**Enterprise Appl Modelling**

**Project**

**Meghna Rajubhai Ramsnehi**

**N01514629**

**Hamed Tara**

**N01540404**

**This part of the project was collaboratively completed by Meghna and Hamed. We worked together in meetings to achieve this. You can review our code on GitHub at the following link: https://github.com/mmistie/SpringBoot-Group-Project-v1**

**BackEnd**

**Admin Controller**

**A screenshot of a computer

Description automatically generated**

1. **Product Management: It includes endpoints to retrieve all products, create a new product, update an existing product by ID, and delete a product by ID. These operations interact with a productService to handle business logic.**
2. **Order Management: There is an endpoint to retrieve all orders, utilizing an orderService for data handling. This allows administrative users to view order details.**
3. **Services Autowiring: The controller uses Spring's @Autowired annotation to inject the necessary services (AdminService, productService, orderService) that the controller methods depend on for processing the data.**
4. **API Mapping: The controller is mapped to handle requests at the /api/admin base URL, indicating that these endpoints are part of the admin functionality of the application.**
5. **Response Entities: It uses ResponseEntity to encapsulate HTTP responses, allowing for rich HTTP status code manipulation, which is suitable for REST APIs ensuring proper client-server communication.**

**Admin Model**

**A screenshot of a computer

Description automatically generated**

**This Java class, Admin, defines an entity representing an administrator in a database. It includes annotations for a JPA entity with fields for ID, username, password, and email. The class uses Lombok to automatically generate constructors, getters, and setters, simplifying the code. The @Table(name = "admin") annotation specifies that this entity is mapped to the "admin" table in the database**

**Image Controller**

**A screenshot of a computer program

Description automatically generated**

1. **Image Upload: It includes a POST endpoint at /images/upload where users can upload images. This endpoint receives an image file, saves it using the ImageService, and returns the saved Image object in the response.**
2. **Fetch Image: There's a GET endpoint at /images/{id} for retrieving an image by its ID. It returns the image data as a byte array and sets the appropriate content type in the response header.**
3. **Service Injection: The controller uses Spring's @Autowired to inject ImageService, which encapsulates the business logic for handling image storage and retrieval.**
4. **Base URL Mapping: The controller is mapped to /images, serving as the base URL for all related image operations.**
5. **Response Handling: The controller uses ResponseEntity to provide HTTP status codes and headers, allowing for detailed client-server communication about the outcome of requests.**

**Image Model**

**A screenshot of a computer

Description automatically generated**

**The Java class Image represents an image entity in a database, annotated with JPA to handle ORM (Object-Relational Mapping). It includes fields for id, filename, contentType, and data, where data is stored as a large object (LOB). The class uses Lombok to generate getters and setters for the fields, simplifying the code structure.**

**Image Service**

**A screenshot of a computer program

Description automatically generated**

1. **Image Storage: It provides a saveImage method that constructs an Image object from a MultipartFile, setting its filename, content type, and data before saving it via ImageRepository.**
2. **Image Retrieval: The getImageById method retrieves an image by its ID using the ImageRepository, returning the image or null if not found.**
3. **Repository Integration: The service is tightly integrated with ImageRepository, leveraging Spring Data JPA for database operations.**
4. **Exception Handling: The saveImage method includes handling for IOException, which is necessary when extracting bytes from the MultipartFile.**

**Orders : Config**

**A screenshot of a computer

Description automatically generated**

1. **Global Settings: It applies the CORS policy to all routes (/\*\*) in the application.**
2. **Allowed Origins: It specifically allows requests from the origin http://localhost:3000, which is useful during development when the front-end runs on a different server or port than the back-end.**
3. **Allowed Methods: It permits HTTP methods such as GET, POST, PUT, and DELETE, enabling these operations from the specified origin.**
4. **Allowed Headers: It allows all headers (\*), ensuring that requests can include any required HTTP headers.**

**Oerders : Controller**

**A screenshot of a computer program

Description automatically generated**

1. **Endpoint Configuration: It's annotated with @RestController and @RequestMapping("/orders"), serving as the base URL for order-related operations.**
2. **CORS Support: It includes a @CrossOrigin(origins = "http://localhost:3000") annotation to allow cross-origin requests from the specified URL, useful in development environments.**
3. **Create Order: The @PostMapping endpoint handles the creation of new orders, saving them via orderService and returning the saved order in the response.**
4. **Get Order by ID: The @GetMapping("/{id}") endpoint retrieves an order by its ID, returning it if found or a 404 status if not.**
5. **List All Orders: The @GetMapping endpoint lists all orders, utilizing orderService to fetch and return them encapsulated in a ResponseEntity.**

**A screenshot of a computer

Description automatically generatedOrders : model (Order)**

1. **Endpoint Configuration: Configured with @RestController and @RequestMapping("/orders") to define the base URL for handling order operations.**
2. **CORS Support: Uses the @CrossOrigin(origins = "http://localhost:3000") annotation to enable handling requests from a specific development environment URL.**
3. **Order Operations: Provides endpoints for creating an order (@PostMapping), retrieving an order by ID (@GetMapping("/{id}")), and listing all orders (@GetMapping).**
4. **Service Integration: Utilizes orderService for the business logic, including saving and retrieving orders, and encapsulates responses in ResponseEntity for proper HTTP response management.**

**Model – order (OrderItems)**

**A screenshot of a computer program

Description automatically generated**

1. **Entity and Table Definition: Marked as an entity with @Entity and mapped to the "order\_items" table using @Table(name = "order\_items").**
2. **Primary Key: It uses @Id and @GeneratedValue(strategy = GenerationType.IDENTITY) for its primary key, automatically generating values.**
3. **Relationships: Includes many-to-one relationships with order and product entities, linking each order item to specific orders and products using @JoinColumn.**
4. **Basic Attributes: Contains a quantity field marked with @Column(nullable = false), indicating it must always have a value.**
5. **Lombok Annotations: Utilizes @NoArgsConstructor, @AllArgsConstructor, @Getter, and @Setter from Lombok to automatically generate constructors, getters, and setters, simplifying boilerplate code.**

**Order (service)**

**A computer screen shot of a black screen

Description automatically generated**

1. **Service Dependencies: It autowires OrderRepository and ProductRepository to access and manipulate order and product data stored in the database.**
2. **Order Saving Logic: Implements a save method that calculates the total price of an order by summing the total prices of all order items, which are derived by multiplying the product price by quantity, and then saves the order using orderRepository.**
3. **Order Retrieval: The findById method retrieves an order by its ID, optionally triggering the lazy loading of associated order items to ensure they are fetched from the database.**
4. **List All Orders: The findAll method fetches all orders from the database, similarly triggering lazy loading for each order's items to populate them completely.**
5. **Business Logic Integration: Incorporates complex business logic to handle calculations and data integrity directly within the service layer, facilitating robust data operations and ensuring accurate order processing**

**Products : Config**

**A screenshot of a computer program

Description automatically generated**

**Global CORS Settings: It applies CORS policies universally to all routes (/\*\*) within the application.**

**Specific Configurations: Allows requests from http://localhost:3000, supporting methods like GET, POST, PUT, and DELETE, and permits all headers, facilitating development and testing across different server environments.**

**Implementation Details: This configuration is part of the Spring MVC framework, implemented through the WebMvcConfigurer interface, ensuring consistent behavior across the application.**

**Product Controllers**

**A screenshot of a computer program

Description automatically generated**

1. **Base URL and CORS: Configured with @RestController and @RequestMapping("/products"), serving as the base URL for product operations, and allows cross-origin requests from http://localhost:3000.**
2. **Create Product: The @PostMapping endpoint enables the creation of a new product, where the product details are passed in the request body, saved using productService, and the saved product is returned.**
3. **Delete Product: The @DeleteMapping("/{id}") endpoint allows for the deletion of a product by its ID, handling the operation via productService and returning a no-content response upon successful deletion.**
4. **Fetch Product by ID: The @GetMapping("/{id}") endpoint retrieves a specific product by its ID, returning the product if found, or a not-found status otherwise.**
5. **List All Products: The @GetMapping endpoint lists all available products, fetching them through productService and returning them in a list encapsulated in a ResponseEntity.**

**Product Model   
  
A screenshot of a computer

Description automatically generated**

1. **The product class defines a JPA entity for managing product data in a database within a Java Spring application:**
2. **Entity Setup and Annotations: Marked as an entity using @Entity, it has fields such as id, name, description, price, stock, and imageId, with id being automatically generated.**
3. **Lombok Integration: Utilizes Lombok annotations @NoArgsConstructor, @Getter, @Setter, and @AllArgsConstructor to automatically generate constructors and accessor methods, reducing boilerplate code.**
4. **Database Mapping and Management: Each product has attributes like price (handled as a BigDecimal for precision) and stock (an integer), mapped to corresponding columns in the database, facilitating CRUD operations on product data.**

**ProductService   
  
A screenshot of a computer

Description automatically generated**

1. **Service Declaration: It's annotated with @Service, indicating that it is a Spring-managed service class dedicated to business logic related to product operations.**
2. **Repository Integration: Uses @Autowired to inject ProductRepository, facilitating direct access to CRUD operations on the product database.**
3. **Product Saving: The save method allows for saving or updating a product entity in the database using the productRepository.**
4. **Product Deletion: The deleteById method provides functionality to delete a product by its ID, simplifying the management of product data.**
5. **Product Retrieval: Includes methods to find a single product by ID (findById), which returns null if not found, and to retrieve all products (findAll), enhancing data access flexibility.**

**Frond End**

**CartContex.js**

**A screen shot of a computer

Description automatically generated**

1. **This React component manages a shopping cart context using React's Context API, providing a way to pass the cart data through the component tree without having to pass props down manually at every level:**
2. **Context Creation: It defines CartContext using createContext(), which enables components to subscribe to context changes without directly interacting with props.**
3. **State Management: Utilizes useState to maintain the cart state, which starts as an empty array and updates through specific actions like adding or removing items.**
4. **Cart Operations: Defines functions within the CartProvider:**
5. **addToCart: Adds a product to the cart.**
6. **removeFromCart: Removes a product from the cart based on its ID.**
7. **clearCart: Clears all items from the cart.**
8. **Provider Setup: The CartProvider component wraps its children with CartContext.Provider, passing the cart state and control functions as context to be available to all nested components.**
9. **Custom Hook useCart: Implements a custom hook that provides an easy way for components to access the cart context. It ensures the hook is used within a component wrapped by CartProvider, throwing an error otherwise, which helps in maintaining proper usage and avoiding runtime errors.**

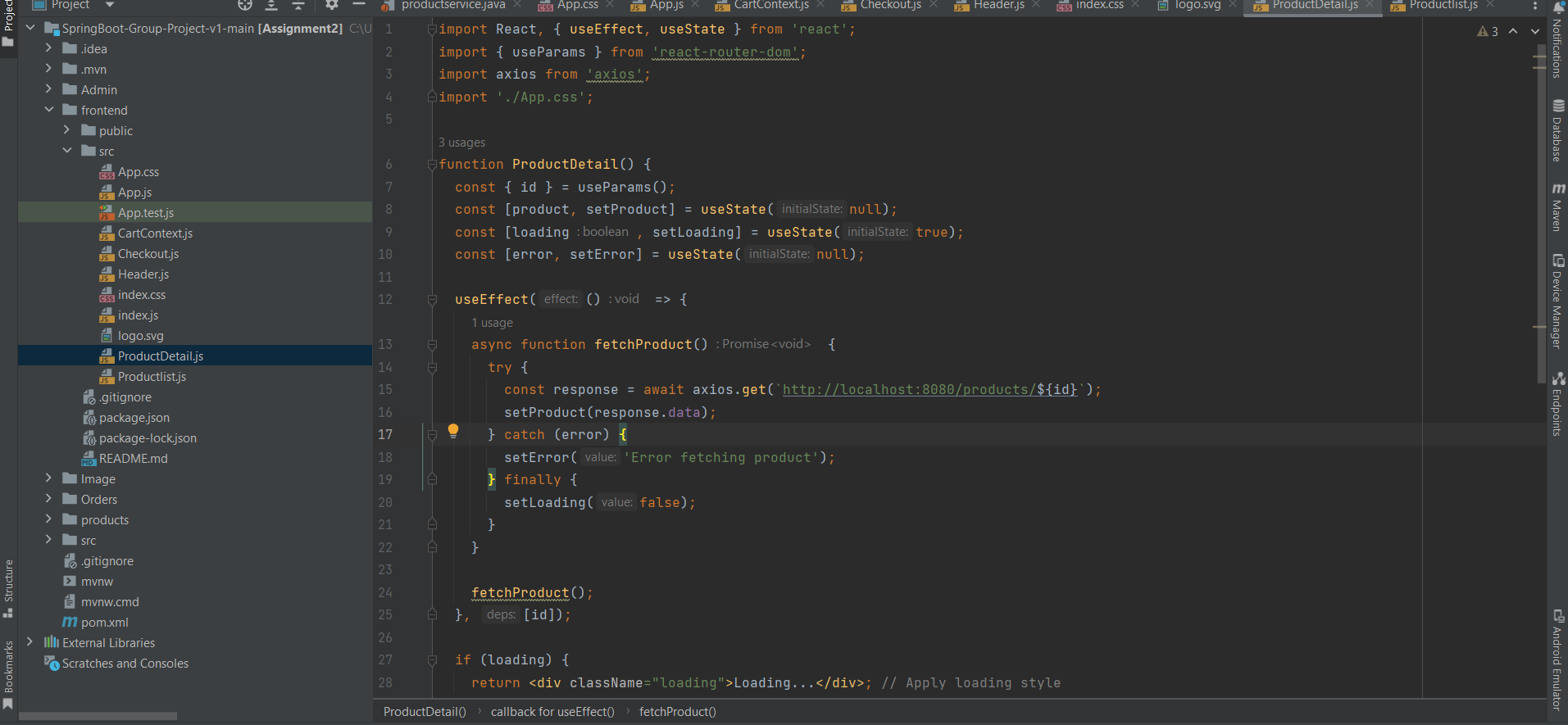
**CheckOut.js**

**A screen shot of a computer

Description automatically generated**

1. **State and Context Use: Utilizes useState to manage local state for error messages, order placement status, and quantities of items. It also integrates the useCart context to access and manipulate the cart items.**
2. **Order Placement: Implements a handlePlaceOrder function that checks if the cart is empty, computes the order details (including adjusting quantities and calculating total prices), and sends the order to a backend service using axios. It also handles the successful and error states of the order placement.**
3. **Price Calculation: Includes functions to calculate the total price of items in the cart and the tax based on a predefined tax rate, facilitating real-time updates in the UI as quantities change.**
4. **Interactive UI Elements: Provides interactive controls for each cart item to adjust quantities, remove items, and view pricing details. After placing an order, it displays a confirmation message or error messages as appropriate.**
5. **Styling and Structure: The component structure includes headers for checkout and cart items, detailed list items for each product in the cart with action buttons for quantity adjustment and removal, and a breakdown of subtotal, tax, and total amounts, enhancing user interaction and visual layout.**

**ProductDetail.js**

****

1. **Dynamic Routing: Utilizes useParams from react-router-dom to capture the id parameter from the URL, which identifies the specific product to fetch.**
2. **State Management: Manages the component's state with useState, tracking the product details, loading status, and any errors that might occur during data fetching.**
3. **Data Fetching: Implements an useEffect hook that triggers fetchProduct, an asynchronous function that requests product data from a backend service using axios based on the product's ID. It handles both the success and error states and stops the loading spinner accordingly.**
4. **Conditional Rendering: Displays a loading indicator while data is being fetched, shows an error message if the fetch fails, and renders the product details (including an image, name, description, and price) upon successful data retrieval.**
5. **Styling and Layout: The component is styled with CSS (referred to by ./App.css), enhancing the visual presentation of the product details such as the product image and descriptive elements.**

**ProductList.js**

**A screen shot of a computer

Description automatically generated**

1. **Data Fetching and State Management: Uses useState and useEffect to manage an array of products. It fetches product data from a backend service using axios upon component mount and handles any errors during the fetch.**
2. **Cart Context Integration: Integrates with the useCart context to utilize the addToCart function, enabling users to add products directly to the shopping cart from this list.**
3. **Quantity Adjustments: Manages individual product quantities with local state, allowing users to increase or decrease the quantity of each product before adding it to the cart.**
4. **User Interaction: Each product item in the list includes interactive elements such as a quantity selector and an "Add to Cart" button, enhancing user engagement and control over purchase decisions.**
5. **Styling and Navigation: Styled with CSS for a consistent and user-friendly layout, each product name is clickable, directing users to a detailed view of the product, and products are displayed with their images for better visualization.**