Implementation Notes

(Back-end role task – Aeontrix AI)

Charaka Jith Gunasinghe
Gunasinghe.info@gmail.com | +94 70 145 4256

Repository: https://github.com/CharakaJith/mern-ecom-website.git

Assumptions

1. Cart handling:
   Shopping cart for non-logged-in users is managed using UI session storage. Cart persistence in the backend is only implemented for logged-in users.

2. Product management:
   All products are pre-seeded in the database. No backend endpoints for adding, updating, or deleting products are required, as the assessment focuses on user-facing functionality.

3. Stock and inventory:
   Inventory is assumed to have unlimited stock. No stock tracking, low-stock alerts, or overselling prevention is implemented.

4. User roles:
   All users are treated as standard users. No admin or multi-tier roles are required at this stage.

5. Email Notifications:
   Order confirmation emails are sent via nodemailer after checkout. No transactional email service integration is assumed.

6. UI:
   The frontend is minimal and functional. Advanced UI features such as animations, lazy loading, or accessibility improvements are not implemented.

Priorities

1. Core functionalities:
   Primarily focused on building the UI and a robust backend for the e-commerce application, adhering to the given guidelines. This includes user management, JWT-based authentication, product management, applying filters from the UI, managing the shopping cart, handling checkout and order placement, and notifying the user about their order via email.

2. Security:
   User passwords are hashed using **bcrypt** before being saved in the database.
   JWT tokens are used to authenticate and validate requests.
   Role-based authorization logic is implemented but not enforced, as all users are treated as standard users.
   Request data and parameters are properly sanitized and validated before performing any business operations to ensure the application follows proper security best practices and prevents common vulnerabilities (e.g: SQL injection, XSS).
   An ORM (Mongoose) has also been used to communicate with the database, helping to prevent server-side attacks such as SQL injection.

3. Logging and error handling:
   Winston logging has been implemented with daily rotation and a 14-day retention period to easily monitor server functionality and check for errors and informational messages. This setup does not log any sensitive information.
   A custom error handler has been implemented to send appropriate HTTP status codes and messages to the UI.

4. Scalability and maintainability:
   The backend is designed in a modular way, **adhering to SOLID principles** as much as possible, with a clear separation of concerns.

The architecture promotes maintainability, scalability, and testability, ensuring that individual modules can be developed, extended, and debugged independently.
This design approach also makes the codebase easier to understand, reduces coupling, and improves overall system flexibility.

5. API standards and consistence:
   RESTful endpoints follow naming conventions, return consistent status codes, and provide structured responses making them intuitive and predictable, making it easier for front-end applications to interact with the API.
   Additionally, endpoints include meaningful error messages to help with debugging and improve overall client-side handling.

6. User experience:
   API endpoints are consistent, returning proper status codes and a success flag, making them easier for the front end to consume.
   The **UI** is designed with a minimalist approach, ensuring simplicity while still allowing users to easily understand and navigate through the features.
   The UI is also **fully mobile-responsive**, providing a seamless experience when accessing the application from any mobile device.
   The combination of a clean UI and well-structured APIs ensure smooth integration between the front end and back end, resulting in an intuitive user experience.

Known Gaps

1. Payment gateway:
   Only a mock checkout has been implemented, which notifies the user via email upon placing an order.
   Real payment gateway integration (e.g., Stripe, PayHere) has not yet been implemented.

2. Testing:
   Currently, only a simple testing has been implemented for the email-sending function.
   Thanks to the modular implementation of the backend, additional unit and integration tests can be easily added to cover each functionality.

3. CI/CD pipeline:
   Automated deployment and continuous testing pipelines were not implemented in the current version, but can be integrated in future iterations to streamline the development and release process.

4. UI enhancements:
   The front end is functional but lacks advanced UI enhancements such as animations, accessibility improvements, better error handling, Lazy loading for images must be implemented for slower connections.

5. Inventory management:
   Currently its assumed as the stock levels are infinite. They are not automatically tracked.
   Implementing stock management in the future would allow real-time tracking, low-stock alerts, and prevention of overselling.

Improvements

- Implement real payment gateway for secure transaction handling.
- Add unit testing and integration testing for the backend using Jest.
- Set up a CI/CD pipeline for automated builds, testing and deployments.
- Implement multi-tier role-based access system for better access control.
- Implement inventory tracking and management system.
- Add product reviews and Wishlist feature.
- Optimize performance and database queries for high-traffic scenarios.
- Implement refresh token mechanism for seamless session management.