Order Management System

Scenario

You are tasked with developing an enhanced Order Management System. The system should allow customers to place orders, view their order history, and allow administrators to manage orders. The business logic includes applying tiered discounts, handling inventory updates, ensuring order validation, supporting multiple payment methods, and generating order invoices. Additionally, implement role-based access control (RBAC) to manage different user permissions.

Requirements

1. **Entities:**
    - **Customer**: `CustomerId`, `Name`, `Email`, `Orders`
    - **Order**: `OrderId`, `CustomerId`, `OrderDate`, `TotalAmount`, `OrderItems`, `PaymentMethod`, `Status`
    - **OrderItem**: `OrderItemId`, `OrderId`, `ProductId`, `Quantity`, `UnitPrice`, `Discount`
    - **Product**: `ProductId`, `Name`, `Price`, `Stock`
    - **Invoice**: `InvoiceId`, `OrderId`, `InvoiceDate`, `TotalAmount`
    - **User**: `UserId`, `Username`, `PasswordHash`, `Role` (Admin, Customer)
2. **Endpoints:**

    Customer Endpoints:

    - `POST /api/customers` - Create a new customer
    - `GET /api/customers/{customerId}/orders` - Get all orders for a customer

    Order Endpoints:

    - `POST /api/orders` - Create a new order
    - `GET /api/orders/{orderId}` - Get details of a specific order
    - `GET /api/orders` - Get all orders (admin only)
    - `PUT /api/orders/{orderId}/status` - Update order status (admin only)

    Product Endpoints:

    - `GET /api/products` - Get all products
    - `GET /api/products/{productId}` - Get details of a specific product
    - `POST /api/products` - Add a new product (admin only)
    - `PUT /api/products/{productId}` - Update product details (admin only)

Invoice Endpoints:

- `GET /api/invoices/{invoiceId}` - Get details of a specific invoice (admin only)
- `GET /api/invoices` - Get all invoices (admin only)

User Endpoints:

- `POST /api/users/register` - Register a new user
- `POST /api/users/login` - Authenticate a user and return a JWT token

3. **Business Logic:**
   - Validate the order to ensure product stock is sufficient for the requested quantity.
   - Apply tiered discounts based on order total (e.g., 5% off for orders over $100, 10% off for orders over $200).
   - Support multiple payment methods (e.g., Credit Card, PayPal).
   - Generate an invoice when an order is placed.
   - Implement role-based access control (RBAC) to manage user permissions.
   - Secure endpoints using JWT authentication.
   - Send email notifications to customers when their order status changes.
4. **Constraints:**
   - Use Entity Framework Core for data access.
   - Ensure proper error handling and validation.
   - Implement unit tests for critical business logic.
   - Document the API using Swagger.
   - Secure sensitive endpoints using JWT authentication.
   - Implement role-based access control (RBAC).

Steps to Implement

1. **Setup the Project:**
   - Create a new ASP.NET Core Web API project.
   - Configure Entity Framework Core with an in-memory database for simplicity.
2. **Define Models:**
   - Create the `Customer`, `Order`, `OrderItem`, `Product`, `Invoice`, and `User` classes.
3. **Setup DbContext:**
   - Create an `OrderManagementDbContext` that includes DbSet properties for each entity.
4. **Create Repositories:**
   - Implement repository classes for handling data access for `Customer`, `Order`, `Product`, and `User`.
5. **Implement Services:**
   - Create services to handle the business logic, such as order validation, applying discounts, handling payments, updating stock, generating invoices, and sending email notifications.
6. **Implement JWT Authentication:**

    - Configure JWT authentication and implement methods for user registration and login.
7. **Create Controllers:**
    - Implement the required API endpoints in `CustomerController`, `OrderController`, `ProductController`, `InvoiceController`, and `UserController`.
8. **Implement Role-Based Access Control:**
    - Secure sensitive endpoints using JWT authentication and implement role-based access control.
9. **Implement Unit Tests:**
    - Write unit tests for the business logic, particularly for order validation, discount application, stock updates, and payment handling.
10. **Add Swagger Documentation:**
    - Configure Swagger to generate API documentation.