School of Arts and Sciences

Department of Mathematics and Computer Science

RideRevive

Automotive Tools E-commerce Web Application

Capstone Project Report

Submitted By:

Abed Almajeed Kanso # 202106219

To:

Dr. Azzam Mourad

Table of Contents

I. Introduction:

RideRevive is your premier online destination for all things automotive. Our platform combines variety, quality, and convenience to provide car enthusiasts and everyday drivers with the ultimate shopping experience. Designed with a keen eye for detail, our website caters to all your automotive needs, offering a diverse selection of premium car products.

Key Features of Our Website:

1. **Streamlined Shopping Experience:** Navigate effortlessly with our user-friendly design. Our sophisticated shopping cart and smooth checkout process are crafted to provide a convenient and hassle-free shopping journey from start to finish.
2. **Responsive Design:** Enjoy a seamless experience on any device. Our website's responsive design ensures that you can browse and shop with ease, whether you're on a desktop, tablet, or smartphone.
3. **User-Friendly Features:** Our platform is built to enhance user engagement and convenience. Easily manage your account, with features like signing in, signing out, and signing up. Strong authentication procedures keep your data secure, while comprehensive product reviews help you make informed decisions.
4. **Effective Search Functionality:** Find exactly what you're looking for with our powerful search feature. Whether you're hunting for a specific item or exploring our extensive catalog, our search function makes navigation easy and efficient.
5. **AI-Powered Assistance:** Optimize your shopping experience with our AI chatbot, designed to assist you with any queries or issues you might encounter. From finding specific products to understanding specifications and addressing order concerns, our intelligent chatbot is here to provide immediate and effective support.

At RideRevive, we are dedicated to delivering an exceptional online shopping experience that meets and exceeds your automotive needs. Whether you're upgrading

your vehicle or searching for the latest accessories, our platform is designed to offer both variety and convenience. Discover the best for your car with RideRevive, where quality and customer satisfaction are our top priorities.

II. Software Architecture and Database:

i. Technology used and Code Architecture:

The project is built using MERN stack, and the code is implemented using the Client-Server Architecture, designed for both public users and administrators use. Some key aspects:

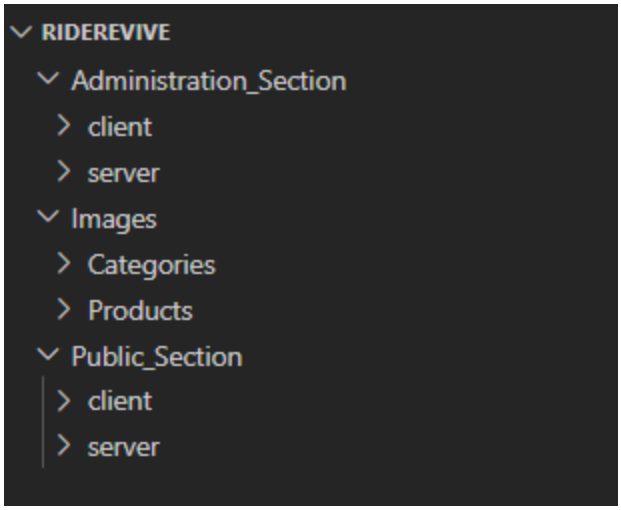
1. **MERN Stack Integration:**
    - **MongoDB:** This NoSQL database is utilized for storing user information, product details, and order records securely and efficiently.
    - **Express.js:** Serving as the backend framework, Express.js manages routing, middleware, and RESTful APIs, facilitating smooth communication between the client and server.
    - **React:** The frontend library, React, provides a dynamic, responsive, and intuitive user interface, enhancing user interaction and overall experience.
    - **Node.js:** This runtime environment allows for executing JavaScript on the server side, ensuring efficient and scalable backend operations.
2. **Separation of Sections:**
    - Ride Revive is divided into two distinct sections, each utilizing its own client-server architecture:

        - **Public Section:** Designed for general users, this section allows browsing products, managing accounts, making purchases, and utilizing features like the shopping cart and Wish List (favorites list).
        - **Administration Section:** Reserved for administrators, this section provides tools for managing user accounts, products, categories, and order records. Administrators can add, update, and delete categories and products, as well as ban users if needed.

3. **Security Measures:**
    - **Password Hashing:** User passwords are securely hashed before being stored in the database. To change a password or update profile information, users must verify their identity by entering their current password.
    - **JSON usage in RESTful APIs:** JSON is used in RESTful APIs for all get and post actions, ensuring that only authorized actions can be performed within the system.
4. **Additional Integrations:**
    - **NodeMailer Library:** NodeMailer is integrated to handle email functionalities such as sending a confirmation email after successfully making a order and other things.
    - **TypeScript**: TypeScript is integrated into the project in some parts to improve code quality and maintainability, providing type safety and reducing potential bugs in both the frontend and backend codebases.
    - **React-Toastify:** React-Toastify is used for displaying notifications in the React application, providing a seamless and user-friendly way to alert users about important actions and events, such as successful form submissions or errors.

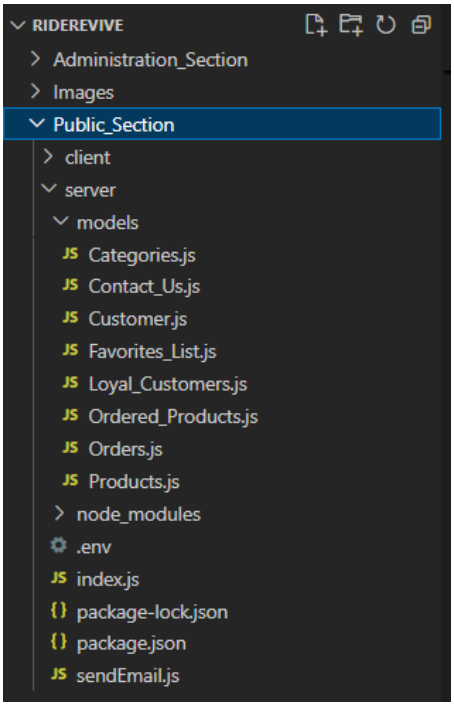
Parts of the Code for illustration:

1. **The project structure is as follows:**



It is worth noting that the content of the Images folder that contained the products and the categories was shared between the two sections, since once an Administrator adds a product for example, it should become available for the users to buy. Of course, there are some other images in the client side of each section that were not included in the folder since they are static. The folder is stored in the folder of the project and can be stored in another place.

2. **How the server of the Public Section Looks Like:**



The models folder includes all the entities in the database used in the Public Section for the CRUD operations. For example, the Categories file represented contains all the attributes of the Category stored in MongoDB. The .env file stores the Port number, the connection string to the local host, and the password and username to NodeMailer. Any necessary additional information can be stored in it. It was better to integrate NodeMailer in the backend rather than the front end since the client part dependencies will crash with the dependency of it in the package files and there is no necessary need for it to be included in the client side which is heavier. The figures below show what the beginnings of the root file in the server look like, and how some of the get and post Restful APIs are placed (all of them follow the same criteria). The examples include Post, Patch, and Get.

index.js

Public_Section > server > index.js > ...

```javascript
app.post('/createCustomer', async (req, res) => {
    try {
        const { Customer_fullname, Customer_email, Customer_number ,Customer_password, Customer_address } = req.body;
        const Customer_state = true;
        if (!Customer_fullname || !Customer_email || !Customer_number || !Customer_password || !Customer_address) {
            return res.status(400).json({ error: 'All fields are required' });
        }

        const existingCustomerEmail = await CustomerModel.findOne({ Customer_email });
        if (existingCustomerEmail) {
            return res.json("EmailExists");
        } else {
            const newCustomer = new CustomerModel({
                Customer_fullname,
                Customer_email,
                Customer_number,
                Customer_password,
                Customer_address,
                Customer_state
            });

            await newCustomer.save();
            res.json(newCustomer);
        }

    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});
```

index.js

Public_Section > server > index.js > ...

```javascript


app.patch('/updateProduct/:id', async (req, res) => {
    try {
        const { Amount } = req.body;

        const product = await ProductModel.findOne({ Product_ID: req.params.id });
        if (!product) {
            return res.status(404).json({ message: 'Product not found' });
        }

        product.Product_current_amount -= Amount;

        if (product.Product_current_amount === 0) {
            product.Product_state = false;
        }

        await product.save();

        res.status(200).json({ message: 'Product updated successfully', product });
    } catch (error) {
        console.error('Error updating product:', error);
        res.status(500).json({ error: error.message });
    }
});



app.get("/getProducts", async (req, res) => {
    try {
        const products = await ProductModel.find({});
        res.json(products);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});

```

```
index.js
Public_Section > server > index.js > ...
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const bcrypt = require('bcrypt');
const CustomerModel = require('./models/Customer');
const ProductModel = require('./models/Products');
const OrderModel = require('./models/Orders');
const ProductOrderModel = require('./models/Ordered_Products');
const CategoryModel = require('./models/Categories');
const FavoritesModel = require('./models/Favorites_List');
const loyalModel = require('./models/Loyal_Customers');
const ContactModel = require('./models/Contact_Us');
const sendEmail = require('./sendEmail');
const multer = require('multer');
const path = require('path');
const fs = require('fs');
const app = express();
app.use(cors());
app.use(express.json());
require("dotenv").config();

mongoose.connect(process.env.MONGO_DB_URI);

app.use('/Categories', express.static(path.join(__dirname, '../../Images/Categories')));
app.use('/Products', express.static(path.join(__dirname, '../../Images/Products')));


const categoryStorage = multer.diskStorage({
    destination: (req, file, cb) => {
      cb(null, path.join(__dirname, '../../Images/Categories'));
    },
    filename: (req, file, cb) => {
      cb(null, Date.now() + path.extname(file.originalname));
    },
  });

  const productStorage = multer.diskStorage({
    destination: (req, file, cb) => {
      cb(null, path.join(__dirname, '../../Images/Products'));
    },
    filename: (req, file, cb) => {
      cb(null, Date.now() + path.extname(file.originalname));
    },
  });
```

3. How send Email works after post APIs:

```
index.js   sendEmail.js
Public_Section > server > sendEmail.js > ...
require("dotenv").config();
const nodemailer = require('nodemailer');

async function sendEmail(to, subject, htmlContent) {
  let transporter = nodemailer.createTransport({
    host: 'smtp.gmail.com',
    port: 587,
    secure: false,
    auth: {
      user: process.env.Email,
      pass: process.env.Secret_key,
    },
  });

  let info = await transporter.sendMail({
    from: 'You <' + process.env.Email + '>',
    to: to,
    subject: subject,
    html: htmlContent,
  });

  console.log(`Message sent: ${info.messageId}`);
}

module.exports = sendEmail;
```

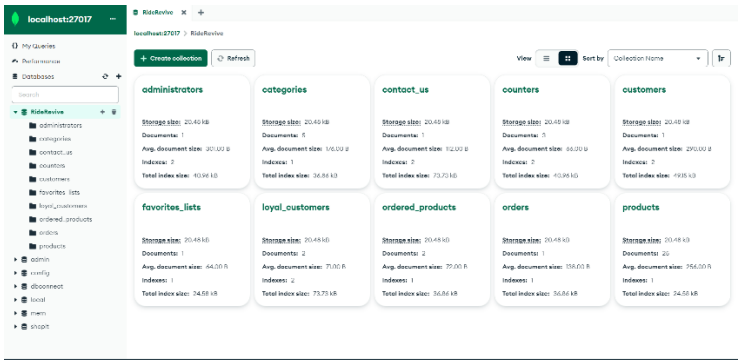
```
index.js
Public_Section > server > index.js > app.get('/getOrdersByCustomerID') callback
app.post('/createOrder', async (req, res) => {
    try {
        const { Customer_ID, Order_payment_method, Order_total_price } = req.body;

        if (!Customer_ID || !Order_payment_method || !Order_total_price) {
            return res.status(400).json({ error: "All fields are required" });
        }

        const newOrder = new OrderModel({
            Customer_ID,
            Order_Date: Date.now(),
            Order_payment_method,
            Order_total_price
        });

        const customer = await CustomerModel.findOne({ Customer_ID });

        await sendEmail(
            customer.Customer_email,
            "Order Confirmation",
            `<h1>Hello ${customer.Customer_Fullname}</h1><p>thank you for your order. Your order has been performed successfully.
            You can view your order on the website in your account if you are authenticated.</p>`
          );

        await newOrder.save();
        res.json(newOrder);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});
```

The senEmail file exports the function that sends an email and it takes three parameters which are the email the system is sending to, the subject, and the

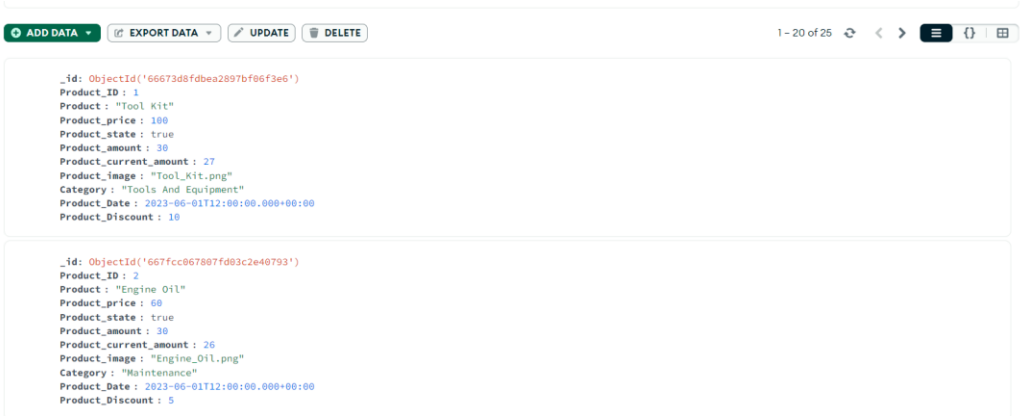
content of the email. The username and the password sending the email are variables stored in the .env file for more security. Let's suppose a customer finishes his order, and receives a confirmation email on his email where the subject and the content of the email sent are constant for all users and the variables in the API are only his email and name. It is worth noting that the API takes the Customer ID, and then fetches his data from his ID accordingly.

ii. Database:

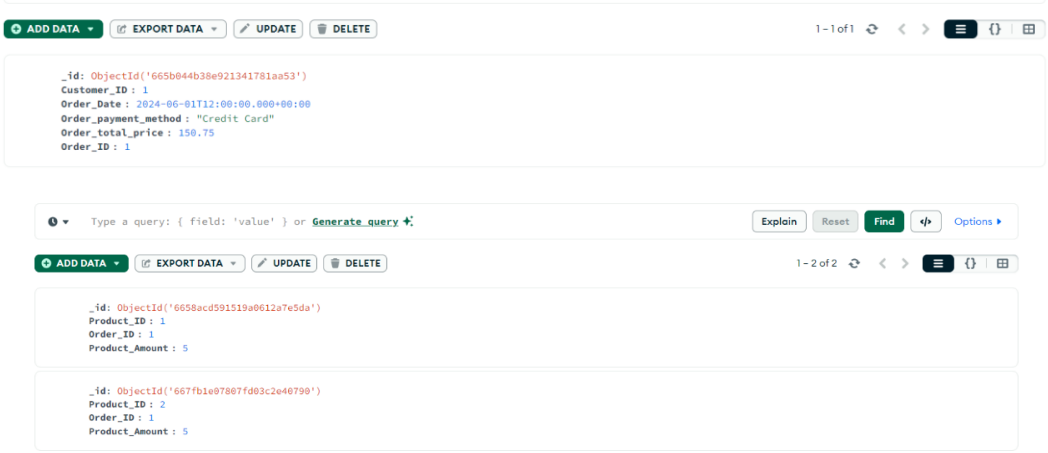
The database is represented in the following figure:



The categories, administrators, customers, and contact us tables are clear. The entity's loyal customers only contain one attribute which is loyal email. It is for subscribing to a service that once a new product is added you get notified. Once you subscribe you get an email by the system email notifier. The entity favorites list only contain two attributes which are the customer ID and the product ID, to know which product each customer have as a favorite. The table counters is for tables that need Autoincrement. Since MongoDB is a NoSQL database, the Autoincrement IDs which are single primary keys should be recorded. Both the processes of incrementing and hashing are done programmatically using special libraries. The table of the products is displayed as follows:



Before inserting a product, it is guaranteed that no other product has the same name. The category in each product is selected among the categories in the system. The product's current amount decreases once a customer makes an order with a specific quantity. Once the current amount becomes 0, the product state will be set to false and it will not be displayed in the public section. The product image is retrieved from the images' common special folder. The product ordering mechanism storage is as follows:



The table order stores the order attributes along with the Order ID which is Autoincremented, without storing the ordered products. The ordered products entity has three attributes: the ordered ID, the Customer ID, and the quantity of

the product. It indicates the ordered products for each order. The previous three figures illustrate one order performed in the system by the customer of ID 1.

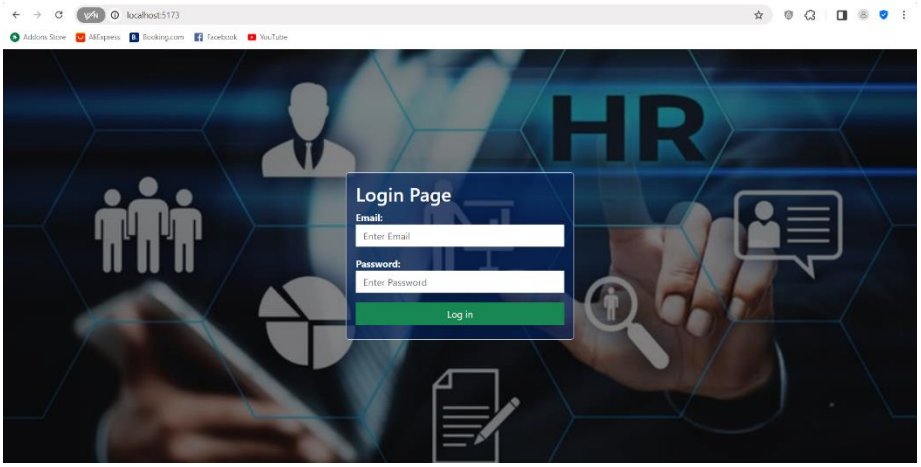
III. Administration Section:

Notes:

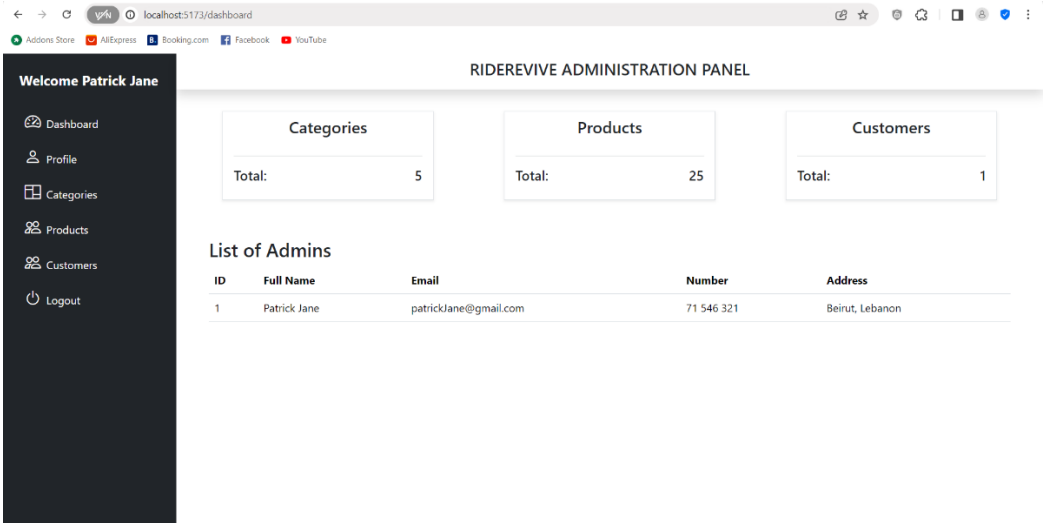
- TypeScript was used in all codes of the Administration section.
- Error handling for empty fields and wrong data types was implemented in all places where it was necessary on the client side.
- React-Toastify was used after almost all actions. The second figure in the profile part shows how it appears.
- Once data changes after any action on any page, the data is re-fetched after 1.5 seconds so that the user will directly see the new effect after the toast.
- If an edit page like edit category depended on a value which is located at the end of the URL, if this value was modified for some reason and it became not in the data collection, the edit page will directly redirect to the previous page.

i. Authentication:

The Login page is as follows:



Once the Administrator Log In successfully, they will be directed to the home page in the Administration panel:



Credentials are checked on the login page to check empty fields, and after the get request is done if they are wrong, the proper message is displayed to the user. Everything is explained in the code below:

```
nboard.tsx   App.jsx   PrivateRoute.jsx   Login.tsx ×   Home.jsx   index.
stration_Section > client > src > Components > Login.tsx > Login > handleEmail
import React, { useState, ChangeEvent, FormEvent } from 'react';
import './style.css';
import axios from 'axios';
import { useNavigate } from 'react-router-dom';

const Login: React.FC = () => {
  const [email, setEmail] = useState<string>("");
  const [password, setPassword] = useState<string>("");
  const [errEmail, setErrEmail] = useState<string>("");
  const [errPassword, setErrPassword] = useState<string>("");
  const navigate = useNavigate();

  const handleEmail = (e: ChangeEvent<HTMLInputElement>): void => {
    setEmail(e.target.value);
    setErrEmail("");
  };

  const handlePassword = (e: ChangeEvent<HTMLInputElement>): void => {
    setPassword(e.target.value);
    setErrPassword("");
  };

  const handleSignIn = async (e: FormEvent<HTMLFormElement>): Promise<void> => {
    e.preventDefault();

    if (!email) {
      setErrEmail("Enter your email");
      return;
    }

    if (!password) {
      setErrPassword("Enter your password");
      return;
    }
```

```
  try {
    const response = await axios.get(`http://localhost:8001/getAdmin?email=${email}&password=${password}`);
    if (response.data !== "Invalid") {
      localStorage.setItem("valid", "true");
      localStorage.setItem("adminId", response.data.Administrator_ID);
      localStorage.setItem("adminName", response.data.Administrator_fullname);
      navigate('/dashboard');
    } else {
      setErrPassword("Invalid email or password");
    }
  } catch (error) {
    console.error("Error:", error);
    setErrPassword("Invalid email or password");
  }
};
```

```
return (
  <div className='d-flex justify-content-center align-items-center vh-100 loginPage'>
    <div className='p-3 rounded w-25 border loginForm'>
      <h2>Login Page</h2>
      <form onSubmit={handleSignIn}>
        <div className='mb-3'>
          <label htmlFor="email"><strong>Email:</strong></label>
          <input
            type="email"
            name='email'
            autoComplete='off'
            placeholder='Enter Email'
            value={email}
            onChange={handleEmail}
            className='form-control rounded-0'
          />
          {errEmail && <div className='text-danger'>{errEmail}</div>}
        </div>
        <div className='mb-3'>
          <label htmlFor="password"><strong>Password:</strong></label>
          <input
            type="password"
            name='password'
            placeholder='Enter Password'
            value={password}
            onChange={handlePassword}
            className='form-control rounded-0'
          />
          {errPassword && <div className='text-danger'>{errPassword}</div>}
        </div>
        <button className='btn btn-success w-100 rounded-0 mb-2'>Log in</button>
      </form>
    </div>
  </div>
);
};

export default Login;
```

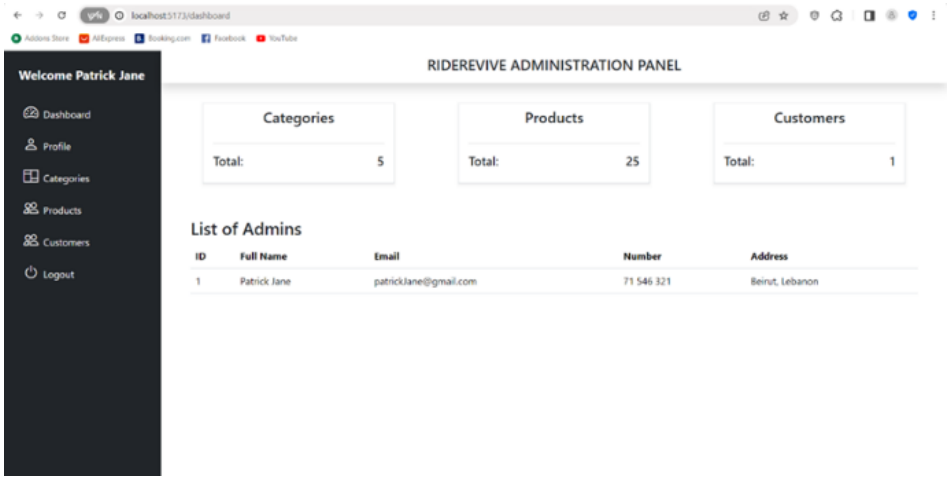
Passwords are hashed in the tables as it was mentioned before:

```
_id: ObjectId('6658afaf91519a0612a7e5e2')
Administrator_ID : 1
Administrator_fullname : "Patrick Jane"
Administrator_email : "patrickJane@gmail.com"
Administrator_password : "$2b$10$tnWGaEDR0MF8r0/J0XpzaeqNq/oZnSn6jAeAyeEwB5CCb3G/9GBSy"
Administrator_number : "71 546 321"
Administrator_address : "Beirut, Lebanon"
```

It is shown in the code, once the administrator logs in, the ID and the name retrieved from the request done to the server are stored in the local storage and valid is set to true. This is done to use the administrator ID for all operations that need the administrator ID like updating their profile. Once they log out the variables in local storage will be removed.

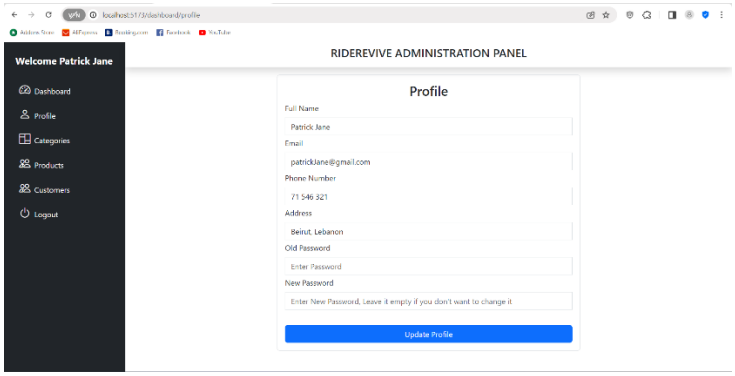
ii. Dashboard:

The dashboard file contains the constant sidebar and header in the panel, it displays the name of the administrator at the top of the sidebar all the time. Once the administrator is logged In, they are directed to the main page (Home page) which contains some number of products, categories, customers, list of admins. Here is part of the fetching code and a figure representing the page:

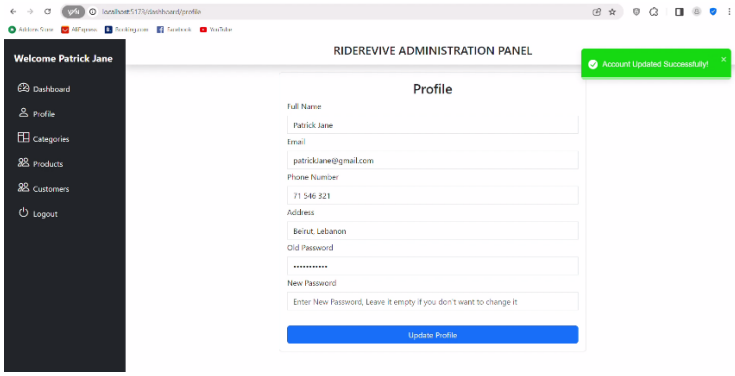


iii. Profile:

The Profile page is displayed in the figure below:



After updating the Profile (Entering the Password is always mandatory), a toast success message will be displayed in the top right of the page for the user:



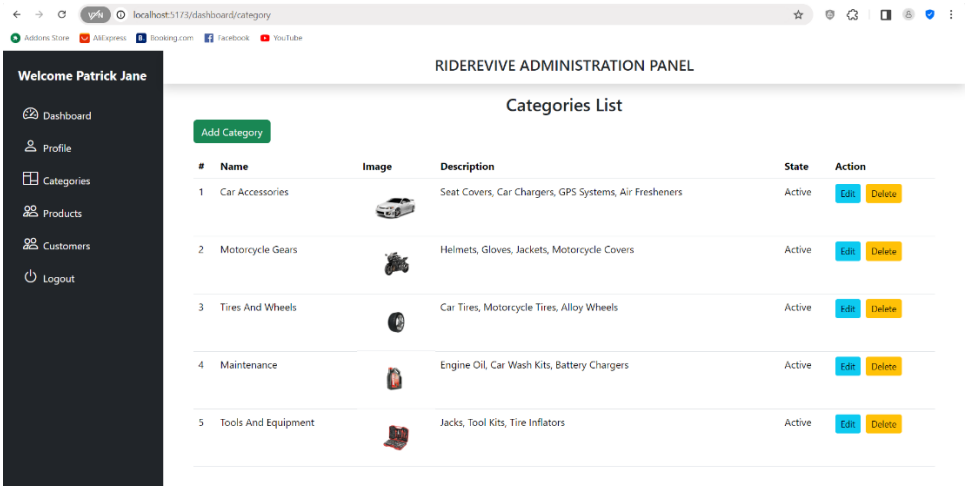
The API which is responsible for the update is represented below:

```
Dashboard.tsx   App.jsx   PrivateRoute.jsx   Login.tsx   Home.jsx   index.js   .env
Administration_Section > server > index.js > app.patch('/updateAdmin') callback

app.patch('/updateAdmin', async (req, res) => {
    const { Administrator_ID, Administrator_fullname, Administrator_email, Administrator_number, Administrator_address, oldPassword, newPassword } = req.body;

    try {
        const admin = await AdministratorModel.findOne({ Administrator_ID });

        if (!admin) {
            return res.status(404).json({ error: 'Administrator not found' });
        }
        const isMatch = bcrypt.compareSync(oldPassword, admin.Administrator_password);
        const emailChanged = await AdministratorModel.findOne({ Administrator_email, Administrator_ID });
        const existingAdminEmail = await AdministratorModel.findOne({ Administrator_email });


        if (!isMatch) {
            return res.json("IncorrectOldPassword");
        }
        else if (!emailChanged && existingAdminEmail) {
            return res.json("EmailExists");
        }
        else {
            if (newPassword.trim("") !== "") admin.Administrator_password = newPassword;
            if (Administrator_fullname) admin.Administrator_fullname = Administrator_fullname;
            if (Administrator_email) admin.Administrator_email = Administrator_email;
            if (Administrator_number) admin.Administrator_number = Administrator_number;
            if (Administrator_address) admin.Administrator_address = Administrator_address;
            const updatedAdmin = await admin.save();
            return res.json(updatedAdmin.Administrator_fullname);
        }
    } catch (error) {
        console.error('Error updating administrator details:', error);
        res.status(500).json({ error: 'Error updating administrator details' });
    }
});
```

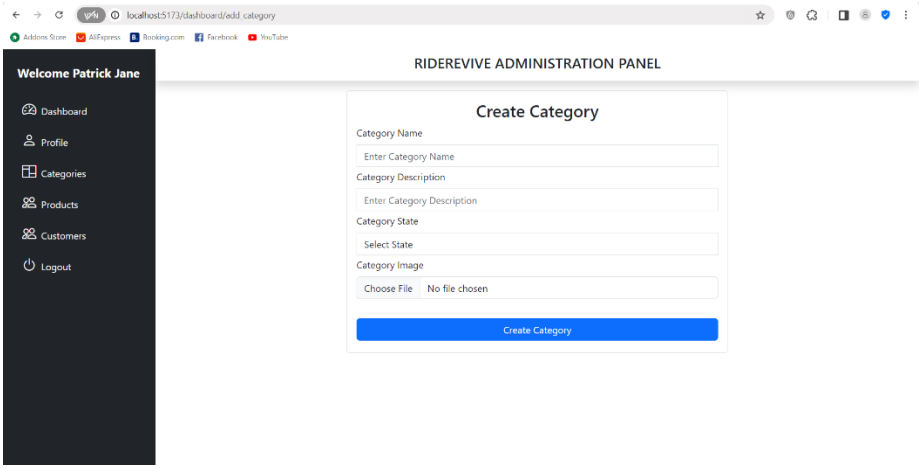
Certain measures are done before patching like checking empty values, correct email format, and password strength, but they are all done on the front-end side. If the email was changed, API as shown above checks if it already exists in the system in another user record, since it is a unique attribute.

iv. Categories:

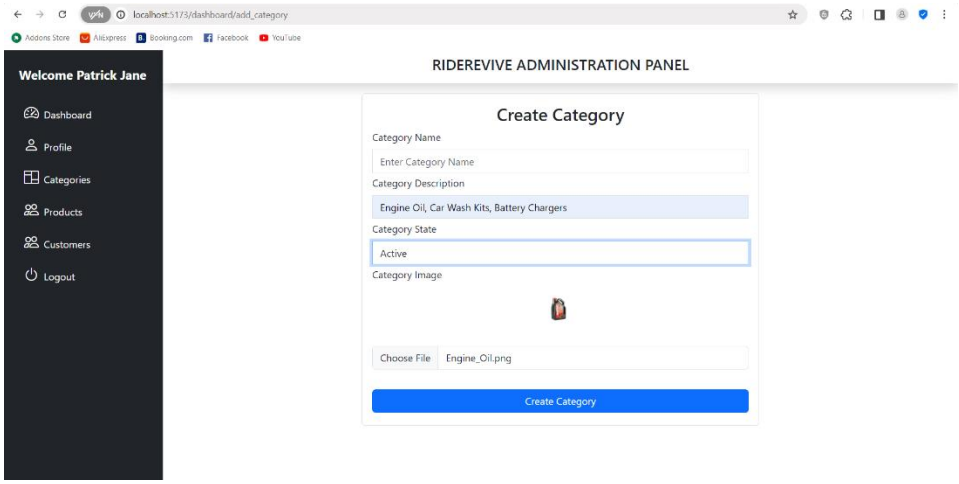
The Categories page is displayed in the figure below:



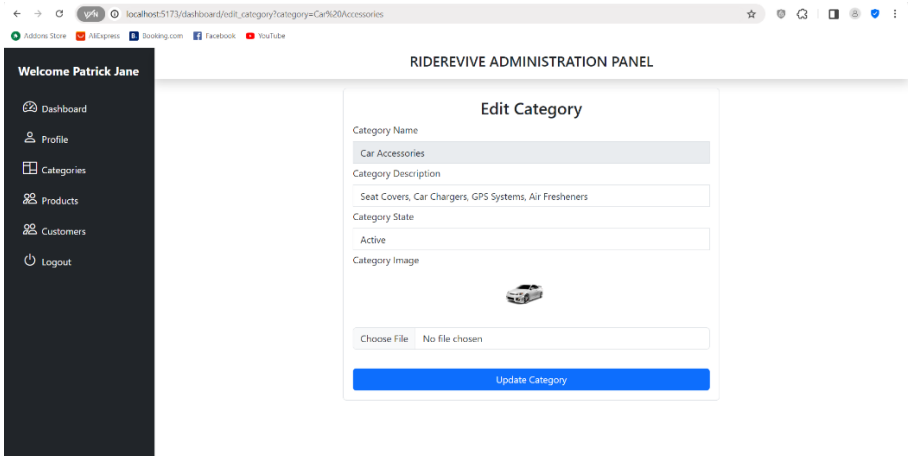
This page is only about fetching data from the Categories table and displaying it. The administrator on this page is also capable of deleting the category. The Add Category Page is displayed in the figure below:



This is the same page with all data fields filled:



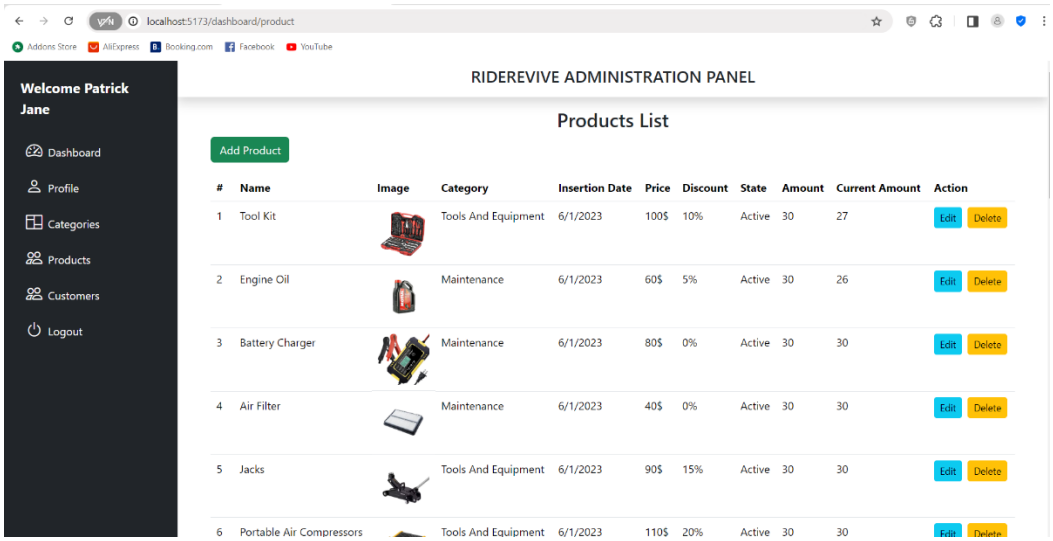
The image will be stored in the Images special directory in the project, to be retrieved when needed. The edit Category page is displayed in the figure below:



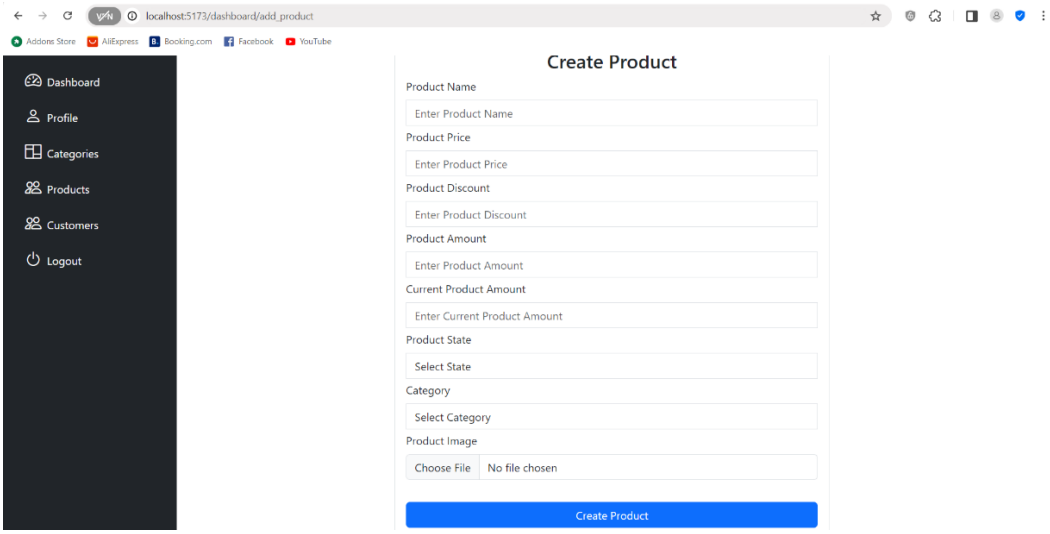
The category name field is disabled, since it should not be changable.

vi. Products:

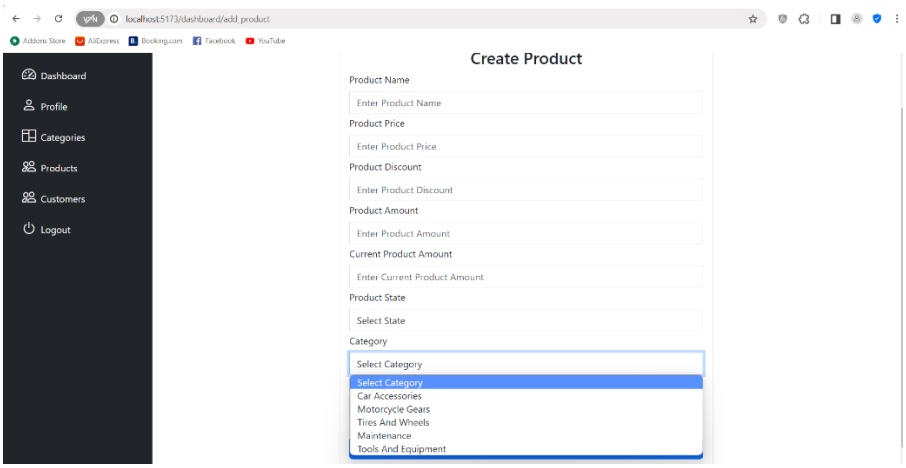
The Products page is displayed in the figure below:



The Add Product Page is displayed in the figure below:



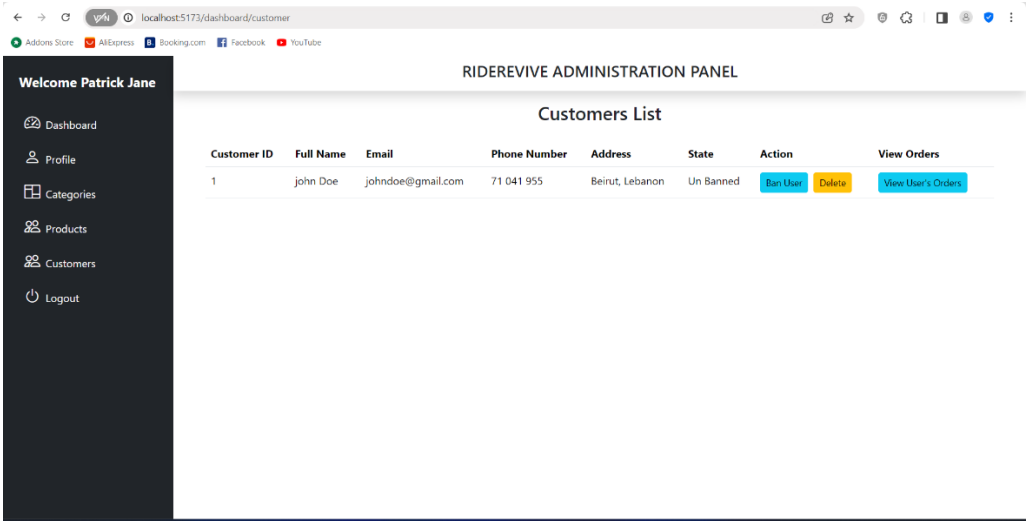
It is worth noting that the categories that can be selected to create a product should be selected from the categories which already exists in the system:



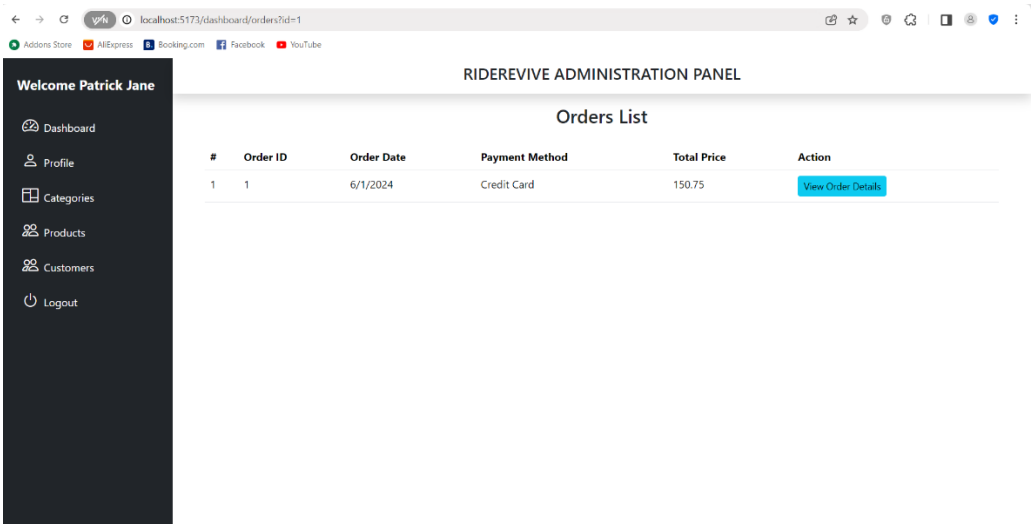
The Edit Product Page have the same structure of the Create Product Page, with the product ID being disabled all values being fetched. (same concept of edit Category).

vii. Customers:

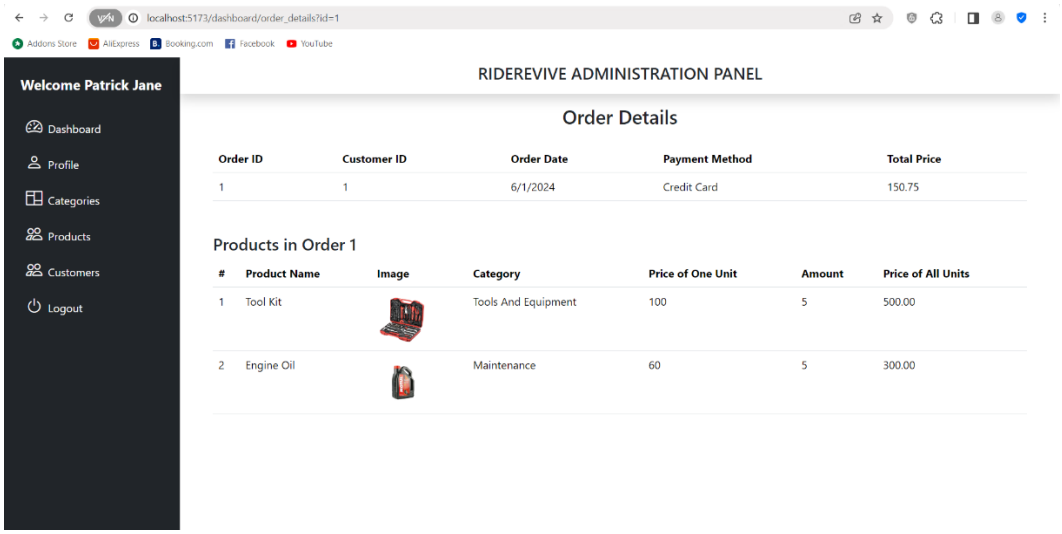
The Customers records are displayed in the figure below:



The Administrator is able to Bann the user. After the administrator ban them, he will not be able to access the public section. The View Orders is displayed in the figure below:



The order details page is displayed in the figure below:



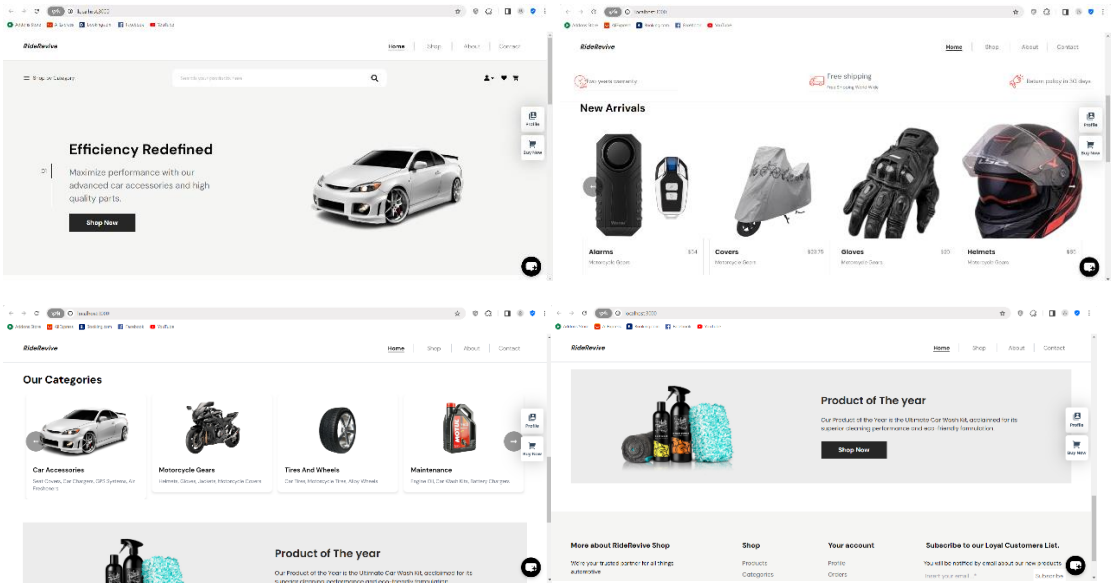
III. Public Section:

Notes:

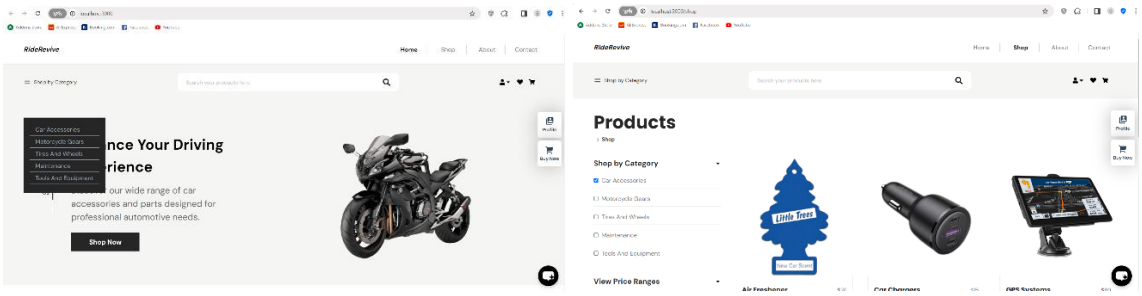
- Error handling for empty fields and wrong data types was implemented in all places where it was necessary on the client side.
- React-Toastify was used after almost all actions. The second figure in the profile part shows how it appears.
- Once data changes after any action on any page, the data is re-fetched after 1.5 seconds so that the user will directly see the new effect after the toast.
- TailWind.css is used in the development of the client side of the public section.

i. Home:

The Home page is displayed in the figures below:



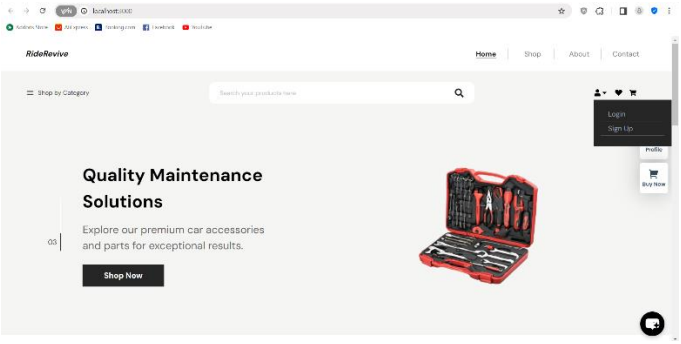
The header part, including the search bar and the icons is constant in all the website sections (except when the user is viewing their information). New arrivals products are products in the system which have their adding date greater than 1/6/2024. The date can be changed. Shop by Category functionality is displayed in the figure below:



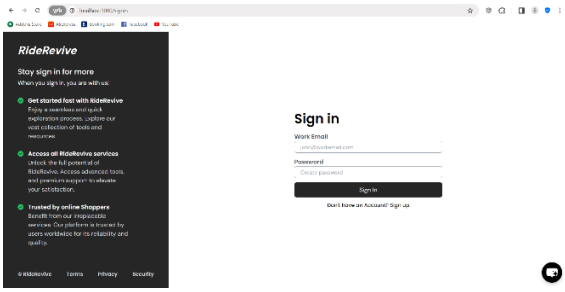
Once the user clicks on the category they want (in any page other than the shop page), they will be directed to the shop page and the search by category functionality will be turned on the category they clicked. Same thing happens once the user clicks on one of the categories displayed under the new arrivals in the home page.

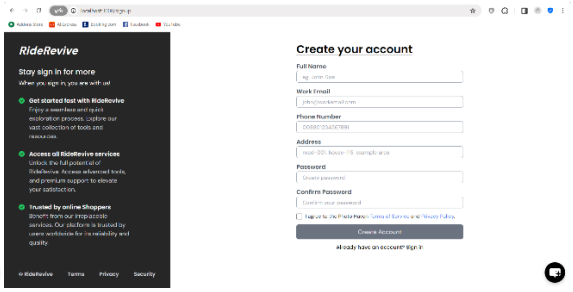
ii. Authentication & Account Management:

For Authentication, the user can either click the profile icon in the top bar, or click the one in the right side. Both are fixed in all pages.

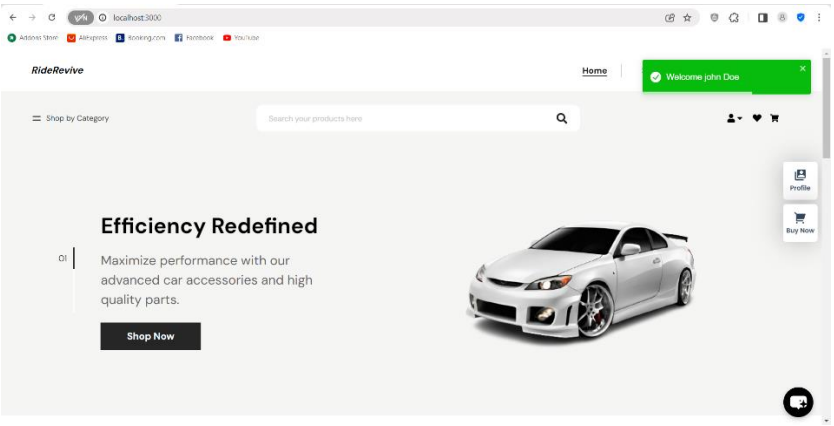


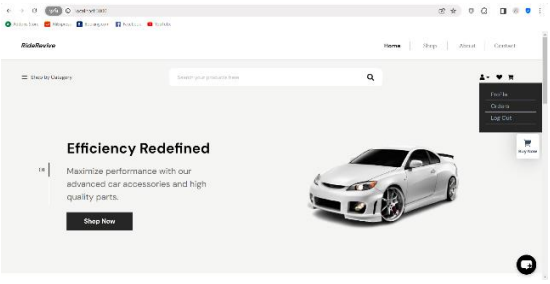
The Log In and Sign Up pages are displayed in the figures below:

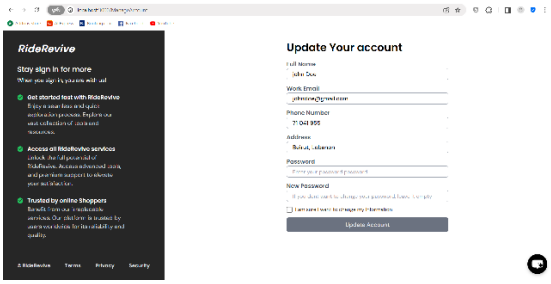




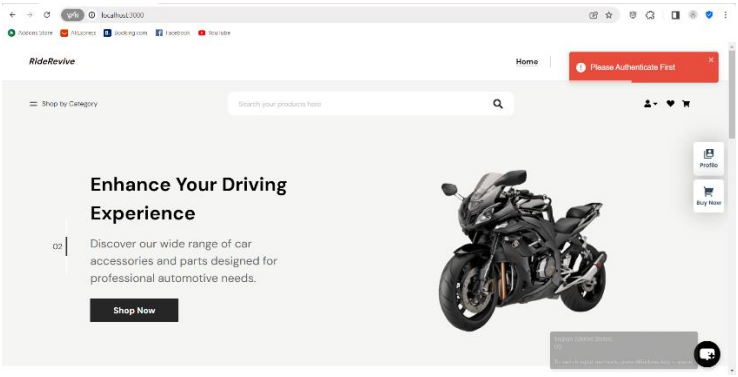
Once the user Authenticates Successfully, they will see a greetings success toast message with their name, and they will be able to access everything was inaccessible before authentication:



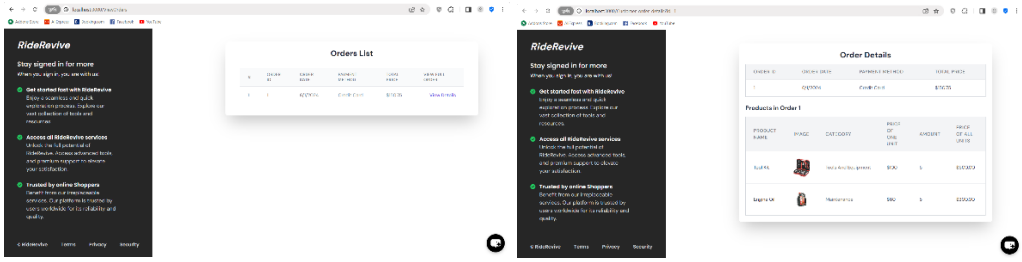




The above figures show what directly visually changes after authentication. The update profile criteria are same as the administrator. Before Authentication the user was not able to access the wish list, cart, add a product to cart or wish list. If they tried, they would receive the following toast error:

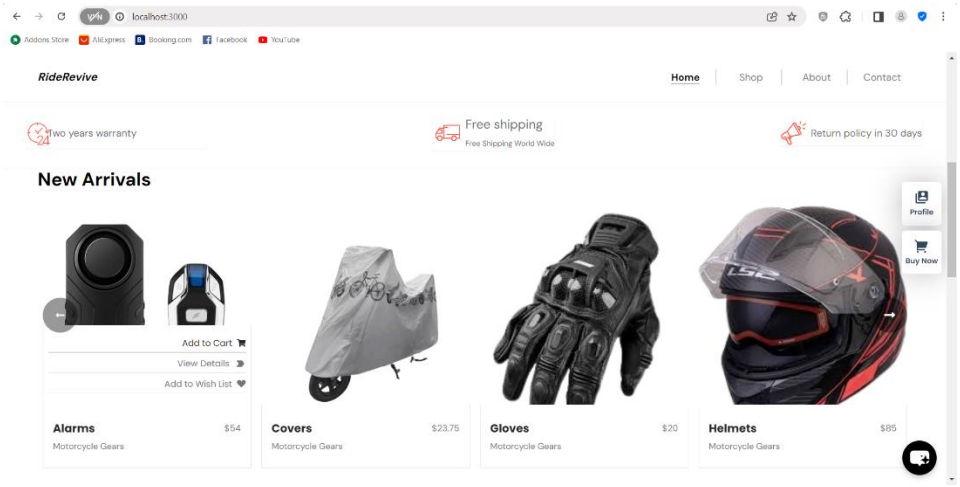


The user orders and orders details are displayed in the figures below:

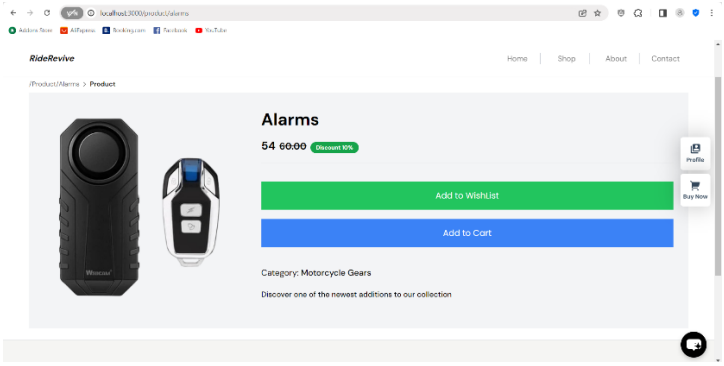


iii. Shop & Products:

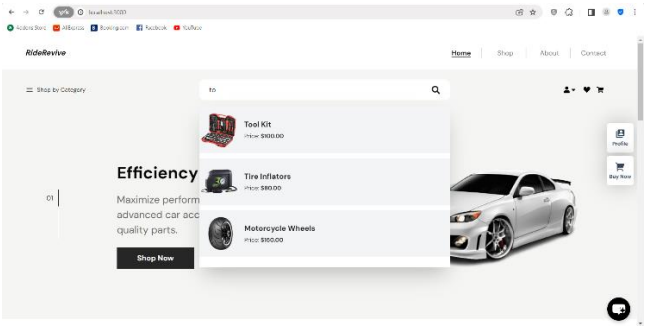
The user can also directly add to cart the products in the new arrivals in the Home Page:



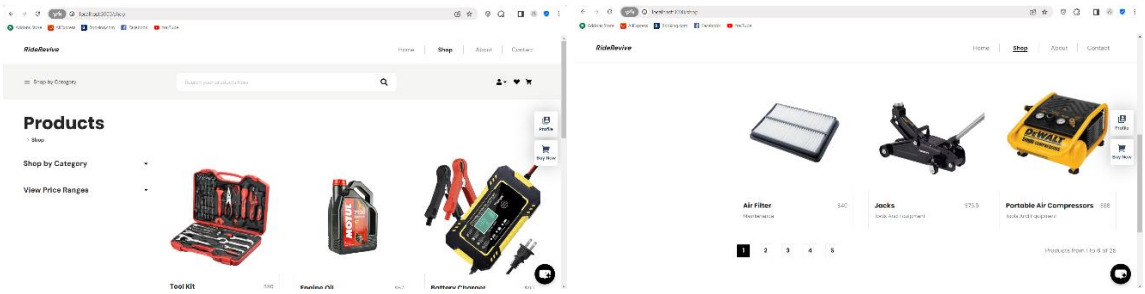
There is also a view details option. Unlike the two other options it do not require authentication. The view details option directs the user to another page where he can see the product details in a larger view:



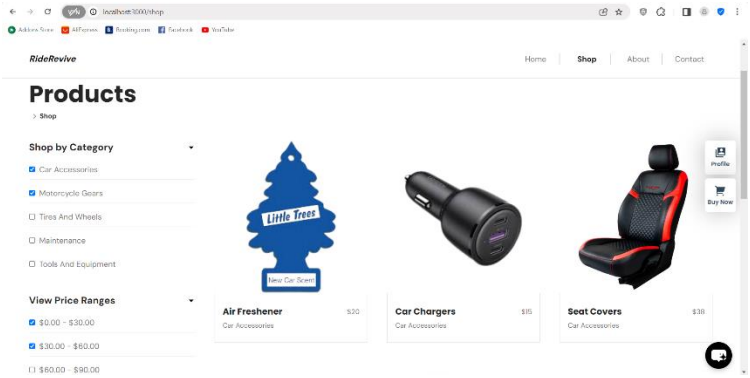
The search bar which is implemented in the header also serve directing to the view product details page:



The shop page is displayed in the figure below, where the products are represented similarly to how they are represented in the new arrivals:

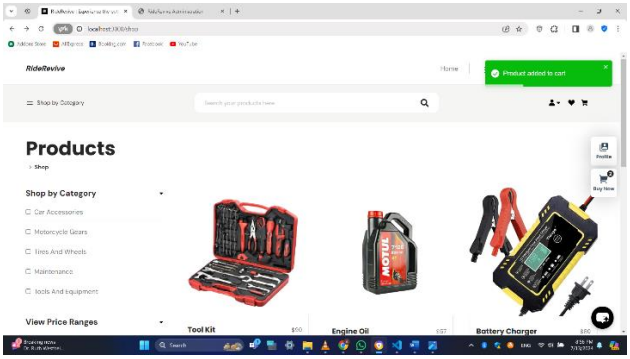


Search by price and category functionalities are provided, and they are shown in the figure below:

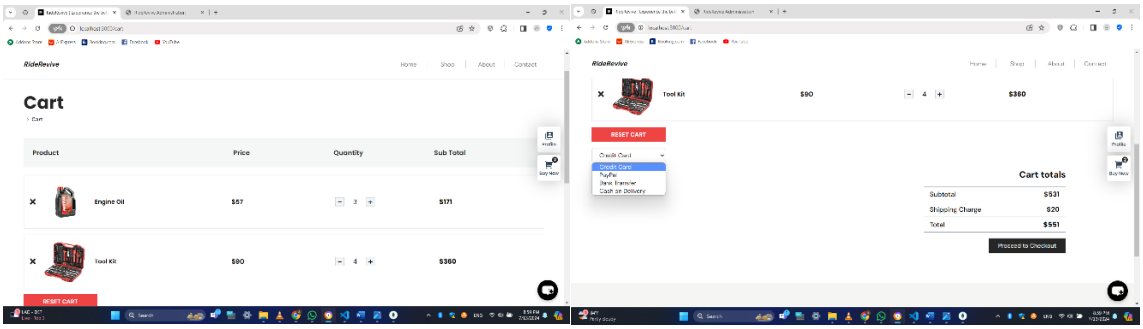


iv. Cart:

If the user is Authenticated, once they add a product to cart, the number of products on the cart changes on the cart in the side and they get a toast success message:



The cart page is displayed in the figure below:

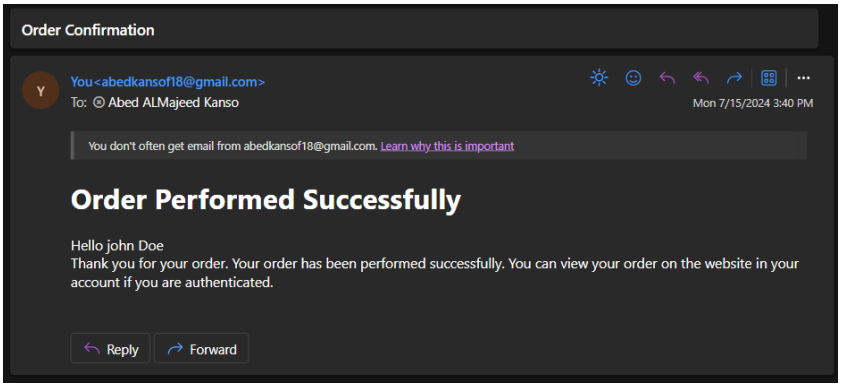


The user can perform the operations shown on the products in the cart. Once the order is proceeded the cart is rested. All the variables in the cart and the Customer ID are stored in a redux file shown in the figure below:

```
import { createSlice } from "@reduxjs/toolkit";
import { toast } from "react-toastify";

const initialState = {
  userInfo: null,
  products: [],
  checkedCategorys: [],
  checkedPriceRanges: [],
  isAuthenticated: -1,
};

export const orebiSlice = createSlice({
  name: "RideRevive",
  initialState,
  reducers: {
    addToCart: (state, action) => {
      const item = state.products.find(
        (item) => item._id === action.payload._id
      );
      if (item) {
        item.quantity += action.payload.quantity;
      } else {
        state.products.push(action.payload);
      }
      toast.success("Product added to cart");
    },
    increaseQuantity: (state, action) => {
      const item = state.products.find(
        (item) => item._id === action.payload._id
      );
      if (item) {
        if(item.quantity < item.amount){
          item.quantity++;
        //  toast.success(item.amount + " " +item.quantity);
        }
        else
          toast.error("No enough products");
      }
    },
```

```
export const orebiSlice = createSlice({
  reducers: {
    drecreaseQuantity: (state, action) => {
      const item = state.products.find(
        (item) => item._id === action.payload._id
      );
      if (item.quantity === 1) {
        item.quantity = 1;
      } else {
        item.quantity--;
      }
    },
    deleteItem: (state, action) => {
      state.products = state.products.filter(
        (item) => item._id !== action.payload
      );
      toast.error("Product removed from cart");
    },
    resetCart: (state) => {
      state.products = [];
    },
    togglePriceRange: (state, action) => {
      const range = action.payload;
      const isRangeChecked = state.checkedPriceRanges.some(
        (r) => r._id === range._id
      );

      if (isRangeChecked) {
        state.checkedPriceRanges = state.checkedPriceRanges.filter(
          (r) => r._id !== range._id
        );
      } else {
        state.checkedPriceRanges.push(range);
      }
    },
    toggleCategory: (state, action) => {
      const category = action.payload;
      const isCategoryChecked = state.checkedCategorys.some(
        (b) => b._id === category._id
      );

      if (isCategoryChecked) {
        state.checkedCategorys = state.checkedCategorys.filter(
          (b) => b._id !== category._id
        );
      } else {
        state.checkedCategorys.push(category);
      }
    },
    signIn: (state, action) => {
      state.userInfo = action.payload;
      state.isAuthenticated = action.payload.Customer_ID;
      toast.success(`Welcome ${action.payload.Customer_fullname}`);
    },
    signOut: (state) => {
      state.userInfo = null;
      state.isAuthenticated = -1;
      toast.success("Successfully signed out");
    },
    AccountUpdated: (state) => {
      toast.success("Successfully Account Updated");
    },
  },
});
```

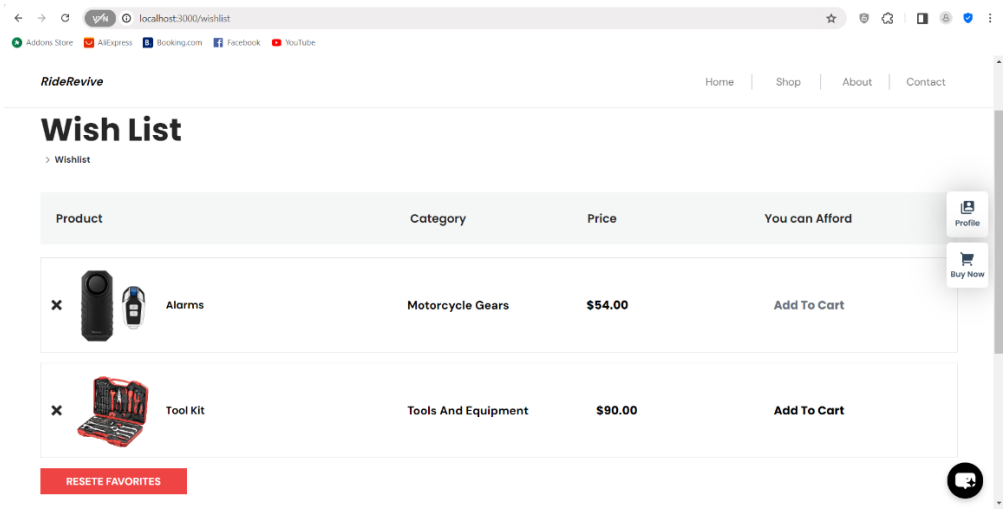
Finally, as it was mentioned before, once an order is performed, an email will be sent to the user using the NodeMailer (integrated in the server side):



v. Wish List:

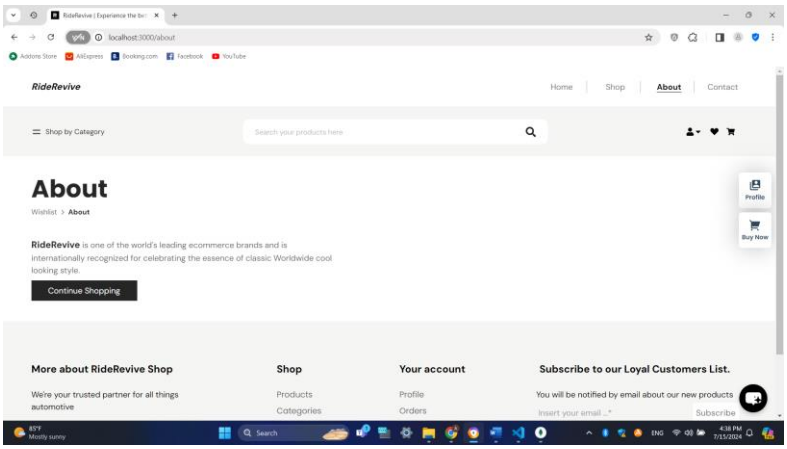
It has the same concept of the cart, but simpler. Once the user adds a product to the Wish List, they will see a toast message. The user is also capable of doing

certain operations in the items in the Wish List page. The Wish List page is in the below figure:



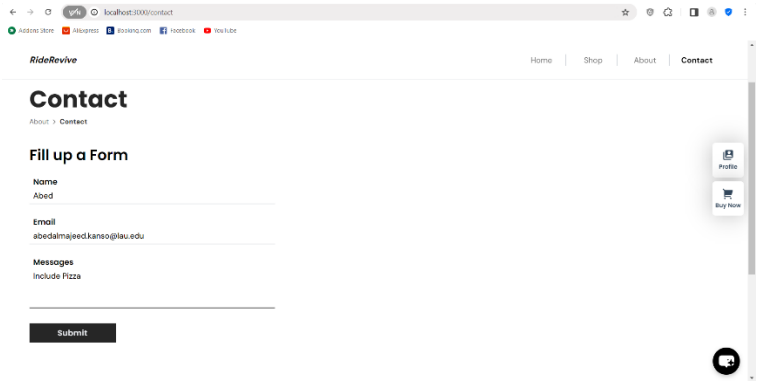
vi. About Us:

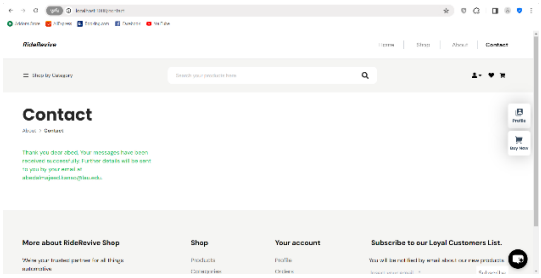
The About page only contains static content and the continue shopping button. It is displayed in the figure below:

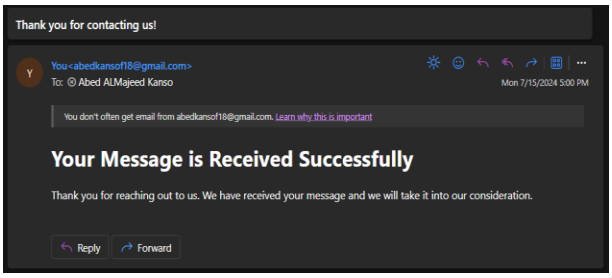


vii. Contact Us:

The Contact Page contains only a form of 3 fields. Once the message submitted, an email also will be sent to the email of the user using NodeMailer and the content of the email is stored in the Contact us table. The whole process is described in the figures below:

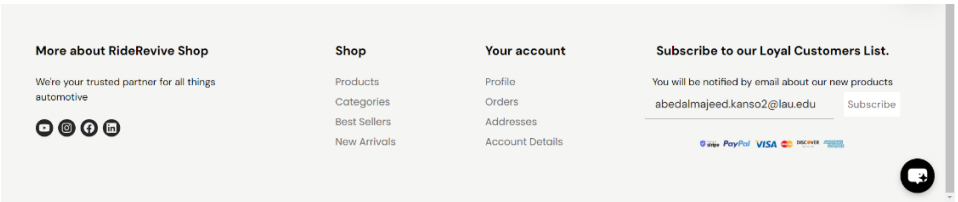


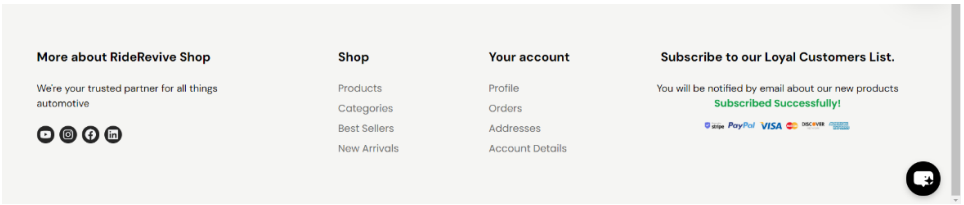




viii. Loyal Customers:

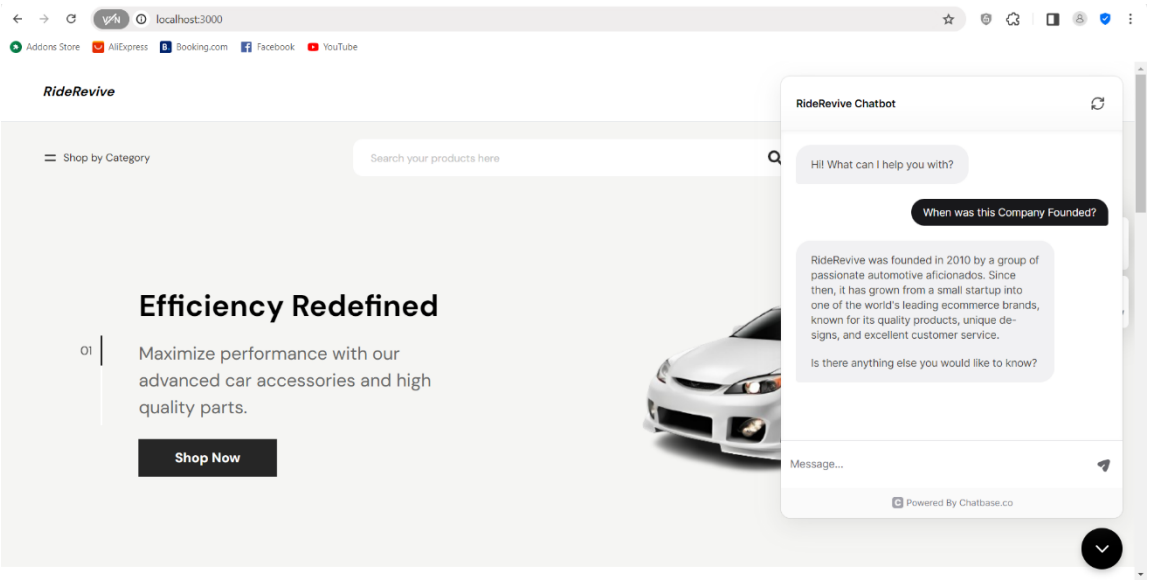
The Loyal Customers part is Embedded in the Footer. It requires customers to add their emails and once they submit it will be added to its table. It is purpose is for future enhancements where they will be informed by specific new products specified by the Administrators. Once the Customer add their email, an email will be sent to them using NodeMailer. (duplicate emails are checked in the API). The process is described in the figures below:





ix. Chatbot Integration:

The Chatbot Integration is done for many purposes. They include: product matching and recommendation, serving as a FAQ for some common user questions, and making the application more user friendly. The small black message icon in the right bottom of all the pages on the public section was the Chatbot icon. Here are some examples on how the interaction between it and the user works:



RideRevive Chatbot

How can I see my previous orders?

To view your previous orders on RideRevive, follow these steps:

1. **Log In to Your Account:**
   - Click on the "Log In" icon, located at the top right corner in the top bar, or click on the profile icon on the right side of the page.
   - Enter your email address and password to access your account.
2. **Navigate to Order History:**
   - Once logged in, click on your account icon, which is typically found at the top right corner of the page.
   - From the dropdown menu, select "Order History."
3. **View Your Orders:**
   - You will be directed to a page that lists all your previous orders.
   - Click on any order to see detailed information, including the items purchased and the price of each.

If you encounter any issues or need further assistance, you can contact RideRevive's customer service for help.

Have a great day! 😊

Message...

Powered By Chatbase.co

RideRevive Chatbot

If I bought 3 tool kits and 4 engine oil. What would be the total Price?

Sure, let's calculate that for you!

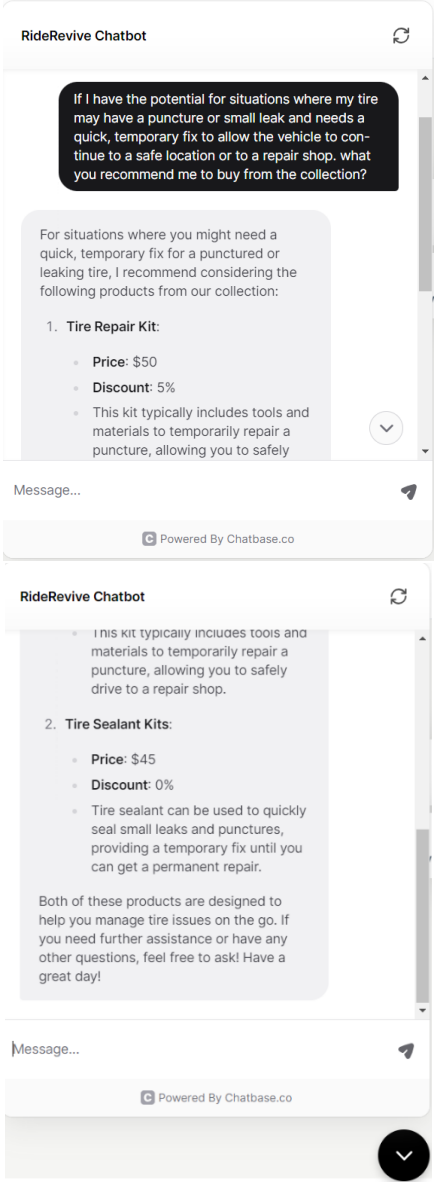
1. **Tool Kit**:
   - Price: $100 each
   - Discount: 10%
   - Discounted Price: $100 - ($100 * 0.10) = $90
   - Total for 3 Tool Kits: 3 * $90 = $270
2. **Engine Oil**:
   - Price: $60 each
   - Discount: 5%
   - Discounted Price: $60 - ($60 * 0.05) = $57
   - Total for 4 Engine Oils: 4 * $57 = $228

Total Price: $270 (Tool Kits) + $228 (Engine Oils) = $498

So, the total price for 3 Tool Kits and 4 Engine Oils would be $498. If you have any more questions or need further assistance, feel free to ask!

Message...

Powered By Chatbase.co



The code of for the chatbot integration is included in the header of the public section, it is shown in the figure below:

```
Header.js
Public_Section > client > src > components > home > Header > Header.js > Header > useEffect() callback
9   const Header = () => {
14    useEffect(() => {
26      window.embeddedChatbotConfig = {
27        domain: "www.chatbase.co",
28      };
29
30      const script = document.createElement("script");
31      script.src = "https://www.chatbase.co/embed.min.js";
32      script.async = true;
33      script.defer = true;
34      document.body.appendChild(script);
35
36      return () => {
37        document.body.removeChild(script);
38      };
39    }, []);
```

IV. Conclusion:

In conclusion, we are committed in providing an outstanding online shopping experience tailored to meet and exceed all your automotive needs. Our platform is designed to offer both variety and convenience, ensuring you find the best products for your car. As we continue to grow, we remain dedicated to expanding our web application's capabilities, introducing more innovative and enhanced features to better serve you. Discover the ultimate in quality and customer satisfaction with RideRevive, your premier destination for automotive excellence.