

**Course:** Web Application Development

**Course ID:** IT093IU



**INTERNATIONAL UNIVERSITY -  
VIETNAM NATIONAL UNIVERSITY**

**SCHOOL OF COMPUTER SCIENCE AND  
ENGINEERING**

**Clothing E-Commerce Website Project**

**Group member:**

| <b>NAME</b>     | <b>STUDENT ID</b> |
|-----------------|-------------------|
| Vũ Nhật Duy     | ITITIUI7047       |
| Hoàng Minh      | ITITIUI9029       |
| Nguyễn Gia Phúc | ITITIUI9041       |

**Instructors:** Dr. Nguyen Van Sinh

**Academic Year:** 5/2023

# Table of Contents

|   |           |
|---|-----------|
| <b>Clothing E-Commerce Website Project</b>      | <b>0</b>  |
| <b>Table of Contents</b>                        | <b>1</b>  |
| <b>ABSTRACT</b>                                 | <b>3</b>  |
| <b>Chapter 1: Introduction to the Project</b>   | <b>4</b>  |
| 1.1. Overview of the Project                    | 4         |
| 1.2. Purpose and Goals                          | 4         |
| 1.2.1. Main Goals of the Project                | 4         |
| 1.2.2. Target Audience                          | 4         |
| <b>Chapter 2: Details and Technologies</b>      | <b>5</b>  |
| 2.1. Details                                    | 5         |
| 2.1.1. Stakeholders                             | 5         |
| 2.2. Technologies                               | 5         |
| 2.2.1. Front-end Technologies                   | 5         |
| 2.2.2. Back-end Technologies                    | 6         |
| <b>Chapter 3: Project Planning</b>              | <b>8</b>  |
| 3.1. Starting the Project (10/4-17/4)           | 8         |
| 3.2. Analyzing the Project (17/4-24/4)          | 8         |
| 3.3. Implementation (24/4-10/5)                 | 8         |
| 3.4. Testing (10/5-End)                         | 9         |
| 3.5. Other tasks (10/5-End)                     | 9         |
| <b>Chapter 4: Analysis</b>                      | <b>9</b>  |
| 4.1. Functional and Non-functional requirements | 9         |
| 4.2. Non-functional requirements                | 10        |
| <b>Chapter 5: System Design</b>                 | <b>11</b> |
| 5.1. Use Case Diagram                           | 11        |
| 5.1.1. Use Case: Customers                      | 11        |
| 5.1.2. Use Case: Administrators                 | 11        |
| 5.1.3. Use Case: Payment Gateway (e.g., Stripe) | 11        |
| 5.2. Sequence Diagram                           | 13        |
| 5.2.1. Sign-up in E-Commerce Firm               | 13        |
| 5.2.2. Login in E-Commerce Firm                 | 14        |
| 5.2.3. Add to Cart in E-Commerce Firm           | 15        |
| 5.2.4. Product Order in E-Commerce Firm         | 16        |
| 5.3: Class Diagram                              | 17        |
| 5.4. Database                                   | 18        |
| 5.4.1. Tables                                   | 18        |
| 5.4.2. Relationships                            | 18        |
| <b>Chapter 6: Implementation</b>                | <b>19</b> |
| 6.1. Setting up Workspace                       | 19        |
| 6.2. Setting up Database                        | 20        |
| 6.2.1. Set up Database Configuration            | 20        |

|   |           |
|---|-----------|
| 6.2.2. Install MySQL Driver                     | 21        |
| 6.2.3. Connect to MySQL from Node.js            | 21        |
| 6.3. Coding Frontend                            | 22        |
| 6.3.1. Create React Components                  | 22        |
| 6.3.2. Use React Router for Routing             | 23        |
| 6.3.3. Use Axios for API Calls                  | 24        |
| 6.3.4. Style your Components                    | 25        |
| 6.4. Coding Backend                             | 25        |
| 6.4.1. Create Express Routes                    | 25        |
| 6.4.2. Implement Controllers and Business Logic | 26        |
| 6.4.3. Connect to the Database                  | 27        |
| 6.4.4. Handle Authentication and Authorization  | 28        |
| 6.4.5. Error Handling and Middleware            | 29        |
| 6.5. Stripe                                     | 29        |
| 6.5.1. Sign up for a Stripe Account             | 29        |
| 6.5.2. Install the Stripe Node.js Library       | 29        |
| 6.5.3. Set up Stripe API Keys                   | 29        |
| 6.5.4. Implement Stripe Integration             | 29        |
| 6.6. Cloudinary                                 | 30        |
| 6.6.1. Sign up for a Cloudinary Account         | 30        |
| 6.6.2. Install the Cloudinary Node.js SDK       | 31        |
| 6.6.3. Set up Cloudinary API Credentials        | 31        |
| 6.6.4. Implement Cloudinary Integration         | 31        |
| 6.7. Admin Dashboards                           | 32        |
| 6.7.1. Design the Admin Dashboard               | 32        |
| 6.7.2. Implement Admin Routes                   | 32        |
| 6.7.3. Build Admin Functionality                | 32        |
| 6.7.4. Apply Security Measures                  | 32        |
| 6.8. Nodemailer                                 | 32        |
| 6.8.1. Installation and Setup                   | 32        |
| 6.8.2. Create a Transporter                     | 33        |
| 6.8.3. Create a Transporter                     | 33        |
| 6.8.4. Verification                             | 34        |
| 6.9. Project Github                             | 34        |
| <b>Chapter 7: Setup and Run Guide</b>           | <b>34</b> |
| 7.1. Prepare the Environments                   | 34        |
| 7.2. Get the Project from GitHub                | 35        |
| 7.3. Installation of MERN                       | 35        |
| <b>Chapter 8: Conclusion</b>                    | <b>36</b> |
| 8.1. Summary of the project                     | 36        |
| 8.2. Challenges faced and lessons learned       | 36        |
| 8.3. Possible future improvements or extensions | 37        |
| <b>References</b>                               | <b>38</b> |

# **ABSTRACT**

An e-commerce website is a reliable tool that businesses create to provide products and services to customers over the Internet. This type of website typically comes with a user-friendly interface with which users can interact, browse products and services inventory, add items to shopping carts and securely complete transactions.

There are multiple features incorporated into the website, such as search functionality, product categories, customer reviews, ratings, and site recommendations. Many sites also offer personalized experiences through the use of data-driven insights and analytics. In order to determine the success of an e-commerce website, user experience is the primary factor, along with other factors such as pricings, range of products, sales and promotions.

Therefore, this Web Application Development project's goal is to design a web application that can operate as an e-commerce website with all of the aforementioned features, together with functional user authentications and databases.

# **Chapter 1: Introduction to the Project**

## **1.1. Overview of the Project**

In today's digital era, e-commerce has revolutionized the way businesses operate and consumers shop. The online marketplace provides immense opportunities for businesses to reach a global customer base, expand their market presence, and increase sales. This project aims to develop an e-commerce website that caters to the growing demands of the digital marketplace.

## **1.2. Purpose and Goals**

The purpose of this project is to create a comprehensive and user-friendly e-commerce website that allows customers to browse and purchase products with ease. The primary goal is to develop a robust platform that seamlessly integrates essential features such as product catalog, shopping cart, secure payment options, order management, and customer support.

### **1.2.1. Main Goals of the Project**

1. Design and develop an attractive and intuitive user interface that enhances the shopping experience for customers.
2. Implement a reliable and secure payment gateway to facilitate safe transactions.
3. Develop an efficient inventory management system to track product availability and update stock levels in real-time.
4. Incorporate personalized recommendations and search functionalities to assist customers in finding relevant products.

### **1.2.2. Target Audience**

The target audience for this e-commerce website includes both individual consumers and businesses seeking to purchase products online. The website will cater to a diverse range of customers, including fashion enthusiasts, tech-savvy individuals, and those looking for convenient and reliable online shopping experiences.

By successfully achieving these goals, the developed e-commerce website will provide a platform that facilitates seamless transactions, enhances customer satisfaction, and drives business growth in the highly competitive online marketplace.

# Chapter 2: Details and Technologies

## 2.1. Details

### 2.1.1. Stakeholders

The development of an e-commerce website involves various stakeholders who play crucial roles in its success. It is important to identify and understand the needs and expectations of these stakeholders to ensure the website meets their requirements. The key stakeholders involved in an e-commerce website project typically include:

- **Customers:** The end-users of the website who browse products, make purchases, and provide feedback on their shopping experience.
- **Team members:** The internal team members who take part in developing the website, managing and maintaining the website. Team members can also become customers of the website, as well as administrators.
- **Suppliers:** The vendors and suppliers who provide the products or services featured on the website.
- **Partners:** Collaborative partners, such as payment gateway providers, shipping and logistics companies, and marketing affiliates, who contribute to the smooth operation of the website.
- **Regulators:** Regulatory authorities and governing bodies responsible for ensuring compliance with legal and industry standards, such as data protection regulations, consumer rights, and taxation requirements.

Understanding the expectations and requirements of these stakeholders is crucial for delivering an e-commerce website that effectively serves their needs and achieves the desired outcomes.

## 2.2. Technologies

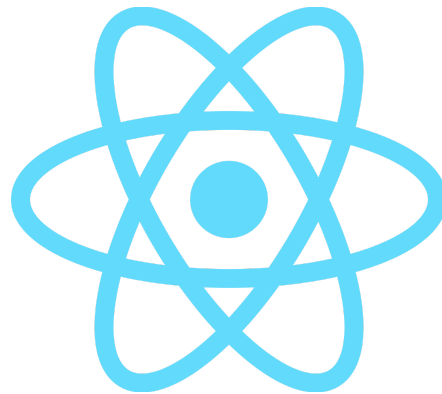
### 2.2.1. Front-end Technologies

The front-end of an e-commerce website focuses on creating an engaging and user-friendly interface that enhances the customer's shopping experience. The following technologies are commonly used in front-end development:

1. **HTML (Hypertext Markup Language):** Used to structure the content and layout of web pages.
2. **CSS (Cascading Style Sheets):** Used to define the visual appearance, layout, and design of web pages.
3. **JavaScript:** A programming language that enables interactive and dynamic elements on web pages.



4. **ReactJS:** A popular JavaScript library for building responsive and interactive user interfaces.

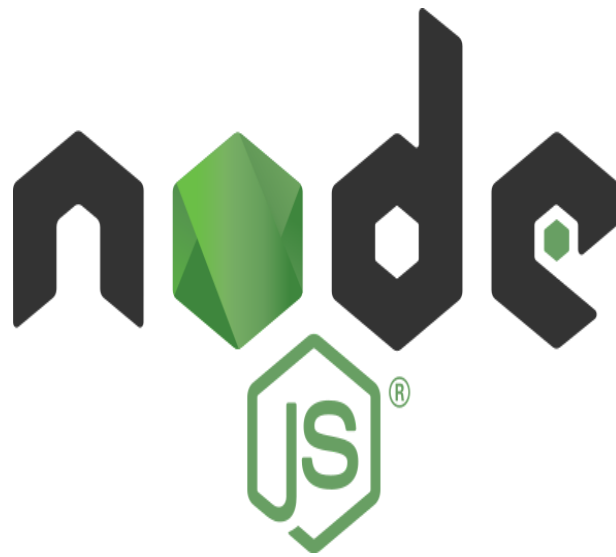


These front-end technologies allow developers to create visually appealing and intuitive interfaces that facilitate seamless navigation and product discovery for customers.

### 2.2.2. Back-end Technologies

The back-end of an e-commerce website involves managing the business logic, data processing, and integration with various systems. The following technologies are commonly used in back-end development:

1. **Node.js:** A JavaScript runtime environment that allows server-side scripting and event-driven programming, facilitating the development of scalable and high-performance web applications.



2. **MySQL:** A popular relational database management system used for storing and managing product data, customer information, and order details.



3. **Express.js:** A flexible and lightweight web application framework for Node.js, enabling the development of robust APIs and handling server-side functionalities.

These back-end technologies provide the necessary infrastructure to handle tasks such as order processing, inventory management, payment processing, and integration with third-party services.

By leveraging the right combination of front-end and back-end technologies, an e-commerce website can deliver a seamless and efficient shopping experience while ensuring the secure management of data and transactions.



## Chapter 3: Project Planning

In this chapter, we will outline the project plan for our upcoming endeavor. A well-structured plan is essential for the successful execution of any project. The following table provides an overview of the tasks, sub-tasks, assignees, deadlines, and reviewers involved. Please note that this table is subject to revisions and updates as the project progresses.

| Period    | Task                  | Sub-task             | Assignee    | Deadline | Reviewer    |
|-----------|-----------------------|----------------------|-------------|----------|-------------|
| 10/4-17/4 | Starting the Project  | Setting up workspace | All members | 17/4     | Duy         |
| 17/4-24/4 | Analyzing the Project | Requirements         | Duy         | 24/4     | All members |
|           |                       | Diagrams             | All members |          |             |
|           |                       | Presentation Slides  | Phuc        | 14/5     |             |
| 24/4-10/5 | Implementation        | MySQL Database       | Duy         | 10/5     |             |
|           |                       | Application          | Duy         | 10/5     |             |
| 10/5-End  | Testing               | Testing              | All members | 26/5     |             |
|           |                       | Other tasks          | All members | 26/5     |             |

### 3.1. Starting the Project (10/4-17/4)

During this phase, the team will focus on setting up the workspace required for the project. This task will be collectively handled by all team members to ensure a smooth start. The deadline for completing this task is set for 17/4, and Duy will review the progress.

### 3.2. Analyzing the Project (17/4-24/4)

In this period, the team will analyze the project in detail. Duy will take the lead in identifying and documenting the project's requirements. The deadline for completing the requirements analysis is set for 24/4, and all team members will review and provide their inputs.

All team members will work on creating diagrams to visualize the project's structure, workflows, or any other relevant aspects. Additionally, Phuc will be responsible for preparing presentation slides to communicate the analyzed project details effectively. The deadline for the diagrams and presentation slides is set for 24/4, and will be reviewed by all members.

### 3.3. Implementation (24/4-10/5)

During this phase, the team will focus on implementing the project. Duy will handle the implementation of the MySQL database, ensuring its proper setup and functionality. The deadline for completing the database implementation is set for 10/5, with no specific reviewer assigned.

Furthermore, Duy will also be responsible for developing the application itself. He will work on coding the necessary functionalities and ensuring a seamless user experience. The deadline for completing the application development is set for 10/5. The work will be presented with all members, gathering their opinions for further development.

### 3.4. Testing (10/5-End)

Once the implementation phase is complete, the team will shift their focus to testing the project. All team members will actively participate in testing the application, ensuring that it meets the specified requirements and functions as intended. The results of testing are contributed to the new development cycle of the website. The deadline for completing the testing phase is set for 26/5.

### 3.5. Other Tasks (10/5-End)

Throughout the project, there might be additional tasks that arise, which are not explicitly mentioned in the table. These tasks will be collectively handled by all team members to ensure their timely completion. The deadline for completing any other tasks is set for 26/5.

## Chapter 4: Analysis

### 4.1. Functional Requirements

As every other project, the requirements analysis is the highest priority in the work chart. Understanding requirements is the key in creating a satisfactory application for clients. Thus, the team has tackled this problem first and created a detailed functional and non-functional requirements table to keep track and modify.

| Req.ID | Requirement Name              | Detailed Descriptions  | Type                   |
|--------|-------------------------------|--|------------------------|
| 001    | Register and login an account | Users can login to their accounts or register an account for the site. Upon registering, an email will be sent to the user's mail to verify the login. | Functional Requirement |
| 002    | Product Catalog               | The website should display all relevant information about the products (price, image, availability,...).   | Functional Requirement |
| 003    | Shopping Cart                 | Users can add items into the shopping carts, remove items and view their carts before checking out.  | Functional Requirement |
| 004    | Payment Gateways              | Payment gateways are required to facilitate transactions between users and   | Functional Requirement |

|     |                      |  |                        |
|-----|----------------------|--|------------------------|
|     |                      | business.  |                        |
| 005 | Order Management     | Order Management allows users to view their orders and businesses to manage their orders.                      | Functional Requirement |
| 006 | Item Search Function | Search Function allows users to find items based on their requirements.  | Functional Requirement |
| 007 | Admin Dashboard      | A dashboard where admin can monitor site's activities, or create, retrieve, update, delete products and users. | Functional Requirement |
| 008 | User Profile         | User profile where users can change their account information, or keep track of their orders' status.          | Functional Requirement |

## 4.2. Non-functional Requirements

| Req.ID | Requirement Name | Detailed Descriptions   | Type                       |
|--------|------------------|---|----------------------------|
| 001    | Performance      | The website should have fast loading times and be able to handle high volume or traffic without crashing or slowing down.                             | Non-Functional Requirement |
| 002    | Security         | The website should be secure, with measures to protect customer data and prevent unauthorized access.   | Non-Functional Requirement |
| 003    | Usability        | The website should be easy to navigate, with a clear and intuitive interface that allows users to quickly find what they are looking for.             | Non-Functional Requirement |
| 004    | Reliability      | The website can be available most of the time, except for downtime for maintenance or update.   | Non-Functional Requirement |
| 005    | Scalability      | The website should be able to handle growth and expansion, with the ability to add new products, users and features without compromising performance. | Non-Functional Requirement |

# Chapter 5: System Design

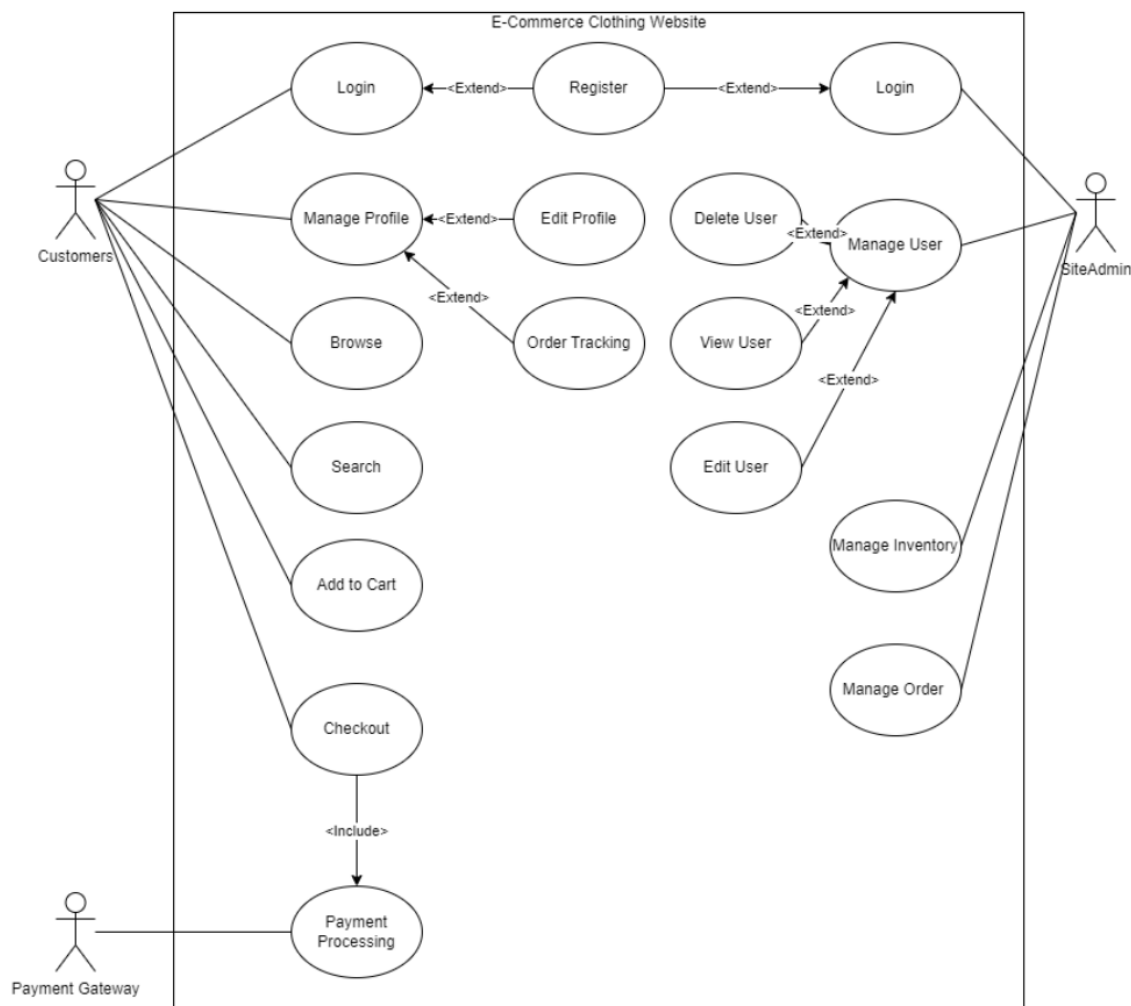
System design involves many complicated phases which include various types of diagrams for the visualization of the system. With these diagrams, system design follows a strictly-observed principle. In this project, three types of diagrams will be used for various purposes.

## 5.1. Use Case Diagram

Use case diagrams provide a visual representation of the interactions between actors and the system, illustrating the functionalities that the system needs to support.

**Actors:** Customers, Administrators, Payment Gateway (e.g., Stripe).

**Use Cases:** Customers, Administrators, Payment Gateway (e.g., Stripe).



**Diagrams:** [Use Case Diagram](#)

### 5.1.1. Use Case: Customers

**Customer use case:**

- **Login:** Customers can log in to their accounts.

- **Register:** Customers can create new accounts.
- **Edit Profile:** Customers can modify their profile information.
- **Order Tracking:** Customers can see their order status from the user profile tab.
- **Browse Catalog:** Customers can explore the website's product catalog, filter and search for clothing items based on various criteria such as category, description, etc.
- **Add to Cart:** Customers can add selected products to their shopping cart for future purchase.
- **Checkout:** Customers can initiate the checkout process, providing billing and shipping information, and selecting a payment method.
- **Payment Processing:** The payment gateway handles the payment processing, verifying customer payment details and authorizing transactions.

### **Use case description:**

**Title:** Customer Checkout Process

### **Description:**

When visiting the online clothing e-commerce website, customers have various features and options available to enhance shopping experience.

1. **Login or Register:** Customers can log in to their existing accounts or create new accounts to access personalized features.
2. **Browse Catalog:** Customers can explore our extensive product catalog, using search and filters to find desired clothing items.
3. **Add to Cart:** Customers can add selected products to their shopping cart for future purchase.
4. **Edit Profile:** Customers can modify their profile information, such as updating their shipping address and contact details.
5. **Checkout:** Customers can proceed to the checkout process, providing billing and shipping information and selecting a payment method.
6. **Payment Processing:** The payment gateway handles the payment processing, verifying customer payment details and authorizing transactions.
7. **Order Tracking:** Customers receive an order confirmation with a unique order ID, enabling them to track their order's status and estimated delivery date from their user profile.

### 5.1.2. Use Case: Administrators

#### Administrator use case:

- **Login:** Administrators can log in to their accounts.
- **Manage Users:** Administrators can manage user accounts, including creating new accounts, updating account information, and resetting passwords.
- **Manage Inventory:** Administrators can manage the product inventory, including adding new products, updating prices and descriptions, and removing out-of-stock items.
- **Manage Orders:** Administrators can manage customer orders, such as processing refunds, handling returns, and addressing customer service inquiries.

#### Use case description:

**Title:** Administrator Order Management

#### Description:

As an administrator of an online clothing e-commerce website, the admin has access to a range of features and functionalities that effectively manage various aspects of the business.

1. **Login:** Administrators can log in to the administrator dashboard using their authorized credentials.
2. **User Management:** Administrators can manage user accounts, including creating new accounts, updating account information, and resetting passwords.
3. **Inventory Management:** Administrators can manage the product inventory, such as adding new products, updating prices and descriptions, and removing out-of-stock items.
4. **Manage Orders:** Administrators can view and manage customer orders, including processing refunds, handling returns, and addressing customer service inquiries.

### 5.1.3. Use Case: Payment Gateway (e.g., Stripe)

#### Payment Gateway use case:

- **Payment Processing:** The payment gateway handles the payment processing, securely managing customer payment information, and facilitating transactions between customers and the business.

#### Use case description:

**Title:** Order Processing

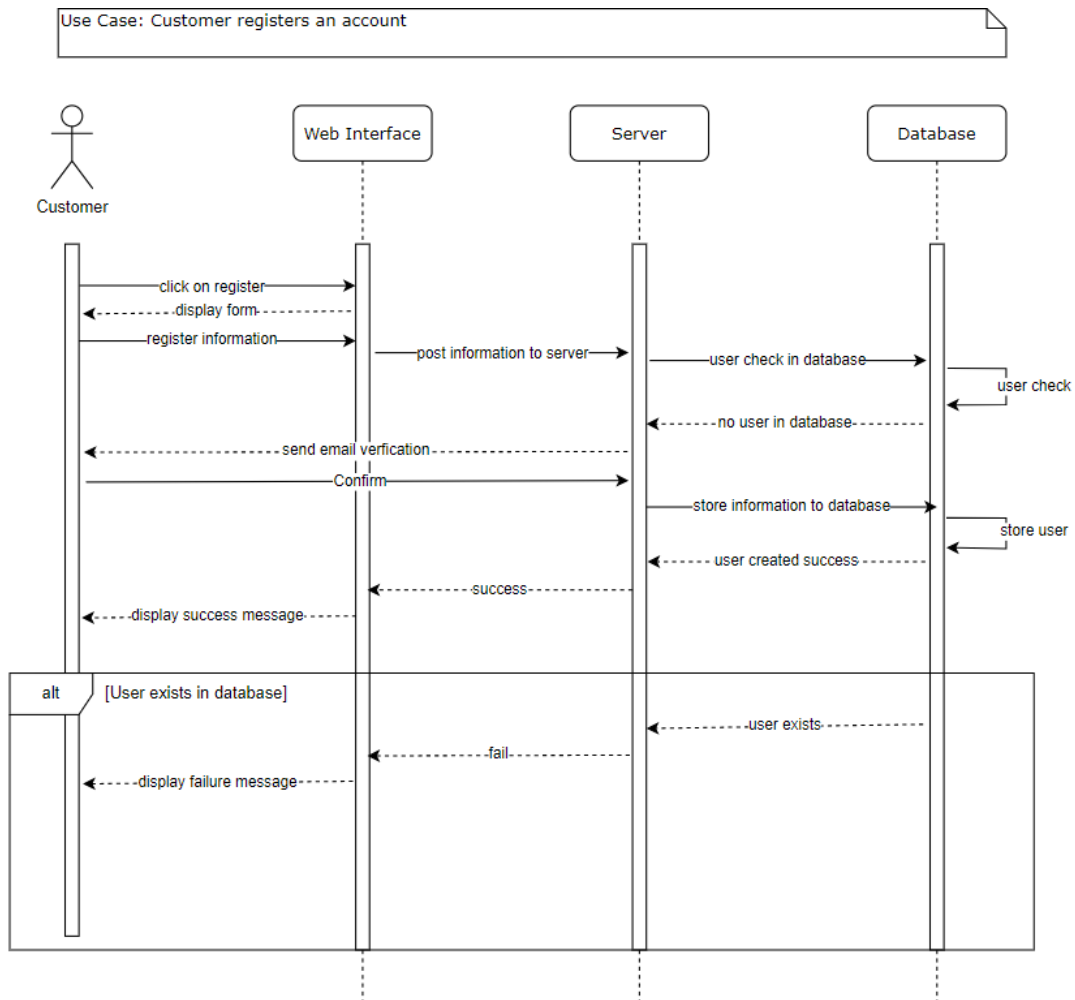
#### Description:

When customers perform checkout, payment gateways handle the payment process.

1. **Payment Processing:** The payment gateway handles the payment processing, securely managing customer payment information, and facilitating transactions between customers and the business.

## 5.2. Sequence Diagram

### 5.2.1. Sign-up



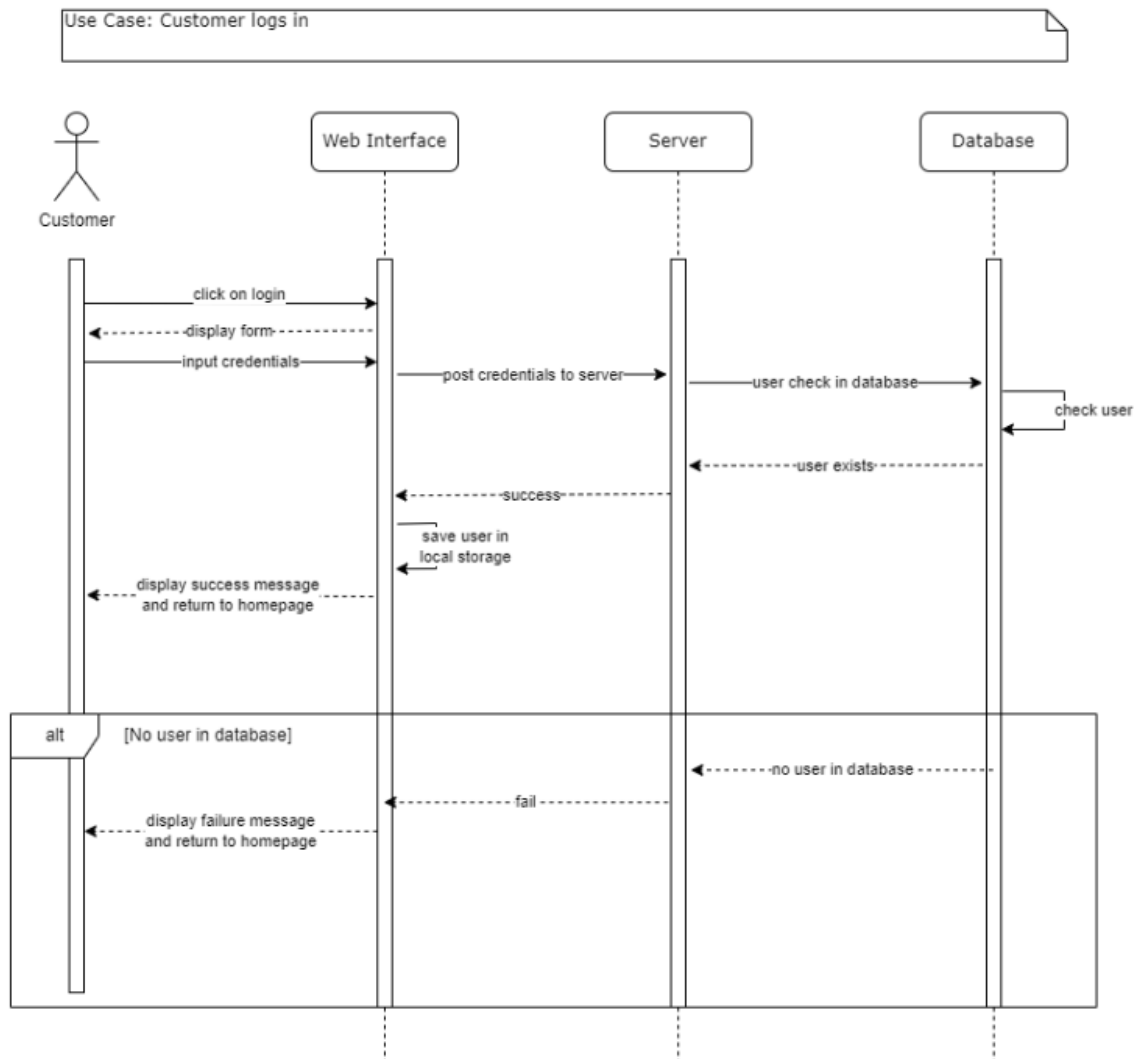
**Diagrams:** [Sequence Diagrams](#)

**Use case:** Customer registers an account.

**Use case description:**

1. Customer visits page and clicks on register. A registration form is displayed.
2. Customer inputs information and clicks register.
3. If there is no user in the database, an email is sent to the customer for verification.
4. Customer clicks verified. Database stores user information. Web interface displays a registration success message.
5. If user information exists or has error information, a failure message is displayed to the customer.

## 5.2.2. Login



**Diagrams:** [Sequence Diagrams](#)

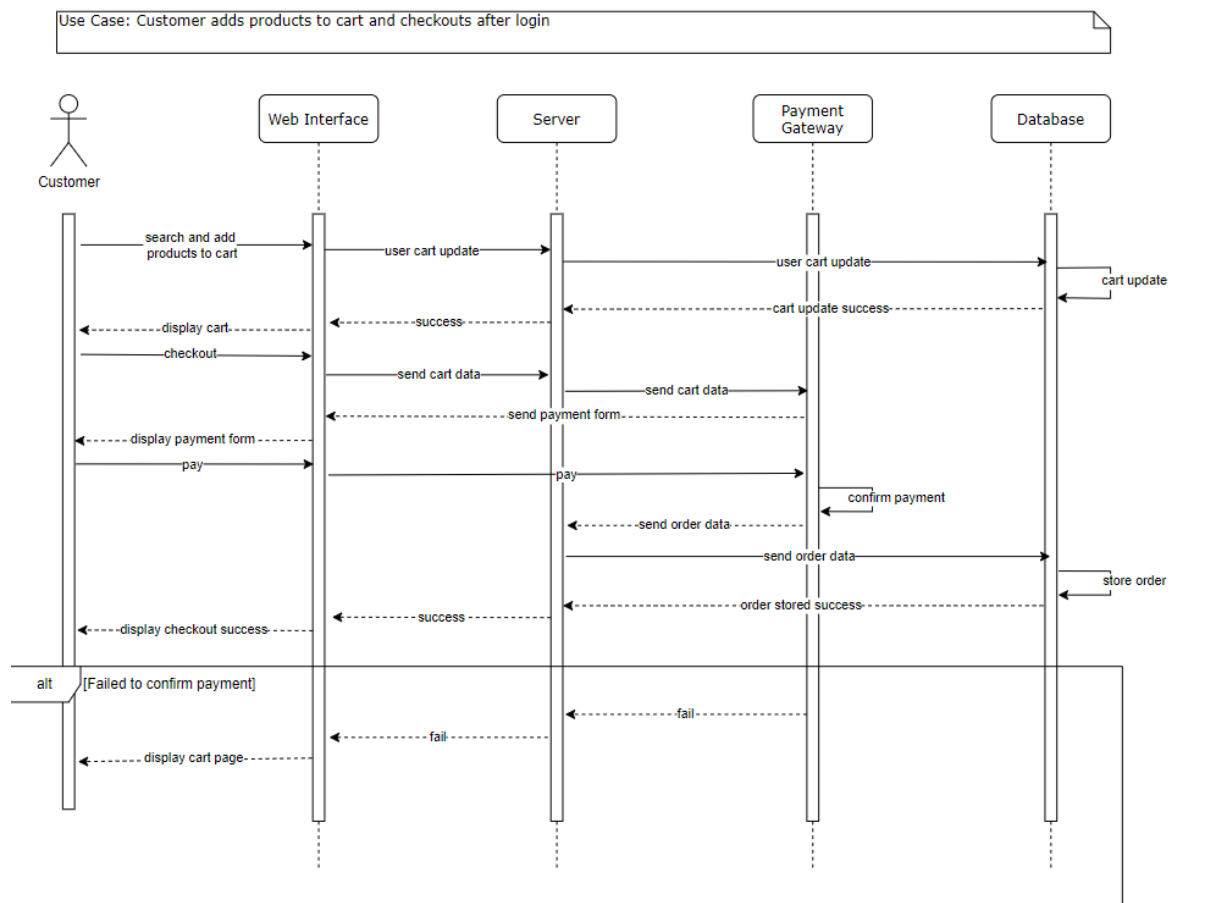
**Use case:** Customer logs in.

**Use case description:**

1. Customer visits page and clicks on login. A login form is displayed.
2. Customer inputs credentials and clicks login.
3. If user credentials exist, the server sends a success message to the web interface. The interface displays a success message and returns to the homepage with the customer logged in.
4. If there is no user in the database, a failure message is displayed to the customer.



### 5.2.3. Add to Cart and Checkout



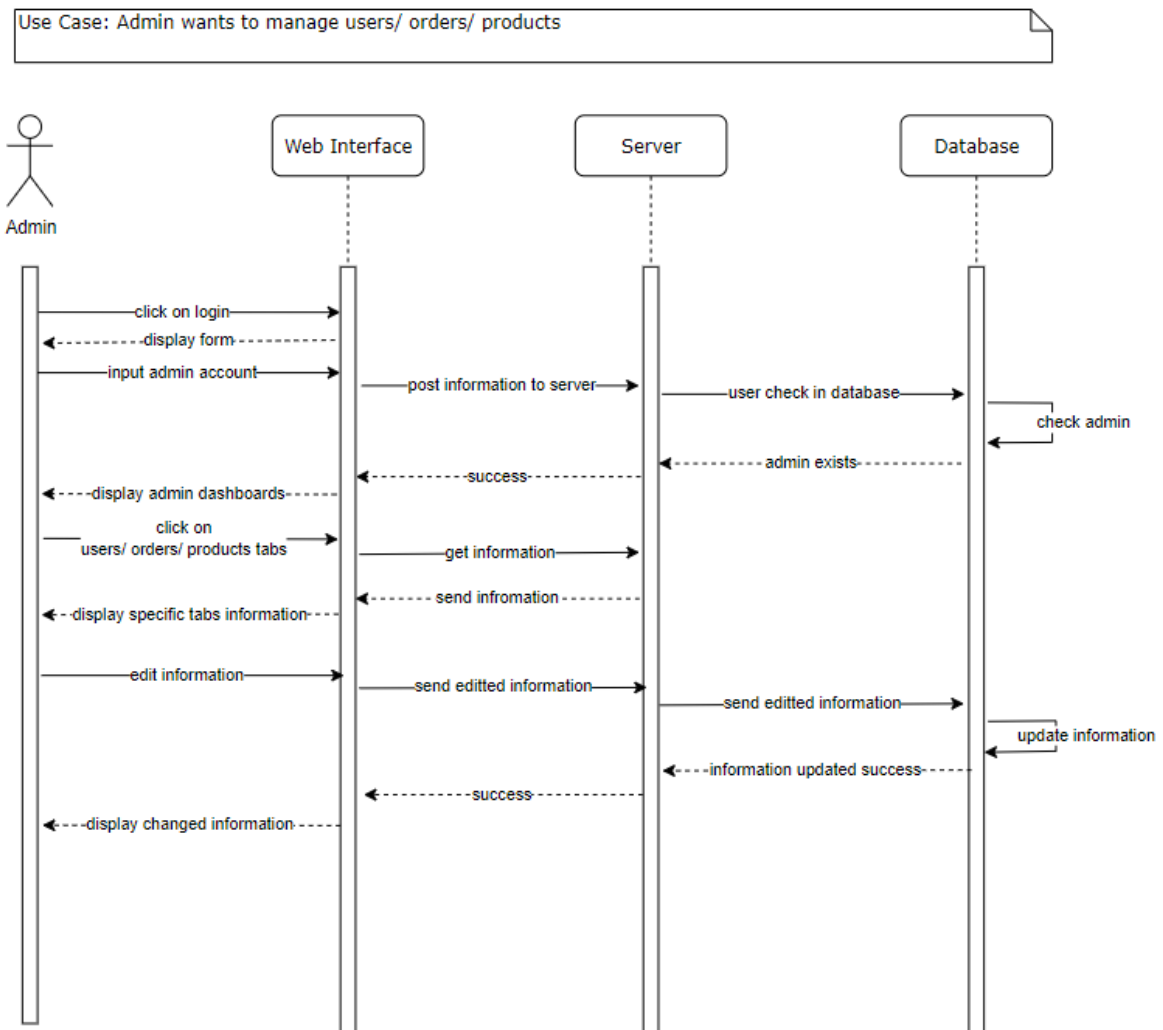
**Diagrams:** [Sequence Diagrams](#)

**Use case:** Customer adds products to cart and checkouts after login.

**Use case description:**

1. Customer searches and adds products to cart. The changed cart data is sent to the server. Server sends data to the database to update. The updated cart is displayed after the update is completed.
2. Customer clicks checkout. The cart data is sent to the server, which is subsequently sent to the payment gateway. The payment gateway displays a payment form to the customer through the web interface.
3. Customer clicks pay. The gateway confirms the payment and sends order data to the server. Server sends order data to the database for storage. A checkout success page is displayed to the customer.
4. If payment confirmation fails, a failure message is displayed to the customer.

## 5.2.4. Admin Managements



**Diagrams:** [Sequence Diagrams](#)

**Use case:** Admin wants to manage users/ orders/ products.

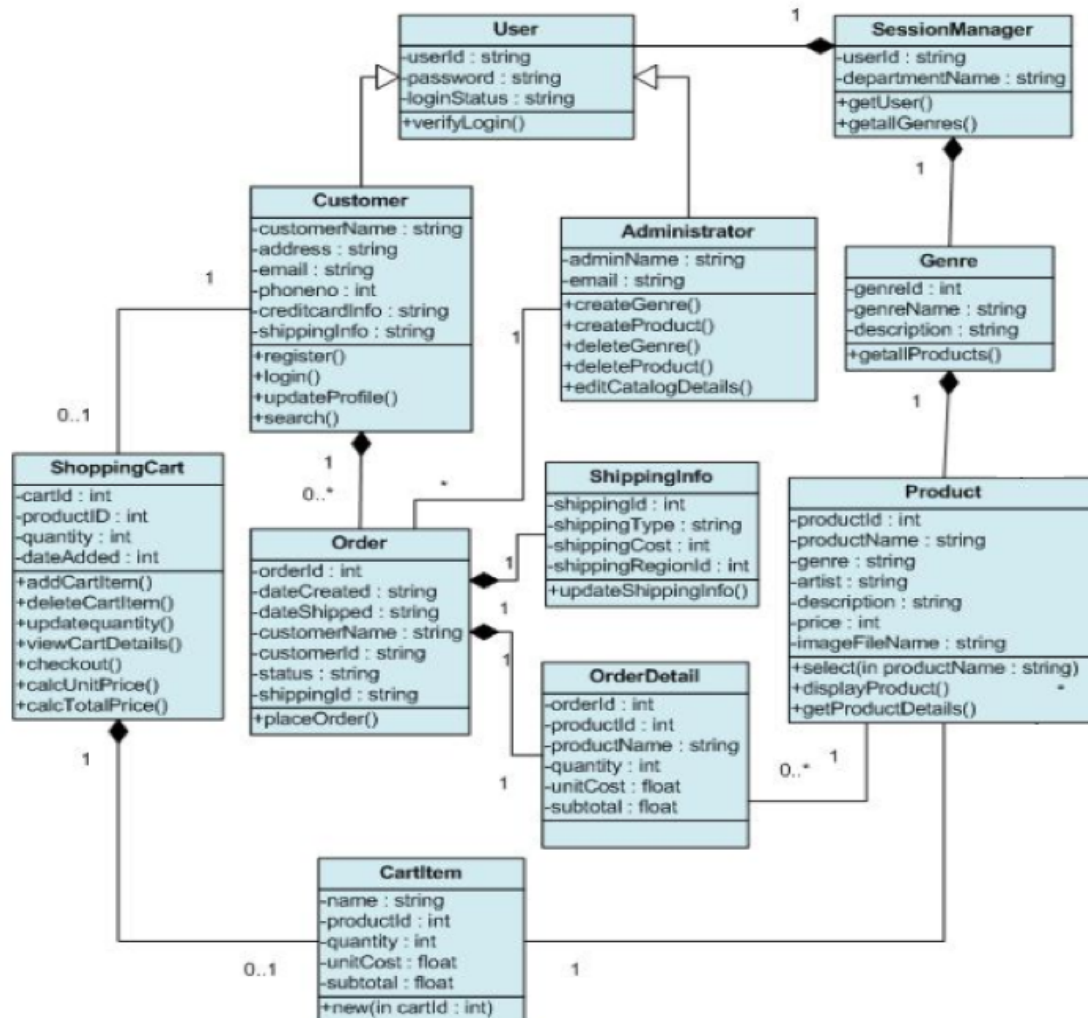
**Use case description:**

1. Admin visits the page and clicks on login. A login form is displayed.
2. Admin inputs credentials and clicks login.
3. If admin credentials exist, the server sends a success message to the web interface. The interface displays a success message and an admin dashboard is displayed.
4. Admin clicks on the users/ orders/ products tabs. Information is displayed in a list for the admin to edit. Upon edit, the information is sent to the server which is later sent to the database to update the changes. Updated information is then displayed for the admin to see.

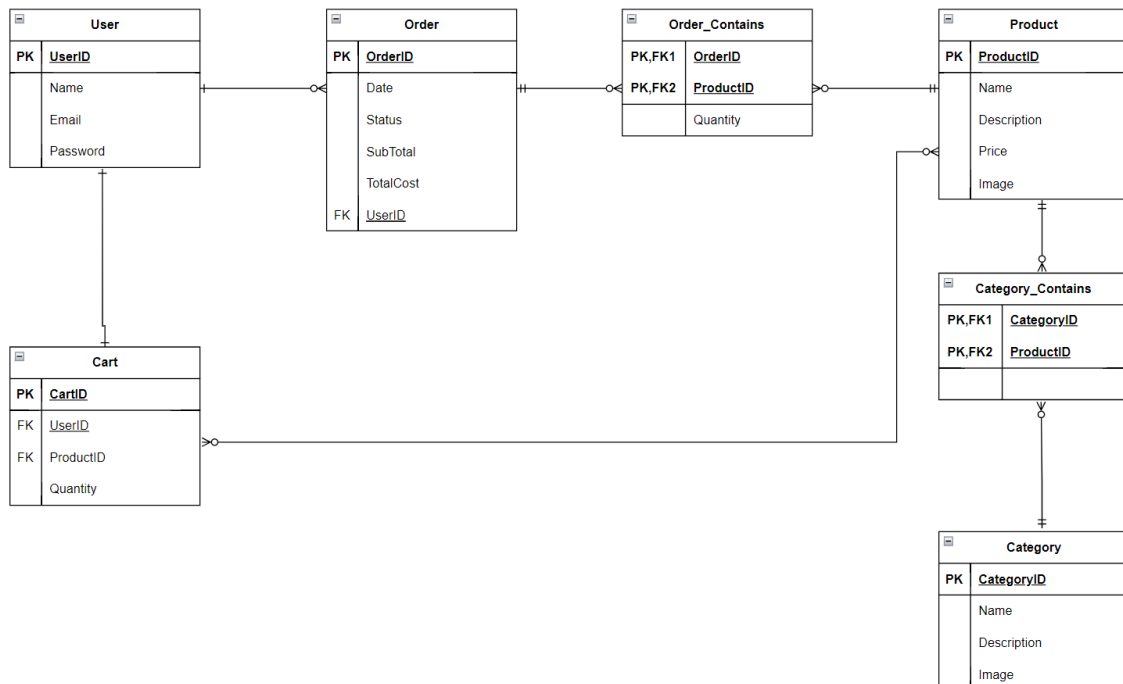
## 5.3: Class Diagram

The system's class schema includes the following entities/attribute/operation:

1. **User**: Information and status of the customer's account.
2. **Customer**: Include customer information and edit account information.
3. **ShoppingCart**: Product information and the ability to edit cart.
4. **CartItem**: Display product information in cart.
5. **Order**: General information and status of the order.
6. **OderDetail**: Detailed information about order, product quantity, price.
7. **Product**: Information about products available in the store.
8. **Genre**: Classification of products.
9. **Administrator**: Management information and the ability to edit products.
10. **SessionManager**: Manage customers and product categories.



## 5.4. Database



Diagrams: [ERD Diagram](#)

### 5.4.1. Tables

The system's database schema includes the following tables/entities:

1. **User:** This table stores user information, including their name, email, and password. Each user can have a cart and proceed with payments.
2. **Product:** This table holds information about the available products for purchase on the website. It includes details such as the product name, description, price, and image. Each product can belong to multiple categories, and each category can contain multiple products. Additionally, each product can appear in multiple orders.
3. **Category:** This table stores information about the categories to which products belong, such as men's clothing, women's clothing, and shoes. Each category can contain multiple products, and each product can belong to multiple categories.
4. **Category\_Contains:** This table serves as a bridge between the product and category tables. It contains the ProductID and CategoryID columns, enabling efficient query execution and association between products and categories.
5. **Cart:** This table stores information about user carts, including the CartID, UserID, ProductID, and Quantity. It keeps track of the products added to the cart and their respective quantities.
6. **Order:** This table holds information about the payments made for orders. It includes details such as the order amount, date, payment status, UserID, and shipping information.
7. **Order\_Products:** This table stores information about the products included in each order and their respective quantities.

### 5.4.2. Relationships

The tables/entities are interconnected through the following relationships:

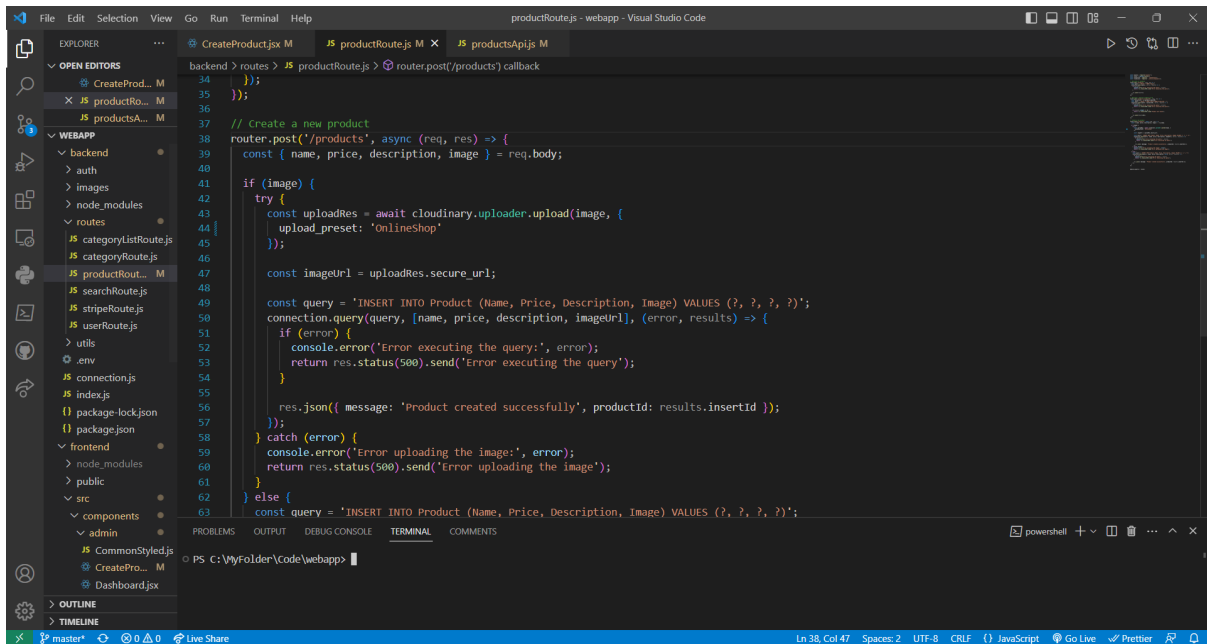
1. **User and Cart:** These entities have a one-to-one relationship, indicating that each user has a single cart and can add products to it.
2. **Product and Category:** These entities have a many-to-many relationship, indicating that each category can contain multiple products, and each product can belong to multiple categories. This relationship is facilitated by the `Category_Contains` table, which holds the `ProductID` and `CategoryID` for efficient query execution.
3. **User and Order:** These entities have a one-to-many relationship, indicating that each user can have multiple orders, but each order belongs to only one user.
4. **Product and Order:** These entities have a many-to-many relationship, indicating that each product can appear in multiple orders, and each order can contain multiple products. The `Order_Products` table, along with the quantity of each product, facilitates this relationship.
5. **Cart and Product:** These entities have a many-to-many relationship, indicating that each product can appear in multiple carts, and each cart can contain multiple products.

## Chapter 6: Implementation

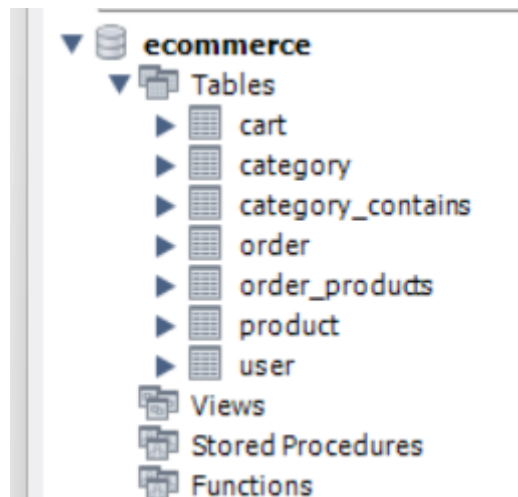
### 6.1. Setting up Workspace

In this section, we will set up the workspace for React, Node.js, Express, MySQL and Visual Studio Code (VSC).

- For VSC: [Visual Studio Code - Code Editing. Redefined](#)
- For Node.js: [Node.js \(nodejs.org\)](#)
- For React: [React](#)
- For Express: [Express - Node.js web application framework \(expressjs.com\)](#)
- For MySQL: [MySQL](#)



## 6.2. Setting up Database



### 6.2.1. Set up Database Configuration

- In the project directory, create a file called connection.js to store environment variables.
- Inside the connection.js file, add the following line to configure your MySQL database connection:

```
DB_CONNECTION=mysql
DB_HOST=your_mysql_host
DB_PORT=your_mysql_port
DB_DATABASE=your_database_name
DB_USERNAME=your_mysql_username
DB_PASSWORD=your_mysql_password
```

- Replace all ``your_mysql_host``, ``your_mysql_port``, ``your_database_name``, ``your_mysql_username``, and ``your_mysql_password`` with your actual MySQL connection details.

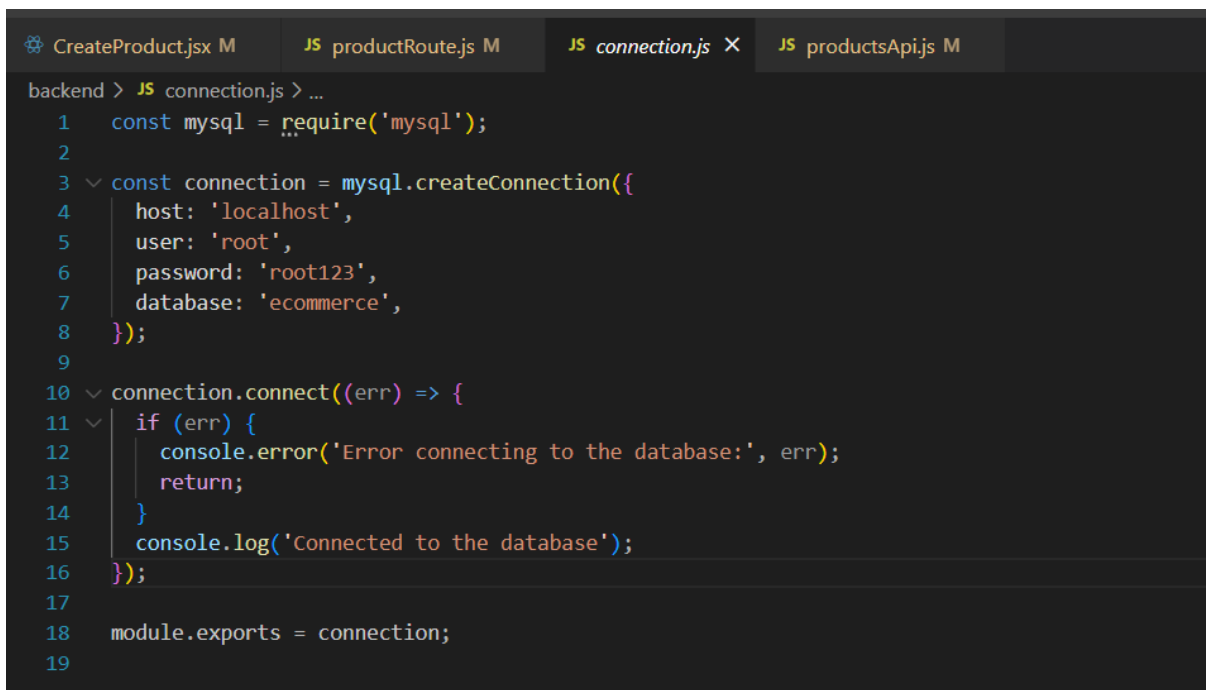
### 6.2.2. Install MySQL Driver

- In project directory, run the following command to install the MySQL driver for Node.js:

```
npm install mysql
```

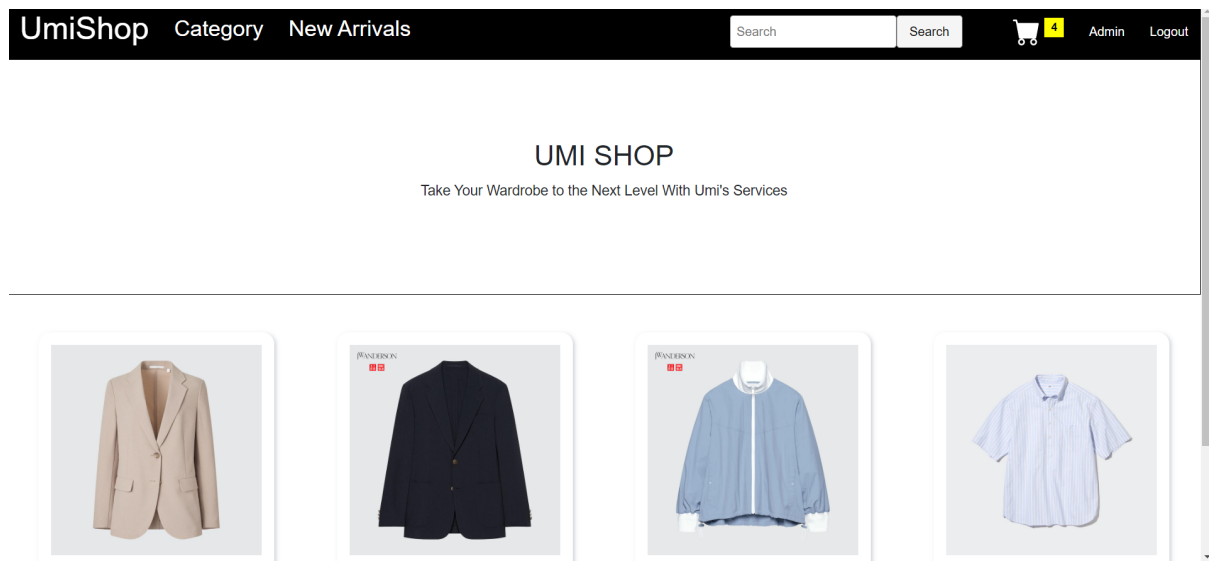
### 6.2.3. Connect to MySQL from Node.js

- In Node.js code, establish a connection to the MySQL database using the ``mysql`` package and the provided configuration.



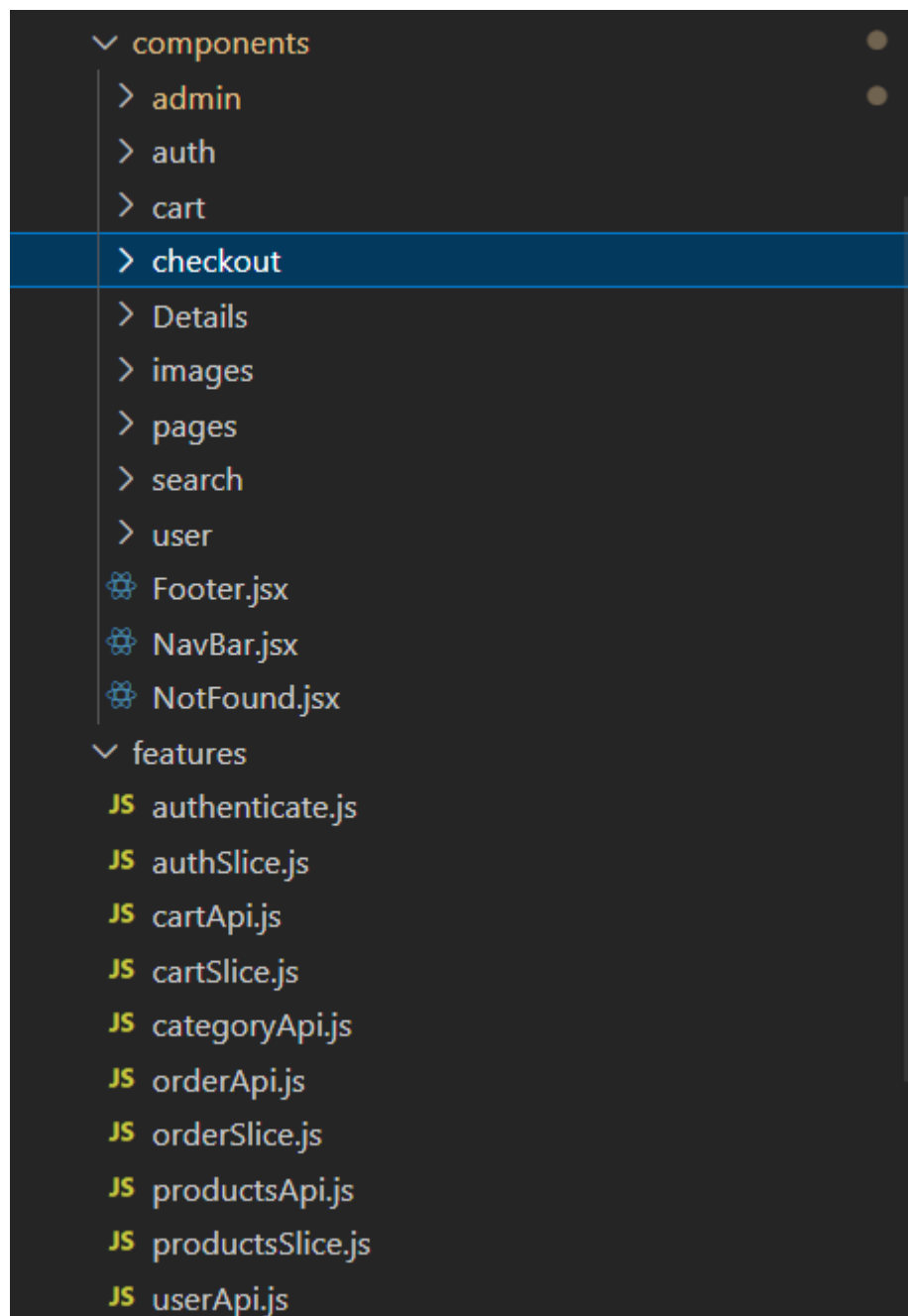
```
backend > JS connection.js > ...
1  const mysql = require('mysql');
2
3  const connection = mysql.createConnection({
4    host: 'localhost',
5    user: 'root',
6    password: 'root123',
7    database: 'ecommerce',
8  });
9
10 connection.connect((err) => {
11   if (err) {
12     console.error('Error connecting to the database:', err);
13     return;
14   }
15   console.log('Connected to the database');
16 });
17
18 module.exports = connection;
19
```

## 6.3. Coding Frontend





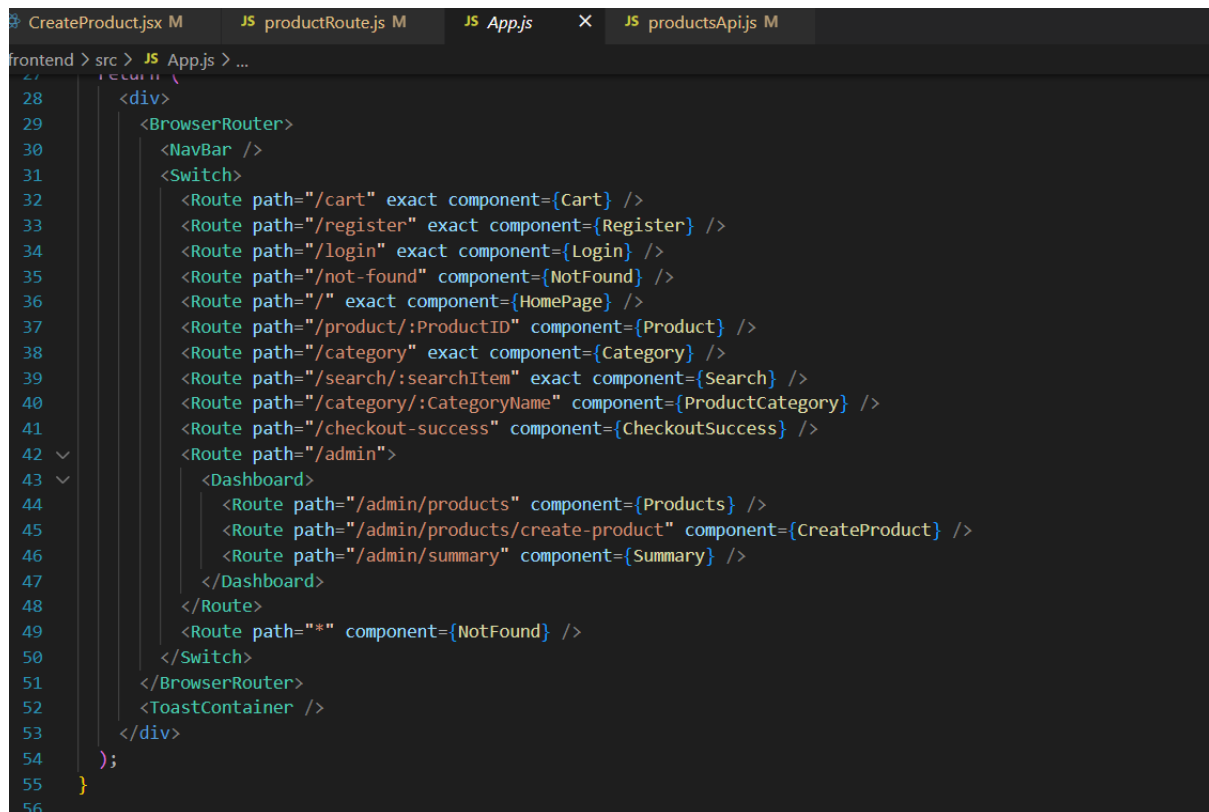
### 6.3.1. Create React Components



- Inside the `src/components` directory, create individual files for React components.
- Define and implement each component according to application's requirements.
- Features the location for all the Api files to communicate with the backend. All the slices files are used to store information in redux store for ease of access across all files in frontend.
- App.js stores the web structure and connects all components into a single website.
- Index.js acts as a redux store for all components to share and access information.

### 6.3.2. Use React Router for Routing

- Install React Router by running the following command in project directory:  
npm install react-router-dom
- Define the application's routes using React Router's `` component and implement the necessary components for each route.



```
28 <div>
29   <BrowserRouter>
30     <NavBar />
31     <Switch>
32       <Route path="/cart" exact component={Cart} />
33       <Route path="/register" exact component={Register} />
34       <Route path="/login" exact component={Login} />
35       <Route path="/not-found" component={NotFound} />
36       <Route path="/" exact component={HomePage} />
37       <Route path="/product/:ProductID" component={Product} />
38       <Route path="/category" exact component={Category} />
39       <Route path="/search/:searchItem" exact component={Search} />
40       <Route path="/category/:CategoryName" component={ProductCategory} />
41       <Route path="/checkout-success" component={CheckoutSuccess} />
42       <Route path="/admin">
43         <Dashboard>
44           <Route path="/admin/products" component={Products} />
45           <Route path="/admin/products/create-product" component={CreateProduct} />
46           <Route path="/admin/summary" component={Summary} />
47         </Dashboard>
48       </Route>
49       <Route path="*" component={NotFound} />
50     </Switch>
51   </BrowserRouter>
52   <ToastContainer />
53 </div>
54 );
55 }
56
```

### 6.3.3. Use Axios for API Calls

- Install Axios, a popular HTTP client, by running the following command:  
npm install axios
- Use Axios to make API calls to the backend server and handle the responses.

```
CreateProduct.jsx M    JS productRoute.js M    JS productsApi.js M    JS authenticate.js X
frontend > src > features > JS authenticate.js > ...
1  import axios from "axios";
2
3  const authenticate = async (formData, endpoint) => {
4    try {
5      const response = await axios.post(
6        `http://localhost:5000/${endpoint}`,
7        formData
8      );
9      return response.data;
10   } catch (error) {
11     console.error(error);
12     throw error;
13   }
14 };
15
16 export default authenticate;
17
```

#### 6.3.4. Style your Components

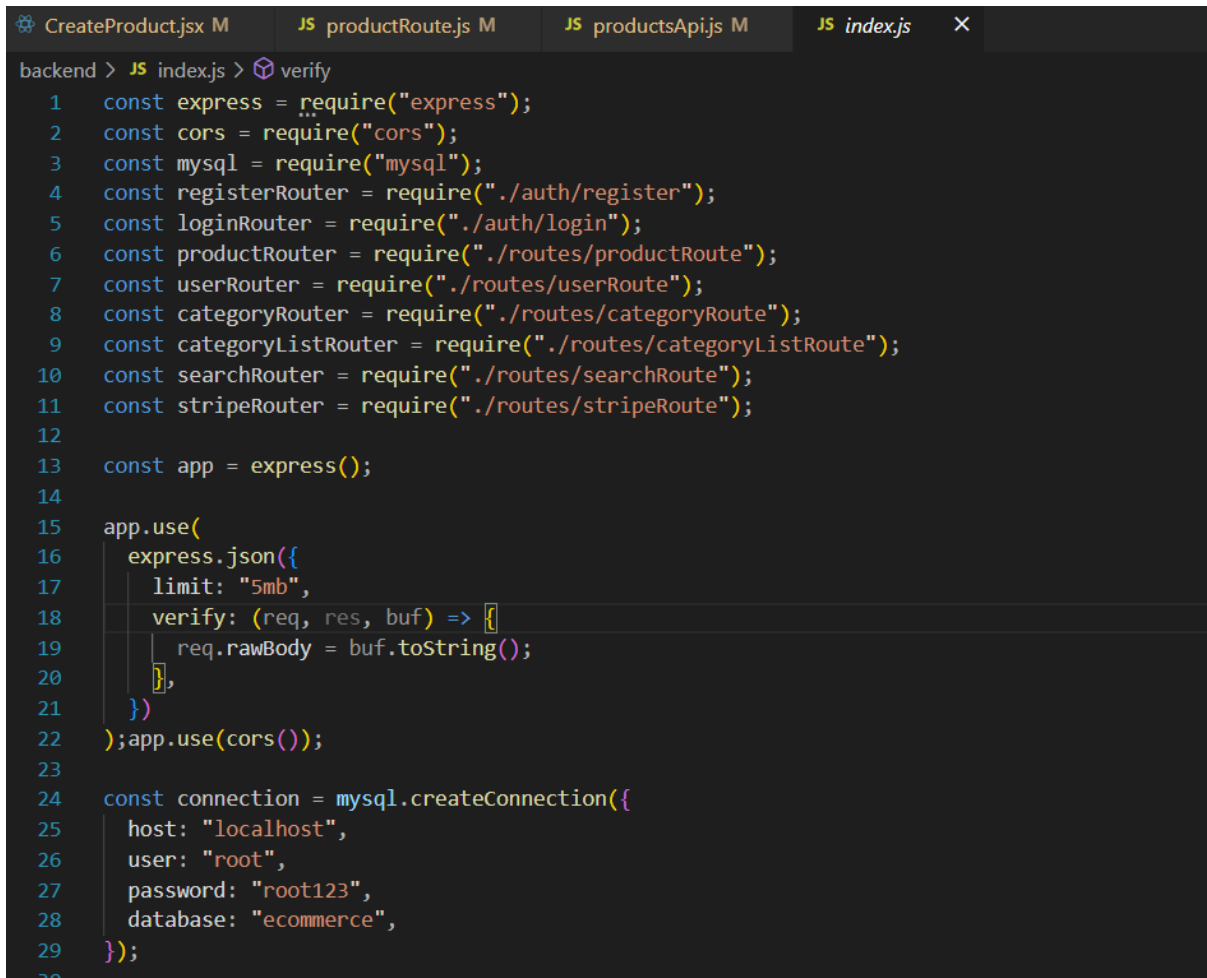
- Utilize CSS frameworks (e.g., Bootstrap, Material-UI) or custom CSS to style components.
- Create CSS files or use CSS-in-JS libraries (e.g., styled-components) to manage component styling.
- App.css is also the location to store CSS.

```
CreateProduct.jsx M    JS productRoute.js M    JS productsApi.js M    # App.css X
frontend > src > # App.css > .nav-bar
1  @import url("https://fonts.googleapis.com/css2?family=Nunito+Sans:wght@200;400;700&display=swap");
2
3  * {
4    padding: 0;
5    margin: 0;
6    box-sizing: border-box;
7    font-family: "Nunito", sans-serif;
8  }
9
10 /* NavBar */
11
12 .nav-bar {
13   height: 70px;
14   background-color: black;
15   display: flex;
16   justify-content: space-between;
17   align-items: center;
18   padding: 0 1rem;
19 }
20
21 .nav-bar a {
22   text-decoration: none;
23   color: white;
24 }
25
26 .nav-bar h2 {
27   font-size: 40px;
28 }
```

## 6.4. Coding Backend

### 6.4.1. Create Express Routes

- Inside the `index.js` file, define and implement Express routes to handle various API requests.
- Use the `express.Router()` function to create modular route handlers for different parts of application.



```
backend > JS index.js > verify
1  const express = require("express");
2  const cors = require("cors");
3  const mysql = require("mysql");
4  const registerRouter = require("../auth/register");
5  const loginRouter = require("../auth/login");
6  const productRouter = require("../routes/productRoute");
7  const userRouter = require("../routes/userRoute");
8  const categoryRouter = require("../routes/categoryRoute");
9  const categoryListRouter = require("../routes/categoryListRoute");
10 const searchRouter = require("../routes/searchRoute");
11 const stripeRouter = require("../routes/stripeRoute");
12
13 const app = express();
14
15 app.use(
16   express.json({
17     limit: "5mb",
18     verify: (req, res, buf) => {
19       req.rawBody = buf.toString();
20     },
21   })
22 ); app.use(cors());
23
24 const connection = mysql.createConnection({
25   host: "localhost",
26   user: "root",
27   password: "root123",
28   database: "ecommerce",
29 });
30
```

### 6.4.2. Implement Controllers and Business Logic

- Create separate controller files to handle the logic for each route.
- Implement the necessary functions to process incoming requests, interact with the database, and send appropriate responses.

The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with a 'backend' folder containing 'routes' and 'utils' subfolders. The 'categoryListRoute.js' file is selected in the 'routes' folder. The code editor shows the following code:

```
backend > routes > JS categoryListRoute.js > ...
1  const express = require('express');
2  const router = express.Router();
3  const connection = require('../connection');
4
5  // Retrieve category
6  router.get('/category', (req, res) => {
7    const categoryName = req.params.categoryName;
8    const query = `
9      SELECT * FROM Category
10    `;
11    connection.query(query, [categoryName], (error, results) => {
12      if (error) {
13        console.error('Error executing the query:', error);
14        return res.status(500).send('Error executing the query');
15      }
16      res.json(results);
17    });
18  });
19
20
21 module.exports = router;
22
```

### 6.4.3. Connect to the Database

- Use the MySQL driver (as described in section 6.2) to connect to the MySQL database from backend code.
- Implement database queries and operations within controllers to perform CRUD (Create, Read, Update, Delete) operations.

```

1  const mysql = require('mysql');
2
3  const connection = mysql.createConnection({
4    host: 'localhost',
5    user: 'root',
6    password: 'root123',
7    database: 'ecommerce',
8  });
9
10 connection.connect((err) => {
11   if (err) {
12     console.error('Error connecting to the database:', err);
13     return;
14   }
15   console.log('Connected to the database');
16 });
17
18 module.exports = connection;
19

```

#### 6.4.4. Handle Authentication and Authorization

- Implement user authentication and authorization mechanisms.
- Secure routes based on user roles and permissions.

```

1  const express = require("express");
2  const bcrypt = require("bcrypt");
3  const connection = require('../connection');
4  const router = express.Router();
5
6  router.post("/login", (req, res) => {
7    const { Username, Password } = req.body;
8
9    // Check if the provided username and password match a user in the database
10   const query = "SELECT * FROM User WHERE Username = ?";
11   connection.query(query, [Username], (error, results) => {
12     if (error) {
13       console.error(error);
14       return res.status(500).json({ message: "Failed to login" });
15     }
16
17     if (results.length === 0) {
18       return res.status(401).json({ message: "Invalid username or password" });
19     }
20
21     const user = results[0];
22
23     // Compare the provided password with the stored hashed password
24     bcrypt.compare>Password, user.Password, (error, isMatch) => {
25       if (error) {
26         console.error(error);
27         return res.status(500).json({ message: "Failed to login" });
28       }
29
30       if (!isMatch) {

```

### 6.4.5. Error Handling and Middleware

- Implement error handling middleware to catch and handle errors that occur during API requests.

## 6.5. Stripe

- For Stripe: [Stripe | Payment Processing Platform for the Internet](#)

The screenshot displays a Stripe payment interface. On the left, a cart summary shows two items: 'shirt 1' (Qty 3, Brown) for ₫1,350,000 and 'test' (Qty 1) for ₫100,000. The subtotal is ₫1,450,000, and shipping is free. The total due is ₫1,450,000. On the right, the 'Pay with card' section includes a 'Shipping information' form with fields for Email, Name, Country (Vietnam), Address line 1, Address line 2, City, Province, and Postal code (091 234 56 78). Below this is a 'Shipping method' section with two options: 'Free shipping' (5-7 business days, Free) and 'Next day air' (1 business day, ₫50,000).

### 6.5.1. Sign up for a Stripe Account

- Visit the Stripe website at <https://stripe.com> and sign up for an account.
- Follow the verification process and complete the necessary steps to activate the account.

### 6.5.2. Install the Stripe Node.js Library

- In project directory, run the following command to install the Stripe library:  
`npm install stripe`

### 6.5.3. Set up Stripe API Keys

- Retrieve Stripe API keys from Stripe account dashboard.
- Store API keys and access them in backend code.

### 6.5.4. Implement Stripe Integration

- Use the Stripe Node.js library to handle payment processing within the application.



- Follow the Stripe API documentation to implement functionalities such as creating charges, managing customers, and handling webhooks.

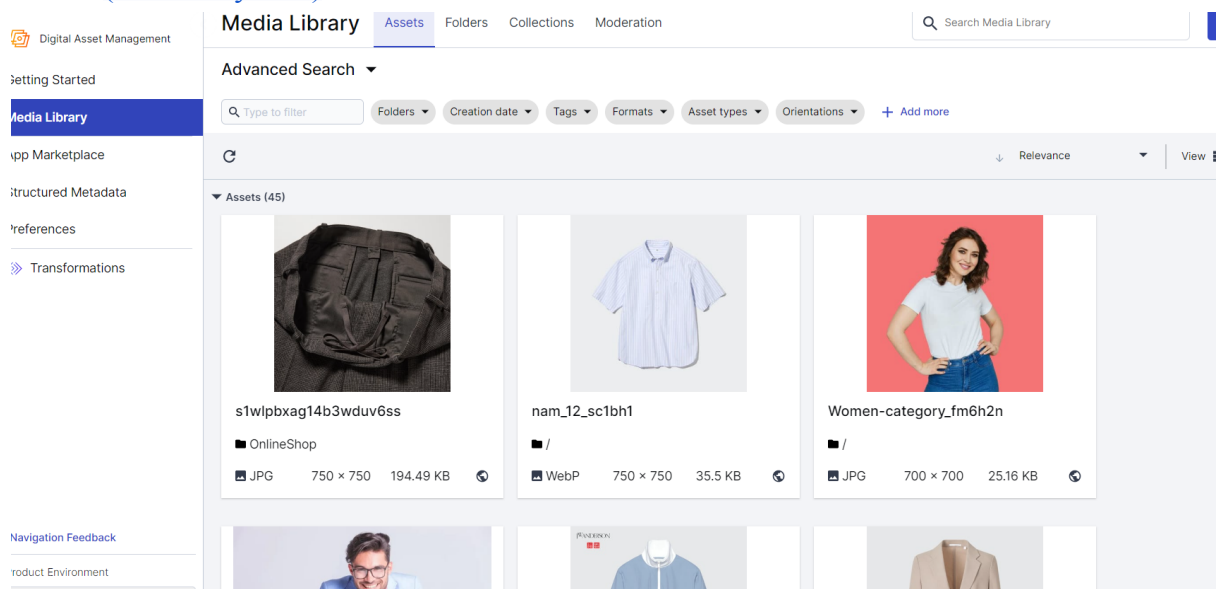
```

1 const express = require("express");
2 const router = express.Router();
3 const connection = require("../connection");
4 const util = require("util");
5 const queryPromise = util.promisify(connection.query).bind(connection);
6 const stripe = require("stripe")
7   ("sk_test_51NAV7gIu8TWD2EDaCDoCplcoP03rOug1aVumex3l7oqaHVkYNBHD85Stq9Z10fuz05w1x4hu2YbYtT0rs10DhKsU00z6lwn8S4");
8 );
9 const bodyParser = require("body-parser");
10 router.use(bodyParser.raw({ type: "application/json" }));
11
12 router.post("/create-checkout-session", async (req, res) => {
13   const customer = await stripe.customers.create({
14     metadata: {
15       userID: req.body.userID,
16       cart: JSON.stringify(req.body.cartItems),
17     },
18   });
19
20   const line_items = req.body.cartItems.map((item) => {
21     return {
22       price_data: {
23         currency: "vnd",
24         product_data: {
25           name: item.Name,
26           images: [item.Image],
27           description: item.Description,
28           metadata: {
29             id: item.ProductID,
30           },

```

## 6.6. Cloundinary

- For Cloundinary: [Image and Video Upload, Storage, Optimization and CDN \(cloudinary.com\)](https://cloudinary.com)



### 6.6.1. Sign up for a Cloundinary Account

- Visit the Cloudinary website at <https://cloudinary.com> and sign up for an account.
- Complete the registration process and create a new cloudinary project.

### 6.6.2. Install the Cloudinary Node.js SDK

- In project directory, run the following command to install the Cloudinary SDK:  
npm install cloudinary

### 6.6.3. Set up Cloudinary API Credentials

- Retrieve Cloudinary API credentials from Cloudinary account dashboard.
- Store API credentials securely, and access them in backend code.

### 6.6.4. Implement Cloudinary Integration

- Use the Cloudinary Node.js SDK to upload, manage, and serve media assets within the application.
- Follow the Cloudinary API documentation to implement functionalities such as uploading images, generating transformations, and handling media assets.

```
// Create a new product
router.post('/products', async (req, res) => {
  const { name, price, description, image } = req.body;

  if (image) {
    try {
      const uploadRes = await cloudinary.uploader.upload(image, {
        upload_preset: 'OnlineShop'
      });

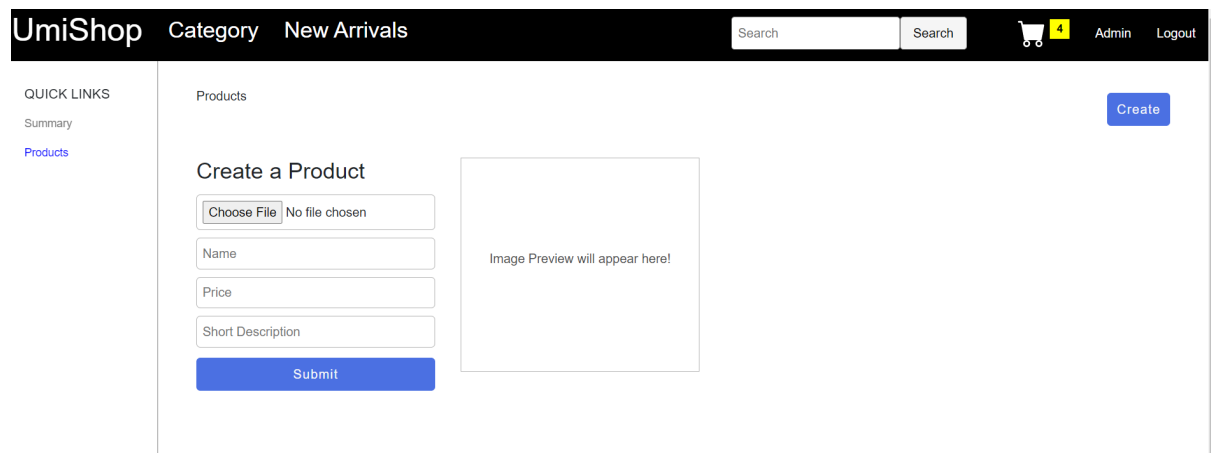
      const imageUrl = uploadRes.secure_url;

      const query = 'INSERT INTO Product (Name, Price, Description, Image) VALUES (?, ?, ?, ?)';
      connection.query(query, [name, price, description, imageUrl], (error, results) => {
        if (error) {
          console.error('Error executing the query:', error);
          return res.status(500).send('Error executing the query');
        }

        res.json({ message: 'Product created successfully', productId: results.insertId });
      });
    } catch (error) {
      console.error('Error uploading the image:', error);
      return res.status(500).send('Error uploading the image');
    }
  } else {
    const query = 'INSERT INTO Product (Name, Price, Description) VALUES (?, ?, ?)';
    connection.query(query, [name, price, description], (error, results) => {
      if (error) {
        console.error('Error executing the query:', error);
        return res.status(500).send('Error executing the query');
      }

      res.json({ message: 'Product created successfully', productId: results.insertId });
    });
  }
});
```

## 6.7. Admin Dashboards



### 6.7.1. Design the Admin Dashboard

- Plan the layout and features of the admin dashboard.
- Consider using UI libraries or frameworks (e.g., React Admin, Material-UI) to expedite the development process.

### 6.7.2. Implement Admin Routes

- Create separate routes and components specifically for the admin dashboard.
- Secure these routes to ensure only authorized users can access the admin features.

### 6.7.3. Build Admin Functionality

- Implement CRUD operations for managing various data entities within your application.
- Create forms, tables, and other UI components to facilitate data management.

### 6.7.4. Apply Security Measures

- Implement authentication and authorization mechanisms to ensure only authorized administrators can access and modify data.
- Set up role-based access control (RBAC) to manage user permissions and restrict access to sensitive functionalities.

## 6.8. Nodemailer

### 6.8.1. Installation and Setup

- To begin using Nodemailer, we need to install it as a dependency in our Node.js project. This can be done by running the following command:  
`npm install nodemailer`

- Once installed, we can import the Nodemailer module in our application using the require statement

```
const nodemailer = require('nodemailer');
```

### 6.8.2. Create a Transporter

To send emails using Nodemailer, we need to create a transporter object. The transporter specifies the configuration for the email service provider (e.g., Gmail, Outlook). Here's an example of creating a transporter for a Gmail account

```
const transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: 'your-email@gmail.com',
    pass: 'your-password'
  }
});
```

In the above code, replace 'your-email@gmail.com' with your Gmail email address and 'your-password' with the corresponding password. Make sure to use a dedicated email account for sending application-related emails.

### 6.8.3. Create a Transporter

Once the transporter is created, we can use it to send emails. Nodemailer provides various methods and options to customize the email content, recipients, attachments, and more.

```
// Construct the verification link
const verificationLink = `http://localhost:5000/verify/${verificationToken}`;

const mailOptions = {
  from: "duyvudrive01@gmail.com",
  to: Email,
  subject: "Account Verification",
  text: "Please click the link to verify your account.",
  html: `<p>Please click the link to verify your account: <a href="${verificationLink}">Verify Account</a></p>`,
};

transporter.sendMail(mailOptions, (error, info) => {
  if (error) {
    console.error("Error sending verification email:", error);
    return res
      .status(500)
      .json({ message: "Failed to send verification email" });
  }

  console.log("Verification email sent:", info.response);
  return res.status(200).json({
    message: "Registration successful. Verification email sent.",
  });
});
```

#### 6.8.4. Verification

Upon registration, an email to verify the account will be sent to the email address registered. User needs to click verify to complete the registration.

```
router.get("/verify/:token", async (req, res) => {
  const { token } = req.params;

  const selectQuery = "SELECT * FROM User WHERE verificationToken = ?";
  connection.query(selectQuery, [token], (error, results) => {
    if (error) {
      console.error(error);
      return res.status(500).json({ message: "Failed to verify account" });
    }

    if (results.length === 0) {
      return res.status(400).json({ message: "Invalid verification token" });
    }

    const user = results[0];

    const updateQuery = "UPDATE User SET isVerified = 1 WHERE UserID = ?";
    connection.query(updateQuery, [user.UserID], (updateError) => {
      if (updateError) {
        console.error(updateError);
        return res.status(500).json({ message: "Failed to verify account" });
      }

      // Send an HTML response with a delay of 2 seconds
      setTimeout(() => {
        res.status(200).json({ message: "Account verified successfully" });
      }, 2000);
    });
  });
});
```

## 6.9. Project Github

Github: [DuyVu285/E-Commerce-Website-For-Clothes \(github.com\)](https://github.com/DuyVu285/E-Commerce-Website-For-Clothes).

## Chapter 7: Setup and Run Guide

This chapter will introduce the necessary steps for the setup and run of a MERN (MySQL, Express, React, Node.js) project in Visual Studio Code (VSC).

## 7.1. Prepare the Environments

1. **Install Node.js:** Visit the official Node.js website (<https://nodejs.org>) and download the latest LTS (Long Term Support) version for your operating system. Follow the installation instructions provided.

2. **Install MySQL:** Download and install MySQL from the official website (<https://www.mysql.com/downloads/>). Follow the instructions specific to your operating system.

## 7.2. Get the Project from GitHub

1. **Install Git:** Download and install Git from the official website (<https://git-scm.com/downloads>) if you don't have it already. Follow the instructions specific to your operating system.
2. **Clone the GitHub repository:** Open a terminal or command prompt and navigate to the directory where you want to store the project. Run the following command to clone the repository:  
`git clone https://github.com/DuyVu285/E-Commerce-Website-For-Clothes`

## 7.3. Installation of MERN

1. **Open Visual Studio Code:** Launch Visual Studio Code on your system.
2. **Open the project folder:** In Visual Studio Code, go to "File" -> "Open Folder" and select the folder where you cloned the GitHub repository.
3. **Install backend dependencies:** Open a new terminal in Visual Studio Code (press ``Ctrl` + `Shift` + ``` or go to "View" -> "Terminal"). Change the directory to the backend folder of your project using the ``cd`` command:  
`cd backend`
4. **Install the necessary Node.js dependencies** for the backend:  
`npm install`
5. **Configure the database:** Open the backend project folder in Visual Studio Code's file explorer. Look for a configuration file that specifies the database connection settings (`connection.js`). Modify the configuration file to provide the correct database connection details such as the host, port, username, and password. Also, run the database SQL file in database folder to setup the database in MySQL
6. **Install frontend dependencies:** Open a new terminal in Visual Studio Code. Change the directory to the frontend folder of your project using the ``cd`` command:  
`cd frontend`
7. **Install the necessary Node.js dependencies** for the frontend:  
`npm install`
8. **Start the project:** In the terminal, make sure you are still in the frontend folder. Run the following command to start the frontend development server:  
`npm start`
9. In a separate terminal, **navigate to the backend folder:**  
`cd ../backend`
10. Run the following command to **start the backend server using Nodemon** (a utility that automatically restarts the server when changes are made):  
`nodemon`

11. Go to stripe (after stripe registration) [Dashboard – New Business – Stripe \[Test\]](#), and perform all the necessary steps in the backend terminal where you type nodemon. If the backend displays Webhooks verified, this shows that the connection to Stripe is completed. All the successful payments are stored in the order table in the database.

The frontend development server will be running on a specified port (port 3000) and the backend server will be running on a different port (port 5000). Access localhost:3000 to see and use the frontend site and localhost:5000 to view the database API.

## Chapter 8: Conclusion

### 8.1. Summary of the project

Throughout this project, we embarked on the development of an e-commerce website using the MERN stack, which consists of MySQL, Express, React, and Node.js. Our objective was to create a robust and user-friendly platform that facilitates online transactions and provides an engaging shopping experience for customers.

Starting with the backend, we utilized Node.js and Express to build a scalable and efficient server. By integrating MySQL as our database management system, we were able to store and retrieve data related to products, user information, and transactions. This allowed us to create a solid foundation for the website's functionality.

On the frontend, we employed React, a popular JavaScript library, to design the user interface and handle the dynamic rendering of components. This enabled us to build a responsive and interactive website, providing seamless navigation and an immersive shopping experience for our users.

Key features of the e-commerce website included product listings, shopping cart functionality, secure payment processing, user authentication and authorization. These features were implemented using various technologies from the MERN stack, allowing us to create a comprehensive and fully functional online shopping platform.

### 8.2. Challenges faced and lessons learned

Throughout the development process, we encountered several challenges and gained valuable insights that contributed to our growth as developers. Some of the challenges we faced included:

1. **Learning and Utilizing the MERN Stack:** As our team was relatively new to the MERN stack, we had to invest time in understanding and effectively utilizing each technology. This involved learning the nuances of Node.js, Express, React, and MySQL, and integrating them seamlessly to create a cohesive web application.
2. **Learning New Technologies:** In addition to the MERN stack, we had to learn and incorporate other technologies and libraries into our project. This included using payment gateways, implementing secure authentication and authorization mechanisms, and integrating third-party APIs for enhanced functionality.

3. **Developing an E-commerce Website:** Building a robust and user-friendly e-commerce website involves tackling various complexities, such as handling product catalogs, managing inventory, implementing shopping cart functionality, and securely processing payments. We had to ensure the website was responsive, intuitive, and optimized for performance.

From these challenges, we learned important lessons that will guide us in future projects:

1. **Continuous Learning:** Technology evolves rapidly, and it is crucial to stay updated with the latest advancements. By maintaining a mindset of continuous learning, we can adapt to new technologies and leverage them effectively in our projects.
2. **Effective Collaboration:** Building a complex web application requires collaboration and effective communication within the development team. Regular meetings, clear task allocation, and maintaining documentation helped us stay organized and meet project goals efficiently.
3. **User-Centric Approach:** Throughout the development process, we realized the importance of focusing on the needs and preferences of our users. Regular user feedback and usability testing helped us improve the website's design and functionality, ensuring a positive user experience.

### 8.3. Possible future improvements or extensions

While we have successfully developed a functional e-commerce website using the MERN stack, there are several potential areas for future improvements and extensions. Some of the possibilities include:

1. **Improved User Profile Editing:** Implementing a user profile editing feature would allow users to update their personal information, manage saved addresses, and modify their preferences.
2. **Improved Login and Registration Process:** Enhancing the login and registration process with additional security measures, such as two-factor authentication or social media integration, can provide users with a more secure and convenient experience.
3. **Search Optimization:** Enhancing the search functionality to provide more accurate and relevant results can improve the overall user experience. This could involve implementing advanced search algorithms, autocomplete suggestions, and filters for refined search results.
4. **Footer:** Designing and incorporating a footer section on the website can provide important links, contact information, and additional navigation options, ensuring easy access to essential pages.
5. **Branding:** Developing a distinct brand identity for the e-commerce website, including a logo, color scheme, and visual elements, can help establish a strong brand presence and enhance user recognition.

By addressing these potential areas for improvement, the e-commerce website can be further enhanced to meet evolving user expectations and industry standards.

In conclusion, the development of the MERN-based e-commerce website was a challenging yet rewarding experience. We successfully utilized the MERN stack, learned new



technologies, and developed a fully functional e-commerce platform. The project taught us valuable lessons in utilizing the MERN stack effectively, learning new technologies, and building an e-commerce website. Moving forward, the identified future improvements and extensions provide a roadmap for enhancing the website's functionality, user experience, and overall success.

## References

- For VSC: [Visual Studio Code - Code Editing. Redefined](#)
- For Node.js: [Node.js \(nodejs.org\)](#)
- For React: [React](#)
- For Express: [Express - Node.js web application framework \(expressjs.com\)](#)
- For MySQL: [MySQL](#)
- For Stripe: [Stripe | Payment Processing Platform for the Internet](#)
- For Cloudinary: [Image and Video Upload, Storage, Optimization and CDN \(cloudinary.com\)](#)
- For Nodemailer: [Nodemailer :: Nodemailer](#)
- Diagrams: [Use Case Diagram](#)
- Github: [DuyVu285/E-Commerce-Website-For-Clothes \(github.com\)](#)