Exam Question

"Build a Full-Stack E-Commerce Application with an MVC Back End (Node.js + Express) and a Next.js Front End, Integrating Both SQL and MongoDB."

Your task is to demonstrate your proficiency as a full-stack developer by creating a mini E-commerce application according to the following specifications:

1. Architecture & Technologies

1. **Back End (Node.js + Express in MVC)**
    a. **Models**: Communicate with the databases (SQL for orders/users, MongoDB for products).
    b. **Views**: If returning JSON, you can treat this lightly. However, keep a /views folder for structure if needed.
    c. **Controllers**: Contain the application logic (no direct DB queries here—must call the Models).
        i. You may create additional controllers (beyond the minimum required) to handle different functionalities as needed.
    d. **Routes**: Each set of endpoints (auth, products, cart, orders, reports) in a separate file, delegating to controllers.
    e. **Authentication**: Secure user registration/login (hashed passwords with something like bcrypt). Use JWT or sessions for protected routes.
2. **Databases**:
    a. **SQL** (PostgreSQL or MySQL):
        i. Tables: users, orders, order_items.
        ii. Show at least one advanced SQL query (e.g., daily revenue, top 3 spenders).
    b. **MongoDB**:
        i. Collections: products (and optionally carts or categories).
        ii. Implement at least one MongoDB aggregation or advanced query (e.g., grouping, indexing).
3. **Front End (Next.js)**:
    a. **SSR (Server-Side Rendering)** for the product listing page to demonstrate Next.js SEO and performance.
    b. Implement the Next.js front end in TypeScript (using .tsx files).

    c. **Dynamic Routes** for product detail pages.
    d. Basic cart page and checkout flow.
    e. Display at least one “report” (data from your advanced SQL or MongoDB queries).

2. Core E-Commerce Features

1. **User Accounts**:
    a. Users can register and log in. Passwords must be stored securely (hashed).
    b. Logged-in users can manage a shopping cart, place orders, and view past orders.
2. **Product Catalog**:
    a. **CRUD**: Show (list) products and their details.
    b. **Search/Filter**: Use a MongoDB aggregation or text/regex query to search by name or category.
    c. **Pagination**: If there are many products, implement pagination to avoid large data loads.
3. **Shopping Cart & Checkout**:
    a. Let users add products to their cart (stored in MongoDB or SQL—your choice).
    b. At checkout, create an order in SQL, linking order_items to each product the user is purchasing.
    c. Clear the cart after successful checkout.
4. **Reports**:
    a. **SQL Example**: Daily revenue for the last 7 days, or top spenders.
    b. **MongoDB Example**: Summarize sales by category.
    c. Expose these in JSON via an /reports endpoint and display them in a simple “Reports” page in Next.js.

3. MVC Structure Clarification

- **Models** (in separate folders for SQL and MongoDB, or clearly named files):
    - SQL Models: User.js, Order.js, OrderItem.js
    - MongoDB Models: Product.js, possibly Cart.js or Category.js

- **Views**:
    - If you return JSON only, you can keep a /views folder as placeholders or skip implementing templates.
- **Controllers**:
    - *Minimum recommended controllers*:
        - AuthController.ts (login/register/logout)
        - ProductController.ts (list/search products)
        - CartController.ts (add/remove items)
        - OrderController.ts (checkout, order history)
        - ReportController.ts (SQL & MongoDB reports)
    - *You are free to add more controllers if you need to separate logic further.*
- **Routes**:
    - authRoutes.ts, productRoutes.ts, cartRoutes.ts, orderRoutes.ts, reportRoutes.ts, each calling the appropriate controller.

4. Performance & Security

- **Query Efficiency**:
    - Use indexes on frequently queried fields in MongoDB.
    - Avoid N+1 queries in SQL.
    - Use bulk operations or transactions where relevant.
- **Time Complexity**:
    - Implement pagination for product listing.
    - Don't fetch an entire dataset if only partial data is needed.
- **Security**:
    - Hash passwords, never store them in plain text.
    - Protect routes that modify data (e.g., cart, orders) with authentication.
    - Validate user inputs to prevent SQL injection and other vulnerabilities.

5. Testing

- Provide at least **one** test (unit or integration) that covers a critical feature (e.g., the checkout process or authentication).
- Ensure you test the deployed (hosted) version of your application to confirm all features work before final submission.

6. Submission & Documentation

- **README** file with:
  - Installation steps (dependencies, environment variables).
  - Database setup instructions (SQL schema, MongoDB connection).
  - How to run the server (Node/Express) and the front end (Next.js).
  - Brief explanation of each folder or file.
- Provide your **code** in a repository that clearly shows your **MVC back-end** structure and **Next.js** front-end.

Rules

1. **Public GitHub Repository**
   a. Set your repository to public.
   b. Avoid mentioning any specific company names in the code, comments, or commit messages.
   c. **Naming Convention:** Use the format FullStackExam<yourname><dateofsubmission> for the GitHub repo name.

2. **No AI-Generated Code**
   a. Ensure that the code you submit is your own.
   b. The use of AI-generated code will lead to disqualification.
3. **Deployment Linked to GitHub**
   a. Make sure the GitHub repository is linked to the deployment platform for continuous deployment (e.g., Vercel, Netlify, Heroku, AWS, etc.).
   b. **Deployment Name:** Also use a similar naming convention for the deployed application if possible (e.g., FullStackExam<yourname><dateofsubmission>).
4. **No Edits After Submission**
   a. After submitting, do not make any further changes or commits to the repository.
5. **Basic but Not Limited Controllers**
   a. You must implement the core controllers listed above, but feel free to expand your controller logic or add new controllers if you need more features.

Goal of This Exam

We want to see:

1. **MVC organization** in the Node.js back end.
2. Good usage of **SQL** and **MongoDB** together, with real queries/aggregations.
3. A functional **Next.js** front end showcasing SSR, dynamic routes, and a minimal but clear UI.
4. Attention to **best practices** in performance, security, and time complexity.
5. Ability to write **clean, maintainable code** and a helpful README.