Backend Documentation

Overview

This document provides a detailed description of the backend setup and implementation, highlighting the structure, functionalities, and approach taken in building the application. The frontend directory in this project is based on a prior implementation from a Django-based project, reused with minor adjustments to integrate smoothly with the Node.js backend environment. This documentation will focus exclusively on the backend aspects.

---

1. Project Structure and Environment Setup

The backend server uses Express.js with MongoDB as the database, and Mongoose as an ODM for easier interaction with MongoDB collections. The project is divided into two primary directories: `server` (backend) and `client` (frontend). Each runs independently on separate terminals to handle the backend and frontend environments concurrently.

Key Steps to Run the Application:

1. **Backend Setup**:
    - Navigate to the server directory using `cd server`.
    - Install dependencies with `npm install`.
    - To start the backend server, use either `npm start` or `npm run server` (the latter runs `nodemon` for live reloading during development).
    - The backend runs on `localhost:4000`.
2. **Frontend Setup**:
    - Navigate to the client directory using `cd client`.
    - Install frontend dependencies with `npm install`.
    - To start the frontend server, use `npm start`.
    - The frontend runs on `localhost:3000`.

Environment Variables:

To configure the backend, an environment file (`.env`) is required in the `server` directory, containing the following variables:

- `MONGO_URI`: MongoDB connection URI.
- `JWT_SECRET`: Secret key used for signing and verifying JWT tokens.

---

2. Backend Dependencies and Key Libraries

The backend leverages several core libraries, including:

- **Express.js**: Handles the server setup and routing.
- **Mongoose**: Interfaces with MongoDB for schema definitions, model creation, and CRUD operations.
- **JWT (jsonwebtoken)**: Manages user authentication by issuing and validating tokens.
- **bcrypt**: Hashes and verifies user passwords for secure authentication.
- **dotenv**: Loads environment variables from `.env` files into `process.env`.
- **cors**: Enables cross-origin requests from the frontend running on a different port.

---

3. Authentication and Middleware

Auth Middleware (`authMiddleware.js`):

The backend uses JSON Web Tokens (JWT) for secure authentication. Middleware functions were implemented to:

1. **Protect Routes**: Ensure only authenticated users can access certain routes.
2. **Admin Restriction**: Restrict certain routes for administrative users only.

When an incoming request requires authentication, the `protect` middleware validates the JWT token in the request headers. If valid, it attaches the user information to the `req.user` object; otherwise, it responds with an "Unauthorized" message. The `admin` middleware further restricts access by checking if `req.user.isAdmin` is true.

---

4. Routes and Controllers

4.1 User Routes (`/api/user`):

The user routes handle authentication and CRUD operations for users. The following routes were implemented:

- **POST `/register`**: Registers a new user with hashed passwords stored in MongoDB.
- **POST `/login`**: Authenticates a user and returns a JWT token.
- **GET `/profile`**: Retrieves the profile of the authenticated user.
- **PUT `/profile`**: Allows an authenticated user to update their profile information.

Controllers in `userController.js` handle these routes by leveraging Mongoose models, bcrypt hashing, and JWT generation.

4.2 Product Routes (`/api/products`):

The product routes allow both CRUD operations on products and stock level adjustments, accessible based on user roles.

- **GET `/`**: Retrieves all products, including category data.
- **POST `/`**: Adds a new product to the database (admin-only).
- **PUT `/:id`**: Updates an existing product by ID (admin-only).
- **DELETE `/:id`**: Deletes a product by ID (admin-only).

The product controller (`productController.js`) uses Mongoose to interact with the MongoDB products collection. Each product is linked to a category using a reference to the category ID. Notably, quantity adjustments were handled using `quantityChange`, which allows for stock level increases or decreases without directly modifying the stock count.

4.3 Category Routes (`/api/categories`):

The category routes allow for CRUD operations on product categories and retrieving products within a category.

- **GET `/`**: Retrieves all categories.
- **POST `/`**: Creates a new category (admin-only).
- **PUT `/:id`**: Updates a category by ID (admin-only).
- **DELETE `/:id`**: Deletes a category by ID (admin-only).
- **GET `/:id/products`**: Retrieves all products within a specified category.

The `getProductsByCategory` controller retrieves products by searching for products with a matching category ID, enabling dynamic filtering of products by category.

---

5. Database Schema and Models

User Schema (`User.js`):

Defines user information, including:

- **Username**: Unique identifier for login.
- **Password**: Hashed with bcrypt for security.
- **isAdmin**: Boolean flag for role-based access control.

Product Schema (`Product.js`):

Defines each product, including:

- **name**: The product's name.

- **category**: A reference to the Category model.
- **price**: The product's price.
- **stock_level**: Tracks the quantity in stock, adjusted by `quantityChange` operations.

Category Schema (`Category.js`):

Each category contains:

- **name**: The name of the category, used for reference in the Product model.
- **description**: Brief description of the category.

---

6. Error Handling

Error handling was implemented in middleware functions and within controllers using `express-async-handler`. All errors return JSON responses with appropriate HTTP status codes:

- **401 Unauthorized**: Returned for invalid or missing JWT tokens.
- **404 Not Found**: Returned when a requested resource, such as a product or category, does not exist.
- **400 Bad Request**: Used for validation errors, such as when attempting to create a duplicate product or category.

---

7. Security and Validation

Password Hashing:

Passwords are hashed using bcrypt upon user registration, with authentication verification also utilizing bcrypt's hashing to ensure password security.

JWT Authentication:

JWTs are issued upon successful login and are required for protected routes. Tokens are verified with the `JWT_SECRET`specified in the `.env` file.

---

8. Running and Testing

To test the application:

1. **Ensure MongoDB is connected** by setting the `MONGO_URI` in the `.env` file.

2. **Run the backend** by navigating to the server folder and executing `npm start` or `npm run server`.
3. **Run the frontend** in a separate terminal window by navigating to the client folder and executing `npm start`.
4. **Use Dummy Logins**:
    - Non-Admin Users: `bari/password123`, `shafquat/password123`
    - Admin Users: `admin/password123`, `admin2/password123`

Once running, you can interact with the backend by sending HTTP requests through the frontend or directly with tools like Postman. Both admin and non-admin users can interact with products, but certain routes are restricted to admin users.