# STUDENT EXAM PAPER

**E-COMMERCE DEVELOPMENT EXAM**

## 1. Objective

Develop a production-ready full-stack **E-Commerce Web Application** that demonstrates both backend and frontend development skills using the MVC pattern.

The application must be secure, well-structured, and fully deployed at the time of submission.

**Use of AI-generated code or copied content will result in disqualification.**

## 2. Technology Stack

**Backend**

- Node.js + Express.js (MVC structure)
- SQL Database (MySQL or PostgreSQL)
- MongoDB (for product catalog)
- Prisma ORM or native database drivers
- JWT authentication with bcrypt password hashing

**Frontend**

- Next.js 14+ (App Router, TypeScript)
- Tailwind CSS or CSS Modules for styling

**Deployment**

- You may use **Vercel, Render, Railway, AWS, Azure, or any other free service** you are comfortable with.

Both the frontend and backend must be publicly accessible and working correctly at the time of evaluation.

## 3. Application Architecture

**Backend Folder Structure**

```
/backend
 /models
 /controllers
 /routes
 /config
 /tests
 server.js
```

**Frontend Folder Structure**

```
/frontend
 /app
  /products
  /cart
  /orders
 /components
 /lib
```

**Responsibilities**

- **Models:** Database schema and queries
- **Controllers:** Application and business logic
- **Routes:** HTTP endpoints and middleware
- **Views:** Next.js pages and components

# 4. Database Schema

**SQL (PostgreSQL or MySQL):**

- users (id, name, email, passwordHash, role)
- orders (id, userId, total, createdAt)
- order_items (id, orderId, productId, quantity, priceAtPurchase)

**MongoDB:**

- products (id, sku, name, price, category, updatedAt)

# 5. Core Functional Requirements

## Authentication

- Implement secure registration, login, and logout using JWT.
- Store passwords as bcrypt hashes.
- Include two roles: **admin** and **customer.**
- Display the logged-in user's name in the navigation bar or header.

## Product Management

- Allow admins to create, update, and delete products (stored in MongoDB).
- Include search functionality, category-based filtering, and pagination.
- Product listings must be sorted by **price in descending order by default**.
- The sorting logic must be implemented **on the server side** and must be able to adjust based on **conditions defined by the evaluator in their request** (such as a specific header or query parameter).

Sorting only on the client side after fetching data will not be accepted.

**Example (for understanding):**

When the evaluator accesses your API endpoint for products, the server should automatically decide the sorting order before sending data back. For instance, under a normal request, the products might be sorted by highest price first. However, if the

evaluator sends a special type of request (for example, one that includes a certain identifier or flag), the server should respond differently—such as by sorting prices in ascending order. The key point is that the sorting change must happen on the server side, not through client-side JavaScript.

### Cart and Checkout

- Customers should be able to add and remove items from their cart.
- Checkout should create new entries in the **orders** and **order_items** tables in SQL.
- Total order amount must be calculated on the server.

### Reports

- Include at least one SQL aggregation (for example, daily revenue or top customers).
- Include at least one MongoDB aggregation (for example, category-wise sales).
- Display both results on a simple “Reports” page.

## 6. Performance and Security

- Validate all input data on the server.
- Add indexes for frequently queried fields (sku, category, updatedAt).
- Protect all sensitive or data-modifying routes using JWT authentication and role-based access control.
- Use pagination for large datasets to improve performance.
- Store all credentials and secrets in .env files; never commit them to GitHub.

## 7. Testing Requirements

Include at least **one automated test** (unit or integration) that verifies a key feature such as authentication, checkout, or product sorting.

**Requirements**

1. Use Jest or Mocha/Chai for testing.

2. Place all test files inside /backend/tests/.
3. Tests must be executable with:

npm run test

4. Your README must include:
    a. The testing framework used
    b. Command to run tests
    c. A short description of what is being tested

**Example Description (for README):**

"This test verifies that the product sorting function returns items in descending order by default and can handle alternate request conditions."

## 8. Submission Rules

1. **Repository Name (lowercase):** your_name_rollid

Example: rahul_sharma_21cs045

2. **Deployment Name:** Same as repository name.
3. **Repository Visibility:** Public.
4. **Deployment:**
    a. Both backend and frontend must be deployed and working.
    b. Test your URLs on multiple browsers and devices.
    c. Broken or non-functional deployments will not be evaluated.
5. **Company Name Restriction:**

The name "Kaushalam" must not appear anywhere in code, comments, commits, or deployment URLs.

6. **AI Restriction:**

Use of AI tools (e.g., ChatGPT, GitHub Copilot) is prohibited.

7. **Plagiarism:**

If code similarity is found between candidates, all involved submissions will be rejected.

8. **README File:** Must include:
    a. Overview and key features
    b. Tech stack and dependencies
    c. Setup and environment variable details
    d. Database configuration and migration steps
    e. Testing instructions
    f. API and frontend route summaries
    g. Deployment URLs
    h. **Admin login credentials (username/email and password)** for evaluation
9. **Deadline Compliance:**

Submissions received after evaluation begins will not be accepted.

## 9. Notes

- Maintain a clean and logical code structure.
- Use meaningful commit messages.
- Include sufficient seed data for easy verification.
- Ensure your application performs well across devices and browsers.
- Non-functional, broken, or incomplete submissions will not be reviewed.