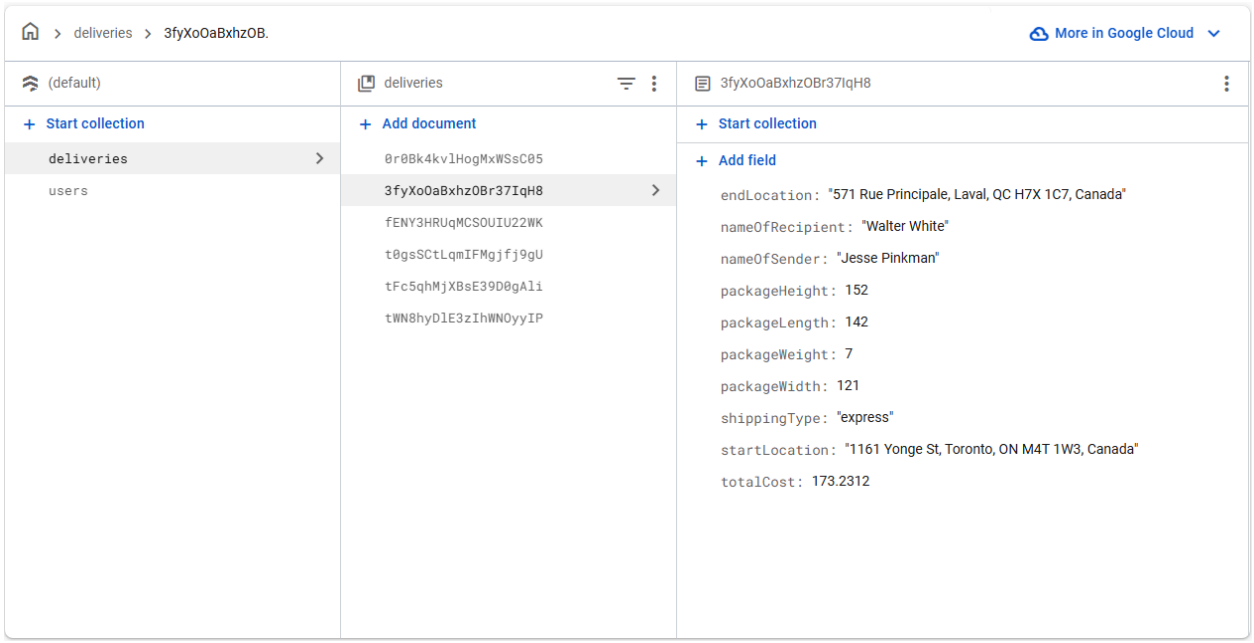


Figure 17: Firebase stores all deliveries and users in Firestore

10. Google Maps API: Google Maps API helps users easily find specific addresses across Canada and calculates the distance between locations to estimate costs. This ensures a smooth and accurate experience for navigation and delivery needs.

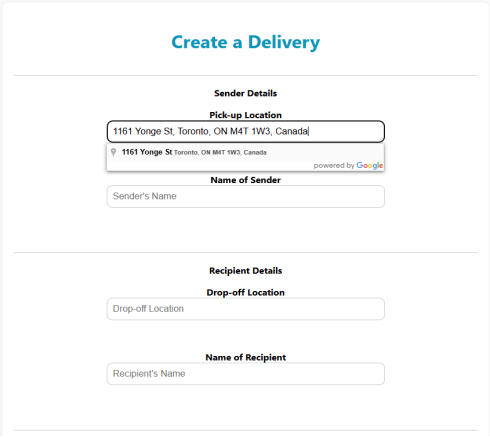


Figure 18: Searching for the address around Canada using Google Maps API

11. Review of service (extra feature): A Review of Service gathers user feedback to assess satisfaction and identify areas for improvement. It helps ensure the service meets expectations and can be enhanced.

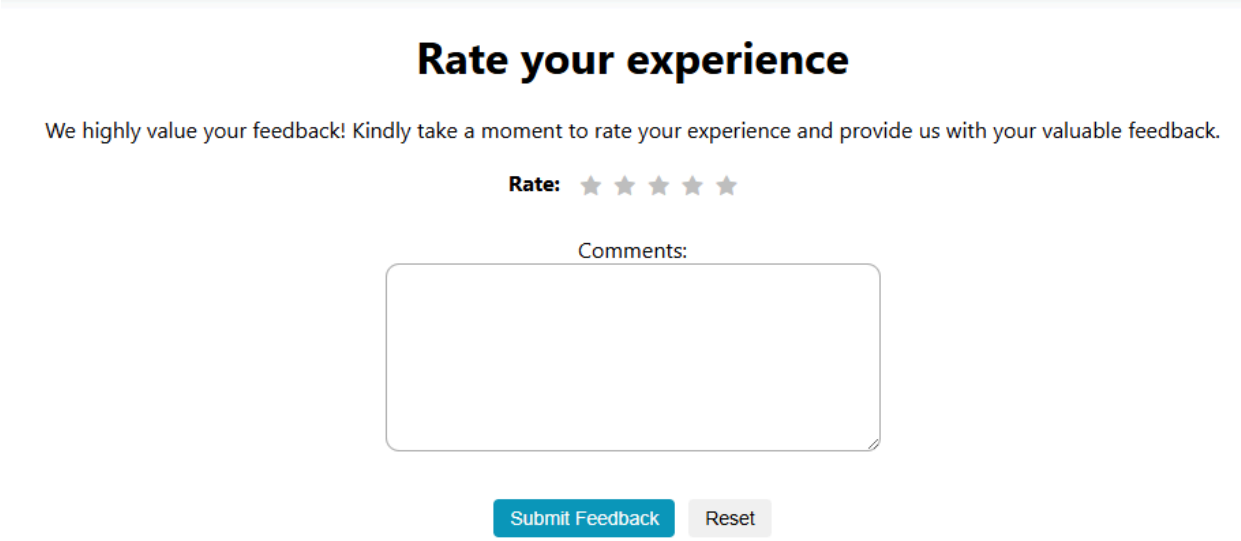


Figure 19: Feedback form

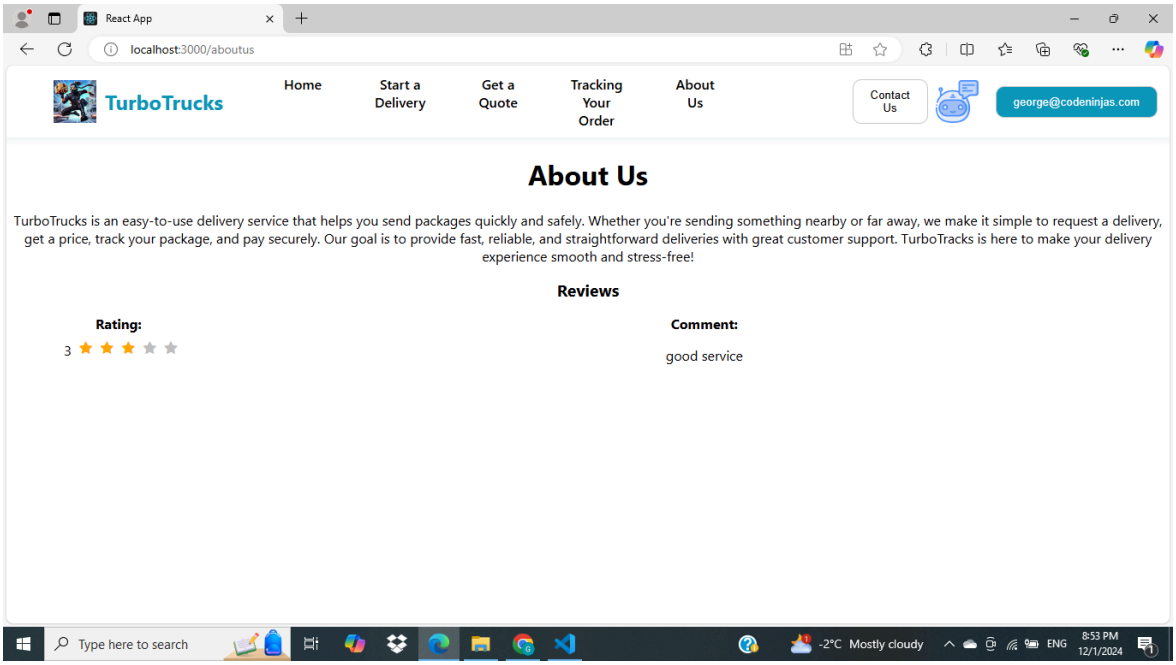


Figure 20: Feedbacks in about us

Conclusion:

The key accomplishments that we made during this sprint were finalizing our system architecture, using design patterns when structuring and refactoring our codebase, and completing the implementation of the core features. We tested and ensured that our codebase is functional and includes the functional requirements that were required to implement the core features. The Google Maps API was successfully implemented to assist us when specifying locations for package pickup and dropoff. Additionally, Firebase was used to assist us with our user authentication as well as our database. The frontend of our codebase was well-organized by many categories such as components, models, and other functions that encouraged a cohesive design.

The main challenge our team faced was the team size. There were many features that were planned but could not be implemented due to time constraints. Furthermore, there weren't many extra features that were implemented, which are crucial in making the project unique in this final sprint. It was an unexpected challenge, however we did our best to remain organized and finish a fully functional project that covers all the core features.

This sprint involved improving our class diagram and sequence diagrams, finishing the codebase and refactoring, and implementing features and testing.

Since the majority of this sprint involves the codebase, it's important for the class diagram and sequence diagrams to already be complete, and only small modifications, if any, are needed. Having well-formed diagrams make the process of creating components and writing code much simpler and faster, since it serves as a foundation for the entire implementation. Furthermore, we learned that it's important for the core features to be at least nearly complete by the end of sprint

3, since sprint 4 should be dedicated to refining existing features, adding any extra features and testing the entire codebase.

Sprint 4 showed us the benefits of having a well-defined system architecture prior to completing the codebase and emphasized the importance of teamwork by having an evenly-distributed workflow, good communication and time-management.