E-Commerce API Design Choices

Technology Stack

- **Node.js and Express**: Chosen for their robust ecosystem, excellent performance, and ease of use for building RESTful APIs.
- **TypeScript**: Provides strong typing, enhancing code quality and developer experience.
- **TypeORM**: An ORM that integrates well with TypeScript, providing a robust way to interact with the database.
- **SQLite (in-memory)**: Used for simplicity in this demonstration. In a production environment, a persistent database like PostgreSQL would be more appropriate.

Database Schema

1. **User**
    - id: number (PK)
    - username: string (unique)
    - email: string (unique)
    - password: string (hashed)
2. **Product**
    - id: number (PK)
    - name: string
    - description: string
    - price: number
    - inventoryCount: number
3. **Order**
    - id: number (PK)
    - userId: number (FK to User)
    - totalAmount: number
    - status: string
    - createdAt: Date
4. **OrderItem**
    - id: number (PK)
    - orderId: number (FK to Order)
    - productId: number (FK to Product)
    - quantity: number
    - price: number

Authentication

JWT (JSON Web Tokens) is used for authentication. This stateless approach is scalable and works well with RESTful APIs.

Design Patterns and Principles

- **MVC-like structure**: The project is organized into models (entities), routes (controllers), and services.
- **Dependency Injection**: TypeORM's repositories are used, allowing for easier testing and separation of concerns.
- **Single Responsibility Principle**: Each module (like CartService) has a single, well-defined purpose.

Third-Party Libraries

- **bcryptjs**: For secure password hashing.
- **jsonwebtoken**: For generating and verifying JWTs.
- **express**: Web application framework for Node.js.
- **typeorm**: ORM for TypeScript and JavaScript.
- **sqlite3**: SQLite driver for Node.js.

Areas for Improvement

1. **Error Handling**: Implement a global error handling middleware for consistent error responses.
2. **Validation**: Add request validation using a library like Joi or class-validator.
3. **Logging**: Implement a logging solution for better debugging and monitoring.
4. **Testing**: Add unit and integration tests for robustness.
5. **Environment Configuration**: Use environment variables for configuration in different environments.
6. **API Documentation**: Implement Swagger or a similar tool for API documentation.

This design aims to provide a balance between simplicity for demonstration purposes and scalability for potential real-world use.