UseCase diagram

Use Case: Place Order

- **Actor(s):** Customer
- **Preconditions:**
  - User must be logged in.
  - Shopping cart must contain at least one product.
- **Description:**
  - The customer places an order after selecting products from the cart.
- **Flow of Events:**
  1. User navigates to the shopping cart.
  2. User reviews the selected items.
  3. User proceeds to checkout.
  4. The system verifies payment details.
  5. If payment is successful, the system confirms the order.
  6. The system updates the order history.
- **Postconditions:**
  - Order is placed successfully.
  - Order appears in the order history.
- **Exceptions:**
  - Payment failure leads to order cancellation.
  - Out-of-stock items prevent order confirmation.

Use Case: Login and Signup

- **Actor(s):** Customer
- **Preconditions:**
  - User must not be logged in.
- **Description:**
  - Allows a customer to create an account or log into an existing account.
- **Flow of Events:**
  1. User navigates to the login/signup page.
  2. User enters login credentials or chooses to sign up.
  3. If logging in:
     - System verifies credentials.
     - If correct, the user is granted access.
     - If incorrect, an error message is shown.
     4. If signing up:
     - User provides necessary details (name, email, password, etc.).
     - System validates the input and creates the account.
     - System sends a confirmation email.
- **Postconditions:**
  - If successful, the user is logged in or account is created.
  - If unsuccessful, appropriate error messages are displayed.
- **Exceptions:**
  - Incorrect credentials result in a failed login attempt.
  - Weak passwords are rejected during signup.

Use Case: Add Product (Admin Only)

- **Actor(s):** Admin
- **Preconditions:**
  - Admin must be logged in.
- **Description:**
  - Admin adds a new product to the system.
- **Flow of Events:**
  1. Admin navigates to the "Manage Product" section.
  2. Admin selects the "Add Product" option.
  3. Admin enters product details (name, description, price, stock, etc.).
  4. System validates the input.
  5. If valid, the product is added to the database.
- **Postconditions:**
  - The new product is available for customers to view.
- **Exceptions:**
  - If mandatory fields are missing, an error message is displayed.

Use Case: Cancel Order

- **Actor(s):** Customer
- **Preconditions:**
  - User must have placed an order.
- **Description:**
  - Customers can cancel an order before it is shipped.
- **Flow of Events:**
  1. User navigates to "Order History."
  2. User selects the order they want to cancel.
  3. User clicks on "Cancel Order."
  4. System verifies order status.
  5. If cancellation is allowed, the order is canceled.
  6. System updates the order status to "Canceled."
- **Postconditions:**
  - Order status changes to "Canceled."
- **Exceptions:**
  - If the order is already shipped, cancellation is not allowed.

Use Case: Search Product

- **Actor(s):** Customer
- **Preconditions:**
  - User must have access to the product catalog.
- **Description:**
  - Allows customers to search for products by name, category, or keywords.
- **Flow of Events:**
  1. User enters search keywords in the search bar.
  2. System retrieves matching products.
  3. System displays search results.
- **Postconditions:**
  - Search results are shown based on user input.

- **Exceptions:**
  - No matching products lead to a "No results found" message.

Sequence Diagram

Scenario 1: Customer Adds Product to Cart

- **Actors Involved:** Customer, Frontend, Backend, Database
- **Preconditions:**
  - The customer is logged in.
  - The product catalog is loaded.
- **Flow of Events:**
  1. Customer selects a product and adds it to the cart.
  2. Frontend sends an "Update Cart" request to the Backend.
  3. Backend stores the cart data in the Database.
  4. Database confirms the cart update to the Backend.
  5. Backend sends a confirmation response to the Frontend.
  6. Frontend displays the updated cart to the customer.
- **Postconditions:**
  - The product is added to the cart.
  - The cart reflects the updated product list.
- **Exceptions:**
  - If the product is out of stock, the cart update fails.
  - If the customer’s session expires, they must log in again.

Scenario 2: Customer Makes Payment

- **Actors Involved:** Customer, Frontend, Backend, Database, Payment Gateway
- **Preconditions:**
  - The customer has added items to the cart.
  - The checkout process is completed.
- **Flow of Events:**
  1. Customer initiates the payment process.
  2. Frontend sends the payment request to the Backend.
  3. Backend forwards the request to the Payment Gateway.
  4. Payment Gateway processes the payment and confirms success or failure.
  5. If successful, Backend updates the order status in the Database.
  6. Database stores the order details and confirms the update.
  7. Backend sends an "Order Confirmation" to the Frontend.
  8. Frontend displays the confirmation to the Customer.
- **Postconditions:**
  - If successful, the order is placed.
  - If unsuccessful, the customer is prompted to retry payment.
- **Exceptions:**
  - Payment failure due to insufficient balance.
  - Network issues causing a timeout.

Class Diagram

1. User Class

- **Attributes:**
  - Contains user details such as UserName, UserEmail, UserType, and Password.
- **Methods:**
  - login(): Authenticates a user.
  - logout(): Logs the user out.
  - updatePassword(): Changes the password after validation.

2. Admin Class

- **Attributes:**
  - Stores admin details including adminName, adminEmail, and adminContact.
- **Methods:**
  - manageProducts(): Allows the admin to add, edit, or delete products.
  - viewSalesAnalytics(): Retrieves sales analytics data.
  - manageUsers(): Enables user account management.

3. Customer Class

- **Attributes:**
  - Includes customer-related fields like CustomerName, CustomerEmail, and CustomerContact.
- **Methods:**
  - browseProducts(): Fetches a list of available products.
  - addToCart(): Adds a product to the shopping cart.
  - placeOrder(): Places an order using the cart.
  - viewOrderHistory(): Retrieves past orders.

4. Address Class

- **Attributes:**
  - Stores location details (cityVillage, pincode, state, country, streetOrSociety).
- **Methods:**
  - updateAddress(): Modifies the address details.
  - validatePincode(): Ensures the pincode is valid.

5. Product Class

- **Attributes:**
  - Contains product information (Name, Description, Price, Stock, Rating, Category).
- **Methods:**
  - updateStock(): Adjusts stock quantity.
  - calculateDiscountedPrice(): Computes the final price after applying a discount.

6. ShoppingCart Class

- **Attributes:**
  - Represents a customer's cart containing multiple CartItem objects.
- **Methods:**
  - addItem(): Adds a new item to the cart.
  - removeItem(): Removes an item from the cart.
  - calculateTotal(): Computes the total cart price.
  - clearCart(): Empties the cart.

7. Order and OrderItem Classes

- **Attributes:**
  - Order contains orderItems, totalPrice, and orderStatus.
  - OrderItem represents individual products in an order.
- **Methods:**
  - updateOrderStatus(): Modifies the order status.
  - cancelOrder(): Cancels an order.
  - calculatePrice(): Computes the total cost of order items.

Relationships

- User can either be an Admin or a Customer.
- Admin manages Products.
- Customer owns a ShoppingCart and places Orders.
- Orders contain multiple OrderItems.
- CartItem references a Product.
- OrderDetail links to Order and Address.

State Diagram

1. Initial State: Idle

- This is the **starting state** of the system.
- The system remains idle until an **admin or customer interacts** with it.

2. User Authentication

- A user (either **customer** or **admin**) initiates login (User_Login).
- If authentication is successful, the user moves to the Authenticated state.
- If authentication fails, the system remains at the Idle state until correct credentials are provided.

3. Customer Workflow

- Once authenticated, the customer can enter the Browsing state.
- If the customer adds a product to the cart, the system transitions to Cart_Management.
- When the customer proceeds to checkout, the state changes to Order_Placed.
- The system enters the Payment_Processing state, where:
    - If payment is **successful**, it moves to Order_Confirmed.
    - If payment **fails**, it transitions to Order_Failed, and the user can retry.
- After order confirmation, the **admin updates** the order to Shipped.
- Finally, the order reaches the Delivered state, completing the customer transaction.

4. Admin Workflow

- If an **admin logs in**, they are directed to the Admin_Dashboard.
- The admin can perform various tasks:
    - Product_Management (Adding, Editing, Deleting products)
    - Order_Management (Tracking orders and updating order status)
    - User_Management (Managing customer profiles)
- After performing tasks, the admin can **log out**, returning to the Idle state.

5. Logout and Termination

- A user (admin or customer) can log out at any time, returning to Idle.
- Once an order is successfully delivered, the transaction is **completed**.

GreenCart System Architecture Diagram - Description

1. Overview

The **GreenCart System Architecture Diagram** illustrates the high-level structure of the GreenCart platform, defining interactions between different system components. It follows a **three-tier architecture**, consisting of the **Client-Side (Frontend)**, **Server-Side (Backend)**, and **Database Layer**.

2. Architectural Components

A. Client-Side (Frontend)

The frontend serves as the user interface for both customers and administrators. It is built using **ReactJS**, ensuring a responsive and dynamic experience.

- **ReactJS User Interface**:
  - Acts as the main interface for both customers and admins.
  - Renders UI components, handles user interactions, and communicates with the backend via API requests.
- **Customer Interface**:
  - Provides functionalities such as product browsing, cart management, and order placement.
- **Admin Interface**:
  - Allows administrators to manage products, orders, and user data.

Communication:

- The **ReactJS User Interface** interacts with the backend via **HTTP requests** to perform authentication, product management, and other operations.
- Both **Customer and Admin Interfaces** rely on the ReactJS frontend to access the system.

B. Server-Side (Backend)

The backend is implemented using **Node.js**, handling business logic, API endpoints, and secure communication with the database.

- **Node.js Business Logic**:
  - Processes requests from the frontend and interacts with the database.
  - Implements core functionalities such as authentication, product management, order processing, and cart operations.
- **APIs for Authentication, Product Management, etc.**:
  - Exposes RESTful APIs to facilitate communication between the frontend and backend.
  - Ensures secure user authentication, product listing updates, order processing, and cart management.

Communication:

- The backend receives **HTTP requests** from the frontend and processes them accordingly.
- It interacts with the **MongoDB database** for data retrieval and updates.

C. Database Layer (MongoDB)

The database layer utilizes **MongoDB**, a NoSQL database, to store and manage data efficiently.

- **Product Details**:
  - Stores product-related information such as name, description, price, stock, and categories.
- **Customer Profiles**:
  - Maintains user details, including credentials, addresses, order history, and preferences.
- **Order & Cart Information**:
  - Tracks user orders, payment status, and shopping cart items.

Communication:

- The backend communicates with **MongoDB** through queries and updates.
- Data is retrieved and sent to the frontend via APIs.

3. Data Flow and Interactions

1. The **Customer/Admin Interface** interacts with the **ReactJS User Interface**.
2. The **ReactJS frontend** sends **HTTP requests** to the backend APIs for authentication, product retrieval, cart management, and order placement.
3. The **Node.js backend** processes the requests, executes business logic, and interacts with **MongoDB** to store or retrieve data.
4. The **MongoDB database** responds to backend queries, providing necessary data for user requests.
5. The backend sends the response back to the frontend, which then updates the UI accordingly.