Backend Developer Practice Assignment:
Microservice-Based Event Management System

Overview

Design and implement a microservice-based event management system with real-time synchronization capabilities. The system will consist of two microservices that communicate through a message broker, using MongoDB for data persistence and JWT for authentication.

Requirements

1. **Event Service**
    - Create a FastAPI application that manages events (create, read, update, delete)
    - Implement MongoDB integration using an ODM (pymongo, motor, or beanie)
    - Implement JWT authentication for API endpoints
    - When events are created/updated/deleted, publish messages to a message broker
2. **Notification Service**
    - Create a second FastAPI application that subscribes to event changes
    - Implement a WebSocket endpoint for real-time notifications
    - Process incoming messages from the message broker
    - Store notification history in MongoDB
3. **Message Broker Integration**
    - Implement RabbitMQ (or Kafka) for communication between services
    - Create appropriate exchange/queue configurations
    - Handle reconnection scenarios and message persistence
4. **Authentication**
    - Implement JWT-based authentication
    - Create endpoints for user registration and login
    - Apply authentication middleware to protected routes
5. **Testing & Documentation**
    - Write unit tests for core functionality
    - Document API endpoints using OpenAPI/Swagger
    - Include setup instructions and examples in the README

Bonus Points

- Docker containerization of the application
- Simple AWS integration (S3 for file uploads or Lambda function)
- Performance optimization strategies
- Implement rate limiting

Deliverables

1. Source code for both microservices
2. Documentation (API, setup instructions)
3. Unit tests
4. Brief explanation of design decisions and potential improvements

Evaluation Criteria

- Code quality, structure, and readability
- API design and implementation
- Proper error handling
- Testing approach
- Documentation quality
- Performance considerations

Supplementary Information

1. Load Requirements/Number of Clients

The system should be designed to handle the following load:

- **Concurrent Users**: Support up to 1,000 concurrent users for the Event Service
- **WebSocket Connections**: Handle up to 500 simultaneous WebSocket connections for the Notification Service
- **Request Rate**: Process up to 100 requests per second for the Event Service
- **Event Creation Rate**: Support up to 50 new events per minute
- **Performance Requirements**:
    - API response time should be less than 300ms for 95% of requests
    - WebSocket message delivery should occur within 1 second of the triggering event
- **Scalability**: Include considerations for horizontal scaling in your design documentation

2. Event Data Structure

An event should include the following data:

- **Required Fields**:

    - `title`: String (1-100 characters)
    - `description`: String
    - `location`: String
    - `start_time`: DateTime
    - `end_time`: DateTime

    - `created_by`: User reference
- **Optional Fields**:
    - `tags`: Array of strings
    - `max_attendees`: Integer
    - `status`: String (e.g., "scheduled", "canceled", "completed")
    - `attachment_url`: String (for uploaded files)
    - `coordinates`: GeoJSON point (latitude, longitude)
- **Validation Requirements**:
    - End time must be after start time
    - Title must not be empty
    - Location must not be empty
    - Event dates should not be in the past when created

3. WebSocket Data Transmission

Data Flow

- When events are created, updated, or deleted, notifications should be sent via WebSocket to relevant clients

Payload Format

The WebSocket payload should follow this structure:

```
{
  "type": "notification",
  "notification_type": "event.created|event.updated|event.deleted",
  "event": {
    "id": "event_id",
    "title": "Event Title",
    "action": "created|updated|deleted",
    "timestamp": "ISO datetime string"
  },
  "user": "username_who_performed_action"
}
```

Recipients

- **All users** should receive notifications for newly created events
- **Only subscribed users** should receive notifications for updates or deletions
- Users should be able to subscribe to specific events to receive updates

- Event creator should always receive notifications about their events

4. User Registration and Login

Registration Requirements

- **Required Fields**:
    - `email`: Valid email format
    - `username`: 3-50 characters, alphanumeric with underscores
    - `password`: Minimum 8 characters, require at least one uppercase letter, one number, and one special character
    - `full_name`: Optional
- **Email Confirmation**:
    - Email confirmation is **not** required for this assignment
    - However, design the system to easily add this feature in the future (include a `is_verified` field in user model)

Login Process

- Login should be possible via:
    - Username + password
    - Email + password

Authentication Flow

- Upon successful login, return a JWT token
- Token should expire after 30 minutes
- Protected routes should validate the token
- Include user identification in the token payload

Password Storage

- Passwords must be securely hashed using bcrypt
- Salt rounds should be configurable

Rate Limiting

- Implement rate limiting for authentication endpoints (max 5 login attempts per minute per IP)

User Roles

- For simplicity, implement a single user role for this assignment

- Design should allow for future role-based authorization

This assignment should be completed within 2 days. Focus on implementing core functionality with clean, well-structured code rather than trying to implement every possible feature.