TurboTrucks: Delivery System Project

SOEN 343 – Software Architecture and Design

Phase 3 submission: GOF design pattern and Class diagram

Dr. Joumana Dargham

Team Code Ninja

Andra Vele – 40175695

Jonathan Della Penta – 40128210

Julie Makary – 40243160

George Ezzat – 40245502

Friday, 15th November 2024

Table of contents

I. Summary of the project

The "Turbo Trucks" service system is an online platform that helps people send and receive packages easily. Customers can request a delivery by giving the pick-up and drop-off details. The system provides a price estimate for the service and allows customers to communicate with the delivery team for updates. It also offers real-time tracking, so users can see where their package is. Payments can be made securely through the platform. Additionally, a chatbot is available to help customers with any questions or issues. The system is built in different layers to keep it organized and efficient. Moreover, a GOF design pattern has three type of patterns to help to solve common design problems, promote code reuse, and improve system maintainability: Creational Patterns Concerned with object creation (e.g., Singleton, Factory Method). Structural Patterns focus on object composition (e.g., Adapter, Decorator). Behavioral Patterns deals with object interaction and responsibilities (e.g., Observer, Strategy).

II. GOF design pattern

a. Strategy Design Pattern

Problem

The "Delivery" service application needs to calculate quotations for different types of delivery services, such as standard, express, or priority delivery. Each type of service uses a unique pricing formula. Currently, all pricing logic is handled in a single place, making the system hard to modify or extend when a new type of delivery service is introduced. This lack of flexibility and maintainability is a major limitation.

Solution

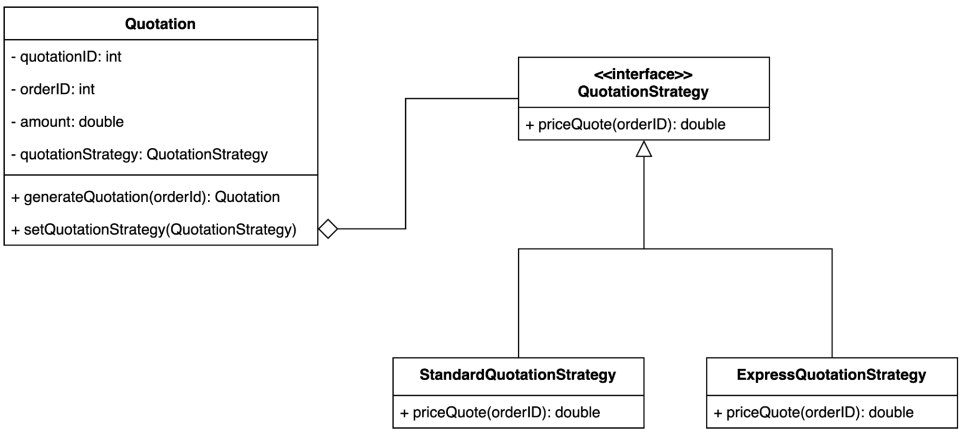
The Strategy Design Pattern solves this problem by allowing the pricing logic for each type of delivery service to be encapsulated in its own class. Instead of having one large block of code with all pricing logic, we define a family of pricing strategies, each handling a specific type of delivery. A context class is used to select and execute the appropriate strategy at runtime, making the system both flexible and easy to extend.

For example:

- A StandardQuotation strategy handles the pricing for standard delivery.
- An ExpressQuotation strategy deals with express delivery pricing.

This approach ensures that adding a new pricing scheme is as simple as creating a new strategy class without affecting the rest of the system.

Structure



b. Singleton Design Pattern

Problem

While each user session needs a unique chatbot instance, the system still requires a single point of control to manage all chatbot instances. Without centralized management, multiple managers could create overlapping or duplicate chatbot instances, leading to inefficiency and errors.

Solution

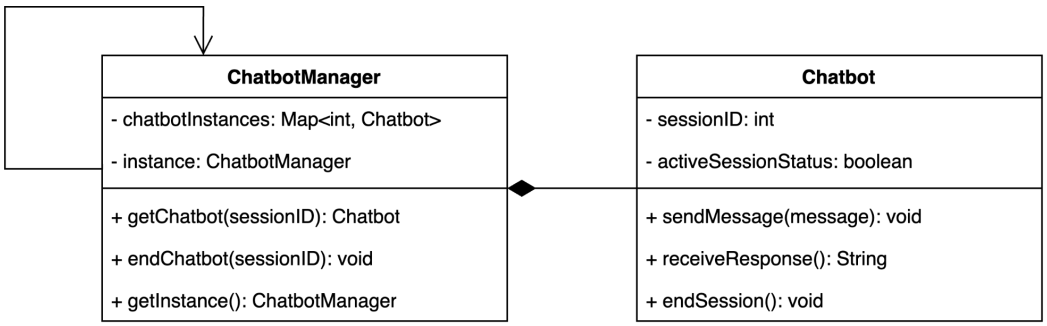
By applying the Singleton Design Pattern to the ChatbotManager, we ensure that:

1. Only one instance of ChatbotManager exists in the application.
2. This single ChatbotManager instance manages all chatbot instances for user sessions.
3. For each user, a single chatbot instance is maintained within the scope of their session.

How It Works:

- ChatbotManager (Singleton Class): The central authority for creating and managing chatbot instances. It uses a map to ensure each session ID corresponds to one chatbot instance.
- Chatbot (Per-Session Instance): Represents a chatbot tied to a specific user session. These are managed and created by the ChatbotManager.

Structure



c. Factory Design Pattern

Problem

The "Delivery" service application needs to manage three types of accounts: Clients, Delivery Drivers, and Admins. Each account type has its own unique features, like requesting deliveries, managing deliveries, or overseeing the system. Manually creating these accounts throughout the code would make the system hard to maintain and prone to errors. Adding new account types in the future would also require changing multiple parts of the system.

Solution

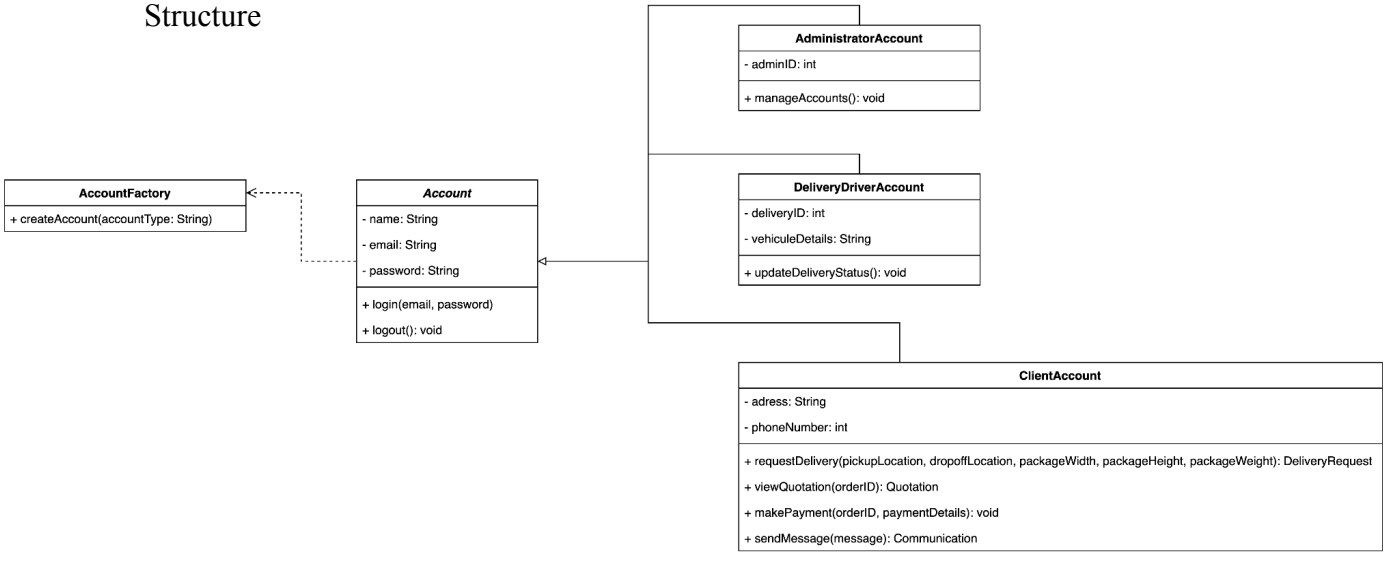
The Factory Design Pattern simplifies this by using a single AccountFactory to handle the creation of accounts. Instead of creating accounts directly, the factory takes care of deciding which type of account to create (e.g., Client, DeliveryDriver, Admin). This makes the system easier to extend and manage.

For example:

- If a user chooses "Client," the factory creates and returns a Client object.
- If a user chooses "Delivery Driver," the factory creates and returns a DeliveryDriver object.

By centralizing account creation, the code becomes simpler and more flexible.

Structure



d. Observer Design Pattern

Problem

One of the problems is that the Clients want to know the real-time status of their deliveries. Without a notification mechanism, clients must manually check the status of their orders, leading to inconvenience and inefficiency.

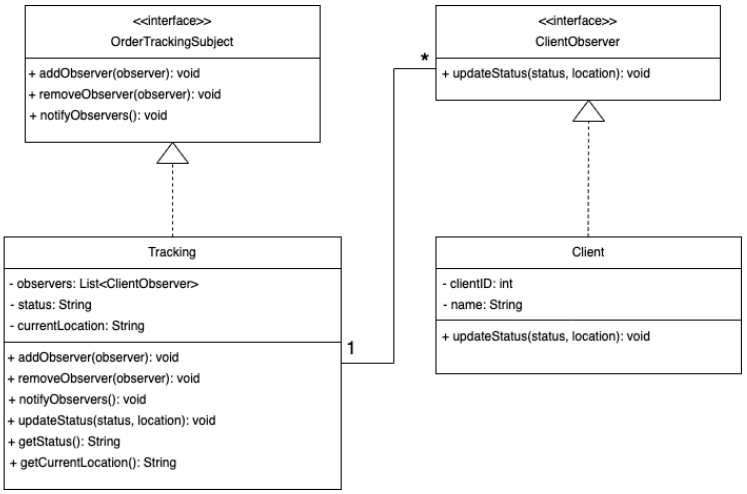
WAnother problem is that the system would have tight coupling between the tracking functionality and the clients. The system would need to keep track of every client who requested updates, which is not scalable or flexible.

Solution

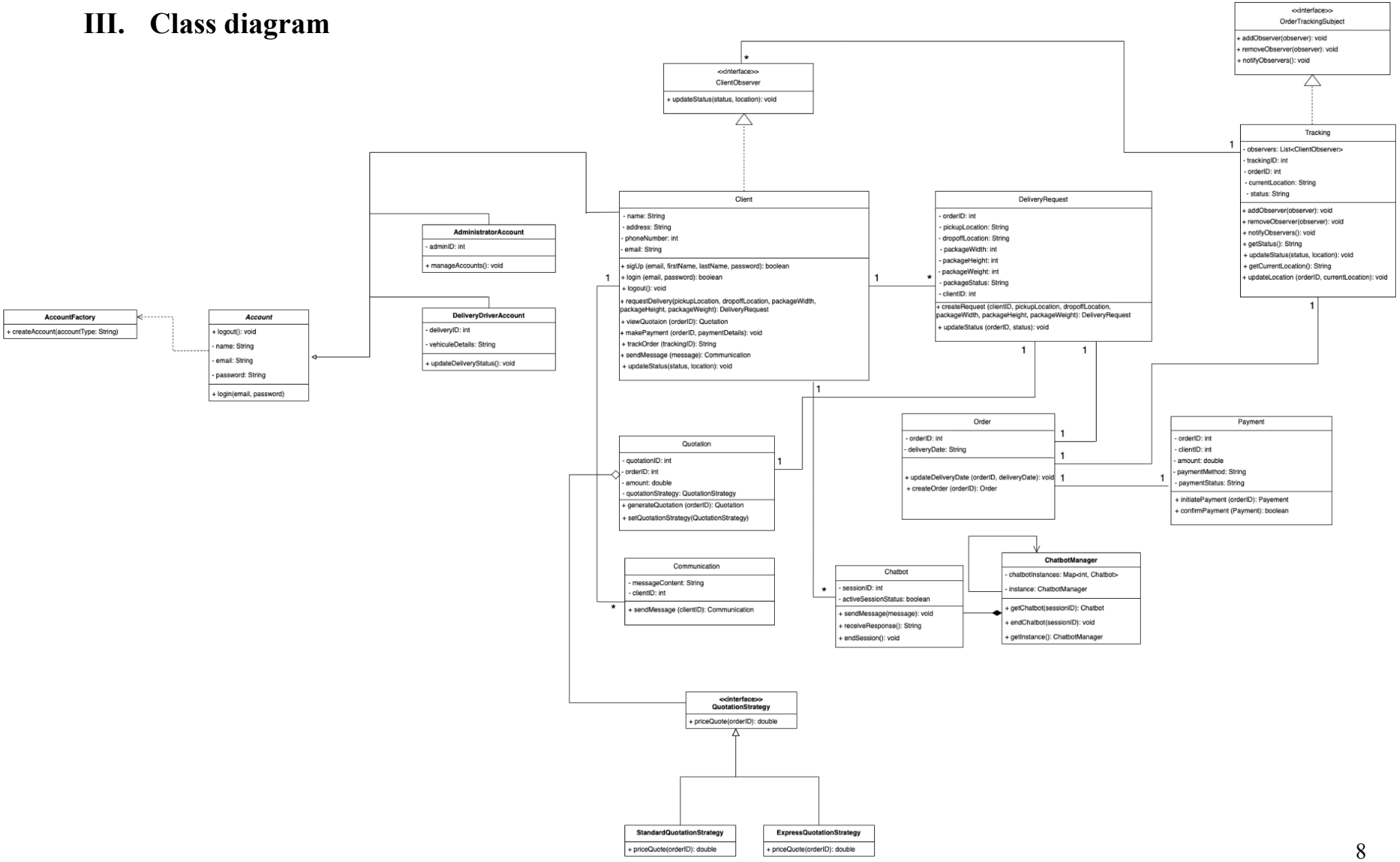
By implementing the Observer Pattern, the Tracking class can automatically notify clients about the status of their deliveries. Clients do not need to manually check for updates; they receive notifications through the update() method whenever there is a status change.

Moreover, the observer pattern decouples the Tracking system from the clients. The Tracking class only needs to maintain a list of subscribed clients and does not need to know any details about the clients it notifies. It simply calls their update() method, ensuring a more flexible and maintainable system.

Structure

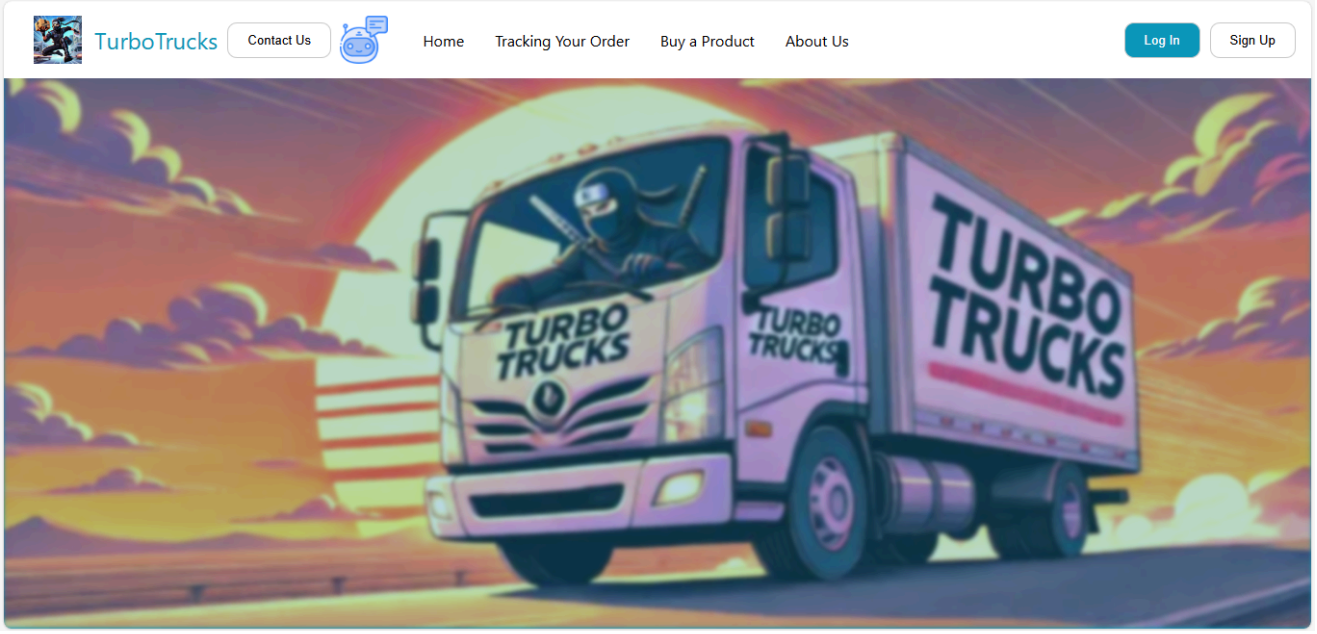


III. Class diagram

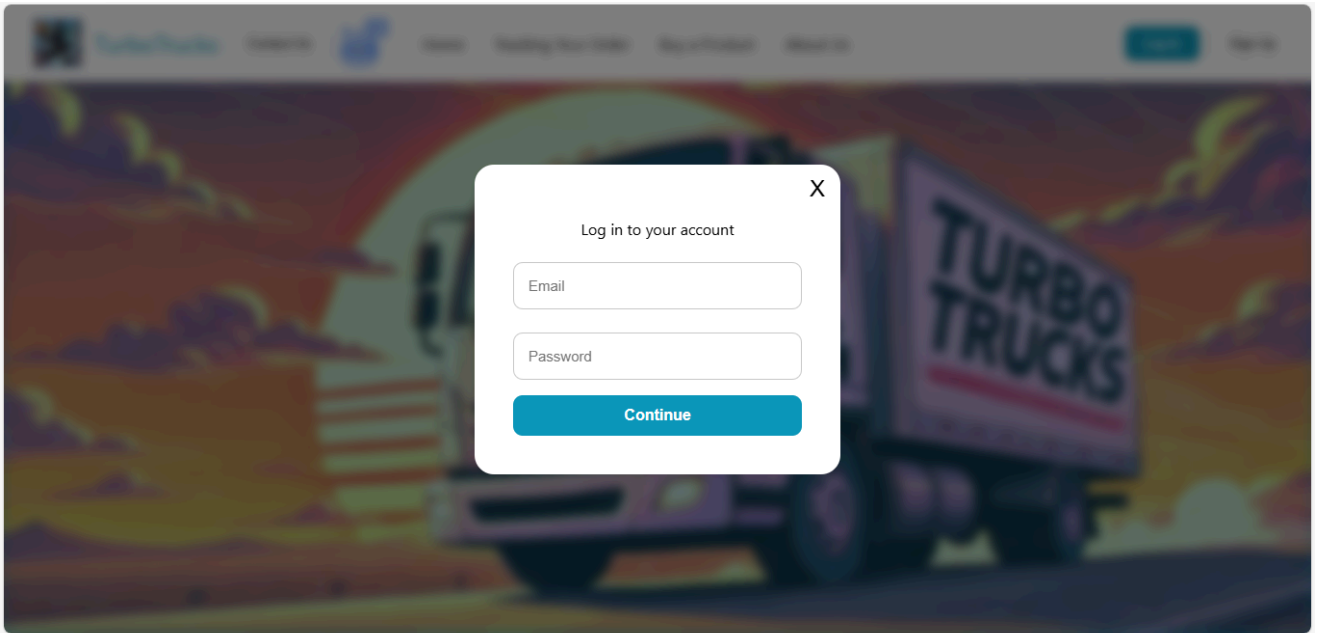


IV. Implementation

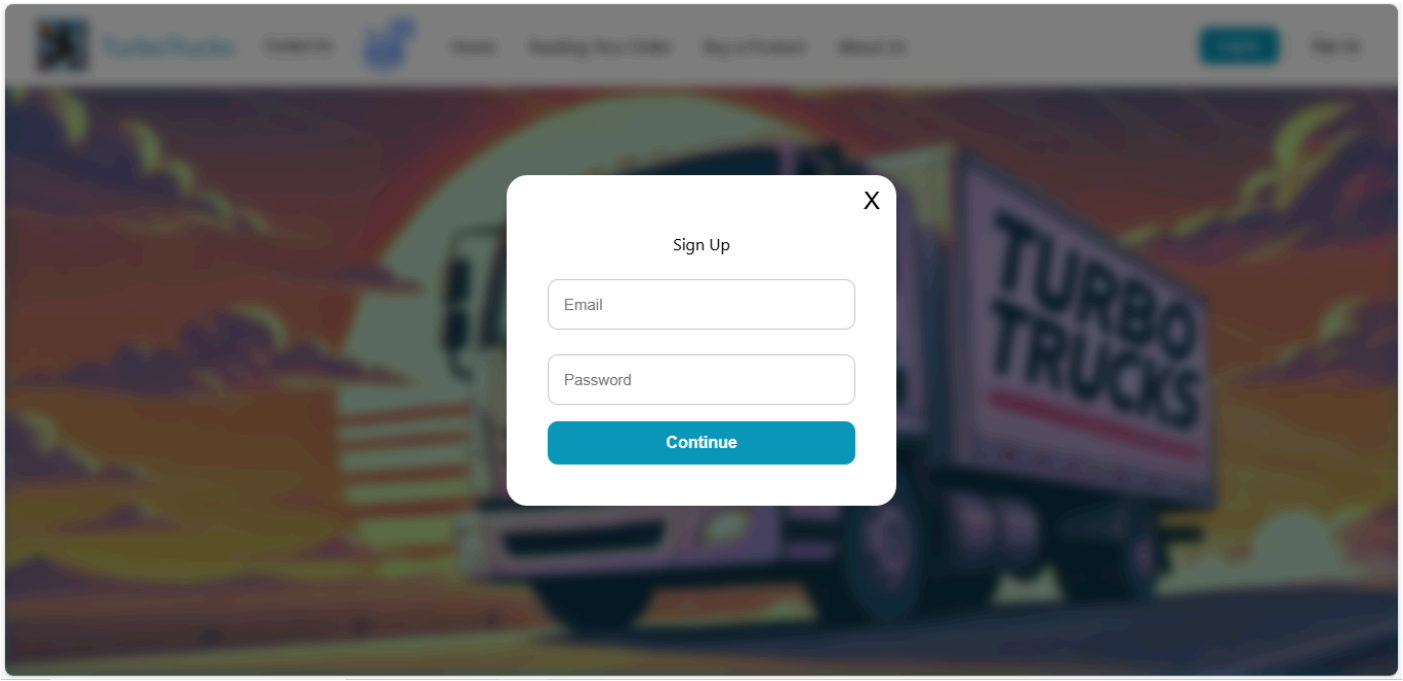
Home page



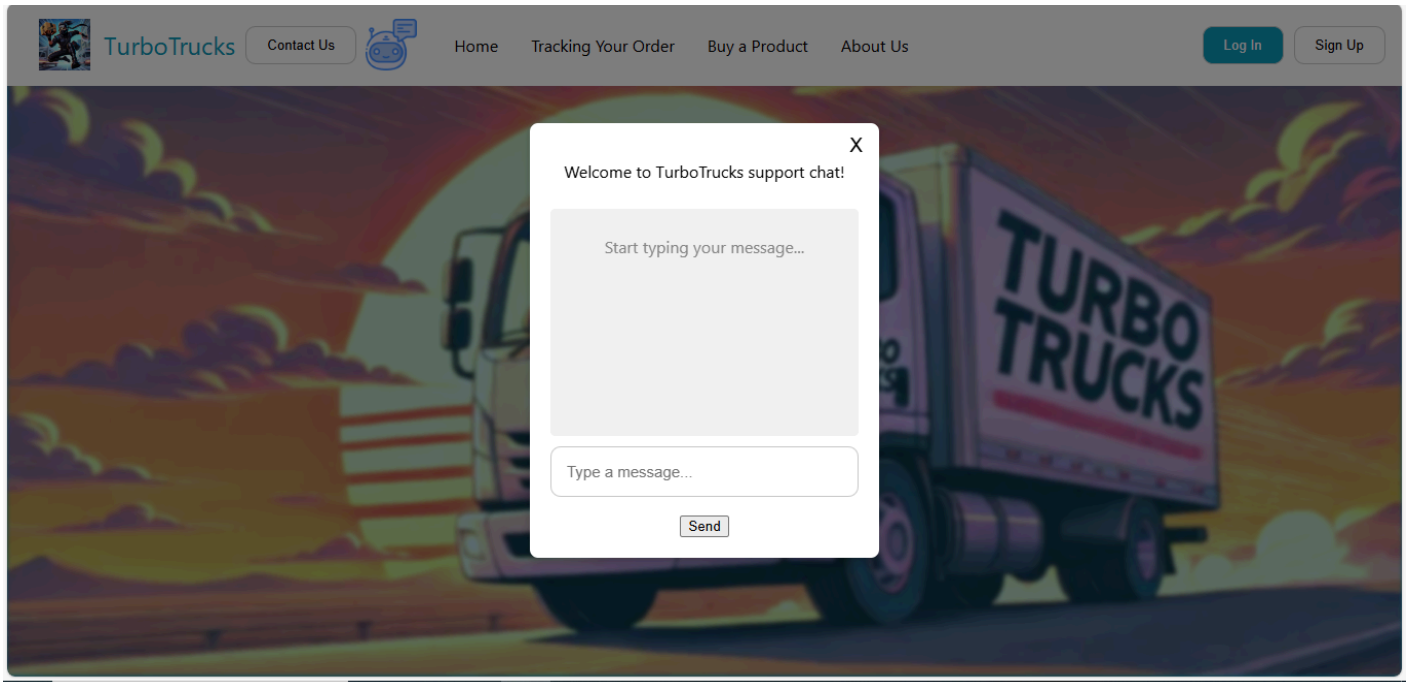
Login page



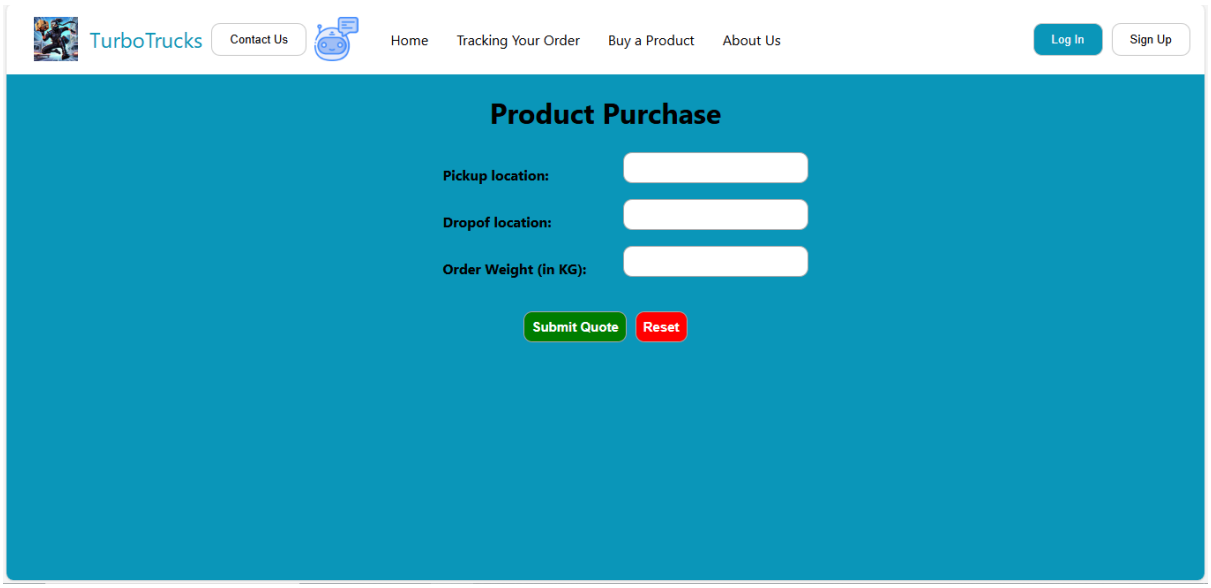
Sign Up Page



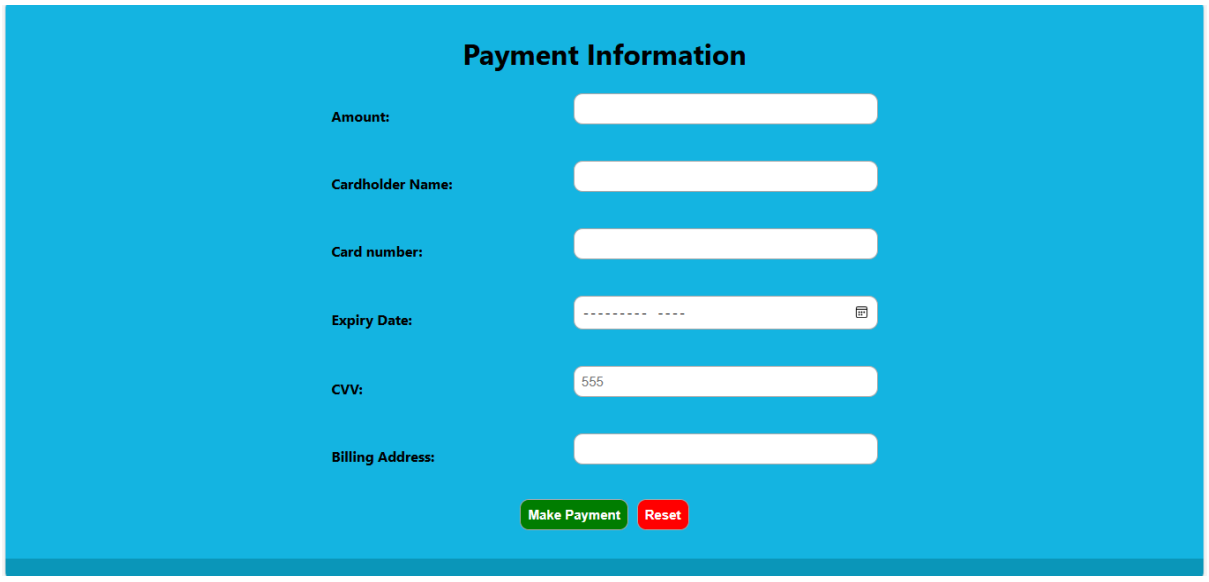
Chatbot page



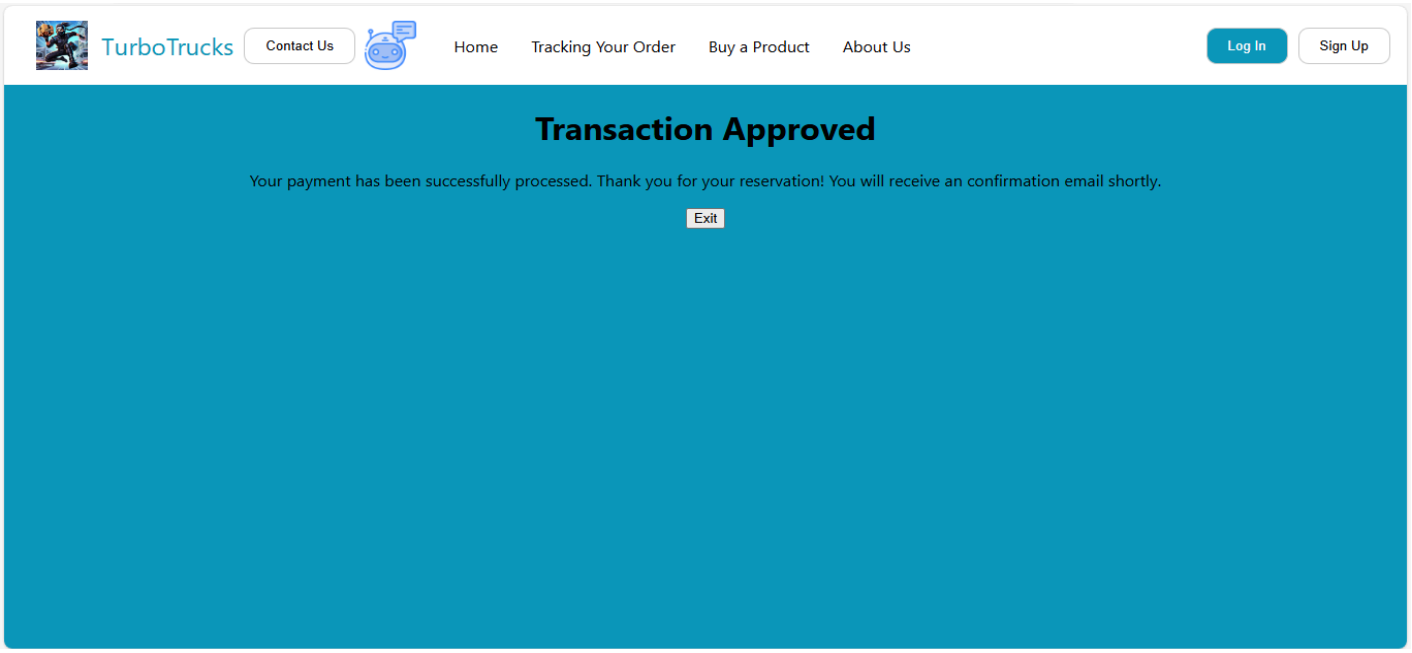
Get a Quote page



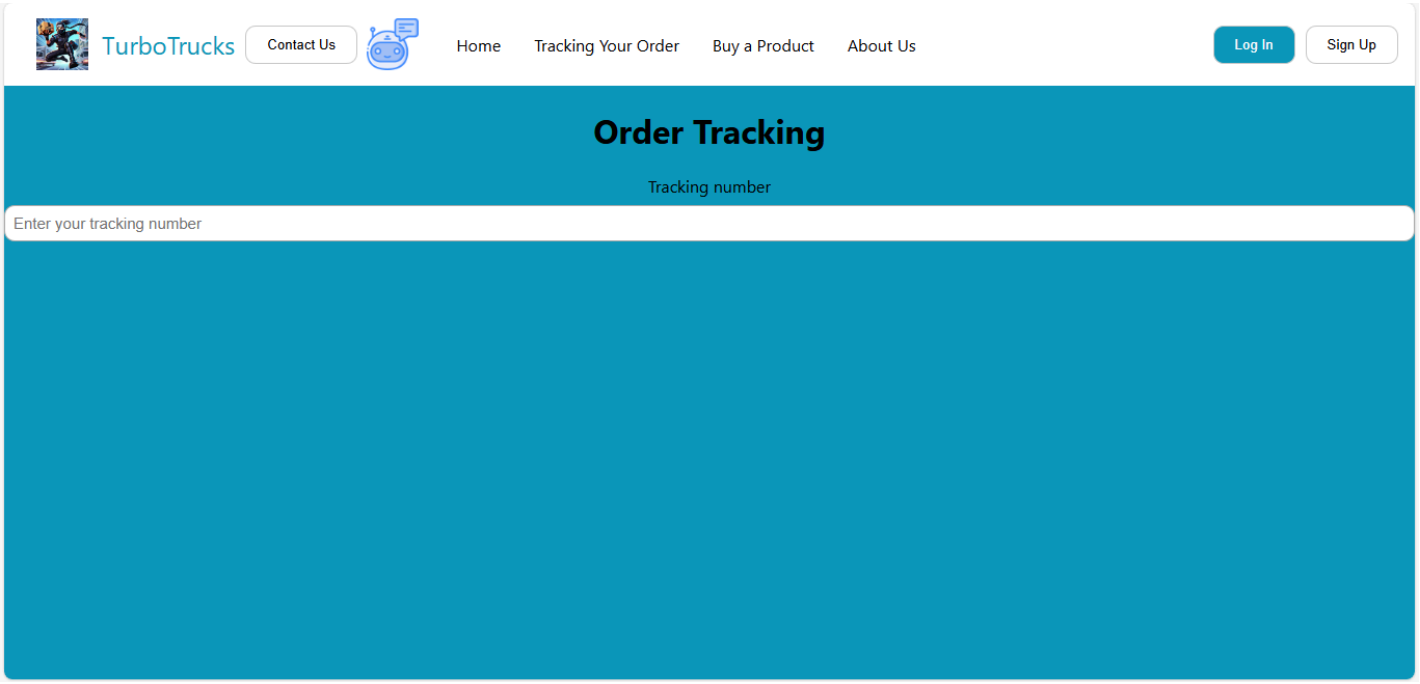
Payment page



Transaction approved page



Tracking order page



About us Page