Assignment: Product Inventory Management System (PIMS)

Objective:

Develop a Product Inventory Management System using .NET Core for the backend, SQL Server for persistent storage, and expose functionality through a secure, versioned RESTful Web API. This system will not only manage product and inventory records but also include user authentication, role-based access control, and logging.

Requirements:

1. **.NET Core Web API:**

   - Utilize .NET Core 6/7/8 for creating the Web API project.
   - Implement versioning in your Web API to allow for future improvements without breaking changes.
   - Secure the API using JWT authentication and implement role-based access control. Define at least two roles: `Administrator` and `User`.

2. **SQL Server Database:**

   - Design the SQL Server database to store products, inventory levels, and user accounts.
   - Include tables with appropriate relationships for Products (ProductID, Name, Description, Price, CreatedDate) and Inventory (InventoryID, ProductID, Quantity, WarehouseLocation).
   - Implement user authentication and store user credentials securely.

3. **Entity Framework Core:**

   - Use Entity Framework Core for ORM with a code-first approach. Demonstrate migration management for database versioning.

4. **Business Logic:**

   4.1 Product Management

   - **4.1.1** **Categorization:** Products can be categorized (e.g., electronics, clothing, groceries). Implement logic to allow products to be assigned multiple categories and enable category-based filtering and search.
   - **4.1.2** **Price Adjustment:** Implement a feature to adjust the prices of products, either individually or in bulk (e.g., seasonal discounts). This might involve percentage decreases or fixed amount deductions. Ensure that there are checks in place to prevent setting negative prices.
   - **4.1.3** **SKU Uniqueness:** Ensure that each product has a unique SKU (Stock Keeping Unit). This requires validation logic to check for SKU uniqueness upon product creation or update.

4.2 Inventory Management

- **4.2.1 Adjustment Transactions:** Inventory levels change through transactions (additions or subtractions). Implement a transactional system that records these changes with timestamps, reasons (e.g., sale, restock), and the user responsible. Transactional system need not be very complex.
- **4.2.2 Low Inventory Alerts:** Develop logic to identify low inventory levels based on predefined thresholds and trigger alerts or notifications for restocking.
- **4.2.3 Inventory Auditing:** Implement a feature for inventory audits, allowing manual adjustments with documented reasons, ensuring that there's a trail for each adjustment made for auditing purposes.

4.3 User Authentication and Authorization

- **4.3.1 Role-Based Access Control (RBAC)**: Differentiate system functionalities based on user roles. For example, only Administrators can adjust prices or perform inventory audits, while Users might only view products and their inventory levels.
- **4.3.2 Secure Credential Storage:** Implement logic to securely store user credentials, likely involving hashing and salting passwords before they are stored in the database.

5. Global Error Handling and Logging:

- Implement global error handling in the Web API to return user-friendly error messages.
- Integrate a logging framework such as Serilog or NLog to log API requests, responses, and application errors.

6. Testing and Documentation:

- Write comprehensive unit and integration tests covering critical paths in the application.
- Use Swagger/OpenAPI for API documentation, ensuring that all endpoints are clearly documented with request/response examples.

7. Performance and Optimization:

- Implement caching to optimize responses for frequently requested data.
- Demonstrate the use of asynchronous programming models to improve API performance.

8. Front-End (Optional):

- For a full-stack experience, develop a simple front-end using a modern JavaScript framework (e.g., Angular, React) or Blazor, consuming the developed Web API.

Evaluation Criteria

- **Architecture:** Well-structured architecture following best practices in API development, security, and database design.
- **Code Quality:** Clarity, maintainability, and use of advanced C# features where appropriate.
- **Security:** Implementation of secure authentication, authorization, and data protection.
- **Testing:** Depth and coverage of tests, including unit and integration tests.
- **Documentation:** Comprehensive documentation, including setup, API endpoint descriptions, and usage.
- **Bonus:** Implementation of additional features such as the front-end application, advanced performance optimization, and real-time updates using SignalR.

Submission Guidelines

- Source code should be provided via a GitHub repository, with clear instructions for setting up and running the project contained within a README file.
- Include all necessary scripts for setting up the database, and any configuration files required to run the project.

This is a test assignment for a .NET Core developer with 3 years of experience, focusing on SQL Server and Web API, we can integrate additional features and technologies that challenge the candidate's skills further.