RMIT University Vietnam
Assignment Cover Page

Subject Code:	COSC2769
Subject Name:	Full Stack Development
Title of Assignment:	Group Project
Student name:	Tran Quoc Trung Huynh Ky Thanh Ho Tran Minh Khoi Tran Minh Khoi
Student Number:	s3891724 s3884734 s3877653 s3916827
Teachers Name:	Tri Dang
Group Number:	4
Number of pages including this one:	13
Word Count:	2450

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

I. Introduction

We are developing an e-commerce website called “Lazada” which allows users to search for and purchase items, sellers can sell and manage products, and admins have permission to manage product categories and sellers' accounts. More details are that users and sellers can register and log in to their accounts on the website. While users can browse, filter or search products and have the functionality to add or remove products from carts, users also can place and manage orders. Sellers can manage the products and orders, and sellers can see the sales statistics on their products based on the orders’ status. The website will have Admins who have permission to manage product categories such as create, update or delete any category. Besides that Admins can manage products that sellers want to sell, and admins can accept or reject sellers depending on whether it is appropriate or not.

II. Design

A. High-level design

For a full stack web application, we need to consider both the backend and frontend. For the backend, we will need a database to store web and user data, and a server to serve that data to the clients in the front end. In the front end, we will need a framework to render the data to the user as well as communicate with the backend through API calls. The front end will perform these API calls using the URLs defined in the backend, each path will request the server to do different actions. When the backend receives a request, it will interact with the database to perform a CRUD action, if it is a GET request, the server is also responsible for returning the database result to the front end, where it will be displayed to the user.

In this project, we will use MongoDB as the database of our application, ExpressJS as the server, and React as the front end framework. Additionally, we will have some additional dependencies to help with the development process. All of these components will be run within the NodeJS environment.

B. Low-level design

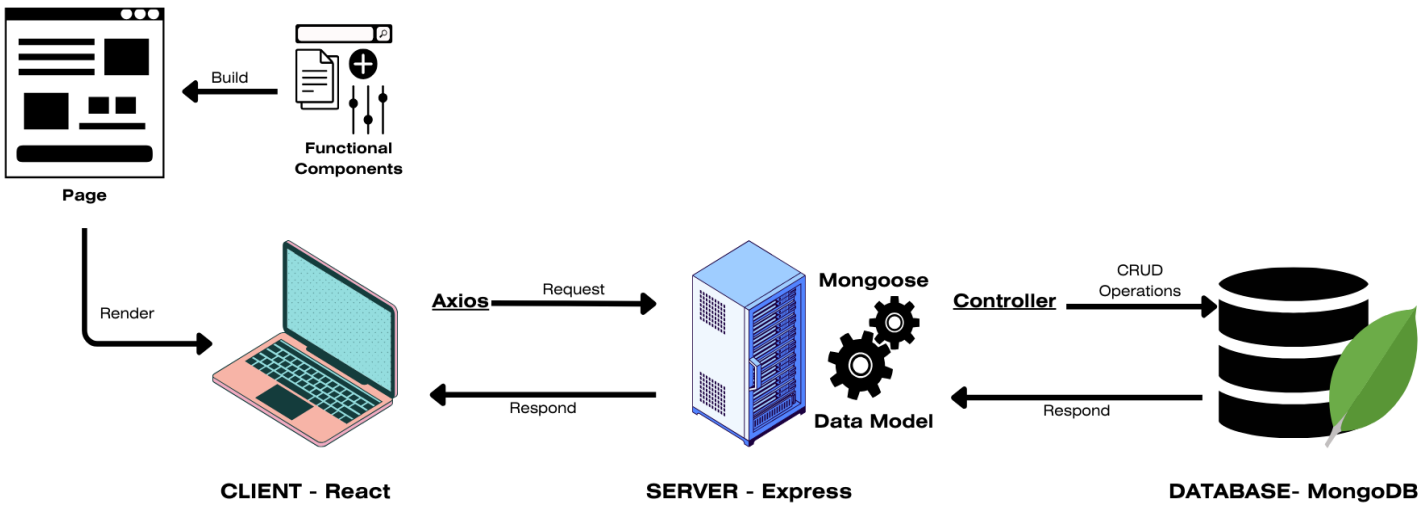

Figure 1. Low-level design diagram

Starting with the front end, we will use React and React Router DOM, to render different functional components to the user. Each main page will be built using smaller functional components instead of building them directly into the page, since this allows us to reuse those components for different pages later on. The components will also be divided for different user types, e.g. Admin, User, Seller. Additionally, to know which user is currently logged in, we also created an AuthContext that will wrap around the main page. By calling useContext, we can get the current user's information within any component. This will allow React to display the correct components to the current user type.

We use Bootstrap library to speed up the designing process of the web page. Bootstrap will provide us with pre-defined, beautiful css components that we can apply to our html documents quickly, with the drawback of having less customization compared to creating the css components ourselves. Additionally, to communicate with the back end, we will use Axios library. With Axios, we can pass in relevant configuration with the server's URL, and it will perform the corresponding HTTP request toward the server, and the response received from server will be converted to object automatically, unlike when using fetch, where an additional json() step is required.

For the back end, we first have to define the models that will be used in our application, this is done using Mongoose. With the models created, we can pass them to our controllers, where they can use these models to interact with the Mongo database that we created in MongoDB Cloud. We use mongoose instead of the usual mongodb library in our NodeJS environment because it makes interacting with the database similar to OOP by creating Schemas that can map to collections in Mongo database, which helps with the readability and querying. However, a drawback is that for larger queries, Mongoose may not be as performance friendly as the mongodb library.

To create the URL path for API requests, instead of defining them all in the index file, which makes applying multiple middlewares very complex, we decided to divide them using route. Each model will have a separate route that will interact with them. This increases the modularity of the project since we can edit routes later on without affecting other API URLs. Particularly, for using middlewares, we can apply certain middlewares to certain routes without it affecting other routes. For example, we can use CustomerAuth middleware for the Cart route so only customers can access cart APIs, but other routes will not be affected by this Auth middleware.

III. Implementation

1. Common implementation

- Axios is used in the frontend to send HTTP requests to the backend. We created an axiosSetting constant in frontend/Cotext/constants.js to set up a standard configuration for calling axios.

frontend > src > Context > JS constants.js > ...

```
1  import axios from 'axios';
2
3  export const backendUrl = 'http://localhost:2222'
4
5  export const axiosSetting = axios.create({
6      baseURL: `${backendUrl}/`,
7      withCredentials: true
8  })
```

Figure 2. Axios setting constant

- For product image, we used Multer middleware to handle uploading image file to store in the server

```
backend > routes > JS Product.js > ...
19   const upload = multer({dest: "./productImgs/"})
20   router.post("/add", upload.single("productImg"), ProductController.addProduct)
21   router.post("/edit/:productId", upload.single("productImg"),ProductController.editProduct)
```

Figure 3. Implementation of multer middleware

- We also used express.json() middleware to convert request body to JSON, cookie-parser middleware to parse cookie header and populate req.cookies with an object keyed by the cookie name and Cors to enable cross-origin resource sharing

```
backend > JS App.js > ...
10   const app = express();
11   app.use(express.json());
12   app.use(cookies(config.COOKIE_KEY));
13   app.use(cors({origin: 'http://localhost:3000', credentials: true}))
```

Figure 4. Usage of express.json(), cookie-parser and cors middleware

2. *Account Register/Login*

- Users can register with two types of accounts, which are Seller and Customer. The password of the user will be hashed and salted before saving into db.
- The customer's data will not consist of a “status” property and can log in and use all the features immediately after registration.
- After registering, the seller's status will be “Pending” by default, admins can either accept or reject the register of the seller. When admins decide to accept or reject the status will be changed to “Approved” or “Rejected”. Only approved sellers can manage their product and this status also can be updated by the admin. The checking of seller status is implemented on both front end (check status on the product page and display a message if the status is not Approved) and back end (implementation of a middleware called SellerAuth)

3. *Authentication*

- After the user login, the backend will use JSON Web Token (JWT) signed with some information of the user and send it back to the client using the signed cookie. Each token will be valid for two hours.

backend > controllers > JS AccountController.js > AccountController > login

```
42          const token = jwt.sign(
43              { userId: user._id, email: user.email, type: user.type, sellerStatus: user.sellerStatus },
44              process.env.TOKEN_KEY,
45              {expiresIn: "2h"}
46          );
47          let data = {};
48          data._id = user._id
49          data.fullName = user.fullName;
50          data.email = user.email;
51          data.type = user.type;
52          data.token = token;
53          data.sellerStatus = user?.sellerStatus;
54
55          // save the token to cookie that send back in response
56          res.cookie('token', token, { httpOnly: true, signed: true });
```

Figure 5. Sign token and assign to signed cookie

- In the Front-end there will be an authentication context to save the authenticated state and check the token every time the website reloads or users log in by calling API (file: frontend/src/Context/LoginSessionContext.js). By using the authenticated state from the context, every time the user accesses a URL of the web, there will be a method placed in the navbar component, which is included on every page, to validate the user and their permission to access that URL. If they accidentally access or access on-purpose to URLs that they are not allowed to, they will be navigated out.

frontend > src > Service > JS CommonService.js > ...

```
export const handleAuth = (isAuthenticated, userTypeUpper) => {
    const guestPath = ["cart", "product"]
    const userPath = ["order"]
    const sellerPath = ["seller"]
    const adminPath = ["admin"]
    const path = window.location.pathname.split("/")[1]

    if ((path === "login" || path === "signup")) {
        if (isAuthenticated) {
            return false
        } else return true
    }
    if (path === "logout") {
        if (!isAuthenticated) {
            return false
        } else return true
    }
    if (sellerPath.some(p => path.includes(p))) {
        if (userTypeUpper === SELLER) {
            return true
        } return false
    }
    else if (adminPath.some(p => path.includes(p))) {
        if (userTypeUpper === ADMIN) {
            return true
        } return false
    }
    else if (userPath.some(p => path.includes(p))) {
        if (userTypeUpper === CUSTOMER) {
            return true
        } return false
    }
    else if (guestPath.some(p => path.includes(p)) || path === "") {
        if (userTypeUpper === ADMIN || userTypeUpper === SELLER) {
            return false
        }
        return true
    }
}
```

Figure 6. Function handleAuth called in navbar

- In the backend, to check whether token validation to any API needs to be authorised or not there will be a middleware called VerifyToken. Also, there will be a few APIs that need some specific middleware to check user type and status including AdminAuth, SellerAuth and UserAuth (files: backend/middleware).

4. Products

- For the seller, there are CRUD APIs and UI for them to manage their products. In the product management page of sellers, there are sorting by name, price and date added. Sellers can filter their products based on these criteria. These sorting and filtering are implemented in the front end, which can ensure smoothness and short execution time

and does not depend on the internet connection. Each filtering method will run one after another before the filtered array is displayed to the user.

- For customers, the product list page has pagination, name-description searching and filtering by category, price, date added and attributes. Because of the large amount of products in the web application, these features are implemented in the backend query which will ensure efficiency and smoothness despite the limitation of user devices and browsers. Attribute filtering is only available when searching or category filtering is active because all of the products on the website attribute value filter amount will be very large (getAllProducts function in backend/controllers/ProductController.js).

5. *Category*

- Category is structured with a property "parentCategoryId" to stored subcategories parent category id.
- Admins have permission to create a new root category or create a subcategory of a category. When adding a new category with existing name or existing attribute name in parent categories, backend will check and respond with an error, then frontend will show an alert.
- Whenever the seller creates a new product belonging to a category that product will have the input to enter every attribute of that category and category parents by calling a function to get attributes of all parent categories by parentCategoryId of that category continuously until null.

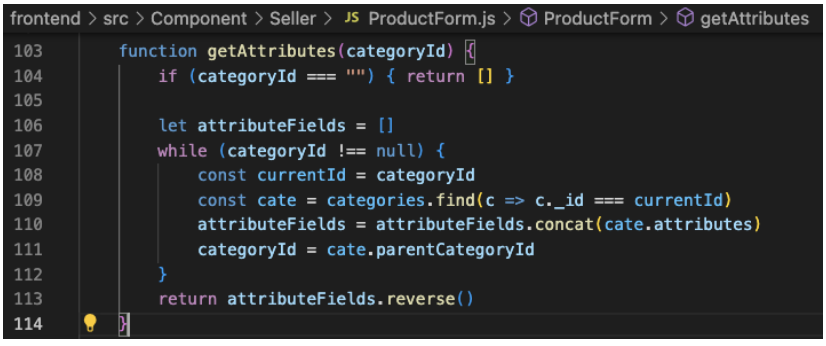

Figure 7. Function to get all attribute of a category

- When users filter products by category, the controller will call a function to get all the children categoryId by having a list of category id, query to get category's id that has parent category id equal to the chosen category id, then add the result to the list. Then continuously get subcategories of categories in the list, then add results to the list and remove itself from the list until the list is empty.

Figure 8. Function to get all children category Id of a category

- In order for the admins to edit or delete a category, that category must not have any sub-category or any products belonging to that category. For checking the two conditions mentioned previously, there is a function to check and then will mark it as a property called "isUpdateable" to the category. This function will be called in the get category API in order to disable edit and delete interaction of category in frontend.

Figure 9. Function to check the updatability of a category

6. Cart

- If the user is not authenticated, every product in the cart will be stored in Local Storage which will be stored as value. Because of this, there is a limitation where if a seller deletes their product, the local storage cart would still display it; however, it is not a significant limitation since users will need to log in to create an order, at which the cart is validated.

Figure 10. Function load cart list

- When the user is authenticated, there will be an API to get the cart data from the database and display all the products in the cart. Every cart product from the database is referenced. So if the product got removed by sellers, it also will be removed from the cart.

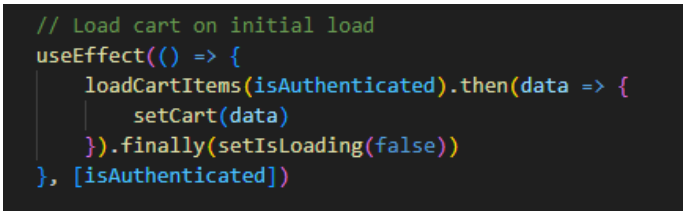

Figure 11. Call API load cart items

- Whenever the cart is updated such as added or deleted or increased quantity, an API will be called through the function “updateCart” to update the whole cart for that user in the database.

```
// Save cart if cart state changes
useEffect(() => {
    if (firstRender.current) {
        firstRender.current = false
        return
    }
    if (isAuthenticated) {
        updateCart(cart).then(res => {
            if (!res) {
                alert("Error updating cart")
            } else {
                localStorage.setItem("cart", JSON.stringify(cart))
            }
        })
    } else {
        localStorage.setItem("cart", JSON.stringify(cart))
    }

}, [cart, isAuthenticated])
```

Figure 12. Call API update cart

7. *Orders*

- Whenever customers place an order, it will add the order to the database.
- The seller can view and edit the status of the product in the order by choosing “Shipped” or “Canceled”. Not only the sellers can edit, but customers also have permission to edit the products in the orders whether “Accept” or “Reject” after the seller already marked the product as “Shipped”.

IV. Testing

During the development, each developed feature will be tested, then a demonstration is given to group members in weekly meetings. Therefore every member can understand the business logic and have their own perspective to use and test the functionality. This activity gives us a higher ability to detect a bug compared to when there is only one person testing with only his own perspective.

For the backend testing, we ran all the app routers in Thunder Client or Postman to test for functionality, database connections, dataflow, function effectiveness and detect bugs.

We ran the whole web application to test for performance, and we created an account for every role: admin, seller, and customer. Then we test each role's functionality to see if the frontend and backend work well with each other. Lastly, we check for integration between each role and each feature.

For usability and user experience, we invited five people outside of our group, gave them some goals and asked them to use the web as seller, customer and admin to achieve the goals. By collecting their behaviour while using the web application and then asking them for their feedback, we prioritise selected the most common constructive feedback to enhance our UI, UX.

V. Development Process

- At first, our team divided each part equally for each member of the team. Khoi Tran and Trung worked on Customer and Guest functionality together because this part required lots of function and design. While Khoi Ho worked on sellers and Thanh Huynh worked on Admins.
- We decided to use GitHub to control source code. Each member of the team created a different branch to work on. After finishing each feature, the member can merge the code from the main to their branch first to resolve the conflict then merge back to the main. This will prevent conflict, error or losing code during the development process for every member of the team.
- At the end of the week, our team members will have a meeting called to evaluate and review everyone's codes to see if there is any design or function that can be improved. After that, our team would give a demonstration about features that had been developed in that week with one member to share the screen at a time, so every member could watch and discuss those features.