VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

BCSE301P - Software Engineering Lab

Assessment - 1

VIRTUAL GROCERY STORE

Submitted to –

Dr. Murugan K

Submitted by –

Shreeji Chandgotia (21BCE3988)

Ishita Prakash (21BCE3995)

Suraj Kumar (21BKT0065)

Q1. Analysis and identification of the suitable life cycle models.

For the Virtual Grocery Store project, adopting the Agile software development process model would be the best option.

Reasons:

- Iterative Development: Agile allows for incremental development and frequent releases, which aligns with the need for continuous improvement and adaptation in the digital landscape.

- Customer Collaboration: Agile emphasizes customer collaboration and feedback, ensuring that the Virtual Grocery Store meets the diverse needs of users, including seniors, busy individuals, and those seeking convenience.

- Flexibility: Agile provides flexibility to accommodate changing requirements and priorities, enabling the project team to respond quickly to market demands and user feedback.

- Quality Focus: Agile encourages a focus on delivering high-quality software through continuous testing and refinement, which is essential for ensuring the reliability and usability of the Virtual Grocery Store platform.

Fig. Agile Model

Reasons why other software process models may not be as suitable for the Virtual Grocery Store project:

i. **Waterfall Model:** The Waterfall model involves a linear, sequential approach to development, which may not be conducive to the dynamic nature of the project. It lacks the flexibility to adapt to changing requirements and customer feedback.

ii. **Spiral Model:** While the Spiral model allows for iterative development and risk management, it may involve longer development cycles and higher costs, which could be less suitable for a project aiming for efficiency and cost-effectiveness like the Virtual Grocery Store.

iii. **V-Model**: The V-Model emphasizes the verification and validation of requirements throughout the development lifecycle. However, it may not provide the same level of customer collaboration and responsiveness as Agile, which is crucial for meeting the needs of diverse users in the digital landscape.

iv. **Incremental Model**: The Incremental model involves delivering the system in small, manageable increments, with each increment adding new functionality. While this aligns with the iterative approach favored by Agile, the Incremental model may lack the emphasis on customer collaboration and flexibility that Agile provides. Additionally, the Incremental model may not prioritize the early delivery of value to users, which is crucial for a project like the Virtual Grocery Store aiming to transform the shopping experience.

v. **Rapid Application Development (RAD) Model:** RAD's reliance on active user involvement may be challenging to sustain, especially with diverse user groups like senior citizens or those with limited technological familiarity, potentially hindering effective feedback gathering. The resource-intensive nature of RAD, necessitating skilled developers and stakeholders' time, could pose logistical hurdles in coordinating their participation across rapid development cycles.

vi. **Prototyping Model:** Prototypes might not fully capture system requirements, risking misunderstandings between stakeholders' expectations and the final product, leading to delays and rework. Secondly, prototyping can induce feature creep, expanding the project scope beyond its intended boundaries, which may compromise deadlines and budget constraints. Lastly, prototypes may not accurately reflect the technical intricacies and scalability demands of the Virtual Grocery Store platform.

Q.2 Analyse the software requirements of your Software project.

SOFTWARE REQUIREMENTS

1. FUNCTIONAL REQUIREMENTS

- **Verify user** – This is a system-based verification procedure where data such as password and username are cross verified with previously mentioned details.
- **Show Grocery Items** – This functionality shows all the items available on our database will be shown to the customer, and then can place order for it accordingly.
- **Add To Cart** – Using this feature the user can add items to the cart which might be led by the following functionalities.
- **Delete From Cart** – User can use this functionality to delete items from cart.
- **Show Cart** – User can use this to view all the items present in the cart.
- **Payment** – User gets an option to pay online using various cards and UPI based applications. User also gets the facility to pay during the delivery by using CASH ON DELIVERY facility.
- **Takes User's Address** - User gets a form to fill in their shipping address and confirm their booking.
- **Tracking Order** – The delivery of the order is initiated and can be tracked.
- **User Feedback** - user feedback is taken after delivery.

2. NON-FUNCTIONAL REQUIREMENTS

- **Performance**: Ensure system performance meets acceptable levels during peak usage.
- **Security**: Implement measures to safeguard user data and transactions.
- **Portability**: Ensure the system can be easily transferred or adapted to different environments.
- **Reliability**: Ensure the system functions reliably without unexpected downtime or errors.
- **Scalability**: Ensure the system can accommodate increasing numbers of users and transactions.

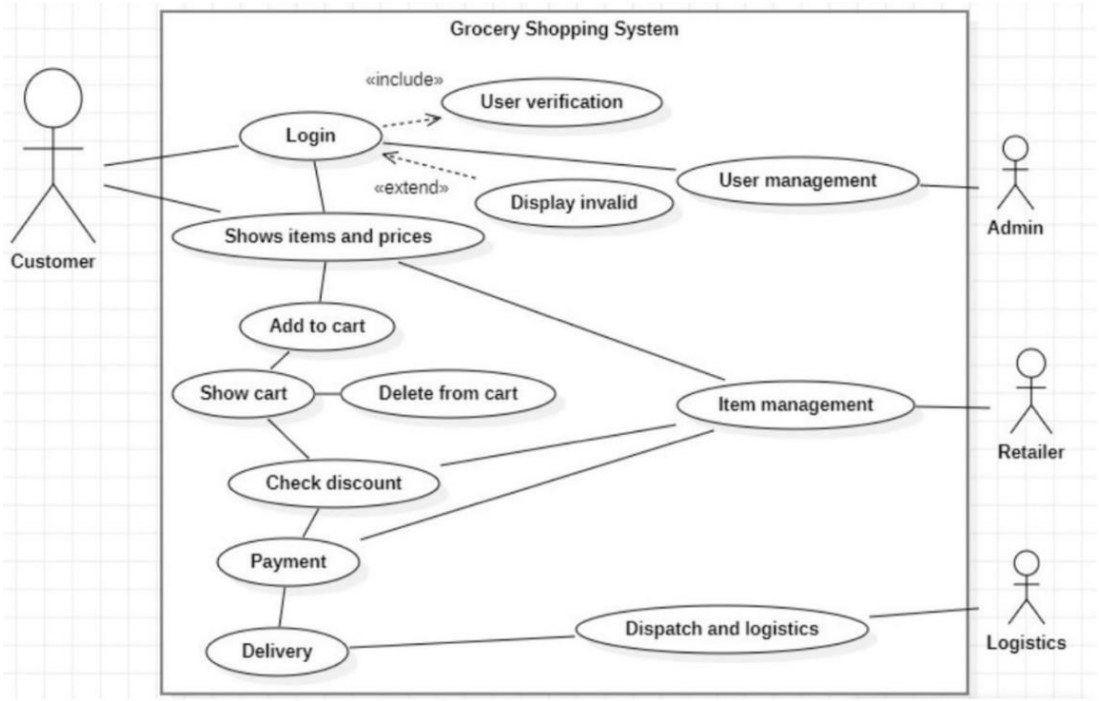
3. SOFTWARE QUALITY ATTRIBUTES

- **Availability**: Ensure orders are available as scheduled, catering to advance reservations.
- **Correctness**: Ensure orders start from the correct terminal and reach the intended destination accurately.
- **Maintainability**: Admins should effectively manage order schedules to maintain accuracy.
- **Usability**: Ensure order schedules meet the needs of a wide range of customers effectively.

SOFTWARE TOOL

React is a versatile JavaScript library perfect for developing the frontend of Virtual Grocery Store. Its component-based approach allows for creating dynamic interfaces with real-time updates and interactive features like product previews. With built-in state management, integration with backend services, and testing tools, React is well-suited for building reliable and scalable grocery store applications, enabling developers to deliver smooth shopping experiences and streamline operations for vendors and administrators.

USE CASE DIAGRAM



VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

BCSE301P - Software Engineering Lab

Assessment - 2

VIRTUAL GROCERY STORE

Submitted to –

Dr. Murugan K

Submitted by –

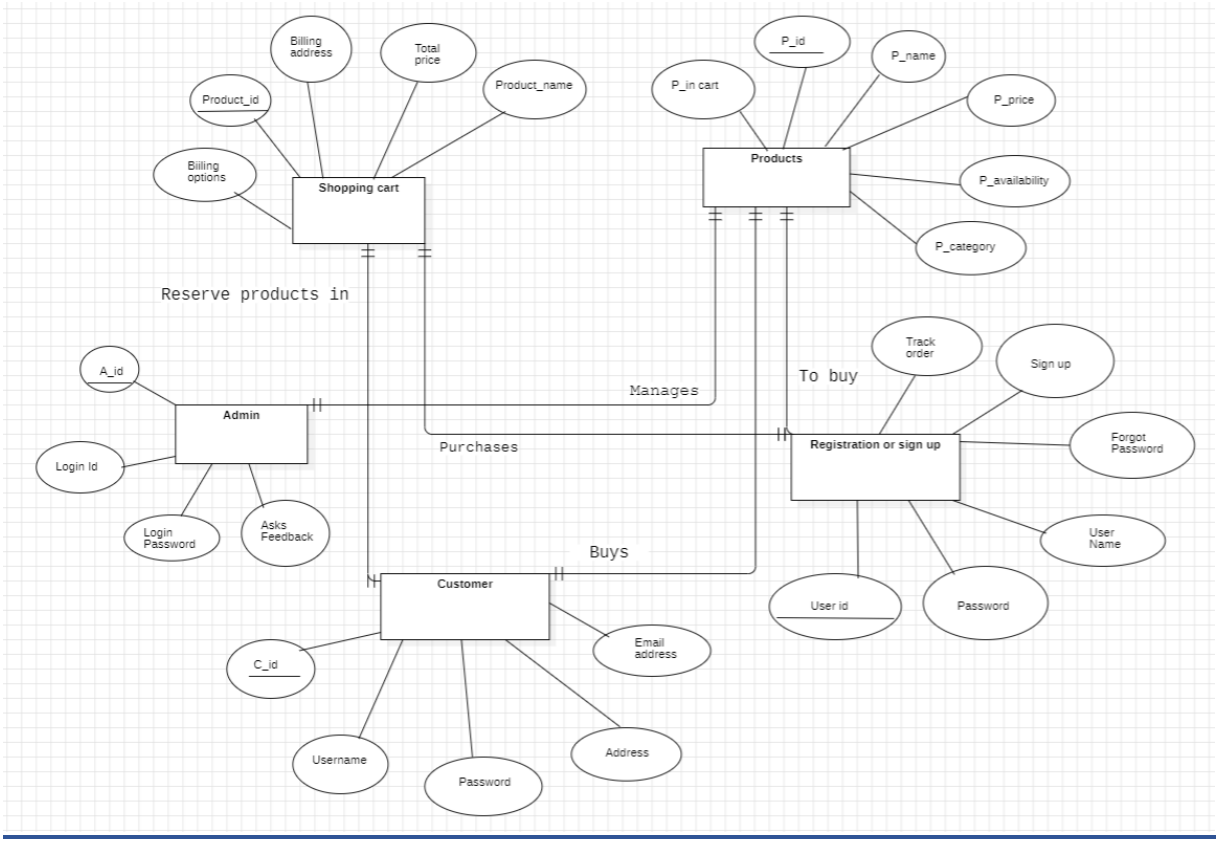
Shreeji Chandgotia (21BCE3988)

Ishita Prakash (21BCE3995)

Suraj Kumar (21BKT0065)

Q1. Requirement modelling using Entity Relationship Diagram (Structural Modelling)

ER DIAGRAM



Entities

- **Customers:** This entity represents the individuals who use the web/mobile application to shop for groceries. Attributes could include Customer ID, Name, Email, Address, etc.

- **Registration or Login:** This entity represents the process of registering a new account or logging into an existing account. It could include attributes like Username, Password, Registration Date, Last Login, Track order etc.

- **Products:** This entity represents the various products available for purchase in the grocery store. Attributes could include Product ID, Name, Description, Price, Quantity, etc.

- **Shopping Cart:** This entity represents the temporary storage of products that customers intend to purchase during their shopping session. It could have attributes like Cart ID, Customer ID (to link to the customer who owns the cart), Total Price, etc.

- **Admin:** This entity represents administrative users who manage the virtual grocery store. Attributes could include Admin ID, Username, Password, receives feedback etc. Admins might have relationships with other entities for managing products, customers, etc.

Relationships:

- **Admin-Products:** This is a one-to-many relationship where one admin can manage multiple products. Each product is managed by only one admin. This means that an admin can be associated with many products, but each product is associated with only one admin.

- **Customer-Product:** This is a one-to-many relationship as well. A customer can buy multiple products in a single transaction. However, each product can be bought by many customers. So, it's a many-to-many relationship, but when we consider a single transaction, it becomes a one-to-many relationship.

- **Customer-Shopping Cart:** This is a one-to-many relationship. A customer can reserve multiple products in their shopping cart, but each product can only be reserved by one customer at a time.

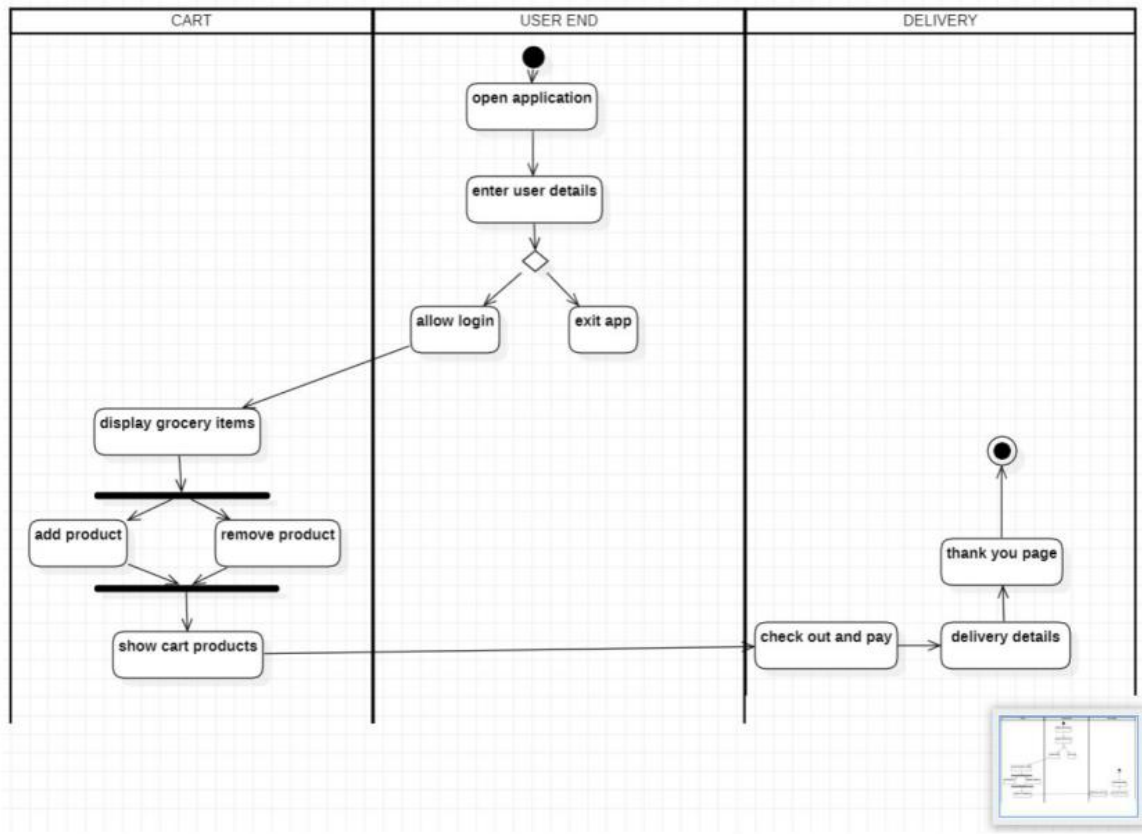
- **Product-Sign up:** This is a one-to-one relationship. Each customer account (sign up) is associated with only one customer. However, a customer can have multiple purchases associated with their account, which makes it a one-to-many relationship in terms of purchases.

- **Sign up-Shopping Cart:** This is also a one-to-one relationship. Each customer account (sign up) is associated with only one customer. Similarly, a customer can have multiple shopping carts (as in abandoned carts), but at any given time, they can only proceed with purchasing from one shopping cart, making it a one-to-one relationship in practice.

Q.2 Requirement modeling using Context flow diagram, DFD (Functional Modelling)

CONTEXT FLOW DIAGRAM:

Activity Diagram:

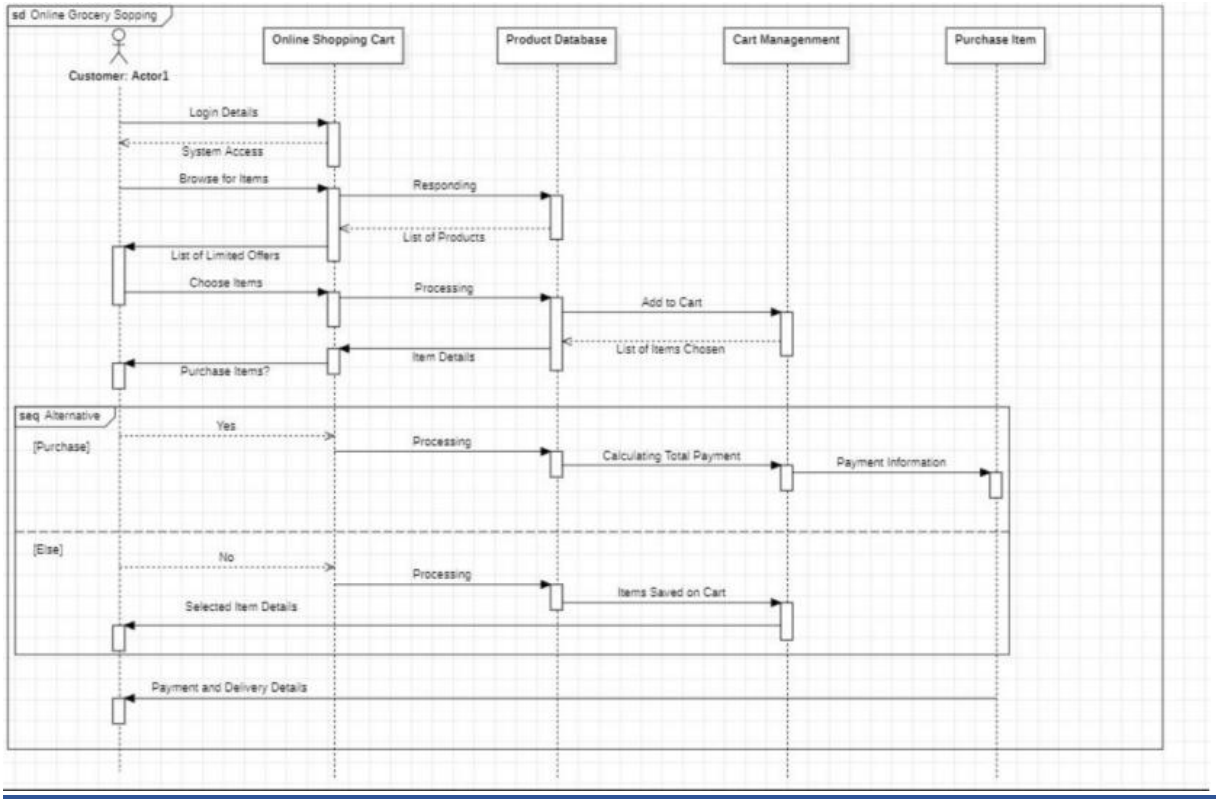


The activity diagram for the Virtual Grocery Store project depicts the flow of actions across three main segments: **Cart, User End, and Delivery.**

In the Cart section, users can view grocery items, add or remove products from their cart, and view the contents of their cart. The User End segment involves opening the application, entering user details, logging in, and exiting the app. Finally, the Delivery part encompasses activities such as navigating the checkout page, entering delivery details, and reaching the thank you page after completing the purchase.

These activities encapsulate the core functionalities of the virtual grocery store software, providing a structured overview of the user journey from product selection to checkout and delivery confirmation.

Sequence Diagram:



A sequence diagram for online shopping illustrates the interactions between various components such as the online shopping cart, product database, cart management system, and purchase process.

1. Online Shopping Cart:

The sequence diagram begins with the user interacting with the online shopping cart interface. This interaction may include actions like adding items to the cart, removing items, or viewing the contents of the cart. The cart interface sends requests to the cart management system to handle these operations.

2. Product Database:

As the user interacts with the cart, the cart management system communicates with the product database to retrieve information about the products. This includes retrieving product details, such as name, price, and availability, to display to the user in the cart interface.

3. Cart Management:

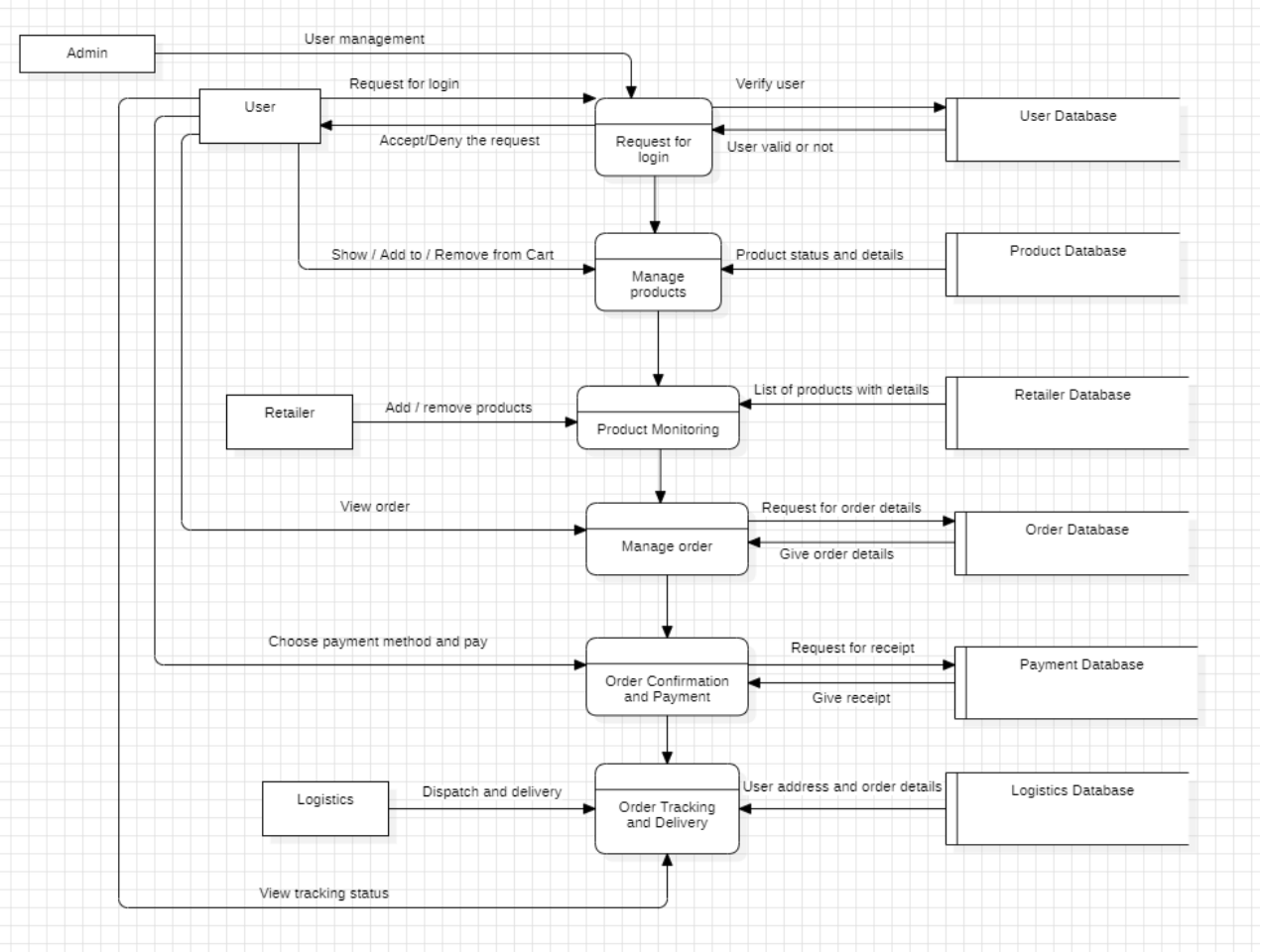
The cart management system processes the user's actions on the cart, such as adding or removing items. It also manages the state of the cart, ensuring that it reflects the user's selections accurately. The cart management system communicates with both the online shopping cart interface and the product database to coordinate these operations.

4. Purchase Items:

Once the user has finished selecting items and proceeds to purchase, the sequence diagram depicts the steps involved in completing the purchase. This may include the user confirming their selections, entering payment information, and submitting the order. The cart management system interacts with the product database to ensure that all selected items are available and updates the database to reflect the purchase. Finally, the system confirms the purchase and provides feedback to the user, completing the sequence.

Overall, the sequence diagram provides a visual representation of the flow of interactions between the components involved in the online shopping process, including the online shopping cart, product database, cart management system, and purchase process.

DATA FLOW DIAGRAM



The data flow diagram illustrates how data flows between the external entities, processes, and data stores within the virtual grocery store system. This representation helps understand the interactions and dependencies among different components of the system.

External Entities:

1. Admin: Responsible for managing the system.
2. User: Interacts with the system to make purchases.
3. Retailer: Manages product inventory and information.
4. Logistics: Handles order tracking and delivery.

Processes:

1. Request for Login: Allows users and administrators to access the system securely.
2. Manage Products: Enables retailers to add, update, or remove products from the inventory.
3. Product Monitoring: Monitors product availability, stock levels, and other related information.
4. Manage Order: Handles order processing, including order placement, modification, and cancellation.
5. Order Confirmation and Payment: Confirms orders and facilitates payment processing.
6. Order Tracking and Delivery: Tracks order status and manages the delivery process.

Data Stores:

1. User Database: Stores user information, including login credentials and order history.
2. Product Database: Contains information about available products, such as name, description, and price.
3. Retailer Database: Stores retailer-specific data, such as product inventory and pricing.
4. Order Database: Maintains records of all orders placed by users.
5. Payment Database: Stores payment-related information, including transaction details.
6. Logistics Database: Stores data related to order tracking and delivery logistics.

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

BCSE301P - Software Engineering Lab

Assessment - 3

VIRTUAL GROCERY STORE

Submitted to –

Dr. Murugan K

Submitted by –

Shreeji Chandgotia (21BCE3988)

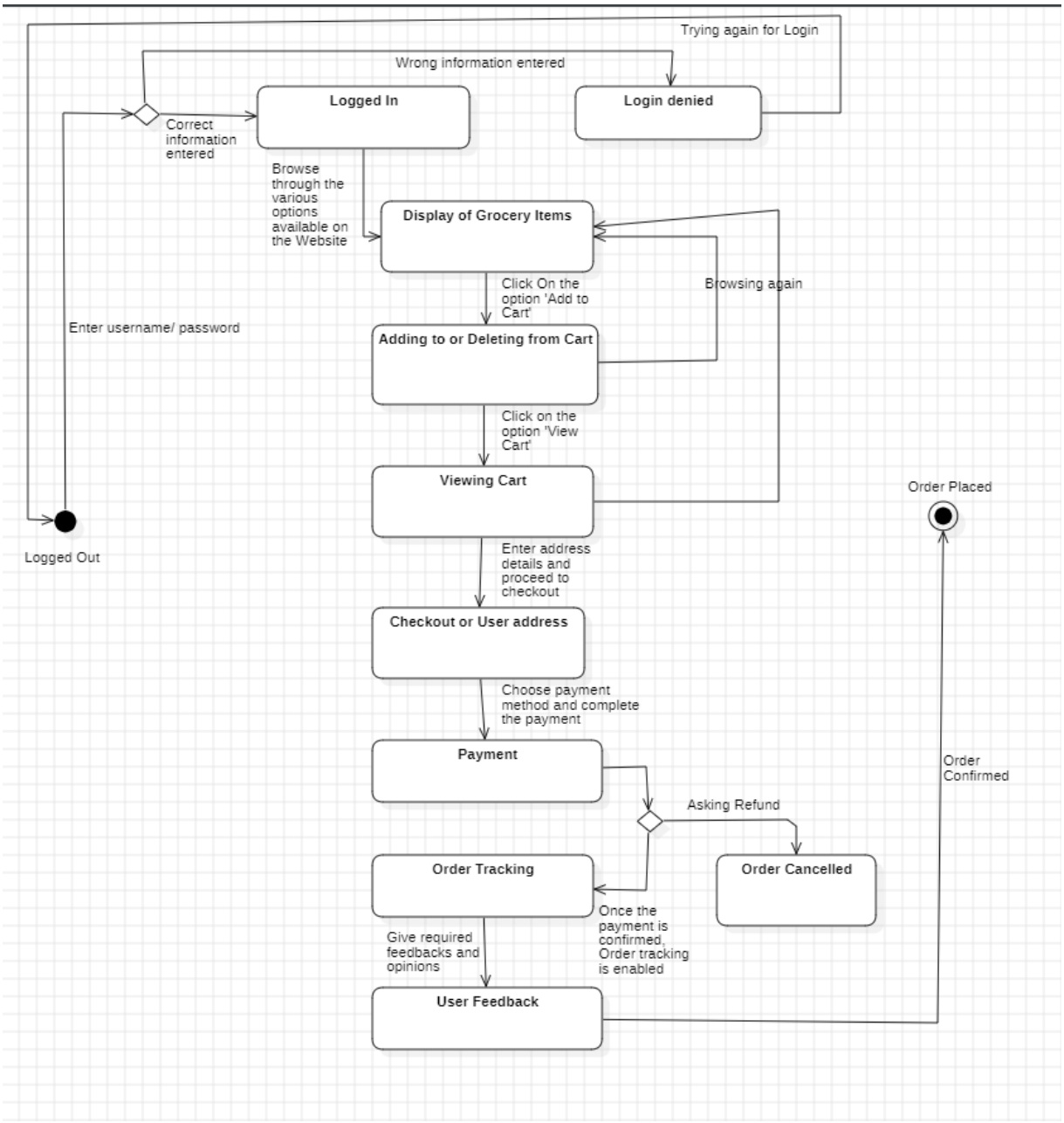
Ishita Prakash (21BCE3995)

Suraj Kumar (21BKT0065)

Q.1 Requirement modelling using State Transition Diagram (Behavioural Modelling).

STATE TRANSITION DIAGRAM:

In the case of the Virtual Grocery Store, we can use state transition diagrams to model the different states the system can be in and how it transitions between these states in response to user interactions or system events.



States:

- Logged Out: Initial state where the user is not authenticated.
- Logged In: State indicating that the user is successfully authenticated.
- Login Denied: State indicating that the user's login attempt was unsuccessful.
- Display of Grocery Items: State where the available grocery items are displayed to the user.
- Adding to or Deleting from Cart: State representing the process of adding or deleting items from the shopping cart.
- Viewing Cart: State where the user can view the contents of their shopping cart.
- Checkout or User Address: State where the user provides their address information during the checkout process.
- Payment: State where the user proceeds to make payment for the selected items.
- Order Tracking: State where the user can track the status of their order.
- Order Cancelled: State indicating that the user has cancelled their order.
- User Feedback: State where the user can provide feedback after completing their order.
- Order Placed: Final state indicating that the order has been successfully placed.

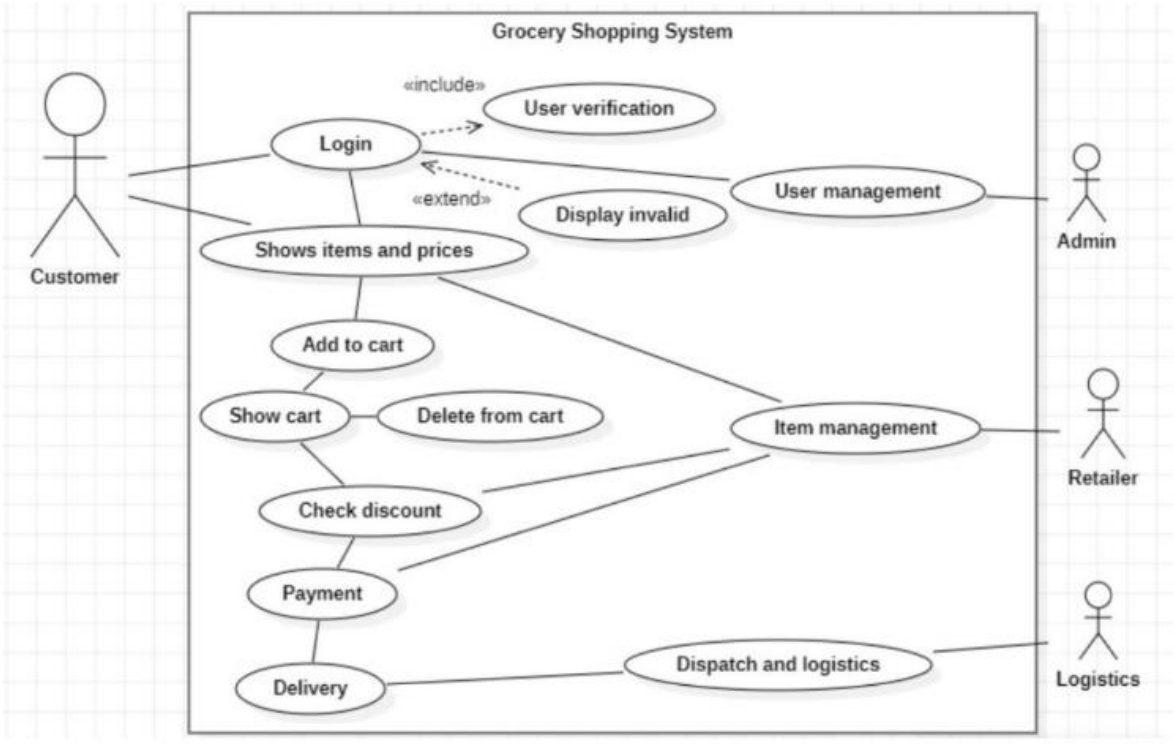
Actions:

- Login: Action triggered when the user attempts to log in to the system.
- Authenticate: Action triggered when the system verifies the user's credentials during the login process.
- Display Grocery Items: Action triggered to display the available grocery items to the user.
- Add to Cart: Action triggered when the user adds an item to their shopping cart.
- Delete from Cart: Action triggered when the user removes an item from their shopping cart.
- Proceed to Checkout: Action triggered when the user decides to proceed to the checkout process.
- Make Payment: Action triggered when the user initiates the payment process.
- Track Order: Action triggered when the user wants to track the status of their order.
- Cancel Order: Action triggered when the user cancels their order.
- Provide Feedback: Action triggered when the user provides feedback after completing their order.

Q.2. OO (Object Oriented) design.

1. USE CASE MODEL:

The use case diagram illustrates the various interactions and functionalities within the Virtual Grocery Store system. Each use case represents a specific action or task that users or system components can perform. Actors are entities external to the system that interact with it, such as customers, admins, and the system itself.



1. **Customer:**
   - Actors: Customer
   - Description: Represents the user of the system who interacts with various functionalities.

2. **Login:**
   - Actors: Customer
   - Description: Allows the customer to authenticate themselves by logging into the system.

3. **User Verification:**
   - Actors: System
   - Description: Verifies the user's credentials during the login process to ensure secure access to the system.

4**. Display Invalid:**
- Actors: System
- Description: Displays a message to the customer if their login attempt is invalid.

5. **User Management:**
- Actors: Admin
- Description: Involves managing user accounts, including creation, modification, and deletion.

6. **Show Items and Prices:**
- Actors: Customer
- Description: Displays the available items along with their prices to the customer.

7. **Add to Cart:**
- Actors: Customer
- Description: Allows the customer to add selected items to their shopping cart for purchase.

8. **Delete from Cart:**
- Actors: Customer
- Description: Enables the customer to remove items from their shopping cart.

9. **Show Cart:**
- Actors: Customer
- Description: Displays the contents of the customer's shopping cart, including selected items and total price.

10. **Check Discount:**
- Actors: System
- Description: Checks for any available discounts or promotions applicable to the customer's order.

11. **Payment:**
- Actors: Customer, System
- Description: Facilitates the payment process for the customer's order, including selecting a payment method and providing payment details.

12. **Delivery:**
- Actors: System
- Description: Manages the delivery process of the customer's order, including dispatch and logistics.

13. **Dispatch and Logistics:**
- Actors: Admin

- Description: Handles the logistics and dispatching of orders, ensuring timely delivery to customers.

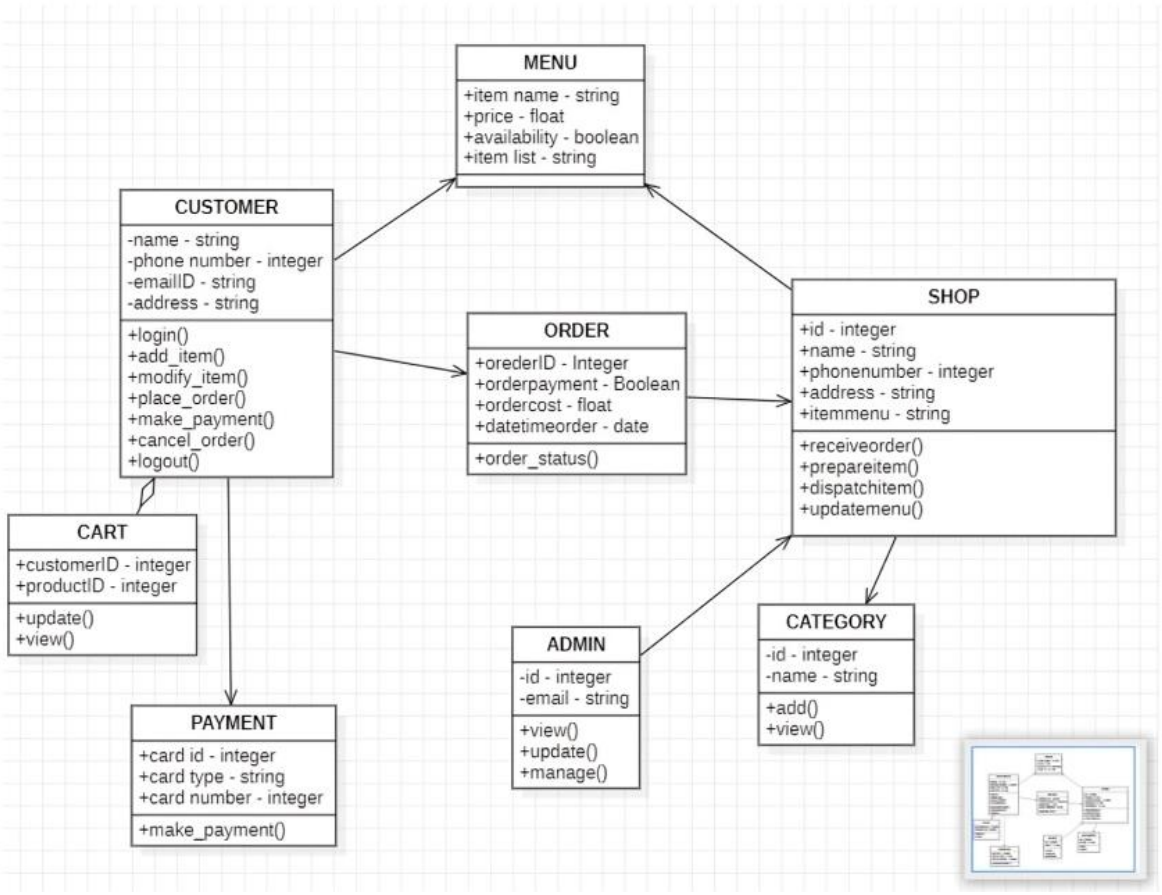
14. **Item Management:**
    - Actors: Admin
    - Description: Involves managing the inventory of items available for purchase, including adding, updating, or removing items.

15. **Retailer:**
    - Actors: Retailer
    - Description: Represents the entity responsible for managing product inventory and information.

16. **Admin:**
    - Actors: Admin
    - Description: Represents the administrative user who manages the virtual grocery store system.

This use case diagram provides a clear overview of the system's functionalities and the interactions between various actors and components. It serves as a valuable tool for understanding the system requirements and guiding the development process.

2. CLASS DIAGRAM:



Class Descriptions:

1. **Customer:**
   Represents a customer who interacts with the system to place orders and make payments.

Attributes:

- name: Name of the customer (String).
- phoneNumber: Phone number of the customer (Integer).
- emailID: Email ID of the customer (String).
- address: Address of the customer (String).

Methods:

- login(): Logs in the customer to the system.
- addItem(): Adds an item to the customer's cart.
- modifyItem(): Modifies an item in the customer's cart.
- placeOrder(): Places an order.

- makePayment(): Makes a payment for the order.
- cancelOrder(): Cancels an order.
- logout(): Logs out the customer from the system.

2. **Cart:**

Represents the shopping cart of a customer.

Attributes:

- customerID: ID of the customer (Integer).
- productID: ID of the product (Integer).

Methods:

- update(): Updates the cart.
- view(): Views the contents of the cart.

3. **Payment:**

Represents the payment process for orders.

Attributes:

- cardId: ID of the card (Integer).
- cardType: Type of the card (String).
- cardNumber: Number of the card (Integer).

Methods:

- makePayment(): Initiates the payment process.

4. **Menu:**

Represents the menu of items available for purchase.

Attributes:

- itemName: Name of the item (String).
- price: Price of the item (Float).
- availability: Availability status of the item (Boolean).
- itemList: List of items (String).

5. **Order:**

Represents an order placed by a customer.

Attributes:

- orderID: ID of the order (Integer).
- orderPayment: Payment status of the order (Boolean).
- orderCost: Cost of the order (Float).
- dateTimeOrder: Date and time of the order (Date).

Methods:

- orderStatus(): Retrieves the status of the order.

6. **Shop:**

Represents a shop where customers can place orders.

Attributes:

- id: ID of the shop (Integer).
- name: Name of the shop (String).
- phoneNumber: Phone number of the shop (Integer).
- address: Address of the shop (String).
- itemMenu: Menu of items available in the shop (String).

Methods:

- receiveOrder(): Receives an order from a customer.
- prepareItem(): Prepares items for dispatch.
- dispatchItem(): Dispatches items to customers.
- updateMenu(): Updates the menu of items.

7. **Category:**

Represents categories of items available in the menu.

Attributes:

- id: ID of the category (Integer).
- name: Name of the category (String).

Methods:

- add(): Adds a new category.
- view(): Views the categories.

8. **Admin:**

Represents an administrator who manages the system.

Attributes:

- id: ID of the admin (Integer).
- email: Email ID of the admin (String).

Methods:

- view(): Views the system.
- update(): Updates the system.
- manage(): Manages the system.

This class diagram provides a detailed representation of the classes, their attributes, methods, and relationships within the system. It captures the essential functionalities and interactions of the system components.

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

BCSE301P - Software Engineering Lab

Assessment - 4

VIRTUAL GROCERY STORE

Submitted to –

Dr. Murugan K

Submitted by –

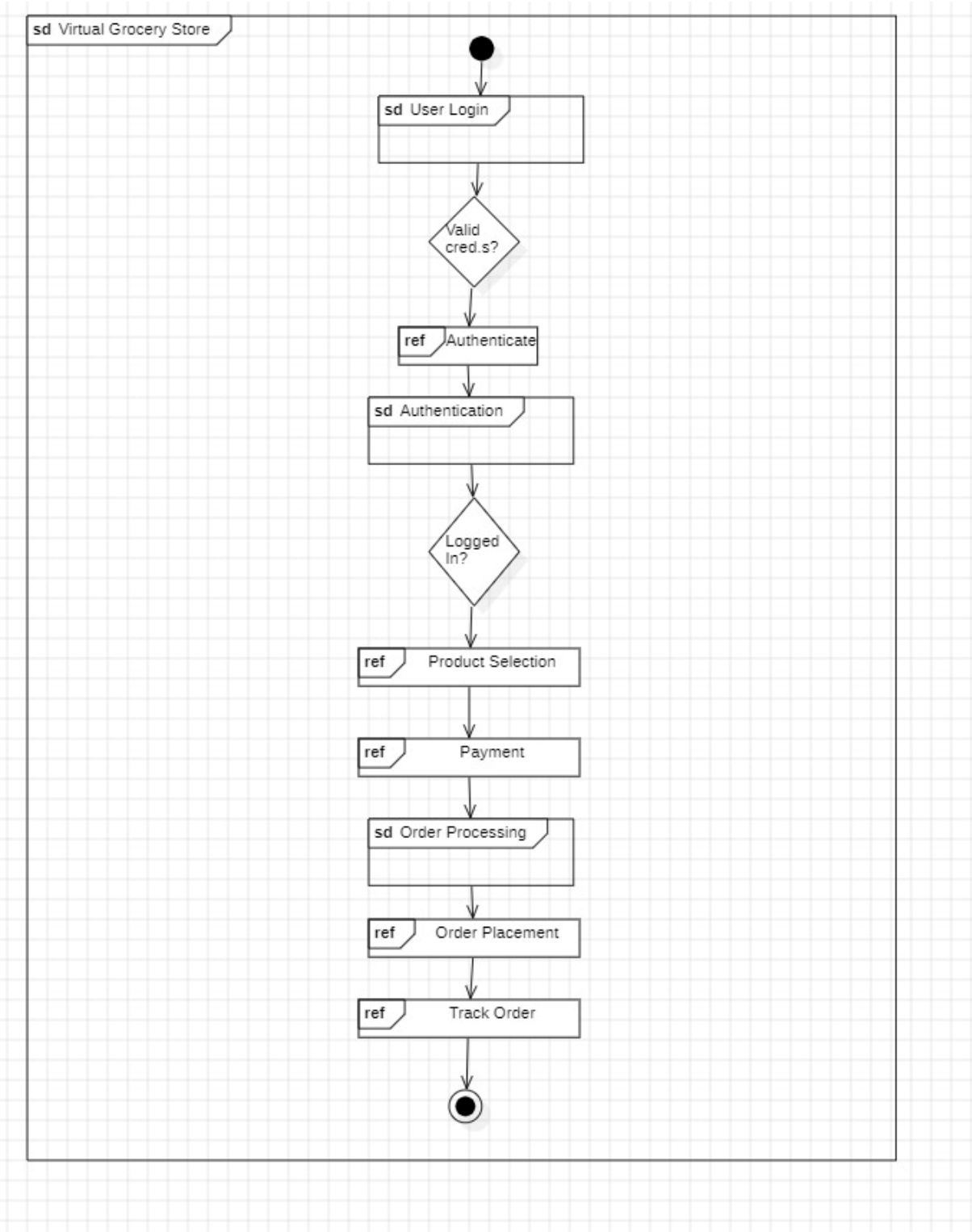
Shreeji Chandgotia (21BCE3988)

Ishita Prakash (21BCE3995)

Suraj Kumar (21BKT0065)

Q.1 OO design – Interaction model.

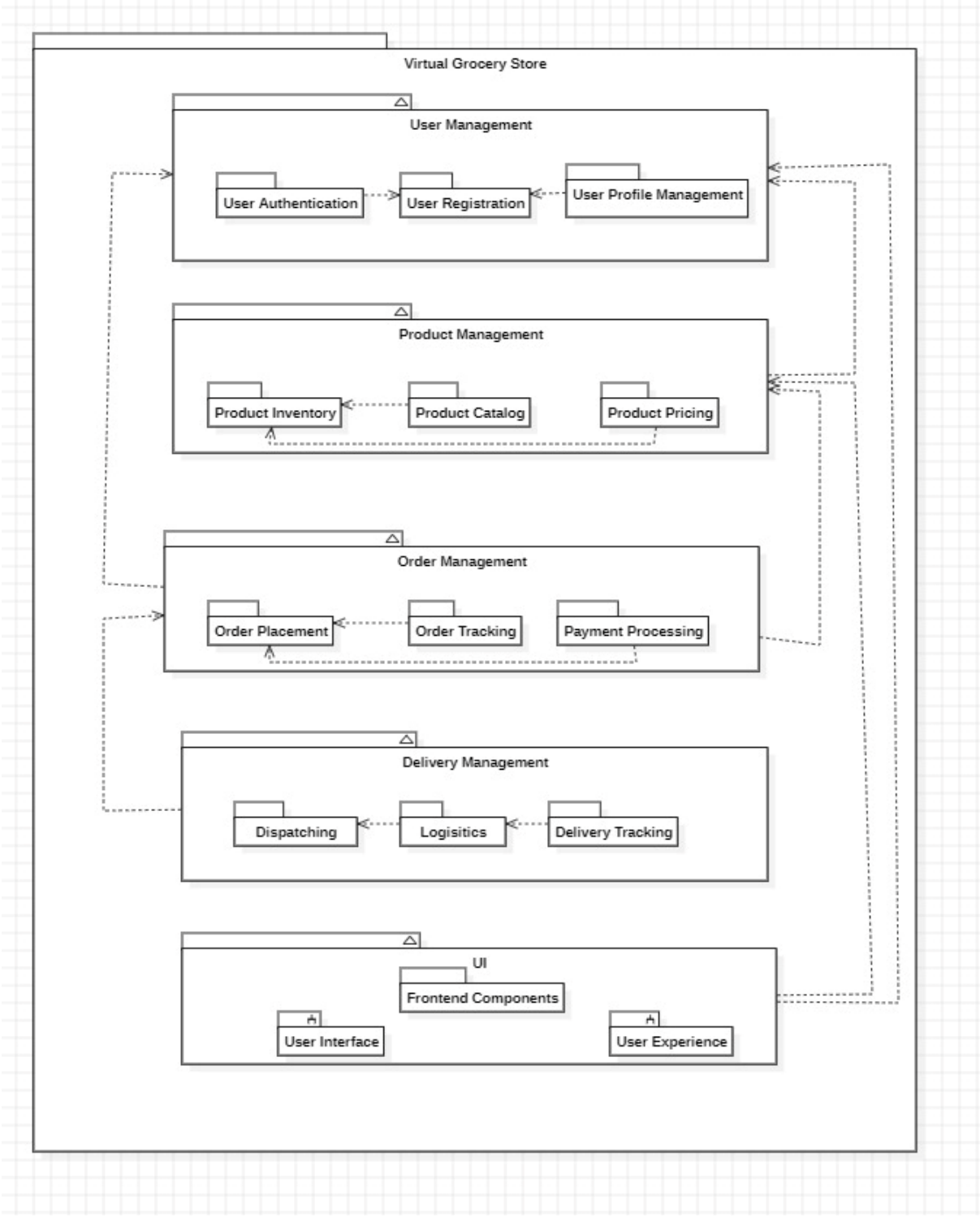
INTERACTION MODEL



- The interaction begins with the Initial state, indicating the start of the process.
- Users attempt to log in (User Login) to the system, leading to a decision point (Decision: Valid Credentials?) to check if the credentials provided are valid.
- If the credentials are valid, the authentication process is initiated (Interaction: Authentication), otherwise, an error message is displayed.
- After successful authentication, another decision is made to check if the user is logged in (Decision: Logged In?).
- If the user is logged in, they proceed with product selection (Interaction: Product Selection), otherwise, they are prompted to log in again.
- The process continues with order processing, forking into parallel activities for order placement and order tracking.
- After both activities are completed, they join back together (Join: Order Processing).
- Finally, the interaction ends with the Final state.

Q.2. OO design Package, Component, and deployment models.

PACKAGE MODEL



Packages:

- UserManagement: Contains modules related to user authentication, registration, and profile management.
- ProductManagement: Contains modules for managing product inventory, catalog, and pricing.
- OrderManagement: Contains modules for processing orders, payments, and order tracking. UI: Contains modules for handling user interface components and interactions.

Modules:

- AuthenticationModule: Manages user authentication functionality.
- InventoryModule: Manages product inventory and stock.
- OrderProcessingModule: Handles order processing, including payment and tracking.
- ShoppingCartModule: Manages user shopping cart functionality.

Subsystems:

- UserSubsystem: Includes modules related to user management.
- ProductSubsystem: Includes modules related to product management.
- OrderSubsystem: Includes modules related to order management.
- UISubsystem: Includes modules related to user interface and presentation logic.

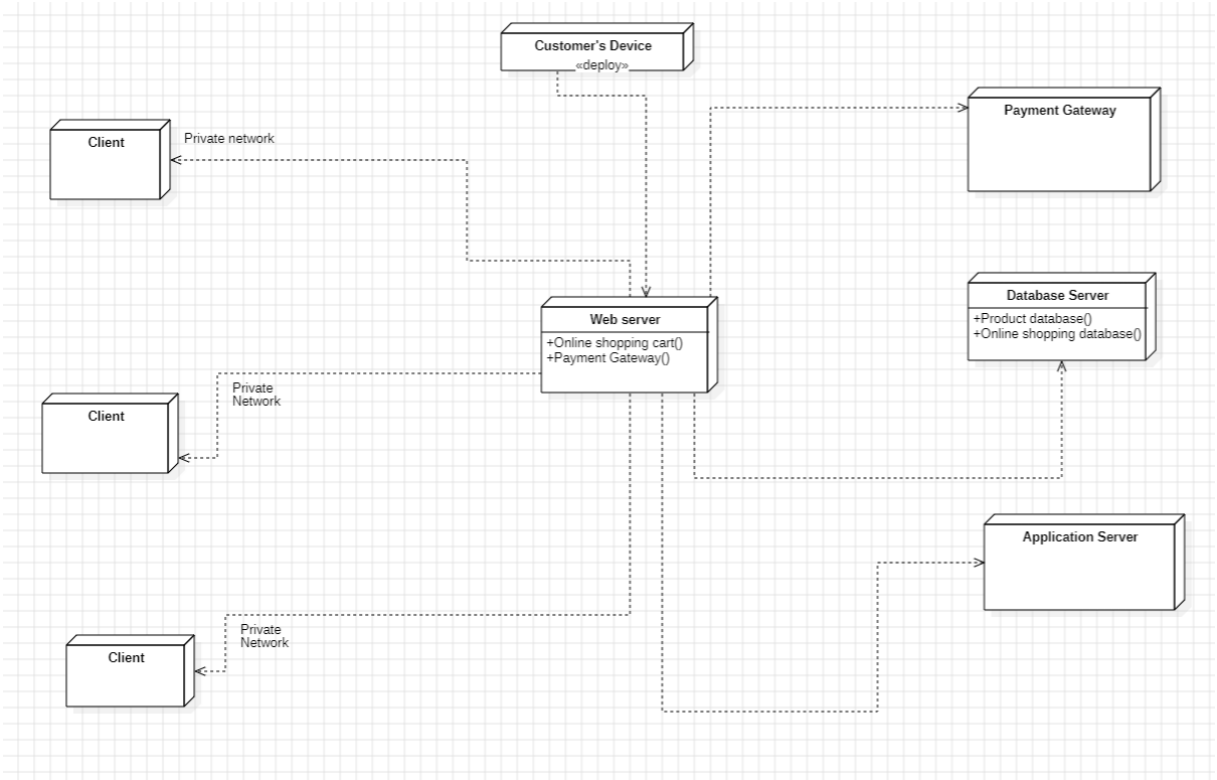
Containment:

- Modules are contained within packages. For example, the AuthenticationModule is contained within the UserManagement package.
- Subsystems contain multiple modules. For instance, the UserSubsystem contains the AuthenticationModule and other modules related to user management.

Dependency:

- The OrderProcessingModule depends on the AuthenticationModule for user authentication.
- The ShoppingCartModule depends on the InventoryModule for product availability.
- The UI package depends on modules from other packages for data display and user interaction.

Deployment Model:



Web Server:

The web server hosts the front-end components of the system accessible to clients and customers. It handles HTTP requests from clients, serves web pages, and executes client-side scripts. The web server communicates with the application server to process dynamic content and retrieve data from the database server.

Clients:

Clients are the end-users or consumers of the system who interact with the web server to access services. Clients can be various devices such as desktop computers, laptops, tablets, or smartphones. They use web browsers or dedicated applications to connect to the web server and perform actions such as browsing products, adding items to the cart, and making purchases.

Customer's Device:

The customer's device represents the specific device used by an individual customer to access the system. It could be a mobile phone, tablet, desktop computer, or any other device capable of connecting to the web server over the internet. The customer's device provides the interface through which the customer interacts with the system, browses products, and makes purchases.

Application Server:

The application server hosts the back-end components of the system responsible for business logic, data processing, and application services. It handles client requests received from the web server, processes dynamic content, and interacts with the database server to retrieve or update data. The application server may also integrate with external systems such as payment gateways to facilitate transactions.

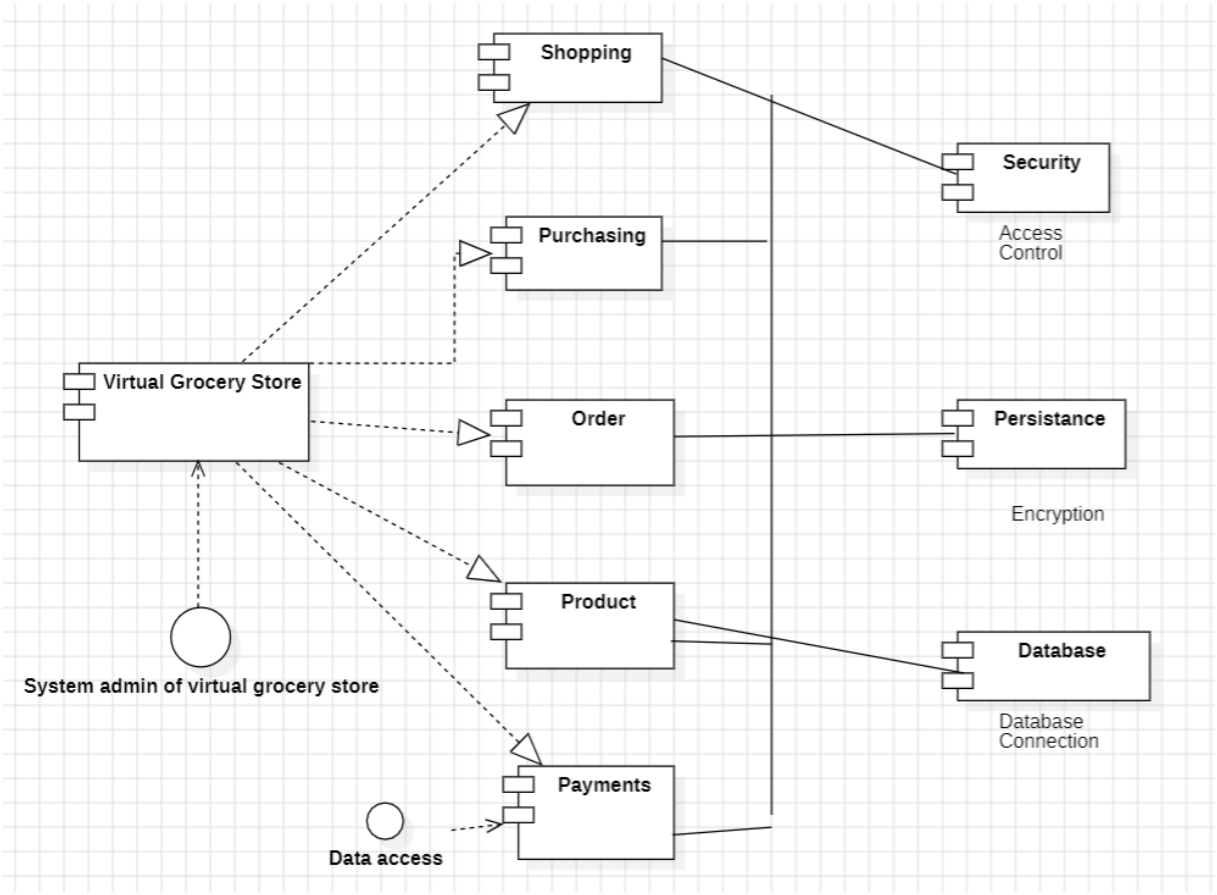
Database Server:

The database server stores persistent data related to the system, including user accounts, product information, order details, and transaction records. It provides a centralized repository for storing and retrieving data, ensuring data integrity, consistency, and security. The database server interacts with the application server to perform database operations such as querying, updating, and deleting data.

Payment Gateway:

The payment gateway is a third-party service that facilitates electronic transactions between clients and merchants. It securely processes payment information provided by clients during checkout, authorizes transactions, and transfers funds between accounts. The payment gateway communicates with the application server to receive payment requests, verify payment details, and provide transaction status updates.

Component Model:



1. Shopping Component:

- Responsible for managing the shopping experience of users within the application.
- Includes functionalities such as browsing products, adding items to the cart, and viewing the shopping cart.
- Interfaces with the purchasing component to facilitate the checkout process.

2. Purchasing Component:

- Handles the process of finalizing orders and making purchases.
- Manages the checkout process, including selecting payment methods, entering delivery details, and confirming orders.
- Interfaces with the shopping component to retrieve the user's selected items.

3. Order Component:

- Manages the lifecycle of orders within the system.
- Tracks order status, updates delivery information, and handles order cancellations if necessary.
- Interfaces with the database component to store and retrieve order-related data.

4. **Product Component:**
- Manages the catalog of products available for purchase in the virtual grocery store.
- Includes functionalities for adding, updating, and removing products from the inventory.
- Interfaces with the database component to store and retrieve product information.

5. Payments Component:
- Facilitates secure payment transactions between users and the virtual grocery store.
- Integrates with payment gateways to process various payment methods, such as credit/debit cards, digital wallets, and cash on delivery.
- Ensures the security and integrity of payment data during transactions.

6. Security Component:
- Implements security measures to protect sensitive data and prevent unauthorized access.
- Includes functionalities such as user authentication, authorization, and encryption of sensitive information.
- Monitors and detects security threats or vulnerabilities within the system.

7. Persistence Component:
- Manages the storage and retrieval of data required by the application.
- Utilizes a database system to store various types of data, including user profiles, product information, order details, and payment records.
- Ensures data consistency, reliability, and durability through proper data management techniques.

8. Database Component:
- Serves as the foundation for storing and managing structured data used by the application.
- Utilizes a relational or NoSQL database system to organize and query data efficiently.
- Stores information related to users, products, orders, payments, and other entities critical to the functionality of the virtual grocery store.

These components collectively form the backbone of the Virtual Grocery Store system, enabling users to browse, purchase, and manage their orders seamlessly while ensuring data security and reliability.

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

BCSE301P - Software Engineering Lab

Assessment - 5

VIRTUAL GROCERY STORE

Submitted to –

Dr. Murugan K

Submitted by –

Shreeji Chandgotia (21BCE3988)

Ishita Prakash (21BCE3995)

Suraj Kumar (21BKT0065)

Q.1. Design and demonstration of test cases. Functional Testing and Non-Functional Testing .

Functional Testing:

Login Functionality:

Test Case 1: Verify that valid credentials allow the user to log in successfully.

Test Case 2: Verify that invalid credentials result in an error message.

Test Case 3: Verify that the system locks the user out after multiple failed login attempts.

Adding Items to Cart:

Test Case 4: Verify that users can add items to their cart by selecting them from the product list.

Test Case 5: Verify that the correct quantity of items is added to the cart.

Test Case 6: Verify that the total price in the cart updates correctly after adding items.

Viewing Cart:

Test Case 7: Verify that users can view the contents of their cart, including item names, quantities, and prices.

Test Case 8: Verify that users can remove items from their cart.

Test Case 9: Verify that the total price updates correctly after removing items from the cart.

Checkout Process:

Test Case 10: Verify that users can proceed to checkout after reviewing their cart.

Test Case 11: Verify that users can enter their delivery address during checkout.

Test Case 12: Verify that users can select a payment method and complete the payment process.

Non-Functional Testing:

Performance Testing:

Test Case 13: Measure the response time of the system when logging in, adding items to cart, and checking out.

Test Case 14: Simulate multiple users accessing the system simultaneously to assess its scalability.

Usability Testing:

Test Case 15: Evaluate the clarity of user interface elements such as buttons, navigation menus, and form fields.

Test Case 16: Conduct user surveys or interviews to gather feedback on the ease of use of the application.

Security Testing:

Test Case 17: Verify that sensitive user information such as passwords and payment details are encrypted during transmission.

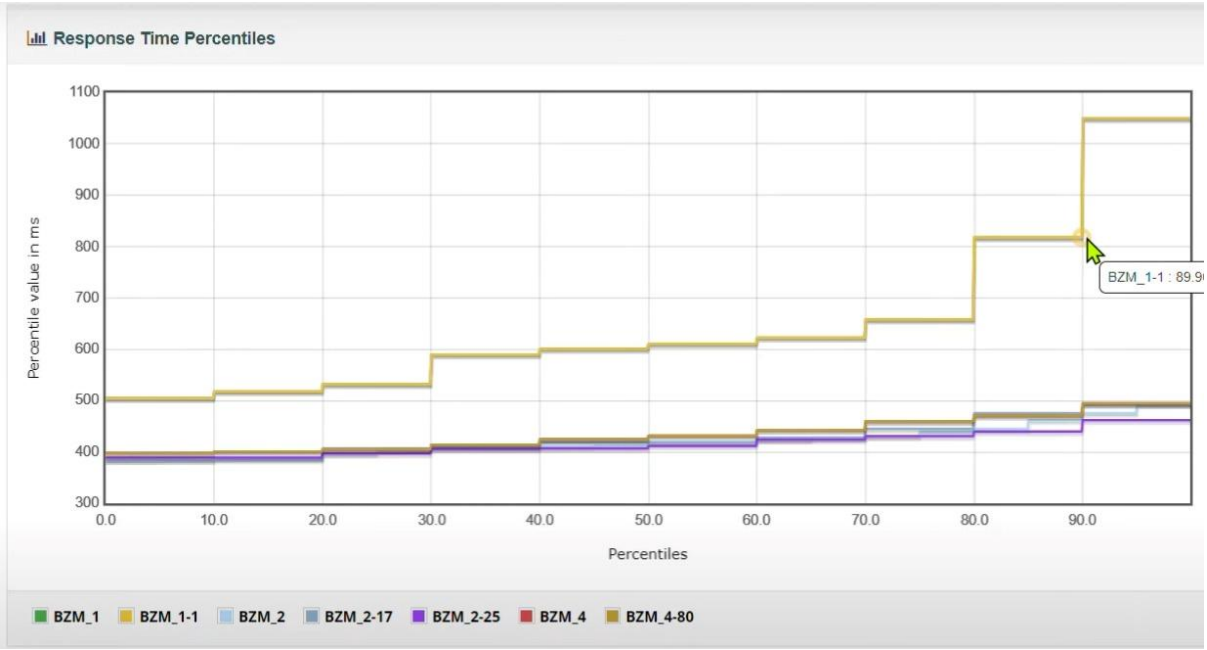
Test Case 18: Test for vulnerabilities such as SQL injection or cross-site scripting attacks.

Compatibility Testing:

Test Case 19: Test the application on different web browsers (Chrome, Firefox, Safari) and devices (desktop, mobile) to ensure compatibility.

Test Case 20: Verify that the application functions correctly on different operating systems (Windows, macOS, Linux).

Testing using open source tool: JMETER



In JMeter, the response time process involves sending requests to a server or application, measuring the time taken for each request to be sent and for the corresponding response to be received, and then analyzing this data to understand the performance characteristics of the system under test. JMeter captures various metrics during the execution of test scenarios, including average response time, minimum and maximum response times, percentiles, and throughput. These metrics provide insights into the overall performance of the system and help identify potential bottlenecks or areas for optimization. By analyzing the response time data collected by JMeter, testers and developers can make informed decisions to improve the performance and scalability of their applications.

Functional testing using Selenium Webdriver

```java
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class LoginTest {

    public static void main(String[] args) {
        // Set the path to the chromedriver executable
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");

        // Create a new instance of the ChromeDriver
        WebDriver driver = new ChromeDriver();

        // Open the login page
        driver.get("https://example.com/login");

        // Find the username and password input fields and enter credentials
        WebElement usernameInput = driver.findElement(By.id("username"));
        WebElement passwordInput = driver.findElement(By.id("password"));

        usernameInput.sendKeys("your_username");
        passwordInput.sendKeys("your_password");

        // Find and click the login button
        WebElement loginButton = driver.findElement(By.id("loginButton"));
        loginButton.click();

        // Wait for the next page to load (you may need to use explicit waits for this)
        // Perform assertions or further actions based on the login result
        // For example, you can check if the login was successful by verifying the presence of a welcome message
        WebElement welcomeMessage = driver.findElement(By.id("welcomeMessage"));
        if (welcomeMessage.isDisplayed()) {
            System.out.println("Login successful!");
        } else {
            System.out.println("Login failed!");
        }

        // Close the browser
        driver.quit();
    }
}
```

In the provided example using Selenium WebDriver, a test case for a login page is automated in Java. The script navigates to the login page, enters predefined credentials into the username and password fields, clicks the login button, and then verifies the success of the login by checking for the presence of a welcome message. This process demonstrates how Selenium WebDriver can simulate user interactions with web elements, enabling automated testing of login functionality and other web-based scenarios.

Test Case Development:

Module 1: Login

Test case ID	Test Objective	Precondition	Steps	Test data	Expected result	Post-condition	Actual Result	Pass/fail
TC001	Verify admin login with username and password	Admin should be registered with valid email and password before login.	Click on Login button Enter valid username and password	Email-id: abc@xyz.com Password: Xyz123	System displays Admin homepage	Admin should be kept logged in until logout.	As Expected,	Pass
sTC002	Verify user login with valid username and password	User should be registered with valid email and password before login.	Click on Login button Enter valid username and password	Email-id: abc@xyz.com Password: Xyz123	System displays User homepage	User should be kept logged in until logout.	As Expected,	Pass
TC003	Verify user login with invalid username and password	User is not registered before with valid email and password.	Click on Login button Enter valid username and password	Email-id: abcxyz.com Password: 123ggh	Display error message login failed.	Redirect to Login page.	As Expected,	Pass
TC004	Forgot password	User should be registered with valid email and password before clicking on forgot password.	Click on forgot password Enter registered email id. Enter OTP sent to email id Verify	Email-id: abc@xyz.com OTP: 1234 Password: Xyz123	User will set new password after verifying OTP and password change successfully message will be displayed.	Redirect to Login page.	As Expected,	Pass

			OTP Enter new password Confirm password	Password: Xyz123				

Module 2 Registration

Test case ID	Test Objective	Precondition	Steps	Test data	Expected result	Post-condition	Actual Result	Pass/fail
TC001	Register new user.	User must have valid details required for registration .i.e. email-id, contact number. Email id must not be used before for registration to this system .i.e. unique email id for each user.	1Click on Register button. 2Fill fields of form .i.e. email id, contact number ,address	Email-id: abc@xyz.com Contact: 9876543210 Address: Aptno12/A, Street,City, State,Country.	All the Required fields are filled	Procced to verification of details via OTP	As Expected	Pass
TC001.2	Verify User with OTP	E-mail address should be correct	1 OTP is sent for verifying email address 2 Verify OTP sent 3 Create password 4 Confirm details and click OK.	OTP:1234 Password: Xyz123 Password: Xyz123	Registration successful message will be displayed.	Redirect to Login page.	As Expected,	Pass

Module 3: Add item to Cart

Test case ID	Test Objective	Precondition	Steps	Test data	Expected result	Post-condition	Actual Result	Pass/Fail
TC001	User should select desired item and add item to cart.	User should be logged in.	Select desired item. Click on add to cart button.	Item name: Item123 Item-id: #123 Specifications: Cost:12RS	Continue shopping	Item gets added to cart successfully. Amount and count are incremented accordingly.	As Expected,	Pass
TC002	Show total amount of items in cart.	Items must be present in the cart.	Open the cart Click on show total amount.	Total amount: 1234RS	Displays the details and total amount of all items in cart.	Proceed to order.	As Expected,	Pass
TC003	Remove item from cart.	At least one item must be present in cart.	Remove desired item. Click on remove from cart button.	Item name: Item123 Item-id: #123	Continue shopping	Item gets removed from cart and decrement the amount and count of items.	As Expected,	Pass
TC004	Proceed to checkout.	Items must be present in the cart.	Open the cart Click on proceed to checkout. Enter shipping address.	Item name: Item123 Item-id: #123 Shipping address:	Displays the details and total amount of all items in cart and shipping address.	Proceed to payment.	As Expected,	Pass

				Aptno12/A, Street, City, State, Country.				

Q.2. Story Boarding and User Interface design Modelling.

Storyboard:

Introduction:

A welcoming screen with the title "Virtual Grocery Store" and a brief description of its purpose.

User Authentication:

Frame 1: Display a login screen with input fields for username and password.

Frame 2: If login credentials are correct, transition to the main interface. If incorrect, display an error message.

Main Interface:

Frame 1: Present the main dashboard with options like "View Items", "View Cart", "Track Orders", etc.

Viewing Items:

Frame 1: Display a list of grocery items available for purchase.

Frame 2: Allow users to filter items by categories like fruits, vegetables, dairy.

Adding to Cart:

Frame 1: Enable users to add items to their cart by clicking on the respective item.

Frame 2: Show a confirmation message after successfully adding an item to the cart.

Viewing Cart:

Frame 1: Navigate to the cart page where users can see the list of items added along with their quantities and prices.

Frame 2: Provide options to edit quantities or remove items from the cart.

Checkout Process:

Frame 1: Guide users through the checkout process, prompting them to enter delivery address and select a payment method.

Frame 2: Display a summary of the order before finalizing the purchase.

Order Tracking:

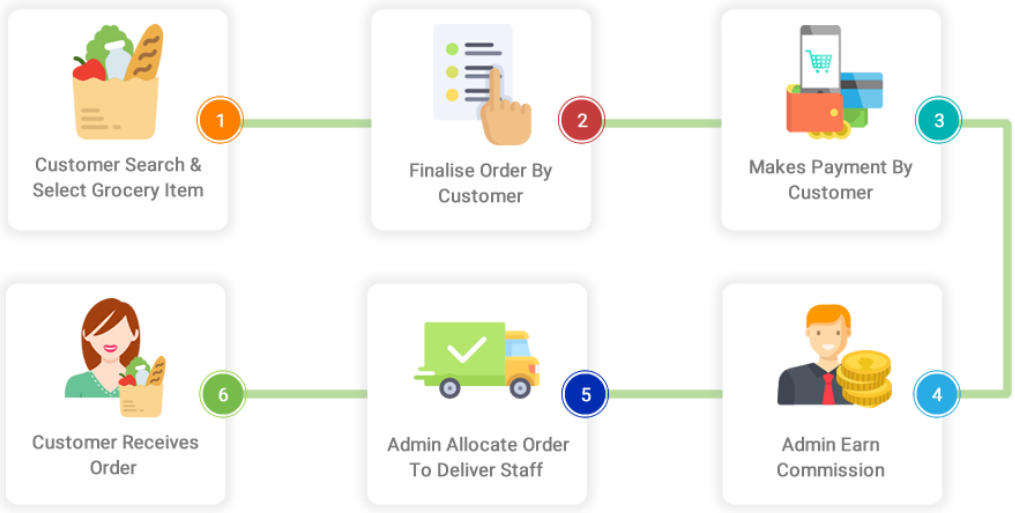
Frame 1: Allow users to track the status of their orders by entering the order ID or accessing their order history.

Frame 2: Provide real-time updates on order status, from processing to delivery.

Feedback Submission:

Frame 1: After order completion, prompt users to provide feedback on their shopping experience.

Frame 2: Thank users for their feedback and offer any incentives for future purchases.



User Interface Design:

Login Screen:

Include input fields for username and password, along with a "Login" button.

Main Dashboard:

Design a visually appealing dashboard with easy-to-navigate options for different functionalities.

Item Display:

Present grocery items in a grid layout with images, names, prices, and an "Add to Cart" button.

Shopping Cart:

Create a cart icon indicating the number of items added, with a dropdown menu to view and manage the cart contents.

Checkout Process:

Design a step-by-step checkout process with clear instructions and form fields for delivery address and payment details.

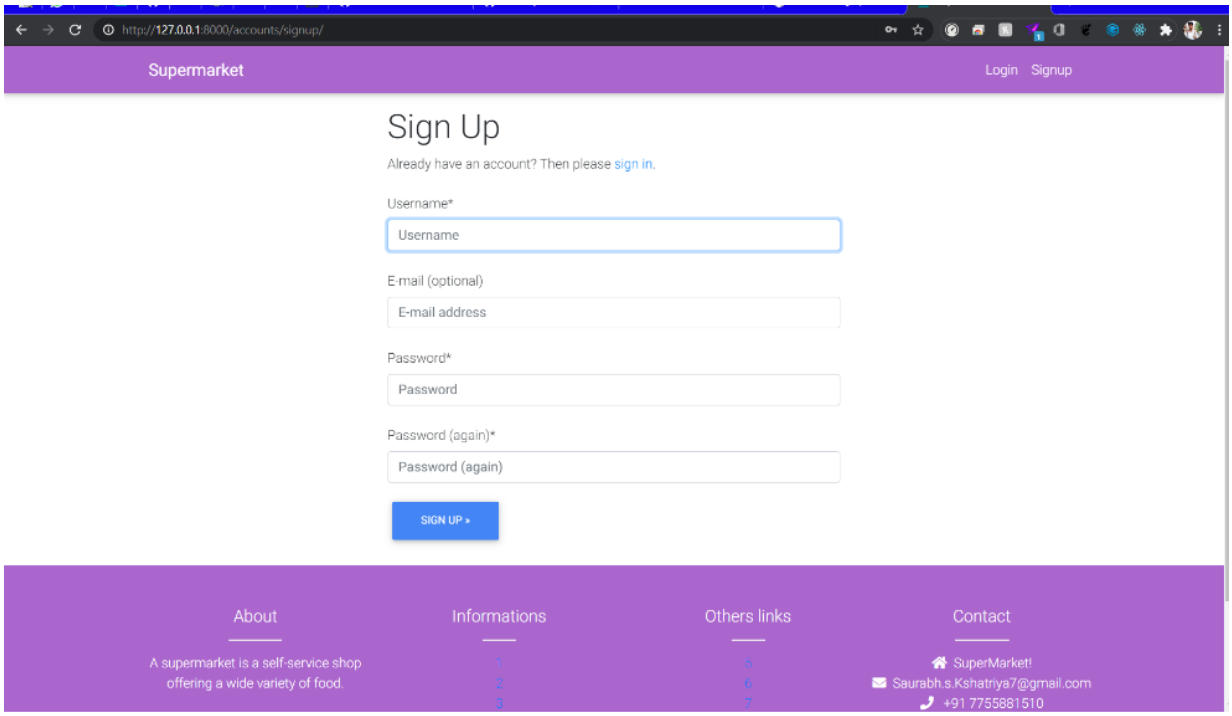
Order Tracking:

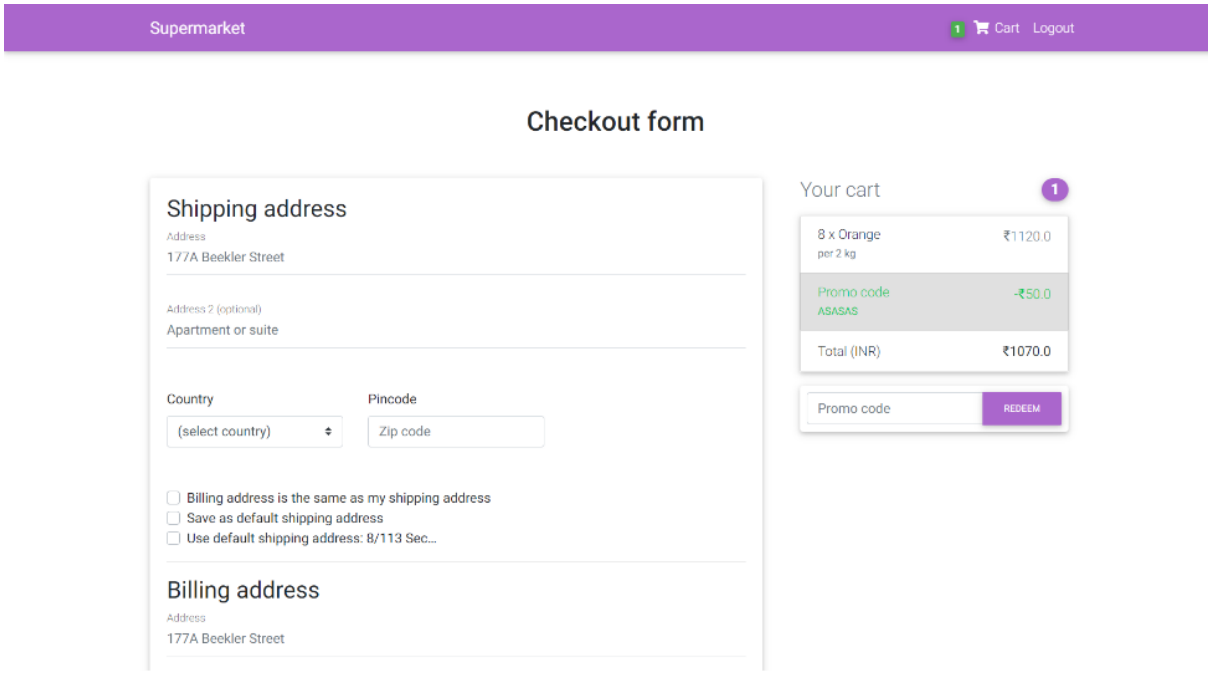
Provide a dedicated section for order tracking, allowing users to enter their order ID or view recent orders.

Feedback Form:

Create a simple form with rating scales and text fields for users to submit feedback.

UI DESIGN:


http://127.0.0.1:8000/accounts/signup/
Supermarket
Login Signup
Sign Up
Already have an account? Then please sign in.
Username*
Username
E-mail (optional)
E-mail address
Password*
Password
Password (again)*
Password (again)
SIGN UP »
About
A supermarket is a self-service shop offering a wide variety of food.
Informations
Others links
Contact
SuperMarket!
Saurabh.s.Kshatriya7@gmail.com
+91 7755881510


Supermarket
1 Cart Logout
Checkout form
Shipping address
Address
177A Beekler Street
Address 2 (optional)
Apartment or suite
Country
(select country)
Pincode
Zip code
Billing address is the same as my shipping address
Save as default shipping address
Use default shipping address: 8/113 Sec...
Billing address
Address
177A Beekler Street
Your cart
1
8 x Orange
per 2 kg
₹1120.0
Promo code
ASASAS
-₹50.0
Total (INR)
₹1070.0
Promo code
REDEEM

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

BCSE301P - Software Engineering Lab

Assessment - 6

VIRTUAL GROCERY STORE

Submitted to –

Dr. Murugan K

Submitted by –

Shreeji Chandgotia (21BCE3988)

Ishita Prakash (21BCE3995)

Suraj Kumar (21BKT0065)

IMPLEMENTATION

FRONTEND

Tech Stack Used:

1. **React :**

   React is a JavaScript library used for building user interfaces, particularly for web applications. In the context of a virtual grocery store, React is utilized to create the various components of the user interface, such as the product listings, shopping cart, and checkout process. It allows for efficient updating of the UI as users interact with the store, providing a smooth and responsive experience.

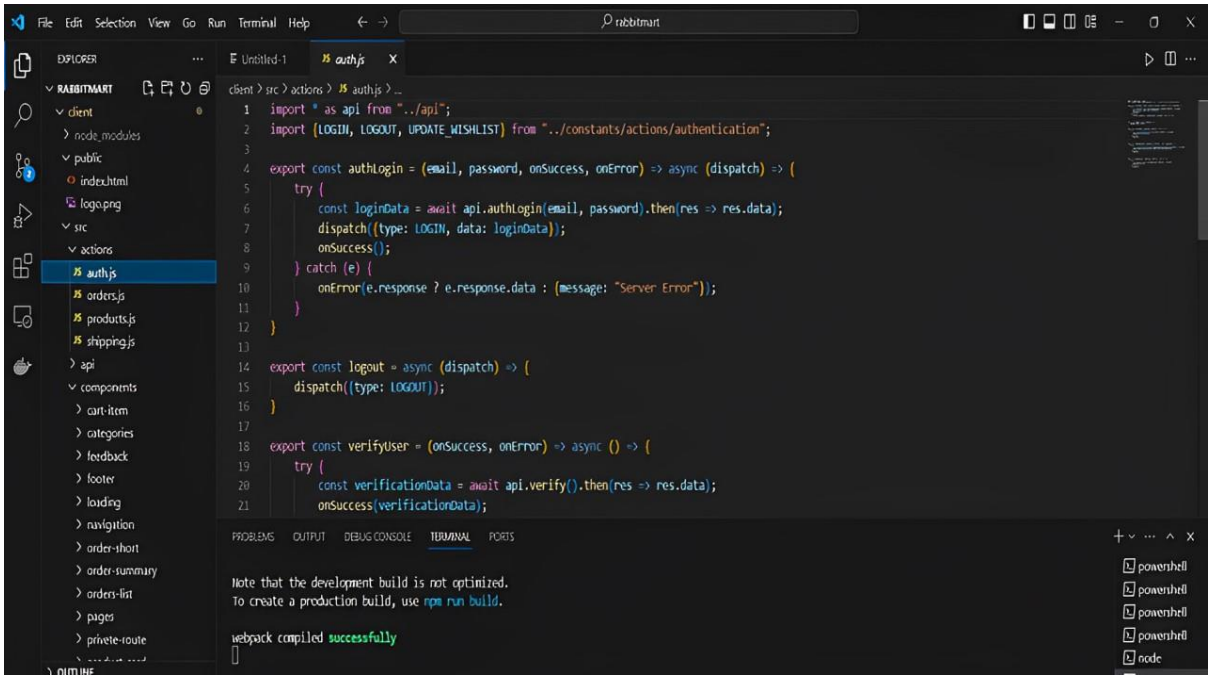
2. **React Router:**

   React Router is a library for managing the routing of a React application. In a virtual grocery store scenario, React Router is employed to handle the navigation between different pages or views within the application. For example, it is used to navigate from the home page to the product listings, then to the shopping cart, and finally to the checkout page. React Router ensures that the UI updates appropriately as users move through different sections of the virtual store.

3. **Redux:**

   Redux is a predictable state container for JavaScript applications, commonly used with React. In the context of a virtual grocery store, Redux is used to manage the application's state, such as the items in the shopping cart, user authentication status, and any other relevant data. By centralizing the state management, Redux helps to maintain a clear and consistent data flow throughout the application, making it easier to manage and debug complex interactions between different components.

Frontend connection established successfully:



Auth.js

```
Untitled-1   auth.js
client > src > actions > auth.js > ...
1  import * as api from "../api";
2  import {LOGIN, LOGOUT, UPDATE_WISHLIST} from "../constants/actions/authentication";
3
4  export const authLogin = (email, password, onSuccess, onError) => async (dispatch) => {
5      try {
6          const loginData = await api.authLogin(email, password).then(res => res.data);
7          dispatch({type: LOGIN, data: loginData});
8          onSuccess();
9      } catch (e) {
10         onError(e.response ? e.response.data : {message: "Server Error"});
11     }
12 }
13
14 export const logout = async (dispatch) => {
15     dispatch({type: LOGOUT});
16 }
17
18 export const verifyUser = (onSuccess, onError) => async () => {
19     try {
20         const verificationData = await api.verify().then(res => res.data);
21         onSuccess(verificationData);
22     } catch (e) {
23         onError(e);
24     }
25 }
26
27 export const updateWishlist = (product_id, onError) => async (dispatch) => {
28     try {
29         const wishlistData = await api.userUpdateWishlist(product_id).then(res => res.data);
30         dispatch({type: UPDATE_WISHLIST, data: wishlistData.wishlist});
31     } catch (e) {
32         onError(e);
33     }
34 }
35
36 export const getWishlist = (onSuccess, onError) => async () => {
37     try {
38         const wishlist = await api.getWishlist().then(res => res.data);
39         onSuccess(wishlist);
40     } catch (e) {
41         onError(e);
42     }
43 }
```

Orders.js

Untitled-1 | orders.js

client > src > actions > orders.js > ...

```
import * as api from '../api/index';
import {ORDERS_FETCH, ORDERS_FETCH_ALL} from "../constants/actions/orders";

export const fetchOrder = (id, onSuccess, onError) => async (dispatch) => {
    try {
        const orderData = await api.fetchOrder(id).then(res => res.data);
        dispatch({type: ORDERS_FETCH, data: orderData})
        onSuccess(orderData);
    } catch (e) {
        onError(e.response.data);
    }
}

export const updateOrder = (id, status, onSuccess, onError) => async () => {
    try {
        await api.updateOrder(id, status);
        onSuccess();
    } catch (e) {
        onError(e.response.data);
    }
}

export const fetchOrders = (page, onSuccess, onError) => async (dispatch) => {
    try {
        const ordersData = await api.fetchOrders(page).then(res => res.data);
        dispatch({type: ORDERS_FETCH_ALL, data: ordersData});
        onSuccess();
    } catch (e) {
        onError(e);
    }
}

export const postOrder = (token, data, onSuccess, onError) => async () => {
    try {
        // const [name, email, phone, country, city, area,
        //     street, building_number, floor, apartment_number] = data;
        const {url} = await api.processPayment(token, data).then(res => res.data);
        onSuccess(url);
    } catch (e) {
        onError(e);
    }
}
```

Products.js

Go Run Terminal Help rabbitmart

Untitled-1 products.js

client > src > actions > products.js > ...

```
import * as api from '../api';
import {FETCH_RECOMMENDATIONS, UPDATE_DATABASE, VALIDATE_CART, VALIDATE_CART_ERR} from "../constants/actions/products";

export const getProductsPerPage = (page, category, onSuccess) => async () => {
    try {
        const products = await api.getProductsPerPage(page, category).then(res => res.data);
        onSuccess(products);
    } catch (error) {
        console.log(error);
    }
}

export const productsSearch = (search, page, onSuccess) => async () => {
    try {
        const products = await api.productsSearch(search, page).then(res => res.data);
        onSuccess(products);
    } catch (error) {
        console.log(error);
    }
}

export const getRecommendations = (onSuccess) => async (dispatch) => {
    try {
        const data = await api.getRecommendations().then(res => res.data);
        dispatch({type: FETCH_RECOMMENDATIONS, data})
        onSuccess(data);
    } catch (e) {
        console.log(e)
    }
}

export const postProduct = (product, onSuccess, onError) => async () => {
    try {
        const data = await api.postProduct(product).then(res => res.data);
        onSuccess(data);
    } catch (e) {
        const {data} = e.response;
        onError(data);
    }
}
```

Untitled-1 products.js

client > src > actions > products.js > ...

```
export const validateCart = (cart, onSuccess, onError) => async (dispatch) => {
    try {
        const cartData = await api.validateCart(cart).then(res => res.data);
        dispatch({type: VALIDATE_CART, data: cartData});
        onSuccess(cartData.token?.split('.')[1]);
    } catch (e) {
        const {data} = e.response;
        dispatch({type: VALIDATE_CART_ERR, data});
        onError(data);
    }
}

export const adminUpdateDatabase = (csv, mode, onSuccess, onError) => async (dispatch) => {
    try {
        const updatedData = await api.adminUpdateDatabase(csv, mode).then(res => res.data);
        dispatch({type: UPDATE_DATABASE, data: updatedData});
        onSuccess(updatedData);
    } catch (e) {
        onError(e);
    }
}
```

Shipping.js

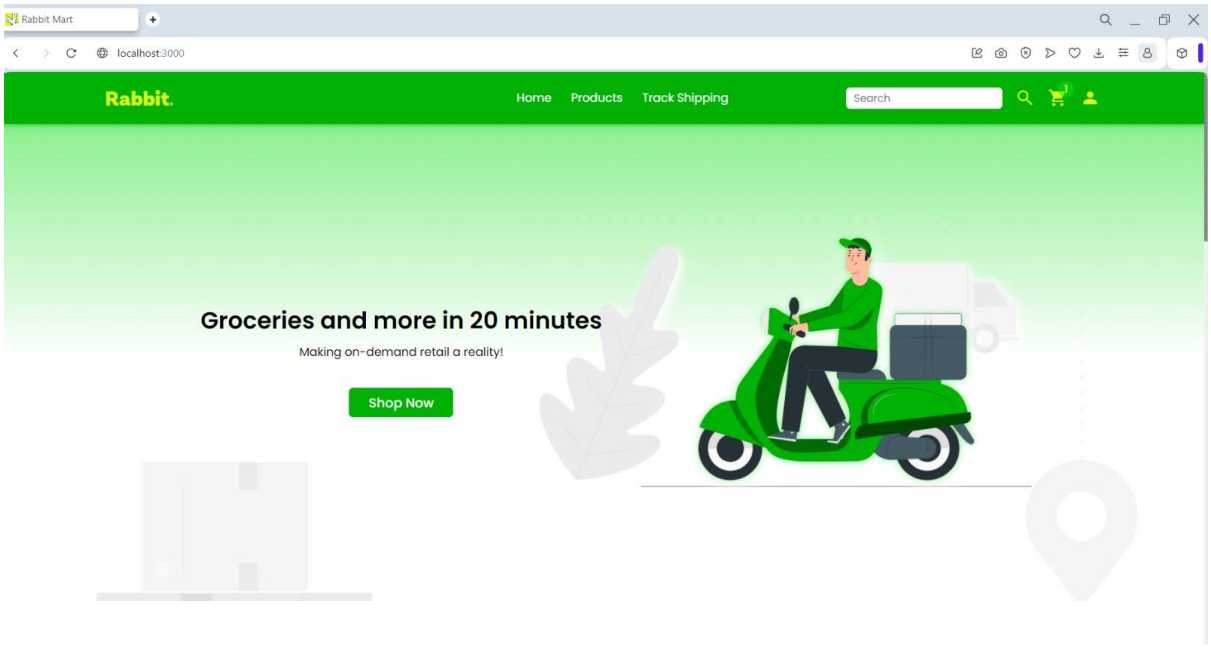
Untitled-1 | shipping.js

client > src > actions > shipping.js > ...

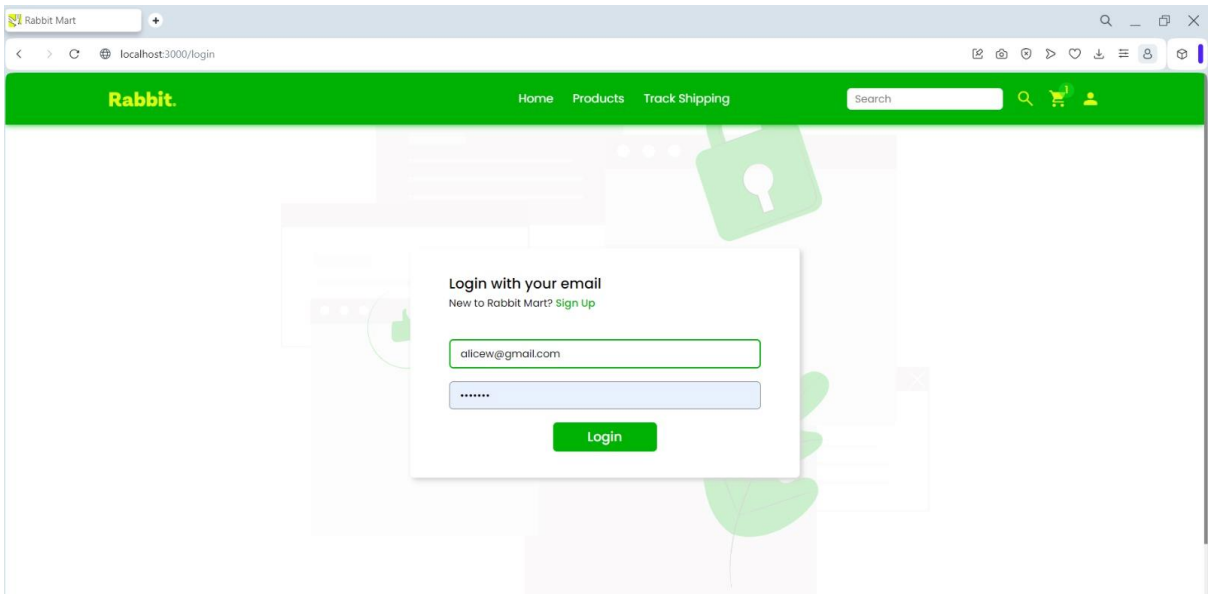
```
import * as api from "../api";
import {SHIPPING_FETCH, SHIPPING_FETCH_ALL} from "../constants/actions/shipping";

export const fetchShipments = (page, onSuccess) => async (dispatch) => {
    try {
        const shippingData = await api.fetchShipments(page).then(res => res.data);
        dispatch({type: SHIPPING_FETCH_ALL, data: shippingData});
        onSuccess();
    } catch (e) {
        console.log(e);
    }
}

export const fetchShipment = (id, onSuccess, onError) => async (dispatch) => {
    try {
        const shipmentData = await api.fetchShipment(id).then(res => res.data);
        dispatch({type: SHIPPING_FETCH, data: shipmentData})
        onSuccess(shipmentData);
    } catch (e) {
        onError(e.response.data);
    }
}

export const updateShipment = (id, status, onSuccess, onError) => async () => {
    try {
        await api.updateShipment(id, status);
        onSuccess();
    } catch (e) {
        onError(e.response.data);
    }
}
```

SCREENSHOTS:

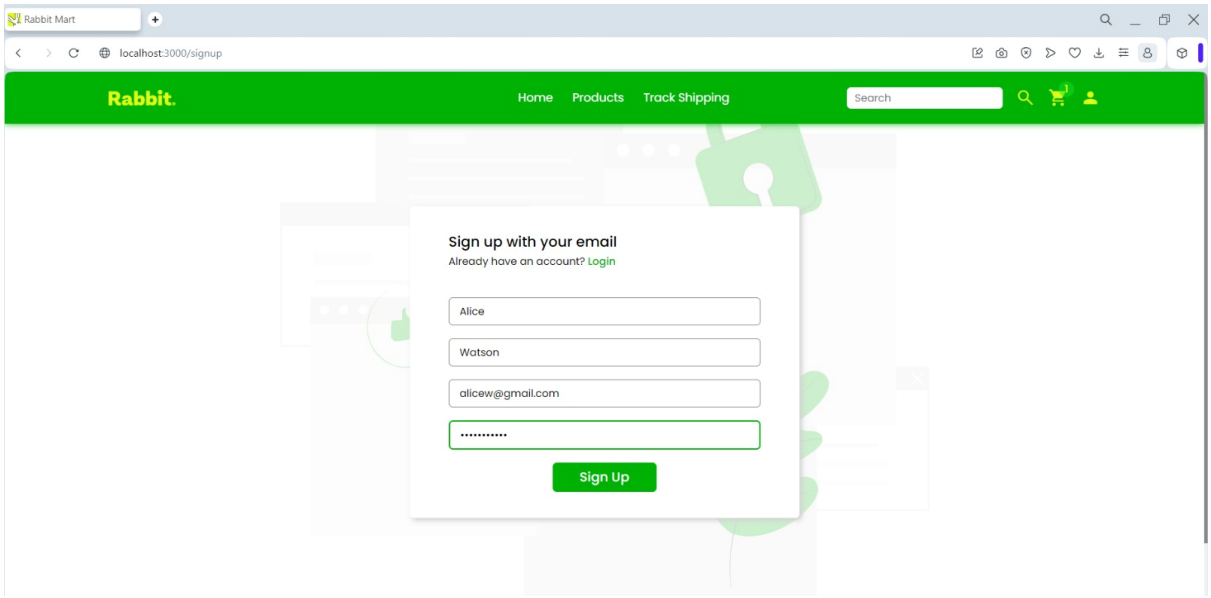
Homepage



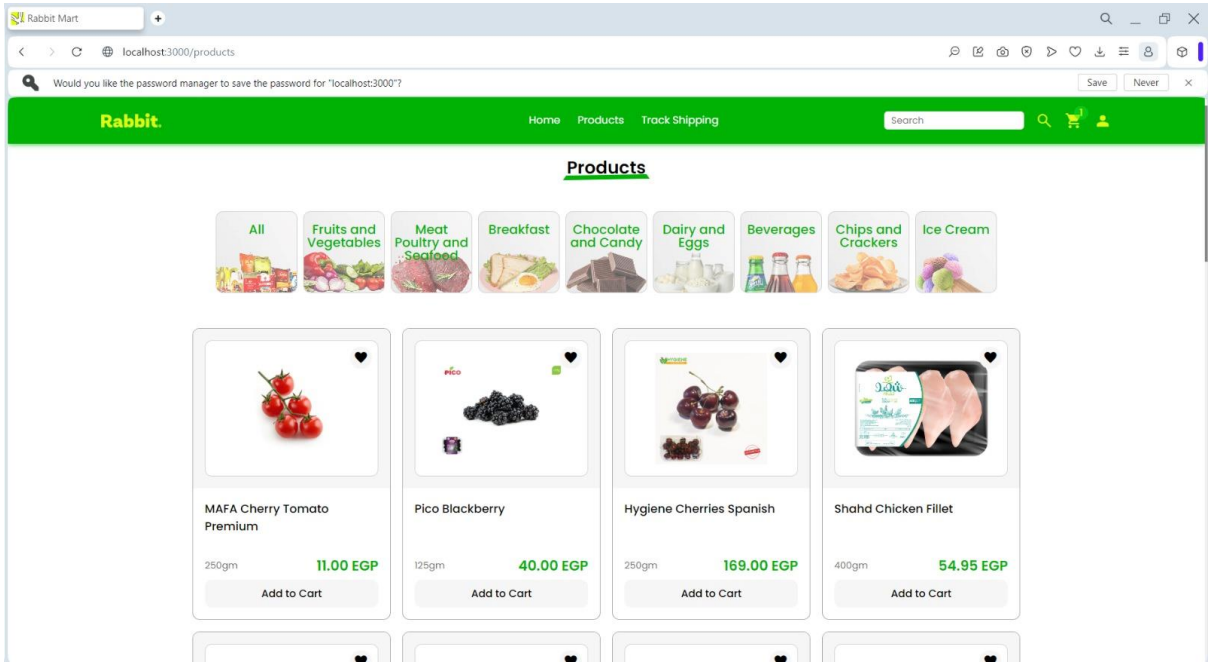
User login



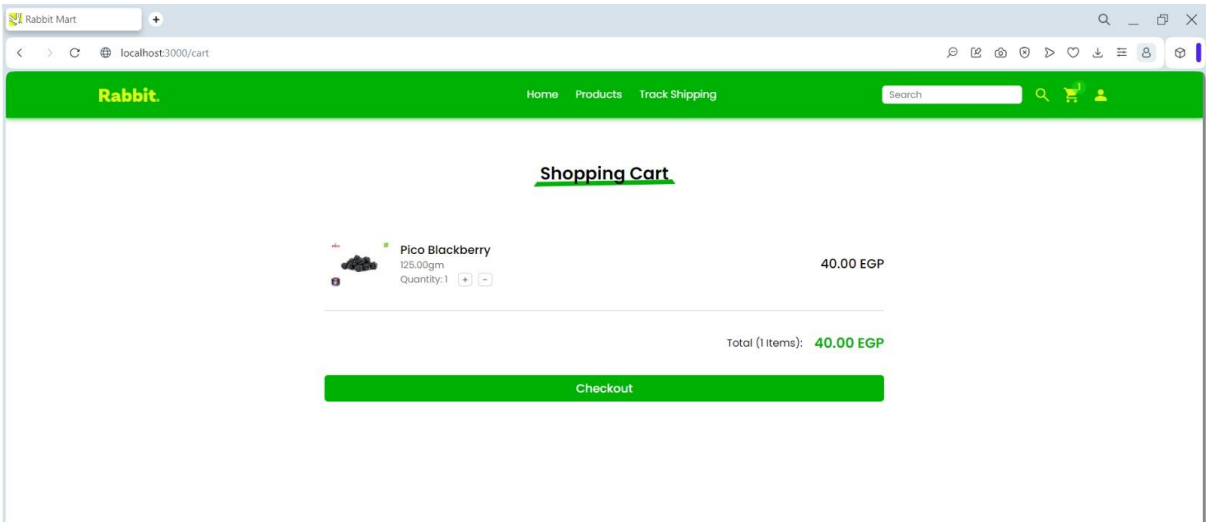
User signup



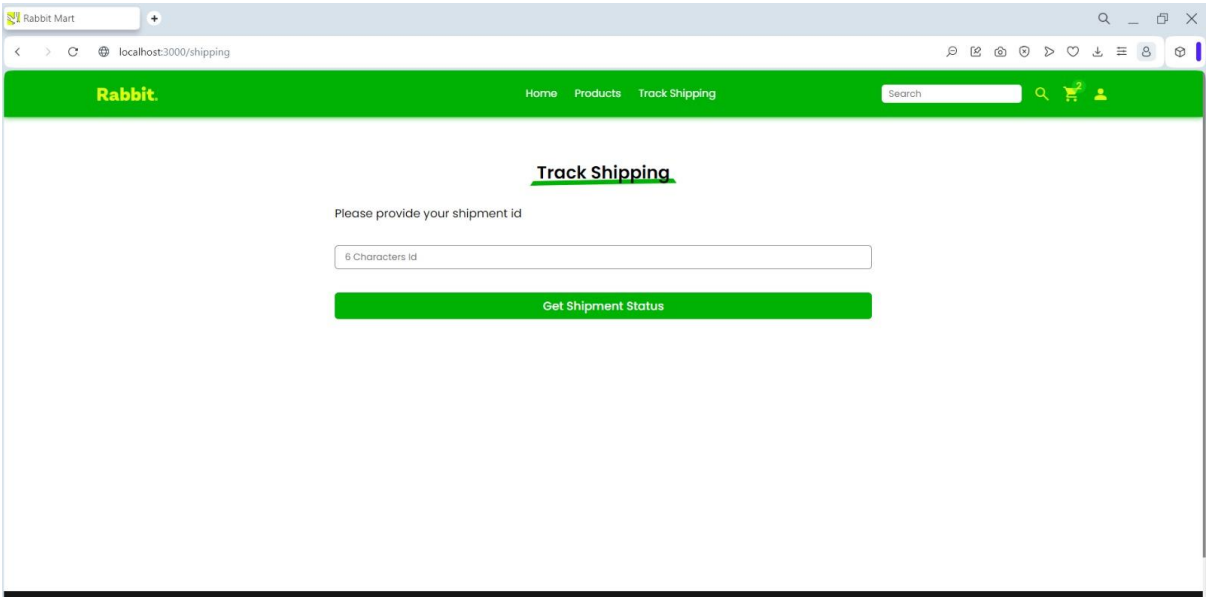
Products page



Cart page



Track shipping page



BACKEND

Tech Stack Used:

1. **Node.js :**
   In the context of a virtual grocery store software project, Node.js is used to build the backend server that handles various tasks such as managing user authentication, handling product inventory, processing orders, and interacting with the database. Node.js provides the runtime environment for executing server-side JavaScript code efficiently, allowing for scalable and high-performance backend services.

2. **Express.js:**
   Express.js is utilized as the web application framework within the Node.js environment for building the backend of the virtual grocery store. It facilitates the creation of RESTful APIs to handle client requests for fetching product information, adding items to the cart, processing payments, and other interactions. Express.js simplifies route handling, middleware integration, and request/response processing, making it an ideal choice for building the backend logic of the virtual grocery store application.

index.js:

Untitled-1 | index.js

server > index.js > ...

```
import express from "express";
import bodyParser from 'body-parser';
import mongoose from 'mongoose';
import cors from 'cors';
import dotenv from 'dotenv';
import products from "./routes/products.js";
import shipping from "./routes/shipping.js"
import orders from './routes/orders.js';
import payments from './routes/payments.js';
import notifications from "./routes/notifications.js";
import sgMail from '@sendgrid/mail';
import me from "./routes/me.js";
import Stripe from "stripe";

const app = express();
dotenv.config();

sgMail.setApiKey(process.env.SENDGRID_KEY);
export const stripe = Stripe(process.env.STRIPE_PRIVATE_KEY);

app.use(bodyParser.json({extended: true}));
app.use(bodyParser.urlencoded({extended: true}));
app.use(cors());

app.use('/orders', orders);
app.use('/payments', payments);
app.use('/products', products);
app.use('/shipping',shipping);
app.use('/notifications', notifications);
app.use('/me', me);

app.get('/', (req, res) => {
    res.status(200).json({
        team_name: "Curious Monkeys",
        dev_team: ["Baraa A.", "Eman S.", "Sary N.", "Youssef S."].sort()
    })
});

const PORT = process.env.PORT || 5000;

const mongooseOptions = {
    useNewUrlParser: true,
    useUnifiedTopology: true
}

const handleServerStartup = () => {
    app.listen(PORT, () => console.log(`Server listening on port ${PORT}`))
}

mongoose.connect(process.env.CONNECTION_URL, mongooseOptions, handleServerStartup)

export default app
```

Authentication.js

Untitled-1 | Authentication.js

server > controller > me > Authentication.js > ...

```
import Users from "../../model/Users.js";
import jwt from 'jsonwebtoken'
import bcrypt from 'bcrypt'

export const login = async (req, res) => {
    const {email, password} = req.body;

    if (!email)
        return res.status(400).json({message: "Email address is not provided"})
    if (!password)
        return res.status(400).json({message: "Password address is not provided"})

    try {
        const user = await Users.findOne({email});

        if (!user)
            return res.status(404).json({message: "User was not found"})

        const isPasswordCorrect = await bcrypt.compare(password, user.password);

        if (!isPasswordCorrect)
            return res.status(400).json({message: "Wrong password"})

        const token = jwt.sign({
            id: user._id,
            email: user.email
        }, process.env.JWT_SECRET_KEY, {expiresIn: process.env.JWT_AUTH_TTL});
        return res.status(200).json({
            user: {
                _id: user._id,
                email: user.email,
                first_name: user.first_name,
                phone: user.phone,
                address: user.address,
                role: user.role,
                wishlist: user.wishlist
            },
            token
        });
```

Untitled-1 | Authentication.js

server > controller > me > Authentication.js > login > token > email

```
5   export const login = async (req, res) => {
40      } catch (e) {
41          return res.status(400).json({message: e.message});
42      }
43  }
44
45  export const verifyUser = async (req, res) => {
46      const {id} = req.body;
47      try {
48          const user = await Users.findById(id, {password: 0})
49          return res.status(200).json({...user?._doc});
50      } catch (e) {
51          return res.status(404).json({message: "User not found"});
52      }
53  }
54
55  export const verifyRole = async (req, res) => {
56      try {
57          const {id, role} = req.body;
58
59          const user = await Users.findById(id, {password: 0});
60
61          if (!user)
62              return res.status(404).json({message: `User ${id} was not found`});
63
64          if (role !== user.role)
65              return res.status(401).json({message: "Unauthorized user"});
66
67          return res.status(200).json({user});
68      } catch (e) {
69          return res.status(400).json({message: e.message});
70      }
71  }
```

Orders.js

Untitled-1 | Orders.js

server > controller > orders > Orders.js > ...

```
import Order from "../../model/Orders.js";
import Pagination from "../../utils/pagination.js";
import axios from "axios";
import {USER_BASEURL, PRODUCTS_BASEURL, NOTIFICATIONS_BASEURL, SHIPPING_BASEURL} from "../../services/BaseURLs.js";

export const createOrder = async (req, res) => {
    try {
        const {data} = req.body;
        const order = new Order({
            order_id: data.order_id,
            name: {
                first: data.firstName,
                last: data.lastName,
            },
            email: data.email,
            phone_number: data.phone_number,
            address: JSON.parse(data.address),
            ordered_at: Date.now(),
            products: JSON.parse(data.products),
            total: data.total,
        });

        await order.save();

        await axios.patch(
            `${PRODUCTS_BASEURL}/updateQuantity`,
            {products: order.products}
        );

        const to = order.email;

        await axios.post(
            `${NOTIFICATIONS_BASEURL}/order-confirmation`,
            {to, order}
        );

        await axios.post(`${SHIPPING_BASEURL}`, {
            ordered_at: order.ordered_at,
            order_id: order.order_id,
            address: order.address,
            total: order.total,
        });

        res.status(200).json({order_id: order.order_id});
    } catch (error) {
        console.log(error);
        res.status(404).json({error: error.message});
    }
};

```

Untitled-1 | Orders.js

server > controller > orders > Orders.js > createOrder

```
export const getOrder = async (req, res) => {
    try {
        const requiredOrder = await Order.findOne({order_id: req.params.id});

        if (!requiredOrder) {
            return res.status(404).json({message: "Order does not exist"});
        }

        const productIds = requiredOrder.products.map(pr => pr.product_id);
        const {data} = await axios.post(`${PRODUCTS_BASEURL}/arr`, {arr: productIds});

        res.status(200).json({...requiredOrder._doc, products: data});
    } catch (error) {
        res.status(400).json({message: error.message});
    }
}

export const getAllOrders = async (req, res) => {
    try {

        const id = req.body.id;

        // verify the user's role by calling the `User` service
        try {
            await axios.post(`${USER_BASEURL}/role`, {id, role: 'ADMIN'})
        } catch (e) {
            const {response} = e;
            return res.status(response.status).json(response.data);
        }

        const orders = await Order.find().sort({ordered_at: -1});
        const ordersPaged = Pagination(req.query.page, orders);

        const total_pages = Math.ceil((await Order.count()) / 20);

        res.status(200).json({total_pages, orders: ordersPaged});
    } catch (error) {
        res.status(404).json({message: error.message});
    }
};

export const updateOrder = async (req, res) => {
    try {

        const orderStatus = req.body.status;
        const id = req.body.id;

        // verify the user's role by calling the `User` service
        try {
            await axios.post(`${USER_BASEURL}/role`, {id, role: 'ADMIN'})
        } catch (e) {
            const {response} = e;
            return res.status(response.status).json(response.data);
        }

        if (!['CREATED', 'PROCESSING', 'FULFILLED', 'CANCELLED'].includes(orderStatus)) {
            return res.status(400).json({message: "Invalid status, has to be CREATED, PROCESSING, FULFILLED, CANCELLED"});
```

Untitled-1 | Orders.js

server > controller > orders > Orders.js > ...

```
export const updateOrder = async (req, res) => {
        }

        const updatedOrder = await Order.findOneAndUpdate({"order_id": req.params.id}, {
            status: orderStatus,
        });

        if (!updatedOrder) {
            return res.status(404).json({message: "Order does not exist"});
        }

        res.status(200).json(updatedOrder);

    } catch (error) {
        res.status(404).json({message: error.message});
    }
};
```

Payments.js

Untitled-1 | Payments.js

server > controller > payments > Payments.js > ...

```
import {stripe} from "../../index.js";
import axios from "axios";
import jwt from "jsonwebtoken";
import generateId from "../../utils/generateId.js";
import {ORDERS_BASEURL, WEBSITE_BASE_URL} from "../../services/BaseURLs.js";

export const createCheckoutSession = async (req, res) => {
    try {
        const {products, total} = jwt.verify(
            req.body.token,
            process.env.JWT_SECRET_KEY
        );
        const {data} = req.body;
        const order_id = generateId();

        const session = await stripe.checkout.sessions.create({
            payment_method_types: ["card"],
            mode: "payment",
            line_items: products.map((product) => {
                return {
                    price_data: {
                        currency: "egp",
                        product_data: {
                            name: product.name,
                        },
                        unit_amount: parseInt(product.price * 100),
                    },
                    quantity: product.quantity || 1,
                };
            }),
            payment_intent_data: {
                metadata: {
                    order_id: order_id,
                    firstName: data.name.first,
                    lastName: data.name.last,
                    email: data.email,
                    phone_number: data.phone_number,
                    address: JSON.stringify({
                        country: data.address.country,
                        city: data.address.city,
                        area: data.address.area,
                        street: data.address.street,
                        building_number: data.address.building_number,
                        floor: data.floor,
                        apartment_number: data.address.apartment_number,
```


Untitled-1
Payments.js
server > controller > payments > Payments.js > createCheckoutSession > session > payment_intent_data
export const createCheckoutSession = async (req, res) => {
const session = await stripe.checkout.sessions.create({
payment_intent_data: {
metadata: {
address: JSON.stringify({
}),
products: JSON.stringify(
products.map((product) => {
return {
product_id: product.product_id,
name: product.name,
quantity: product.quantity,
};
})
),
total: total,
},
},
success_url: `${WEBSITE_BASE_URL}/checkout/success?order=${order_id}`,
cancel_url: `${WEBSITE_BASE_URL}/cart`,
});
res.status(201).json({url: session.url});
} catch (error) {
res.status(500).json({message: error.message});
}
};
export const webhook = async (req, res) => {
const eventType = req.body.type;
const {metadata} = req.body.data.object;
try {
if (eventType === "charge.succeeded") {
await axios.post(ORDERS_BASEURL, {data: metadata});
}
res.status(200).json(metadata);
} catch (error) {
res.status(404).json({message: error.message});
}
};

Products.js

Untitled-1 | Products.js

server > controller > products > Products.js > ...

```
import Products from '../../model/Products.js';
import jwt from 'jsonwebtoken';
import Pagination from '../../utils/pagination.js';
import CSVtoJSON from "../../utils/CSVtoJSON.js";

export const productsSearch = async (req, res) => {
    try {

        const products = await Products.find({"name": {$regex: req.query.search, $options: "i"}});

        const productsPaged = Pagination(req.query.page, products);

        const numberOfPages = Math.ceil(products.length / 2);
        res.status(200).json({total_pages: numberOfPages, products:productsPaged});

    } catch (error) {
        res.status(404).json({message: error.message});
    }
}

export const updateQuantity = async (req, res) => {
    try {
        const { products } = req.body;
        for(const product of products){
            const searchedProduct = await Products.findOne({ product_id: product.product_id });
            if(searchedProduct.stock - product.quantity <= 0){
                await Products.findOneAndUpdate({product_id: product.product_id},{ "stock": 0});
            }
            else{
                await Products.findOneAndUpdate({product_id: product.product_id},{"stock": searchedProduct.stock - product.quantity});
            }
        }
        res.status(200).json({ message: "updated" });
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
}

export const ShowProductsPerPage = async (req, res) => {
    try {
        let products = [];
```

Ln 6, Col 12 Spaces: 4 UTF-8

Untitled-1 | Products.js

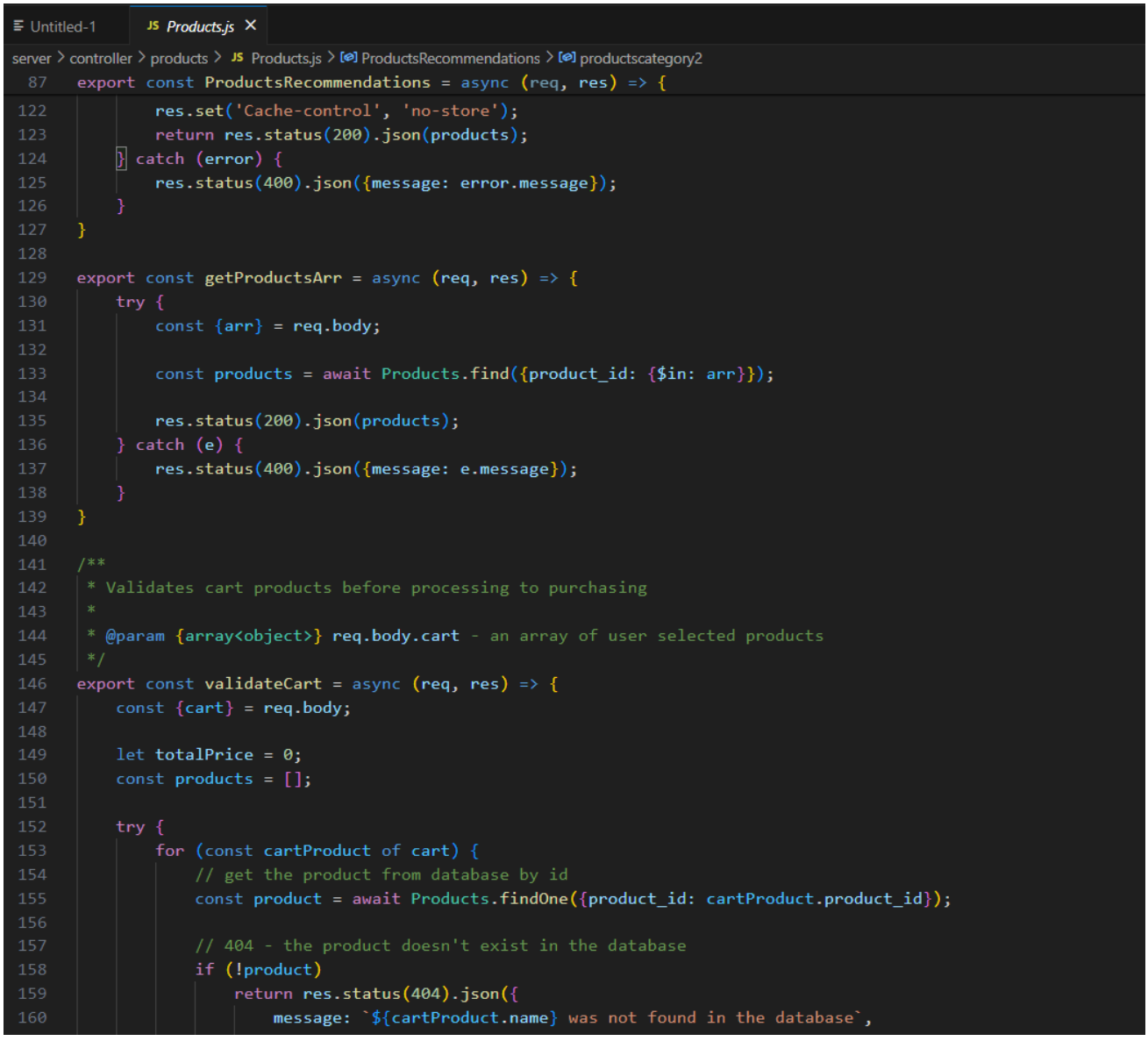
server > controller > products > Products.js > updateQuantity

```
export const ShowProductsPerPage = async (req, res) => {

        const itemsPerPage = 20;

        // If there is category: just filter them by the category,
        // then do the pagination on it.
        if (req.query.category) {
            products = await ShowProductsPerCategory(req.query.category, products);
        } else
            products = await Products.find();
        const numberOfPages = Math.ceil(products.length / itemsPerPage);
        // in both cases you have to paginate the products
        products = Pagination(req.query.page, products, itemsPerPage);


        res.status(200).json({total_pages: numberOfPages, products: products});

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

const ShowProductsPerCategory = async (category, products) => {
    try {

        products = await Products.find({"category": {$regex: category, $options: "i"}});
        return products;

    } catch (error) {
        throw error;
    }
}

export const PostProducts = async (req, res) => {
    const product = req.body;
    const newProduct = new Products(product);
    try {
        await newProduct.save();
        res.status(201).send(newProduct);

    } catch (error) {
```

Untitled-1 | Products.js

server > controller > products > Products.js > ShowProductsPerCategory

```
74  export const PostProducts = async (req, res) => {
82          res.status(409).json({message: error.message});
83      }
84
85  }
86
87  export const ProductsRecommendations = async (req, res) => {
88      try {
89          // get 2 different random categories from the database
90          let categories = await Products.aggregate([
91              {$sample: {size: 2}},
92              {$project: {category: 1, _id: 0}}
93          ]);
94          // if the categories are the same, get another two
95          while (categories[0].category === categories[1].category) {
96              categories = await Products.aggregate([
97                  {$sample: {size: 2}},
98                  {$project: {category: 1, _id: 0}}
99              ]);
100         }
101         let products = [];
102
103         // get the first category products
104         let productscategory1 =
105             await Products.aggregate([
106                 {$match: {category: categories[0].category}},
107                 {$sample: {size: 5}},
108                 {$match: {stock: {$gt: 0}}}
109             ]);
110
111         // get the second category products
112         let productscategory2 =
113             await Products.aggregate([
114                 {$match: {category: categories[1].category}},
115                 {$sample: {size: 5}},
116                 {$match: {stock: {$gt: 0}}}
117             ]);
118
119         products.push({category: categories[0].category, products: productscategory1});
120         products.push({category: categories[1].category, products: productscategory2});
121
```


Untitled-1
Products.js
server > controller > products > Products.js > ProductsRecommendations > productscategory2
87 export const ProductsRecommendations = async (req, res) => {
122 res.set('Cache-control', 'no-store');
123 return res.status(200).json(products);
124 } catch (error) {
125 res.status(400).json({message: error.message});
126 }
127 }
128
129 export const getProductsArr = async (req, res) => {
130 try {
131 const {arr} = req.body;
132
133 const products = await Products.find({product_id: {$in: arr}});
134
135 res.status(200).json(products);
136 } catch (e) {
137 res.status(400).json({message: e.message});
138 }
139 }
140
141 /**
142 * Validates cart products before processing to purchasing
143 *
144 * @param {array<object>} req.body.cart - an array of user selected products
145 */
146 export const validateCart = async (req, res) => {
147 const {cart} = req.body;
148
149 let totalPrice = 0;
150 const products = [];
151
152 try {
153 for (const cartProduct of cart) {
154 // get the product from database by id
155 const product = await Products.findOne({product_id: cartProduct.product_id});
156
157 // 404 - the product doesn't exist in the database
158 if (!product)
159 return res.status(404).json({
160 message: `${cartProduct.name} was not found in the database`,

Untitled-1 | Products.js

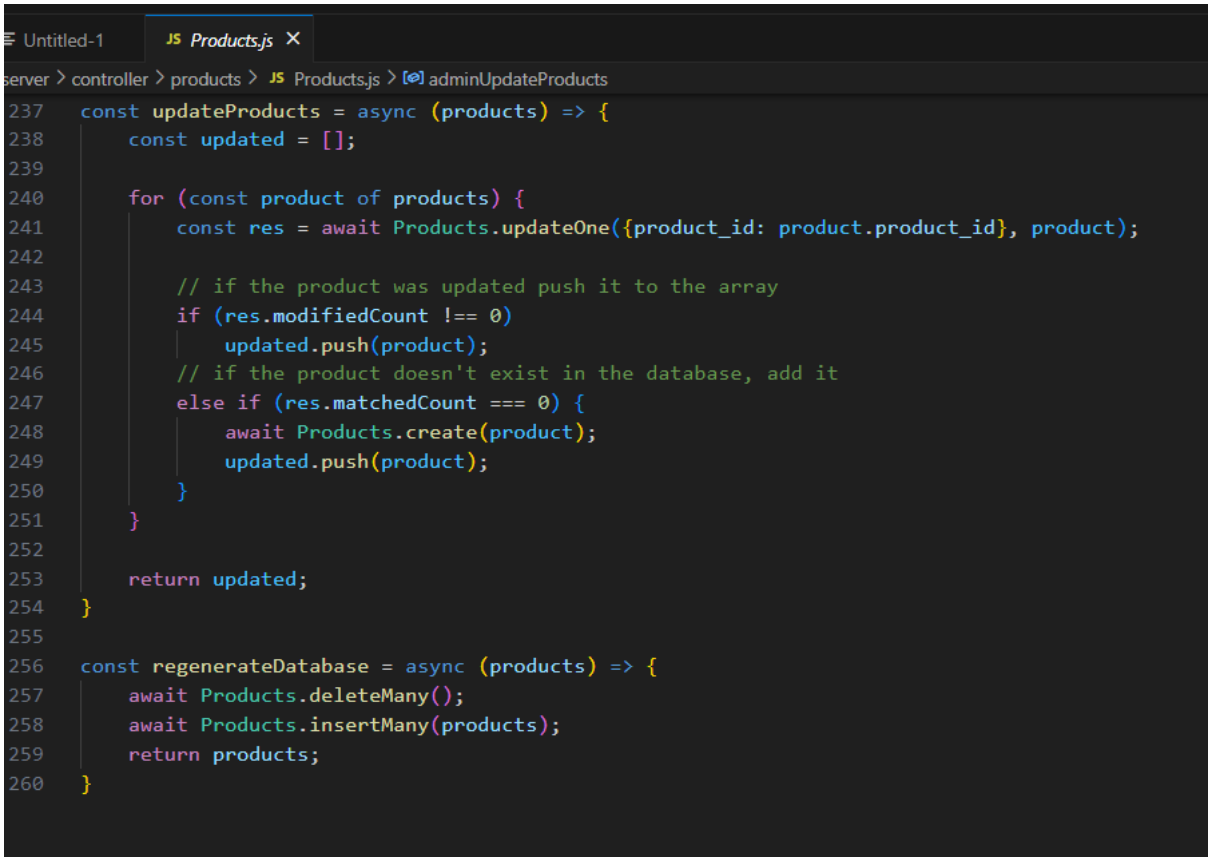
server > controller > products > Products.js > validateCart

```
146  export const validateCart = async (req, res) => {
161                  product_id: cartProduct.product_id,
162              });
163
164          // 400 - the product is out of stock
165          if (!product.stock)
166              return res.status(400).json({
167                  message: `${product.name} is out of stock`,
168                  product_id: product.product_id
169              });
170
171          // 400 - product stock is not enough for purchase
172          if (product.stock < cartProduct.quantity)
173              return res.status(400).json({
174                  message: `Not enough stock for ${product.name} to complete the purchase. Requested items: ${cartProduct.quantity}`,
175                  product_id: product.product_id
176              });
177
178          // calculate total price from the database
179          totalPrice += product.price * cartProduct.quantity;
180
181          // add products data to the `products` array
182          products.push({
183              product_id: product.product_id,
184              name: product.name,
185              price: product.price,
186              quantity: cartProduct.quantity
187          });
188      }
189
190      // round to 2 decimals
191      totalPrice = totalPrice.toFixed(2);
192
193      // generate validation/checkout token
194      const token = jwt.sign(
195          {products: products, total: totalPrice},
196          process.env.JWT_SECRET_KEY,
197          {expiresIn: process.env.JWT_CHECKOUT_TTL});
198
```

Untitled-1 | Products.js

server > controller > products > Products.js > validateCart

```
146  export const validateCart = async (req, res) => {
198
199          // validated successfully
200          return res.status(200).json({
201              total: totalPrice,
202              cart: cart,
203              token: token
204          });
205      } catch (e) {
206          // internal error
207          return res.status(500).json({
208              message: e.message
209          });
210      }
211  }
212
213  export const adminUpdateProducts = async (req, res) => {
214      const {csv, mode} = req.body;
215
216      if (!['UPDATE', 'REGENERATE'].includes(mode))
217          return res.status(400).json({message: " Unknown mode"});
218
219      const productsJson = CSVtoJSON(csv);
220
221      try {
222          if (mode === "UPDATE") {
223              const updatedProducts = await updateProducts(productsJson);
224              return res.status(200).json(updatedProducts);
225          }
226
227          if (mode === 'REGENERATE') {
228              const updatedProducts = await regenerateDatabase(productsJson);
229              return res.status(200).json(updatedProducts);
230          }
231
232      } catch (e) {
233          res.status(400).json({message: e.message});
234      }
235  }
236
237  const updateProducts = async (products) => {
```


Untitled-1
Products.js
server > controller > products > Products.js > adminUpdateProducts
237 const updateProducts = async (products) => {
238 const updated = [];
239
240 for (const product of products) {
241 const res = await Products.updateOne({product_id: product.product_id}, product);
242
243 // if the product was updated push it to the array
244 if (res.modifiedCount !== 0)
245 updated.push(product);
246 // if the product doesn't exist in the database, add it
247 else if (res.matchedCount === 0) {
248 await Products.create(product);
249 updated.push(product);
250 }
251 }
252
253 return updated;
254 }
255
256 const regenerateDatabase = async (products) => {
257 await Products.deleteMany();
258 await Products.insertMany(products);
259 return products;
260 }

Shipping.js

Untitled-1 | Shipping.js

server > controller > shipping > Shipping.js > ...

```
import Shipments from '../../model/Shipments.js';
import Pagination from "../../utils/pagination.js";
import axios from "axios";
import {USER_BASEURL} from "../../services/BaseURLs.js";

export const getShipmentId = async (req, res) => {
    const {id} = req.params;
    try {
        const your_shipment = await Shipments.findOne({"order_id": id});

        if (!your_shipment)
            return res.status(400).json({message: 'Invalid order id'});

        return res.status(200).json({
            order_id: your_shipment.order_id,
            total: your_shipment.total,
            address: your_shipment.address,
            status: your_shipment.status,
            ordered_at: your_shipment.ordered_at
        });
    } catch (e) {
        return res.status(400).json({message: e.message});
    }

}


export const updateShipments = async (req, res) => {
    try {
        const {status, id} = req.body;

        // verify the user's role by calling the `User` service
        try {
            await axios.post(`${USER_BASEURL}/role`, {id, role: 'ADMIN'})
        } catch (e) {
            const {response} = e;
            return res.status(response.status).json(response.data);
        }

        if (!status)
            return res.status(400).json({message: "Please provide the new status"});

        if (status !== 'CREATED' && status !== 'SHIPPED' && status !== 'DELIVERED' && status !== 'RETURNED')
            return res.status(400).json({message: "Please re-type the status correctly "});

        const order_id = req.params.id;

        const shipmentResponse = await Shipments.findOneAndUpdate({order_id}, {status});

        return res.status(200).json(shipmentResponse);
    } catch (e) {
        return res.status(400).json({message: e.message});
    }
}

export const postShipments = async (req, res) => {
    const {ordered_at, order_id, address, total} = req.body;

    const newShippment = new Shipments({
        total: total,
        ordered_at: ordered_at,
        address: address,
        order_id: order_id
    });
```

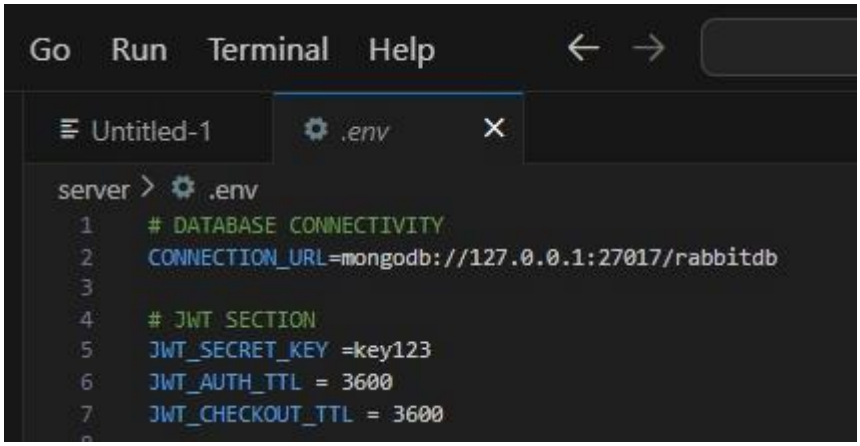

Untitled-1
Shipping.js
server > controller > shipping > Shipping.js > updateShipments > shipmentResponse
export const postShipments = async (req, res) => {
try {
await newShippment.save();
res.status(200).json(newShippment);
} catch (e) {
res.status(409).json({message: e.message});
}
}
export const getShipments = async (req, res) => {
try {
const id = req.body.id;
// verify the user's role by calling the `User` service
try {
await axios.post(`${USER_BASEURL}/role`, {id, role: 'ADMIN'})
} catch (e) {
const {response} = e;
return res.status(response.status).json(response.data);
}
const {page} = req.query;
const shipments = await Shipments.find().sort({ordered_at: -1});
const total_pages = Math.ceil(shipments.length / 20);
const pagedShipments = Pagination(page, shipments, 20);
res.status(200).json({total_pages, shipments: pagedShipments});
} catch (e) {
res.status(400).json({message: e.message});
}
}

DATABASE CONNECTIVITY

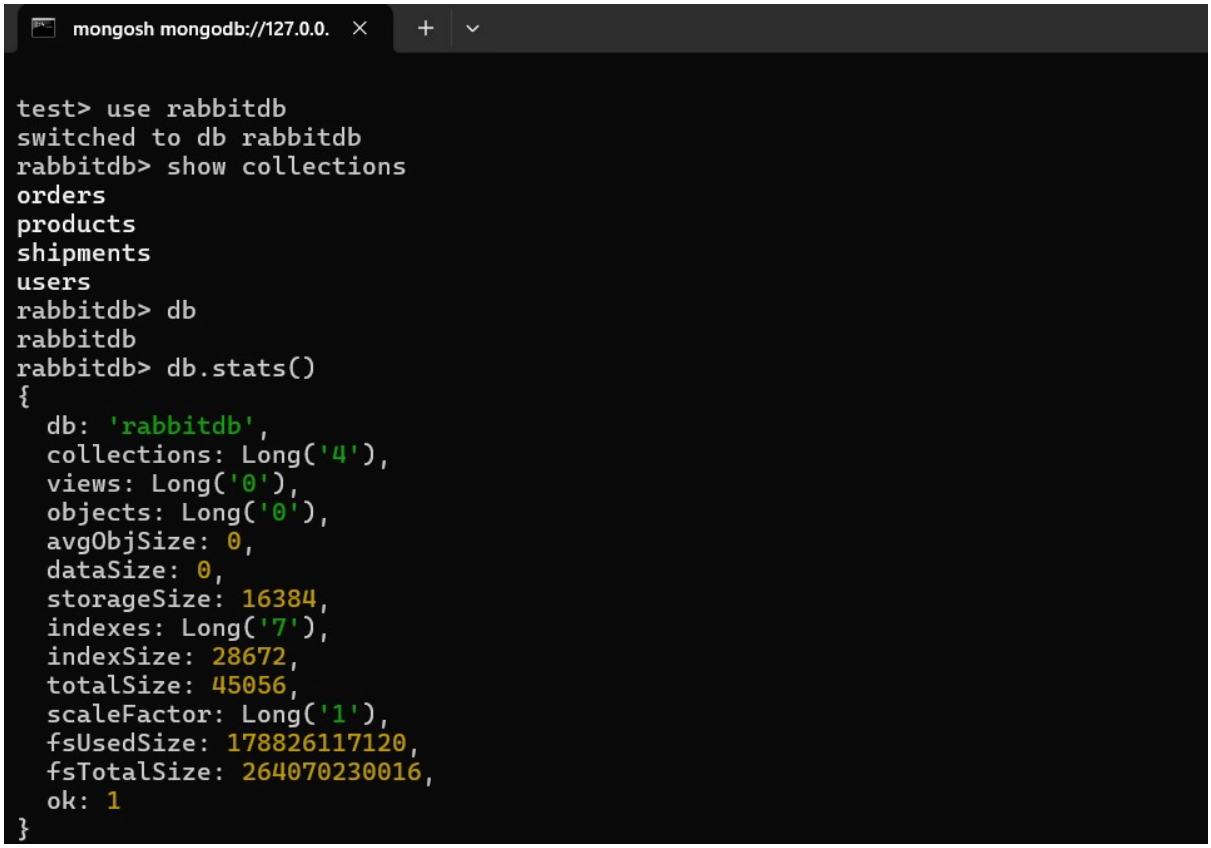
MongoDB:

MongoDB serves as the database management system for storing various data related to the virtual grocery store, such as product details, user profiles, shopping carts, and order history. Its document-oriented data model aligns well with the flexible nature of e-commerce data, allowing for easy storage and retrieval of JSON-like documents representing products, users, and transactions. MongoDB's scalability and performance characteristics make it suitable for handling the dynamic and growing data requirements of a virtual grocery store application.

Code:



Database created successfully according to the format of the project:



Users added successfully to the database: