RMIT UNIVERSITY

RMIT International University Vietnam Assignment Cover Page

Course Code:	COSC2769
Course Name:	Full Stack Development
Location & Campus:	RMIT Vietnam - Saigon South Campus
Title of Assignment:	Group Project
Group Number	14
Group Members:	s3878141 – Nguyen Anh Duy s3938331 – Christina Yoo s3916884 – Junsik Kang s3726123 – SeongJoon Hong
Course Coordinator Name:	Mr. Tran Dang Tri (tri.dangtran@rmit.edu.vn)
Tutorial Group:	01
Assignment due date:	11/09/2023
Date of Submission:	11/09/2023

I declare that in submitting all work for this assessment I have read, understood, and agree to the content and expectations of the Assessment Declaration.

Table of Contents

1. Introduction

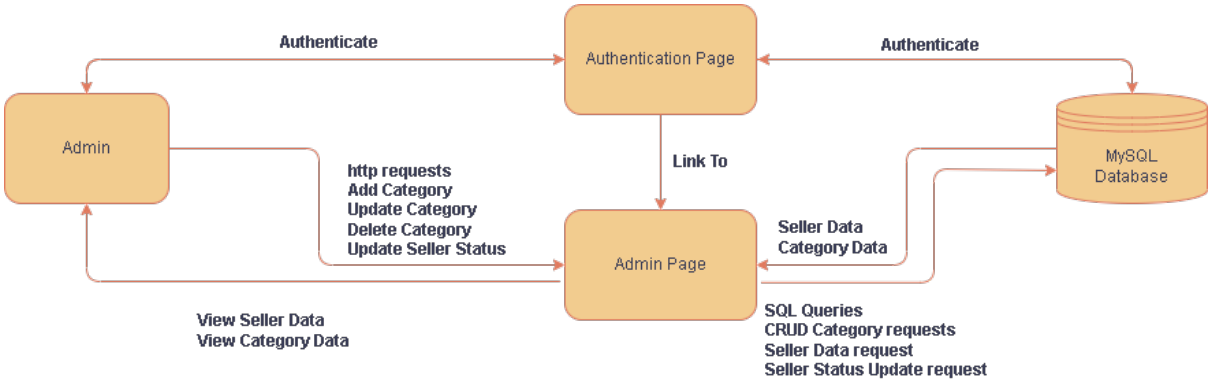
The web application we developed is an e-commerce system, inspired by Lazada, built by React, Node, Express, and MySQL. This application supports mainly three roles: Customers, Sellers, and Admin. Users can become registered-customers by signing up for our system. If the customer is not registered, there may be restrictions on functions such as not being able to place an order. Users on our application are divided into two types: customers who can select and place order the products, and sellers who can sell and manage their products. Customers can search for the products under various conditions and place them in an online shopping cart, then place an order. After that, if the order status changed to shipped by the seller, they can finally decide whether to accept or reject the products. Sellers can also sort and filter the products and decide whether to ship or cancel when the customers order their products. They can also check the statistics of their products as well. The administrator can manage the product category. When sellers apply for registration or anytime, the administrator has the authority to approve or reject it. Only approved sellers by the administrator can manage the products.

2. Design

2.1. Administrator Application Architecture

The following image depicts the application architecture on the administrator side.

- Authentication Page: The interface for authentication of the administrator.
- Admin Page: The interface for the administrator after the authentication which displays seller data, category data and allows access to management of the data. Seller status and category data can be modified.
- MySQL Database: Database storing administrator, seller and category data.



In the authentication process, a front-end for authentication and MySQL Database for retrieving data are needed. Authentication page is where the client mainly interacts in this process so it must be capable of getting client inputs for authentication, in this case, the inputs are email, phone number and the password. Once the web application receives the inputs, it will send a http request to the server and the server will make a query to the database. To make this process available, React js is the one that is used. For the front-end, the react hooks pattern is applied. React hooks is considered as a good approach because of its reusability and code efficiency. Functional components with react hooks need less code to perform compared to class based components. Moreover, unlike other variables in react,

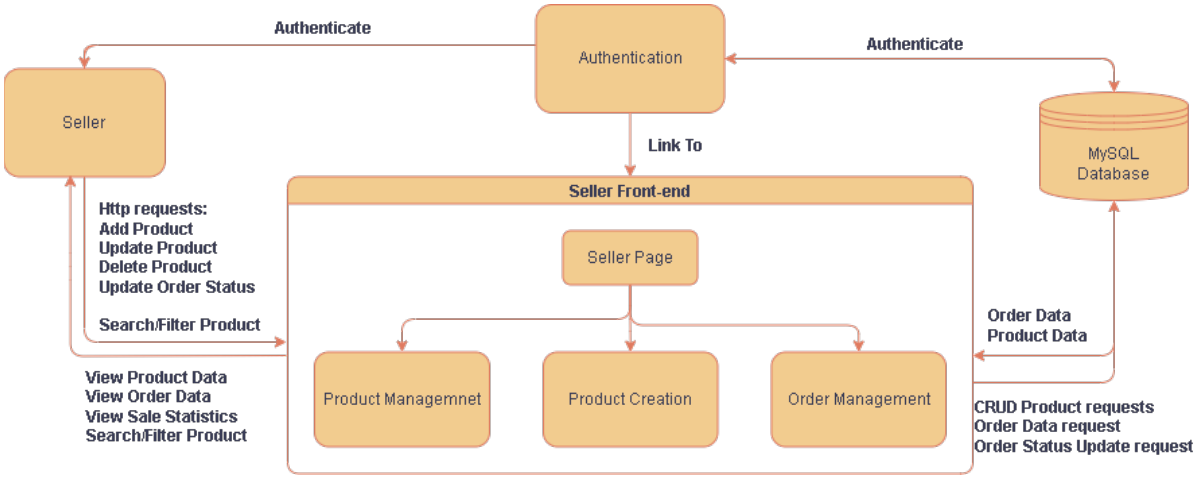
react hooks allows the client to keep track of data modification since it re-renders the component once the property changes.

On the back-end side, MySQL is chosen as a system to form a database for the web application due to its high performance although it needs data formatting to view the data to the client. Once the database gets a query, it sends a result back to the server and based on the result, the client gets access to the administrator page. To build a connection between the server and the database, the server side codes are built in a pattern of MVC. By dividing them into 3 parts, which are model, router and controller, it is much easier to manage the http requests and keep the codes cleaner. This format makes the code readable which leads to painless debugging. After the authentication, the client will be navigated to the administrator page. Administrator has access to the category data and seller data. The page offers CRUD operation of categories, attributes assigned to the categories and view and update the seller status data. According to the http requests the client sends, the server will send a certain query to the database and the database will return the result that will be displayed on the client side. Design patterns applied to both administrator and server are the same with the authentication, but one difference is that the administrator page provides loads of functions related to categories and seller management, it is divided into 2 different components and put into one higher-order component to make the codes maintained easily.

2.2. Seller Application Architecture

The image below describes the application architecture on the seller side.

- Authentication Page: The interface for authentication of the seller.
- Seller Page: The interface for the seller after authentication. Parent component of all pages related to the seller.
- Product Management: The interface for managing products that belong to the seller. All data of the products sold by the seller are accessible and can be modified.
- Product Creation: The interface for product creation. Products created in this page will be stored in the database.
- Order Management: The interface for the sale statistics and order list. Order status can be modified.
- MySQL Database: Database storing seller, product and order data.



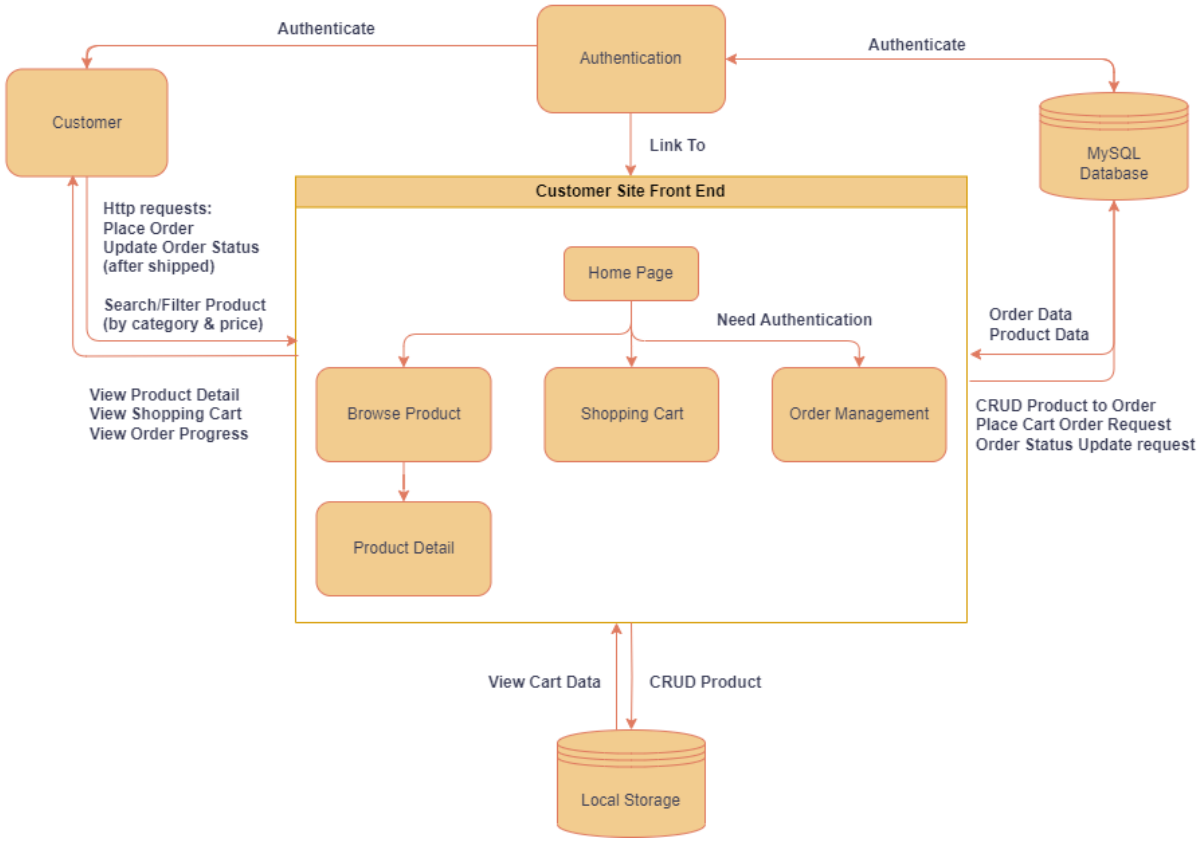
The front-end of the seller is composed of 3 different components. The client has access to the products, orders data and is allowed to modify the products and order status data that belongs to the client. The client must be approved by the administrator to navigate to the seller's front-end. Applying a conditional rendering design pattern, the unapproved client will not be able to view the seller pages.

In the page, one parent component, which is a Seller Page, holds all of the other 3 front-end components. By differentiating components based on its functions, makes the code neat and less complex. First of all, in product management, once it is loaded, the application will retrieve all of the product data that belongs to the client from the database. In the front-end, filtering and sorting by name, date and price can be done. Secondly, products can be created by the client. The data columns required to add a product into products data table are name, price, description, imagePath, category, attribute, quantity, dateAdded and the SellerID. Moreover, update and deletion is also available. Lastly, the client is allowed to manage orders. The front-end will make an api call to the server and the query for retrieving order data will be sent. During the data formatting in the front-end, the component will calculate the sales statistics and display it with the order data. As well as the administrator page, the front-end of the seller side followed react hooks design pattern, HOC design pattern and MVC architecture in the back-end which resulted in increased maintainability.

2.3 Customer Application Architecture

The image below describe the application architecture on the customer side

- Authentication Page: The interface for authentication of the customer (similar to seller and administrator).
- Home Page: The menu navigation page rendered when user firstly visiting the website.
- Browse Product: The page that display all the product information available in the database. User can search and filter products.
- Product Detail: The page store information about a single product when user select this item from the browse product page. User can add this product in the shopping cart
- Shopping Cart: The page store information about all the products user have added, including its quantity, and also the total price of all the item. User can place the order from this cart page
- Order Management: The page displaying the order that user have placed (only accessible through the order navigation link if user has log in). User can update the order status only if the product is shipped
- MySQL Database: Database storing customer, products, and order data (only after user has logged in)
- Local Storage: A temporary database in the client browser to store the shopping cart info



Regarding the customer site front-end, it has navigation to 3 different page components. The primary interface is the browse product page, contain all the product information that has been retrieved from the database. The browse product page is formed by the top Header component, the search & filter section, the list of all product cards, and finally the pagination component. Each product is generating a separate EachProduct component, which is a product card containing brief information of that product including the name, price, available quantity to purchase, and a button that navigates to its product detail page. User can filter the product by the maximum of 4 categories levels, as well as filtering the product by price range or date added range (minimum and maximum price, recent date added and oldest date added). For the product detail component, it contains all the properties of a product information, including the name, price, date added, and product description.

For the next page component - shopping cart, the cart itself contains a list of product items which is converted into a CartRow component that has been appended from the product detail page, which is also saved in the local storage and retrieved during the runtime of the web application. User can add as much product or update the quantity of the product as much as they wanted. User can also be able to place order, but only if they have log in to the website as a customer user only.

Lastly, for the order management page, user need to login to be able in accessing this page through a navigation link name “order” in the header. With similar layout like shopping cart, the order page also contains the list of product that has been placed in the cart, saved and retrieved from the database. However, the OrderRow component will show the shipping status, and the 2 buttons for user to accept or deny the order, only if the product has been shipped. User can see the blur and disabled of the buttons if the product has not been shipped yet.

3. Implementation

3.1. Administrator page

The main features of an administrator page are the CRUD of category data and update of a seller status. The administrator can view and modify the data retrieved from the database.

3.1.1. Category data management

```
const getCategories = () => {
    Axios.get('http://localhost:3001/admin/allCategories')
        .then((response) => {
            setCategories(response.data)
        })
        .catch(() => { console.log('ProductUpdate.js_getCategories: error') });
}
```

The code of an image above will run when the administrator navigates to the admin page after authentication. The code will make an API call on the browser and this call will retrieve the category data from the MySQL database. After it gets a response, by using the react state hooks, the state variable ‘categories’ will be set to the return value, which is a list of categories in this case. Once the state value changes, the component containing the state hooks will re-render so that the admin can view the data.

```
database.query('INSERT INTO categories (name, parentID) VALUES (?,?)', [name, parentID], (err, result) => {
    if (err) {
        return res.send({ err: err });
    } else {
        return res.send({ message: 'New category inserted successfully!' });
    }
});
```

To create a new category, the columns needed are the name of a new category and the id of its parent category since the category can have subcategories. The parent ID can be a null value if the new category is on the toppest level. An image above is a code of the back-end to add a new category into the database. After the admin sends an api call to add a new category, the server will send a query for checking if there are any duplicates in the database first. Then, if there is no duplicate, the code above will run. The update of categories works in a similar process. The only difference is that the update is changing the column value of an existing category in the database. Unlike other processes, some elements should be considered in deletion of categories. Since the categories can only be removed when they have no product categorized into them, at first, the code will check if there are any products related to the category. If it finds out the related product, the front-end will not show buttons for deletion. If not, the front-end will send a http request to the back-end and it will delete the category and also the subcategories. Once it is removed from the database, a front-end component will re-render to update the display.

3.1.2. Seller status update

```
const SellerID = req.body.SellerID;
const status = req.body.newStatus;
database.query('UPDATE sellers SET status = ? WHERE SellerID = ?', [status,SellerID], (err, result) => {
    if (err) {
        return res.send({ err: err });
    } else {
        return res.send({ message: 'Seller status updated successfully!' });
    }
});
```

Aside from category data management, the administrator can also handle seller status. In the front-end, a list of sellers registered will be displayed to the admin with their status. The admin can choose to approve or reject a seller. Once a decision is made by the admin, the front-end component will retrieve a chosen seller ID and status and send an api call to the server. Based on the body parameter, the code will run to update a status of a certain seller in the database accordingly.

3.2. Seller page

3.2.1. Product Management

3.2.1.1. Product CRUD

On the seller side, when the seller gets approved by the admin, the seller can get access to products registration and management. If a seller has products registered in the database, a list of the products will be displayed. In the products data table, there is a column for seller ID to store the seller who registered the product. Using this column, the back-end will get the products data with the ID of the current seller.

```
const formData = new FormData();
formData.append("name", product.name);
formData.append("price", product.price);
formData.append("description", product.description);
formData.append("image", product.image);
formData.append("category", JSON.stringify(product.category));
formData.append("quantity", product.quantity);
formData.append("dateAdded", product.dateAdded);
formData.append("SellerID", product.SellerID);
if (Object.keys(product.attribute).length === 0) {
    formData.append("attribute", null);
} else {
    formData.append("attribute", JSON.stringify(product.attribute));
}
```

Product registration is available in the seller side front-end. As shown in the image above, a lot of columns are needed to fill out the product data. Instead of putting all the properties to the body parameter of the request, the FormData interface enables wrapping all properties into a set of key/value pairs. This FormData object will be sent to the back-end and the addition to the database will be done. Next, deletion of products can also be done by the seller. Once the seller deletes a product, removing the link to the product image will be processed first. After the image is removed, the server will delete a product that has the same id as the one in the body parameter. Update process is the same with the one done in categories data management but only with different data properties.

3.2.1.2. Sort & Filter Products

```
function sortTable(accessor, order) {
    switch (accessor) {
        case "name":
            if (order === "desc") {
                const sortedProducts = products.sort((a, b) => a.name.localeCompare(b.name)).reverse();
                setProducts(sortedProducts);
            } else {
                const sortedProducts = products.sort((a, b) => a.name.localeCompare(b.name));
                setProducts(sortedProducts);
            };
            break;
```

```
const newProducts = [...allProducts];
const filteredProducts = newProducts.filter(product => {
    if (pName !== "") {
        const name = pName.toLowerCase();
        if (!product.name.toLowerCase().includes(name)) {
            return false;
        }
    }
    if (pDate.fromDate !== "" && pDate.toDate !== "") {
        if (!(new Date(pDate.fromDate) <= new Date(product.dateAdded.slice(0, 10)))
            || !(new Date(product.dateAdded.slice(0, 10)) <= new Date(pDate.toDate))) {
            return false;
        }
```

In the list of the front-end, the seller can filter and sort products by name, added date and price. The 2 images above are the code for the sorting and filtering.

First of all, accessor is a parameter for retrieving sorting criteria and order is for determining an order of the list. Using a switch statement, based on the accessor, the sorting process starts. New array with different order will be created and set to the state variable to re-render the component. Furthermore, the second image shows the code for filtering product lists. When the filter starts, a new constant 'newProducts' will be created and duplicate the current product lists. Then, if one of the input fields for name, price and added date is not empty, the filtering will be done. In the case of name, before filtering, all the names in the list will be modified to lowercase for precise comparison. After the modification, if the product name does not include a word typed in as an input the filtering function will return false and the product will be removed from the product lists, but if it does, the code will proceed to the next condition, which can be date of addition and price. In the case of date, each date state variable is formed as a date object for comparison. After all processes are finished, the remaining products will be set as a state variable for the product list and displayed to the seller.

3.2.2. Order Management

The web application provides permission to the modification of order status to the seller. In the order management page, the seller can view the list of order requests from the customers. These data are from the MySQL database. After the retrieval, data formatting to display the order lists is processed and categorized based on their status simultaneously to form a sales statistics. Then, the seller will be able to view the list and deal with the orders. The seller is only allowed to change a status to 'Shipped' or 'Canceled' from 'New'. Once the seller changes the status, api calls containing new status value, order ID and product ID will be sent to the server. The request will be processed on the server. At first, the server will search for an order with the same ID as the order ID of a body parameter. Then, the database will return a JSON object so to make it usable, it needs to be changed into an object. To do so, a parse function is used. Then, the code maps through the object to find an appropriate product with a requested ID and changes its status. Lastly, the update made will be applied to the database.

3.3. Customer page

3.3.1. Browse Products

This page shows all the products that are registered in the system. Customers can search for the products according to various criteria. Information for all products is obtained directly from the server database. There are three ways in which customers browse products.

3.3.1.1. Browse by categories/subcategories

```
const browseByCategory = (req, res) => {
    const category = req.body.category;

    database.query("SELECT * FROM products WHERE JSON_UNQUOTE(JSON_EXTRACT(category, '$[0]')) LIKE ? ||
        if (err) {
            return res.send(err);
        } else {
            return res.send(result);
        }
    });
}
```

```
|| JSON_UNQUOTE(JSON_EXTRACT(category, '$[1]')) LIKE ? || JSON_UNQUOTE(JSON_EXTRACT(category, '$[2]')) LIKE ? || JSON_UNQUOTE(JSON_EXTRACT(category, '$[3]')) LIKE ?"
```

Each product contains one main category and several subcategories. This allows users to browse products by categories. First, get the names of the categories from the ‘categories’ database which saves all the categories. If the user selects the main category, the secondary category imports only the subcategories directly connected to the main category. Below that, it works with an algorithm like this. Each time a user selects a category, the products including that category are displayed. So customers can ultimately choose the optional category to find the products they want.

3.3.1.2. Search by title/description & Filter by name/price/date added

```
const browseByTitle = (req, res) => {
    const value = req.body.value;

    const valueCondition = ("%" + value + "%");

    database.query("SELECT * FROM products WHERE name LIKE ? OR description LIKE ?", [valueCondition, valueCondition], (err,result) => {
        if (err) {
            return res.send(err);
        } else {
            return res.send(result);
        }
    });
}
```

```
const newProducts = [...allProducts];
const filteredProducts = newProducts.filter(product => {
    if (pName !== "") {
        const name = pName.toLowerCase();
        if (!(product.name.toLowerCase().includes(name) || product.description.toLowerCase().includes(name))) {
            return false;
        }
    }
    if (pDate.fromDate !== "" && pDate.toDate !== "") {
        if (!(new Date(pDate.fromDate) <= new Date(product.dateAdded.slice(0, 10)))
            || !(new Date(product.dateAdded.slice(0, 10)) <= new Date(pDate.toDate))) {
            return false;
        }
    }else if (pDate.fromDate !== "" || pDate.toDate !== "") {
        if (pDate.fromDate !== "") {
            if(!(new Date(pDate.fromDate) <= new Date(product.dateAdded.slice(0, 10)))){
                return false;
            }
        } else {
            if (!(new Date(product.dateAdded.slice(0, 10)) <= new Date(pDate.toDate))){
                return false;
            }
        }
    }
```

```
if (pPrice.fromPrice !== 0 && pPrice.toPrice !== 0) {
    if (!(pPrice.fromPrice <= product.price) || !(product.price <= pPrice.toPrice)){
        return false;
    }
} else if (pPrice.fromPrice !== 0 || pPrice.toPrice !== 0) {
    if (pPrice.fromPrice !== 0) {
        if (!(pPrice.fromPrice <= product.price)){
            return false
        }
    } else {
        if (!(pPrice.toPrice >= product.price)){
            return false;
        }
    }
}
return true;
```

The products have names, and include prices, date added and description. Customers can search for products with the prices in between by entering minimum and maximum prices, or by entering start and end dates to search for the products with dates added in between. This compares the value entered by the user with the database of the product, and returns only the matching products. Finally, it is also possible to search for a product that meets all of the above conditions by entering the name or description of the product customers want in the search bar. This filter mirrors the layout and functionality of the Admin one.

3.3.1.3. Pagination

```
const [thisPage, setThisPage] = useState(1);
const [productsPerPage, setproductsPerPage] = useState(4);
const [selection, setSelection] = useState([]);
const [selectionName, setSelectionName] = useState([]);

const lastPage = thisPage * productsPerPage;
const firstPage = lastPage - productsPerPage;
const currentProduct = products.slice(firstPage, lastPage);
```

Pagination which is a function that is essential to show a large amount of data. Save the number of products to be displayed on each page and the variables on the first and last page. The first page(thisPage) starts at 1, of course. Subsequently, ‘products’ variable which stored all the products in the database is sliced by the first page number divided by the last page number, and the products are displayed on the currently seeing page. When the customers click the page number they want to see on the pagination, it sets it to the page number and re-render other products to show them.

3.3.2. Product detail

This is the page that appears when a customer clicks on the product he wants in the product browsing page. Information on the product transmitted to each product detail page is received, and category information is also obtained separately. After that, the information contained in the product data is output to the appropriate location and shown to the customers.

3.3.2.1. Add to cart & remove on cart

```
function removeProduct() {
    if(userType !== "") { // member
        Axios.post('http://localhost:3001/shoppingCart/removeShoppingCart', {
            productID: product.ProductID,
            customerID: user.CustomerID
        }).then((response) => {
            console.log(response);
        });
    }
    if(localStorage.getItem(product.ProductID) != null) {
        localStorage.removeItem(product.ProductID);
    }
    alert("Successfully removed on shopping cart!");
}
```

```
function saveProduct() {
    if(userType !== "") { // member
        Axios.post('http://localhost:3001/shoppingCart/updateShoppingCart', {
            quantity: quantity,
            productID: product.ProductID,
            customerID: user.CustomerID
        }).then((response) => {
            console.log(response);
        });
    }
    if(localStorage.getItem(product.ProductID) == null) {
        localStorage.setItem(product.ProductID, quantity);
    } else {
        localStorage.setItem(product.ProductID, quantity);
    }
    alert("Successfully added on shopping cart!");
}
```

Customers can add it to the shopping cart or delete the added product from the product detail page. Only one product can be added or deleted from this page. Get the CustomerID and ProductID of the product and send it to the server to find the shopping cart of the logged in user. Then the product will be added or deleted. At the same time. addition or deletion is also performed on local storage.

3.3.3 CartPage (Place Order Feature)

```
const orderForm = () => {
    orders.current = [];
    if (userType === "Customer") {
        for (let pid of products.current) {
            const q = localStorage.getItem(pid);
            const order = [{ quantity: parseInt(q), ProductID: pid }, "New"];
            orders.current.push(order);
        }
        Axios.post('http://localhost:3001/shoppingCart/insertToOrder', {
            CustomerID: user.CustomerID,
            products: JSON.stringify(orders.current, replacer),
            price: totalPrice,
            date: `${new Date().getFullYear()}-${new Date().getMonth() + 1}-${new Date().getDate()}`
        })
            .then((response) => console.log(response))
            .catch(err => console.log(err));

        Axios.post('http://localhost:3001/shoppingCart/deleteShoppingCart', {
            CustomerID: user.CustomerID,
        }).then((response) => { })
        alert("Checkout success!")
        Axios.post('http://localhost:3001/shoppingCart/insertShoppingCart', {
            CustomerID: user.CustomerID,
        }).then((response) => { })
        localStorage.clear();
        window.location.reload();
    } else {
        alert("Please login first to checkout!")
    }
```

The website will firstly check if user has been authenticated as a customer user, in order to place order from the shopping cart. The order data will then be retrieved from the local storage, then each product will be then insert into the order object by its id, including all of its information (name, quantity, price, date), and the shopping cart info in the local storage will be reset to empty and prepare to receive another new order.

3.3.4 OrderRow (Accept and Deny product feature)

```
function reject() {
    Axios.post('http://localhost:3001/order/updateOrderStatus', {
        OrderID: OrderID,
        ProductID: ProductID,
        newStatus: "Rejected"
    }).then((response) => {});
    window.location.reload();
}

function accept() {
    Axios.post('http://localhost:3001/order/updateOrderStatus', {
        OrderID: OrderID,
        ProductID: ProductID,
        newStatus: "Accepted"
    }).then((response) => {});
    window.location.reload();
}
```

```
className={`d-block order-btn bg-danger mb-3 ${isShipped ? 'active' : 'blur'}`}
```

For the order information appear in the order page, each product row will contain a shipping status of the product. User can see the button is blur if the product status is not shipped yet, and the classlist will append the “active” class to return its opacity to 1 only when the shipped status is updated. After user have update the order status, there will be no more changed made to the order itself.

4. Testing

In general, the two methods were used the most; the values which are stored on the database were changed as necessary, and the result according to this change was output to the terminal of the Visual Studio for confirmation.

Importing values from a database and using it to needs requires very sensitive and precise coding. In particular, the JSON format from the database required many steps to import, so efforts such as parsing the data by changing the data to various data types were required. It took some time to check whether the data was imported properly through Visual Studio, so we directly entered a query in mySQL Workbench to check the result immediately. If the expected data is output through the query statement here, usually there is no problem using it as it is for backend coding.

After the data was imported, it was necessary to make sure we imported correct data, or that it was imported in the correct format. Even if we brought data with the right value, if it is the wrong format this could lead to problems like errors or poor results. In addition, syntax or methods that fit the data type were not available. Therefore, we checked the value and data type of the data in real time using the syntax ‘console.log’ which can be debugged while outputting the value of the imported data. Thanks to this, we were able to check what the result of the imported data according to our codes was, so we could roughly figure out how to fix the code.

Then we had to print them out in the right place to fit the page design. We also had to check the location of the components and design such as color and font at the page. The ‘run start’ which is included in the ‘npm’ command was used on the ‘server’ which contains all related to the backend and ‘client’ which contains all related to the frontend folders in our workspace respectively, then our work for coding was carried out with the preview displayed. So we could see if the data was printed to the

desired location and if the design was completed as expected. If it came out differently than expected, the result changed as soon as the coding changed so it was not pretty tough to solve the problem.

Finally after all the coding work, we tried using our e-commerce system as if we had become real users. We tested three roles in our system; the admin, the seller and the customer a bunch of times. As a result, we found some bugs we missed before and inconveniences for all users. We have improved the user experience while modifying through testing in this way.

5. Development processes

To ensure an efficient team working flow, our team follows a task-based model approach, which is similar to the agile model but not using the whole scrum system. Each phase (or week) will be divided into a number of tasks needed to be completed, and there will be 2 meetings occurs in that phase (the usual meeting dates will be every Monday and Thursday/Saturday). The first meeting will be to initiate the task for that week and updates on the last week (if there are still some pending task, the whole team will find suitable time to compensate in this phase too), and the second meeting will be a short catch up on the current progress to check if there is any miscalculation about the task workload or missing features so it can be reallocate to all the member. In each phase, the task will be divided into two groups - Front end tasks and Back end tasks. Duy and Seongjoon are responsible for the front end activities, whereas Christina and Junsik are responsible for the back end activities.

Below is an illustration of the task-based model that has been implemented from 14/08/2023 - 11/09/2023:

Phase	Duration	Task Description
1	14/08 - 20/08	General: Finding resources (UI design ideas, database references), analyse the main requirements of the application, setting up project folder hierarchy & repository. Front end: Brief layout for the application (find image references from similar e-commerce app and website) Backend: Setup express environment in server folder and react in client folder. Make connections to each other by Cors and origin.
2	21/08 - 27/08	Front end: Home Page (components: Header, Banner), Customer Site (Browse Product & Product Detail Page, components: EachProduct, BroseProduct, ProductDetail), Seller Page (Product Management, Sales Statistics & Order Management Pages, components: ProductManage, AddProduct, Statistics, OrderList, OrderDetail), set up routes for navigation to other pages

		Backend: Set up connection to MySQL database, create data for products, customers, sellers, and orders table, authentication functions, cookies and session setting, seller functions (CRUD products, order management), customer features (search and filter products by category and price), host database by Amazon RDS.
3	28/08 - 03/09	Front end: Shopping Cart Page (components: CartPage, CartRow), Order Management Page (for customers and sellers, components: OrderPage, OrderRow), Administrator Page (Category Management & Seller Management, components: Category, Seller) Backend: Shopping Cart features - Customer (CRUD product to cart [update product quantity], calculate total price, place order [save order data to database]), Set order placement status - Customer (to accept or deny), create categories table, admin functions: CRUD categories and seller status management.
4	04/09 - 11/09	Front end: Refactoring code for UI components Back end: Edit parts related to categories' attributes. Test and debug all functionalities. Enhance functions and organize codes. Change the database connection information for submission.

Regarding the functional tools, Microsoft Teams is the primary communication platform to hold weekly catch up meetings as well as outline all the pending tasks for that phase. Apart from Visual Studio Code as the main programming IDE, git and github is the source control system to help our team commit, review, merge and update code.

6. Conclusion

To summarize, this e-commerce web application is a web application developed by React, Node, Express, and MySQL database that allows customers to purchase products and sellers to sell products online. The customers can view the list of the products that are on sale, add any products they want to the shopping cart and then place an order. The sellers get access to the products and orders data from the database that belongs to them and manage changes of the data, once the sellers get approval from the administrator. These are the main features of the web application.

For further improvement in the future, the UI of the application should be well polished to make it more user-friendly. In the aspect of function, it may have more browsing filters to help customers narrow the scope of product searching and support payment systems like all other real-life e-commerce web applications.