***Naan Mudhalavan***

***Phase - 4***

***E-Commerce platform with IBM foundry***

***Team Member :***

1. Naranther M
2. Subhash M
3. Sham P
4. Vaikundaraja N

***Problem Statement :***

Creating an e-commerce platform with user authentication, a shopping cart, and checkout functionality is a complex project. Here's a high-level overview of the steps involved:

***E - Commerce Application :***

1. \*\*Backend Development\*\*:

- Choose a backend technology like Node.js or Python for server development.

- Set up a web server to handle HTTP requests and responses.

- Implement user registration and authentication using libraries like Passport.js (Node.js) or Django (Python).

2. \*\*Database Setup\*\*:

- Create a database to store user information, product details, and shopping cart data. You can use databases like MySQL, PostgreSQL, or MongoDB.

3. \*\*User Registration and Authentication\*\*:

- Create endpoints for user registration, login, and logout.

- Use secure password hashing for storing user passwords.

- Implement session management or token-based authentication for user sessions.

4. \*\*Product Catalog\*\*:

- Set up a database table for products and populate it with product details.

- Create endpoints to retrieve and display product information.

5. \*\*Shopping Cart\*\*:

- Develop endpoints to add, update, and remove items from the shopping cart.

- Store the cart data in the database or session.

- Calculate the total price of items in the cart.

6. \*\*Checkout Process\*\*:

- Create a secure endpoint for initiating the checkout process.

- Collect user shipping and payment information.

- Validate and process payments using payment gateways (e.g., Stripe, PayPal).

- Generate an order confirmation and store it in the database.

7. \*\*User Interface\*\*:

- Develop a frontend using HTML, CSS, and JavaScript (or a frontend framework like React or Vue.js).

- Create user interfaces for registration, login, product listings, shopping cart, and checkout.

- Connect the frontend to the backend using API requests.

8. \*\*Security\*\*:

- Implement security measures like input validation, XSS and CSRF protection, and HTTPS for data transmission.

- Regularly update and patch dependencies to protect against vulnerabilities.

9. \*\*Testing\*\*:

- Test the entire system for functionality and security.

- Perform unit, integration, and end-to-end testing.

- Address any bugs or issues that arise.

10. \*\*Deployment\*\*:

- Choose a hosting provider or server for deployment (e.g., AWS, Heroku).

- Deploy the backend and database.

- Configure the domain and SSL certificate for secure access.

11. \*\*Scaling and Optimization\*\*:

- Monitor the platform's performance and optimize code and database queries as needed.

- Implement caching mechanisms to reduce server load.

- Plan for scalability as your user base grows.

12. \*\*Maintenance\*\*:

- Regularly update and maintain the platform, addressing security patches and feature enhancements.

This is a high-level overview, and each step can be quite complex. You may need to break down the development into smaller tasks and work on them incrementally. Additionally, consider consulting with experienced developers and security experts to ensure your e-commerce platform is robust and secure.

***PROGRAM :***

const express = require('express');

const { Pool } = require('pg');

const app = express();

const port = process.env.PORT || 3000;

// PostgreSQL configuration

const pool = new Pool({

user: 'your\_username',

host: 'your\_host',

database: 'your\_database',

password: 'your\_password',

port: 5432,

});

// Database schema creation function

const createTables = async () => {

const createProductsTable = `CREATE TABLE IF NOT EXISTS products (

product\_id SERIAL PRIMARY KEY,

product\_name VARCHAR(255) NOT NULL,

description TEXT,

price DECIMAL,

image\_url TEXT,

category\_id INT

);`;

const createCategoriesTable = `CREATE TABLE IF NOT EXISTS categories (

category\_id SERIAL PRIMARY KEY,

category\_name VARCHAR(255) NOT NULL

);`;

const createUsersTable = `CREATE TABLE IF NOT EXISTS users (

user\_id SERIAL PRIMARY KEY,

username VARCHAR(255) NOT NULL,

password VARCHAR(255) NOT NULL,

email VARCHAR(255) NOT NULL

);`;

const createOrdersTable = `CREATE TABLE IF NOT EXISTS orders (

order\_id SERIAL PRIMARY KEY,

user\_id INT,

product\_id INT,

quantity INT,

total\_price DECIMAL,order\_date DATE,

FOREIGN KEY (user\_id) REFERENCES users(user\_id),

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

);`;

try {

await pool.query(createProductsTable);

await pool.query(createCategoriesTable);

await pool.query(createUsersTable);

await pool.query(createOrdersTable);

} catch (error) {

console.error('Error creating tables', error);

}

};

app.use(express.json());

// User registration endpoint

app.post('/register', async (req, res) => {

try {

const { username, password, email } = req.body;

const insertUserQuery = 'INSERT INTO users (username, password, email) VALUES ($1, $2,

$3)';

await pool.query(insertUserQuery, [username, password, email]);

res.status(201).send('User registered successfully');

} catch (error) {

console.error('Error registering user', error);

res.status(500).send('Internal Server Error');

}

});

// User login endpoint

app.post('/login', async (req, res) => {

try {

const { username, password } = req.body;

const userQuery = 'SELECT \* FROM users WHERE username = $1 AND password = $2';

const { rows } = await pool.query(userQuery, [username, password]);

if (rows.length === 1) {

res.status(200).send('Login successful');

} else {

res.status(401).send('Invalid credentials');

}

} catch (error) {

console.error('Error during login', error);res.status(500).send('Internal Server Error');

}

});

// Add to cart endpoint

app.post('/cart/add', async (req, res) => {

try {

const { userId, productId, quantity } = req.body;

// Implement shopping cart functionality here

// You need to manage user carts and quantities

res.status(200).send('Product added to cart successfully');

} catch (error) {

console.error('Error adding to cart', error);

res.status(500).send('Internal Server Error');

}

});

// Remove from cart endpoint

app.post('/cart/remove', async (req, res) => {

try {

const { userId, productId } = req.body;

// Implement shopping cart functionality here

// Remove products from the user's cart

res.status(200).send('Product removed from cart successfully');

} catch (error) {

console.error('Error removing from cart', error);

res.status(500).send('Internal Server Error');

}

});

// Checkout endpoint

app.post('/checkout', async (req, res) => {

try {

const { userId, products, totalPrice } = req.body;

// Implement the checkout process, including payment handling

// Create an order entry and update product quantities

res.status(200).send('Checkout successful');

} catch (error) {

console.error('Error during checkout', error);

res.status(500).send('Internal Server Error');

}

});

// Endpoint to fetch all productsapp.get('/products', async (req, res) => {

try {

const { rows } = await pool.query('SELECT \* FROM products');

res.json(rows);

} catch (error) {

console.error('Error executing query', error);

res.status(500).send('Internal Server Error');

}

});

app.listen(port, async () => {

console.log(`Server is running on port ${port}`);

await createTables();

});

***THANKING YOU***