**Static Analysis Report**

Submitted By

**Abdul Moiz Hassan (22I-2560)**

**Hassaan Mansur (22I-2421)**

**Syed Muhammad Pasha (22I-2635)**

Submitted to

**Sir Atif Jilani**

11th December

**Fall 2024**

A logo with text and letters

Description automatically generated with medium confidence

**Department of Software Engineering**

FAST – National University of Computer & Emerging Sciences

**Frontend**

**General Analysis of the Codebase**

1. **Component Structure and Ant Design**:
   * **Ant Design (AntD)** is used for UI components like Button, Flex, Row, and Col, ensuring responsive layouts.
   * **Example**: In ProfilePage, the user's avatar is displayed in a circular container using Flex and Row components to ensure a responsive, aligned layout.
2. **State Management and API Calls**:
   * **Redux Toolkit** is employed for state management and API calls. Custom hooks like useCreateBrandMutation and useGetSelfProfileQuery interact with the backend.
   * **Example**: In CreateBrand, useCreateBrandMutation sends a POST request to create a brand, while useState manages form input.
3. **Error Handling and User Feedback**:
   * **Toast Notifications** provide user feedback after API calls, displaying success or error messages.
   * **Example**: In CreateCategory, a success toast is shown after creating a category, and an error toast is displayed on failure.
4. **Routing and Navigation**:
   * **React Router** is used for page navigation with the useNavigate hook for dynamic redirection.
   * **Example**: In NotFound, navigate(-1) redirects the user to the previous page when the "Go Back" button is clicked.
5. **UI Customization and Styling**:
   * **Inline CSS** is used for layout tweaks and responsive design.
   * **Example**: In ProfilePage, the user’s avatar is styled with inline CSS to fit inside a circular container, with Flex ensuring responsive alignment.

**Potential Problems Identified**

1. **Missing Client-Side Validation**:
   * **Problem**: Forms (e.g., CreateBrand, CreateCategory) lack sufficient validation, allowing submission of invalid data.
   * **Fix**: Implement validation with HTML5 attributes or libraries like Formik to ensure inputs are valid before submission.
2. **Unclear Error Messages**:
   * **Problem**: API error messages are often vague or technical, confusing users.
   * **Fix**: Provide user-friendly error messages, like "Brand name already exists" instead of generic technical errors.
3. **Asynchronous Data Loading Without Feedback**:
   * **Problem**: Pages making asynchronous API calls (e.g., SaleHistoryPage) lack loading indicators, leaving users unsure if data is loading.
   * **Fix**: Use loading indicators like spinners or AntD's Skeleton component to inform users when data is being fetched.

**Conclusion**

The codebase is clean, modular, and effectively utilizes Ant Design and Redux. However, areas for improvement include adding client-side validation, more user-friendly error messages, loading states for asynchronous data, and better state persistence across page refreshes. Addressing these would enhance the user experience and reduce potential issues.

**Backend**

**General Overview**

The codebase represents a well-structured Node.js backend application that uses modern JavaScript features like async/await, ES6 imports/exports, and modular architecture. The application covers user authentication (password hashing, JWT generation), purchases, sales, and seller management (CRUD operations), input validation with Zod, and custom error handling.

**Strengths**

1. **Async/Await and Promises**:
   * Async/await is used throughout for better readability and streamlined error handling via a custom asyncHandler utility.
2. **Modular Design**:
   * Clear separation of concerns with controllers, services, and utilities, making the codebase easy to maintain and extend.
3. **Password Security with bcrypt**:
   * Passwords are hashed using bcrypt with a reasonable salt round of 10, ensuring secure password storage.
4. **JWT Authentication**:
   * JWT tokens are generated and signed securely, with a 2-day expiration for stateless user authentication.
5. **Validation with Zod**:
   * Zod schemas validate incoming data, ensuring integrity and reducing the risk of invalid data being processed.
6. **Consistent HTTP Status Codes**:
   * The use of the http-status library ensures standardized response codes across the application.

**Areas for Improvement**

1. **Error Handling**:
   * Inconsistent error handling across services and controllers. Some errors are not consistently wrapped in the custom error class. Adding try-catch blocks in functions like generateToken and hashPassword would improve robustness.
2. **Password Handling**:
   * Hardcoded bcrypt salt rounds (10) should be configurable via environment variables. There is also no password complexity validation, which could be added to enforce stronger passwords.
3. **JWT Token Expiration**:
   * The static 2-day expiration for JWT tokens should be configurable via environment variables, allowing flexibility in token durations.
4. **SQL Injection and Data Integrity**:
   * Although MongoDB is used, precautions against SQL injection (for potential future SQL databases) and additional data validation to prevent malicious input are needed.
5. **Rate Limiting and Security Headers**:
   * The application lacks rate limiting to prevent brute-force attacks on authentication endpoints. Security headers like Content Security Policy and X-XSS-Protection should be added to improve security.
6. **Testability**:
   * The absence of unit tests makes it difficult to ensure code correctness. Adding unit tests and database mocking would improve test coverage and reliability.
7. **Data Pagination**:
   * Pagination logic could be abstracted into a utility to avoid code repetition. Additionally, validation of pagination parameters (e.g., positive integers) would enhance security.

**Suggestions for Improvement**

1. **Centralize Error Handling**:
   * Ensure consistent error handling by using the custom error class across all services and controllers.
2. **Password Complexity Validation**:
   * Implement stronger password rules before hashing passwords.
3. **Improve Token Expiry Configuration**:
   * Make JWT token expiration time configurable through environment variables.
4. **Implement Rate Limiting**:
   * Protect authentication routes with rate limiting to mitigate brute-force attacks.
5. **Add Security Headers**:
   * Use security middleware (e.g., Helmet.js) to add HTTP security headers.

**Conclusion**

The application is well-structured, follows modern JavaScript best practices, and uses secure techniques like bcrypt and JWT authentication. However, improvements can be made in error handling, security, and test coverage. Addressing these areas will enhance the application's robustness, security, and maintainability.