EECS 4413 F24 Team Project Report

Team Composition

Team Name: 404notFound

Team Members:

1. Darren, Tsang (220056677)
2. Raveena, Rainal (217231721)

Contributions

Darren, Tsang (220056677):

Darren led the development of the core e-commerce functionality. He implemented the shopping cart system, and checkout process. He also created the product catalog, search functionality, and user authentication system. Additionally, he set up the project infrastructure, including the database design, API architecture, and deployment configuration. He also developed the admin dashboard and sales history tracking components for monitoring business performance.

Signature: Darren Tsang

Raveena, Rainal (217231721):

Raveena focused on developing the administrative components of the e-commerce system. She implemented the inventory management system which includes out-of-stock monitoring, and restocking functionalities. She also built the user management features that allow administrators to view and manage user accounts details, order history and maintain user data.

Signature: Raveena Rainal

Knowledge Sharing

Throughout the project, team members stayed informed about each other's work through weekly meetings where they demonstrated their progress and explained their code. When adding new features, each member would walk the other through their implementation, ensuring both understood how different parts of the system worked together. This helped when fixing bugs that crossed between the admin and customer sides of the system.

E-commerce System Design Document

Table of Contents

1. Introduction

This project is about creating an online store where customers can browse and purchase products, and administrators can manage inventory, orders, and user accounts. The goal is to provide a simple, reliable, and efficient system that meets the needs of both shoppers and store managers.

The system is built using the MERN stack (MongoDB, Express.js, React, and Node.js) and follows the Model-View-Controller (MVC) design pattern. The frontend, built with React and Vite, focuses on delivering a responsive and easy-to-use interface. The backend, created with Express.js, handles the core functionality like user management, product handling, and order processing.

MongoDB is used to store data, and Docker is used for easy setup and deployment.

Strengths of the system include a responsive user interface that works on various devices, persistent shopping carts that maintain data even after logging out, and user-friendly product search and filtering features. Future improvements could include implementing real-time notifications, expanding payment options, and adding advanced performance optimizations to handle larger databases more effectively.

2. Architecture Description

2.1. System Overview

The e-commerce system is designed as a modern web application that provides a comprehensive online shopping experience. At its core, the system is divided into several key components that work together seamlessly:

1. **User Interface Layer**
   - Customer-facing storefront for browsing and purchasing products
   - Admin dashboard for managing inventory, orders, and analytics
   - Responsive design that adapts to different screen sizes
   - Interactive components for cart management and checkout
2. **Application Logic Layer**
   - Product catalog management and search functionality
   - Shopping cart operations and order processing
   - User authentication and account management
   - Payment processing and transaction handling
3. **Data Management Layer**
   - Product inventory and categorization
   - User profiles and authentication data
   - Order history and transaction records
   - Shopping cart state persistence

The system supports two main user roles:

- **Customers**: Can browse products, manage shopping carts, place orders, and track order history
- **Administrators**: Can manage inventory, process orders, and access analytics dashboard

2.2. Architecture Patterns

The system implements a clear 3-tier architecture with distinct separation between frontend and backend components:

1. **Presentation Tier (Frontend)**
   - Built as a Single Page Application (SPA) using React
   - Communicates with backend exclusively through REST APIs
   - Maintains its own state management for user interface

- Located in `/frontend` directory with clear component organization

2. **Application Tier (Backend)**
   - Implements RESTful API endpoints using Express.js
   - Handles all business logic and data processing
   - Manages authentication and authorization
   - Located in `/server` directory with modular route handling
3. **Data Tier (Database)**
   - Uses MongoDB for persistent data storage
   - Implements data models and validation
   - Handles data relationships and queries
   - Maintains data integrity and consistency

The separation between frontend and backend is achieved through: - Clear API contracts between client and server - Independent deployment capabilities - Distinct error handling and logging mechanisms

2.3. Component Integration

The following architecture diagram illustrates the system's components and their interactions (see Figure 1):

The diagram shows the three main tiers of our application and how they interact:

1. **Frontend Layer**
   - Customer and Admin UIs built with React
   - Core components for product catalog, cart, account, and checkout
   - Service layer for API communication
2. **Backend Layer**
   - Express.js-based API
   - Route controllers for different business domains
   - Authentication middleware for secure access
   - Clear separation of concerns in route handling
3. **Database Layer**
   - MongoDB collections for different data domains
   - Direct interaction with route controllers
   - Persistent storage for all application data

Key interaction flows:

- Frontend components communicate through the API client
- All requests pass through authentication when required
- Each route controller handles specific business logic
- Database operations are isolated to relevant controllers

This architecture ensures:

- Clear separation of concerns
- Secure data access
- Scalable component structure

- Maintainable codebase

2.4. Use Cases

Based on the implemented functionality, here are the key use cases in our e-commerce system:

2.4.1 Product Browsing Use Case The following diagram illustrates the product browsing functionality and related use cases (see Figure 2):

This diagram shows:

- Customer interactions with the product catalog
- Search and filtering capabilities
- Product details and cart functionality
- Admin product management features

The implemented use cases include:

1. **Customer Functions**
   - Browse product catalog with pagination
   - Search products by name, brand, or description
   - Filter products by category
   - Sort products by various criteria
   - View detailed product information
   - Add products to shopping cart
2. **Admin Functions**
   - Manage product listings
   - Update product inventory

2.4.2 User Account Management Use Case The following diagram illustrates the user account management functionality for both customers and administrators (see Figure 3):

This diagram shows:

- Customer account operations
- Admin user management capabilities
- Authentication and security features

The implemented use cases include:

1. **Customer Account Operations**
   - Register new account
   - Sign in with credentials
   - View and update profile information
   - Change password
   - View order history
   - Manage shipping addresses
2. **Admin Management Functions**

- View user accounts
- Manage user status
- View user orders
- Reset user passwords

3. **Authentication System**
   - Verify user credentials
   - Generate authentication tokens
   - Validate user sessions

All operations requiring authentication include session validation to ensure security, and the system maintains clear separation between customer and administrative functions.

2.4.3 Order Processing Use Case The following diagram illustrates the order processing workflow and interactions (see Figure 4):

This diagram shows:

- Shopping cart operations
- Checkout process flow
- Order management capabilities
- Admin order processing features
- Inventory control system

The implemented use cases include:

1. **Shopping Cart Management**
   - View cart contents
   - Update item quantities
   - Remove items from cart
2. **Checkout Process**
   - Initiate checkout from cart
   - Select shipping address
   - Choose payment method
   - Place order
   - View order confirmation
3. **Order Management**
   - View order history
   - Track orders
4. **Admin Functions**
   - View all orders
   - Manage inventory levels

The system ensures proper flow between these components through:

- Automatic stock validation during checkout
- Integration between cart and order processing
- Inventory updates after order placement
- Clear separation between customer and admin functions

3. Design Description

3.1 Core Design Patterns

3.1.1 MVC (Model-View-Controller) Pattern The application implements the MVC pattern to separate concerns and improve maintainability:

Models:

- Implemented through Mongoose schemas in `/server/models/`
- Key models include:
  - `product.model.js` : Manages product data and inventory
  - `order.model.js` : Handles order processing
  - `cart.model.js` : Manages shopping cart state
  - `registration.model.js` : Handles user data and authentication

Views:

- Implemented in React frontend
- Separation of presentation logic from business logic
- Component-based architecture for reusability

Controllers:

- Implemented as Express.js route handlers
- Handle business logic and data flow
- Examples:
  - `productRoutes.js` : Product management and queries
  - `cartRoutes.js` : Shopping cart operations
  - `orderRoutes.js` : Order processing

3.1.2 DAO (Data Access Object) Pattern Implemented through Mongoose models, providing a clean separation between business logic and data persistence:

- **Abstract Interface**: Mongoose models provide a standard interface for data operations
- **Concrete Implementation**: MongoDB handles actual data storage
- **Benefits**:
  - Encapsulation of database operations
  - Consistent data access methods
  - Easy to modify database implementation

3.2 Design Pattern Implementation

Our e-commerce system incorporates several design patterns, though some are partially implemented and could benefit from further refinement:

3.2.1 Facade Pattern

- Implemented in the routes layer (`routes/*.js`)
- API routes provide a simplified interface to complex backend operations
- Examples include:
  - cartRoutes.js for abstracting cart operations
  - orderRoutes.js for handling complex order processing
  - productRoutes.js for managing product-related operations
- Effectively encapsulates multiple model operations behind clean endpoints

3.2.2 Singleton Pattern

- Implemented in database connection management (`config/db.js`)
- Features include:
  - MongoDB connection using mongoose
  - Environment-based configuration using dotenv
  - Connection status logging
- Our code currently relies on Mongoose's internal connection pooling for reuse of connections

3.3 Design Decisions and Trade-offs

3.3.1 Database Schema Design

- **Decision**: Used MongoDB with Mongoose
- **Main Schemas**:
  1. **Product Schema**:

```
{
  item_id: String,
  item_name: [{
    language_tag: String,
    value: String
  }],
  price: Number,
  brand: [{
    language_tag: String,
    value: String
  }],
  main_image: Buffer,
  quantity: Number,
  node: [{
    node_id: Number,
    node_name: String
  }]
}
```

  2. **Order Schema**:

```
{
  userId: String,
  cartId: String,
  items: [{
    product: ObjectId,  // References Product
    quantity: Number,
    price: Number
  }],
  totalAmount: Number,
  shippingAddress: String,
  billingAddress: String
}
```

- **Benefits**:
  1. **Easy to Change**:
     - Can add new product fields without changing code
     - Supports multiple languages for product names
     - Can store images directly in database
  2. **Simple Queries**:
     - No need to join tables for basic operations
     - Easy to get all product info in one query
     - Works well with product catalogs
- **Challenges**:
  1. **Keeping Data Accurate**:
     - Need to handle multiple users buying same product
     - Use Mongoose's version control to prevent conflicts
  2. **Connected Data**:
     - Orders need to link to correct products
     - Use Mongoose's ID system to connect orders to products
  3. **Loading Speed**:
     - Product images can slow down queries
     - Only load necessary fields when querying

3.3.2 Authentication Design

- **Decision**: JWT-based authentication
- **Benefits**:
  - Stateless authentication
  - Easy to implement and maintain
- **Trade-offs**:
  - Token management complexity
  - Need for secure token storage

3.3.3 API Design

- **Decision**: RESTful architecture with Express.js
- **Implementation Details**:
  - HTTP API Endpoints:

```
GET    /api/products         # List products
GET    /api/products/:id     # Get single product
GET    /api/cart/:cartId     # Get cart
POST   /api/cart/:cartId/items  # Add to cart
PUT    /api/cart/:cartId/items/:productId  # Update cart item
DELETE /api/cart/:cartId     # Delete cart
```

- **Benefits**:
  - Clear resource hierarchy and relationships
  - Standard HTTP methods for CRUD operations
  - Stateless nature improves scalability
  - Self-documenting API structure
- **Trade-offs and Challenges**:
  1. **Multiple Requests for Complex Operations**:
     - Example: Checkout process requires multiple calls:
       1. Validate cart items
       2. Check product inventory
       3. Create order
       4. Update inventory
     - Solution: Implemented transaction-like behavior in order creation endpoint
  2. **Data Transformation Overhead**:
     - MongoDB documents need transformation for client consumption
     - Example: Product images converted to base64
     - Solution: Implemented response transformation middleware (`imageUtils.js`)
  3. **N+1 Query Problem**:
     - Cart endpoints need to fetch related product data
     - Solution: Used Mongoose population to perform joins under the hood:
       `cart.populate('items.product', 'item_name price main_image quantity')`
  4. **State Management**:
     - RESTful APIs are stateless by design
     - Challenge with shopping cart persistence
     - Solution: Implemented cart ID system for both authenticated and guest users

3.4 UML Diagrams

3.4.1 Class Diagram The system's class structure is illustrated in the following diagram (see Figure 5):

This diagram shows the main components of the system and their relationships, including:

- Core models (User, Product, Order, Cart)
- Supporting models (CartItem, OrderItem, etc.)
- Relationships between components
- Field constraints and data types

3.4.2 Database Schema The MongoDB database schema is illustrated in the following diagram (see Figure 6):

This diagram illustrates the NoSQL database structure, including:

- Collection structures
- Embedded documents
- References between collections
- Field types and relationships

These diagrams provide a comprehensive view of both the object-oriented design and the underlying database structure of our e-commerce system.

4. Advanced and Distinguished Features

Our e-commerce system implements several advanced features that enhance its functionality and user experience beyond the basic requirements:

4.1. Technology Stack and Architecture

1. **Modern Frontend Development**
   - React with Vite for optimized build performance
   - Component-based architecture for reusability
   - Responsive design using modern CSS frameworks
2. **Advanced Backend Implementation**
   - Express.js with RESTful API architecture
   - MongoDB for flexible document-based storage
   - JWT-based authentication system
   - Modular route handling

4.2. Admin Data Visualization

1. **Admin Dashboard**
   - Sales history and top products charts
   - Customer management system
   - Inventory tracking system

4.3. User Experience Enhancements

1. **Shopping Experience**
   - Persistent shopping cart after log out
   - Auto merge shopping carts

        - If a user adds items to the cart while logged out, upon logging in, the app merges the logged-out cart with the user's existing saved cart.
    - Dynamic catagory filtering based on search results
2. **Security and Authentication**
    - Implementation of JWT (JSON Web Token) for secure authentication and authorization.
    - Stateless session management using JWT

4.4. Design Pattern Implementation

1. **Design Patterns**
    - Singleton pattern for database connection management
    - Facade pattern for API route organization and subsystem abstraction

5. Implementation

5.1. Technology Stack and Decisions

Technologies Considered:

Frontend Options

1. **Traditional HTML/CSS/JavaScript**
    - Pros: Simple implementation, no learning curve
    - Cons: Limited component reusability, complex state management
2. **Angular**
    - Pros: Full-featured framework, strong typing with TypeScript
    - Cons: Steep learning curve, heavier bundle size
3. **React (Selected)**
    - Pros: Component-based architecture, Virtual DOM for performance, rich ecosystem
    - Cons: Requires additional libraries for routing and state management

Backend Options

1. **Servlet/JSP**
    - Pros: Mature technology, covered in class
    - Cons: Verbose implementation, limited modern features
2. **Spring Boot**
    - Pros: Comprehensive framework, strong security
    - Cons: Heavyweight, steeper learning curve
3. **Node.js with Express (Selected)**
    - Pros: Unified JavaScript stack, async I/O, easy setup for RESTful APIs
    - Cons: Single-threaded, though mitigated by the event loop

Database Options

1. **MySQL**
   - Pros: Structured data storage, strong consistency
   - Cons: Fixed schema, less flexibility for diverse data
2. **MongoDB (Selected)**
   - Pros: Flexible schema, JSON-like documents, scalable
   - Cons: Eventually consistent, less suited for complex transactions

Final Decisions and Justifications

Frontend: React with Vite

- Offers a modern development experience, component reusability, and excellent performance with virtual DOM
- Chosen over Angular for its simplicity and smaller learning curve, and over traditional HTML/CSS/JavaScript for its maintainability and scalability

Backend: Node.js with Express

- Unified JavaScript stack simplifies development and integrates well with MongoDB
- Chosen over Servlet/JSP and Spring Boot for its ease of use, flexibility, and asynchronous capabilities

Database: MongoDB

- Chosen for its schema flexibility, natural fit with JSON data, and excellent support for rapid prototyping and scalability
- Outweighed MySQL due to the need for handling diverse and dynamic product data

5.2. Implementation Details

Frontend

- Component Structure: Reusable UI components, admin dashboard
- State Management: React Context and custom hooks
- Routing: React Router for navigation with protected routes
- UI/UX: Responsive design and interactive visualizations

Backend

- API Design: RESTful endpoints, middleware for authentication
- Security: JWT-based authentication, password hashing
- Database Integration: Mongoose schemas and query optimization
- Business Logic: Handles user authentication, order processing, inventory management

Database

- Schema Design: Modular schemas for products, users, orders, carts
- Optimization: Indexed queries, caching, and selective field projection

6. Deployment Efforts

Our deployment strategy uses Docker to containerize and deploy our MERN stack application, with MongoDB hosted on Atlas for a managed database solution. By leveraging Docker and Atlas, we ensure consistency, scalability, and simplified management across environments. Here's how we deployed the project:

1. **Docker Containers**:
   - We created separate containers for the frontend (React with Vite) and backend (Node.js/Express.js).
   - Environment variables are used for configuration, and the backend connects to MongoDB Atlas for database operations.
2. **MongoDB Atlas**:
   - MongoDB is hosted on Atlas, providing a reliable, scalable, and managed database service.
   - Connection strings and credentials are securely managed using environment variables.
3. **Docker Hub**:
   - Built images are pushed to Docker Hub for version control and easy distribution.
   - Deployment environments pull the latest images from Docker Hub.
4. **Deployment Process**:
   - Code changes are pushed to GitHub.
   - Docker images are built, tagged, and pushed to Docker Hub.
   - Docker Compose orchestrates the frontend and backend containers for local and production environments.

7. Conclusion

To complete this project, we used a combination of React, Express.js, and MongoDB to build a structured e-commerce platform. The Model-View-Controller (MVC) design pattern was applied to separate concerns between the frontend, backend, and database layers, while Docker was used for consistent deployment across development environments.

The system's main strengths include its responsive user interface, persistent shopping cart feature, and practical search and filtering capabilities. However, areas needing improvement include expanding payment options, adding real-time notifications, and improving system performance with caching.

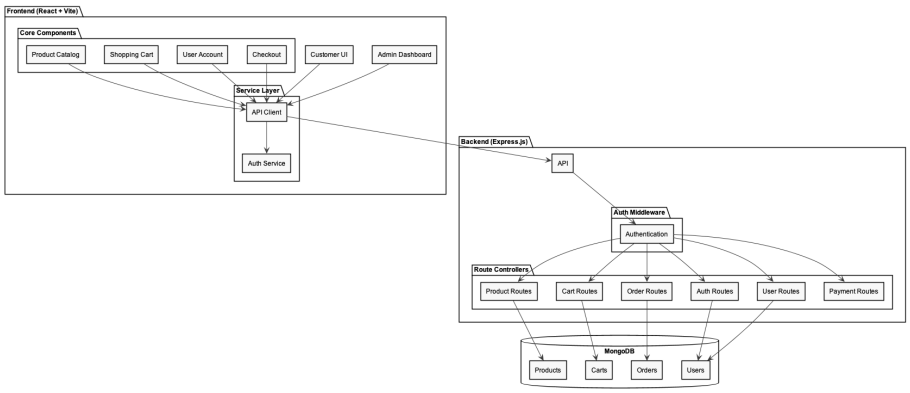
During the project, we successfully collaborated on integrating various components and ensuring functionality for key user features. Challenges included

debugging deployment issues and managing time effectively as a team. Through this process, we gained practical experience in applying software development practices, troubleshooting, and collaborative coding.

Working as a team provided the advantage of diverse perspectives and workload sharing but required careful communication and coordination to stay aligned. Overall, the project was a valuable learning experience in building a functional, user-oriented web application.

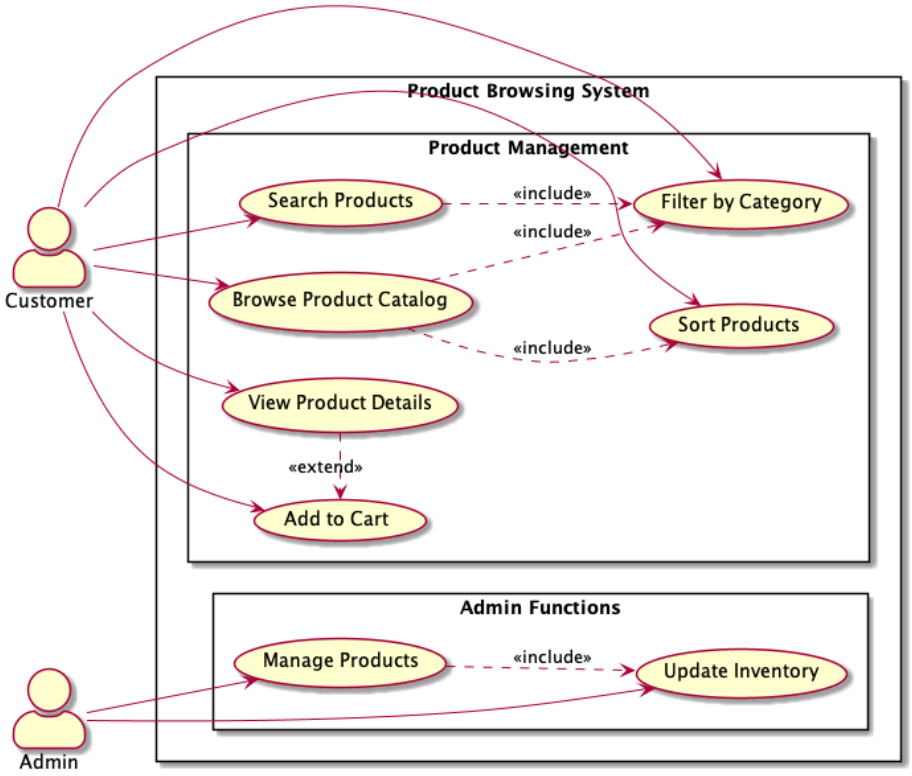
Figures

Figure 1: Architecture Diagram



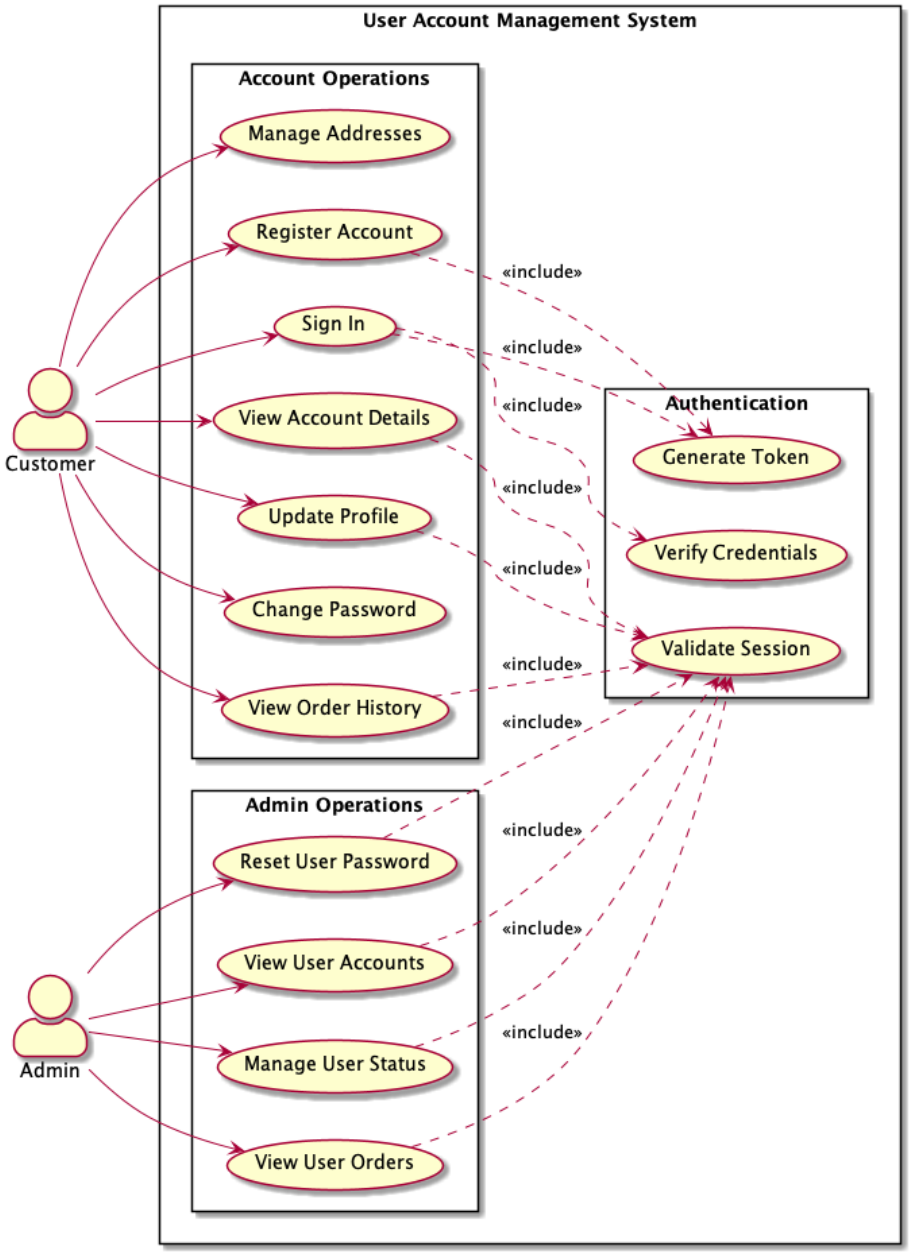
Back to Section 2.3: Component Integration

Figure 2: Product Browsing Use Case



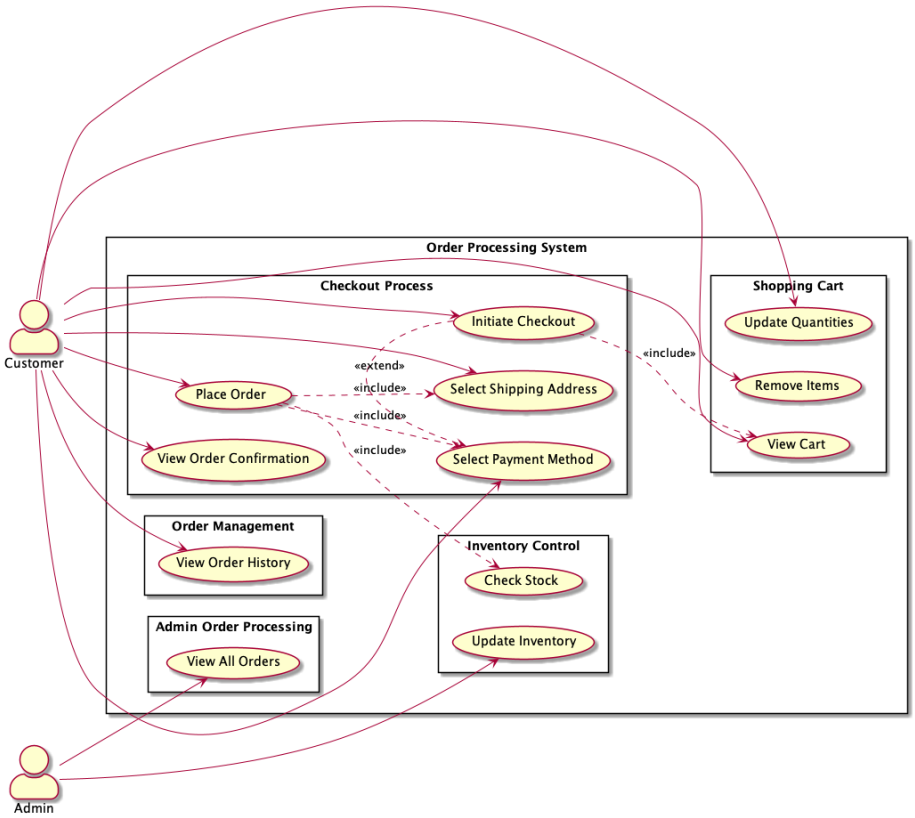
Back to Section 2.4.1: Product Browsing Use Case

Figure 3: User Account Management Use Case



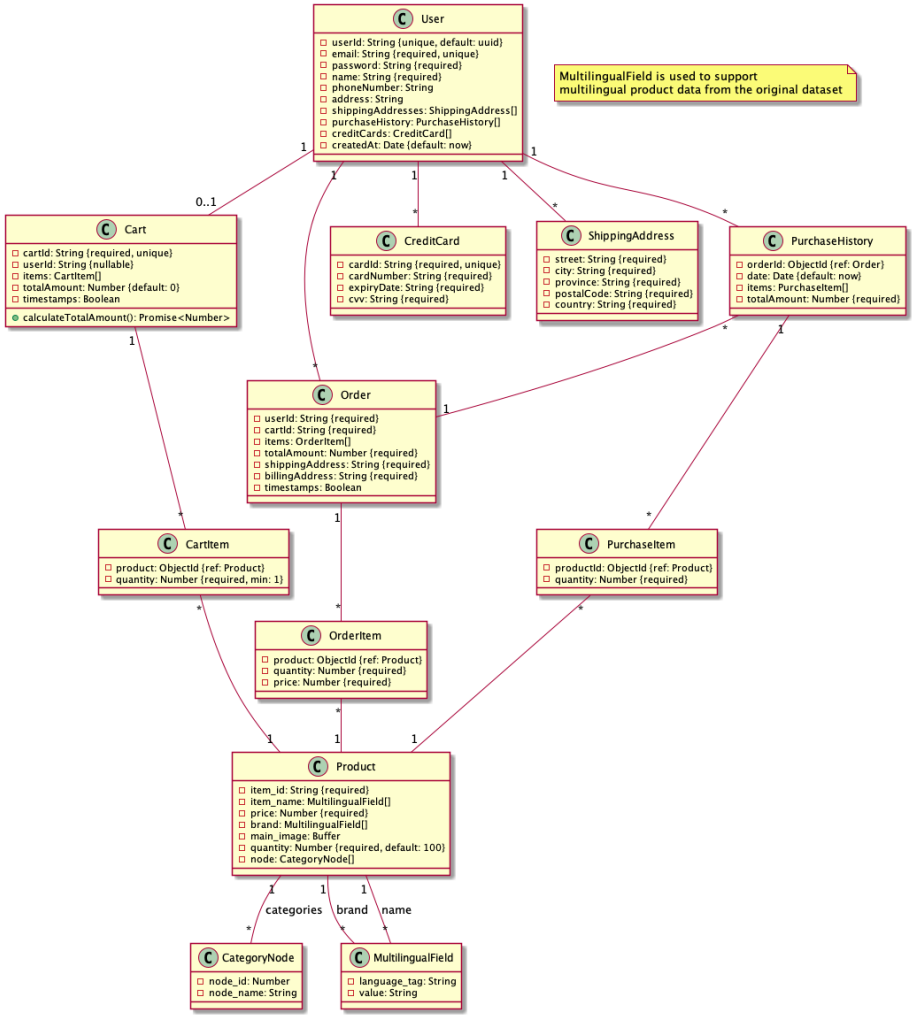
Back to Section 2.4.2: User Account Management Use Case

Figure 4: Order Processing Use Case



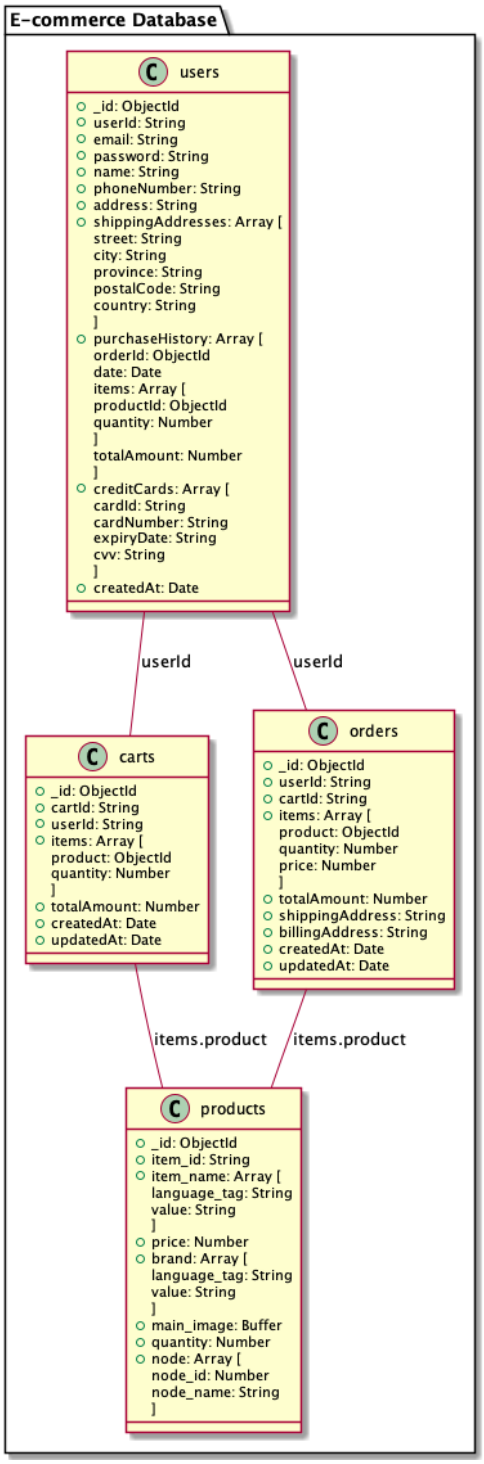
Back to Section 2.4.3: Order Processing Use Case

Figure 5: Class Diagram



Back to Section 3.4.1: Class Diagram

Figure 6: Database Schema



Back to Section 3.4.2: Database Schema