NeoFi Backend Challenge: Collaborative Event Management System

Task Description

Develop a RESTful API for an event scheduling application with collaborative editing features. The application will allow users to create, manage, and share events with role-based permissions and maintain a comprehensive history of changes.

Note: This is a backend challenge, no UI is required.

Core Requirements

Authentication and Authorization

1. Implement a secure authentication system with token-based authentication
2. Create role-based access control (RBAC) with at least three roles: Owner, Editor, and Viewer

Event Management

1. Implement CRUD operations for events
2. Support recurring events with customizable recurrence patterns
3. Enable conflict detection for overlapping events
4. Allow batch operations for creating multiple events

Collaboration Features

1. Create a sharing system with granular permissions
2. Implement real-time notifications for changes
3. Track edit history with attribution

Advanced Features

1. Versioning system with rollback capability
2. Changelog with diff visualization
3. Event conflict resolution strategies
4. Implement a transaction system for atomic operations

Required Endpoints

```
# Authentication
POST /api/auth/register - Register a new user
POST /api/auth/login - Login and receive an authentication token
POST /api/auth/refresh - Refresh an authentication token
POST /api/auth/logout - Invalidate the current token


# Event Management
POST /api/events - Create a new event
GET /api/events - List all events the user has access to with pagination
and filtering
GET /api/events/{id} - Get a specific event by ID
PUT /api/events/{id} - Update an event by ID
DELETE /api/events/{id} - Delete an event by ID
POST /api/events/batch - Create multiple events in a single request

# Collaboration
POST /api/events/{id}/share - Share an event with other users
GET /api/events/{id}/permissions - List all permissions for an event
PUT /api/events/{id}/permissions/{userId} - Update permissions for a
user
DELETE /api/events/{id}/permissions/{userId} - Remove access for a user

# Version History
GET /api/events/{id}/history/{versionId} - Get a specific version of an
event
POST /api/events/{id}/rollback/{versionId} - Rollback to a previous
version

# Changelog & Diff
GET /api/events/{id}/changelog - Get a chronological log of all changes
to an event
GET /api/events/{id}/diff/{versionId1}/{versionId2} - Get a diff between
two versions
```

Technical Requirements

1. Use Python with either **Django, Flask or FastAPI**
2. Implement proper data validation and error handling
3. Create a database schema that efficiently supports all requirements
4. Provide API documentation using OpenAPI/Swagger
5. Implement rate limiting and security measures
6. Support both JSON and optional MessagePack serialization formats
7. Make judicious use of caching where appropriate

8. Tests are optional

Changelog implementation reference:
https://en.wikipedia.org/w/index.php?title=Python_(programming_language)&action=history

Detailed Endpoint Specifications

User Registration

Endpoint: `POST /api/auth/register`

- **Functionality**: Allows users to create an account by providing necessary information such as username, email, and password.
- **Required Fields**: username, email, password
- **Output**:
    - Success: User object with JWT token
    - Error: Appropriate error message with status code

User Login

Endpoint: `POST /api/auth/login`

- **Functionality**: Authenticates a user and provides a JWT token
- **Required Fields**: username/email, password
- **Output**:
    - Success: JWT token with user details
    - Error: Authentication error with status code

Create Event

Endpoint: `POST /api/events`

- **Functionality**: Creates a new event with the user as owner
- **Required Fields**: title, description, start_time, end_time, location (optional), is_recurring, recurrence_pattern (if recurring)
- **Output**:
    - Success: Event object with ID
    - Error: Validation error with status code

Get Event

Endpoint: `GET /api/events/{id}`

- **Functionality**: Retrieves an event by ID if the user has access

- **Output**:
    - Success: Event object with details including permissions
    - Error: Permission denied or not found error

Share Event

Endpoint: `POST /api/events/{id}/share`

- **Functionality**: Shares an event with other users with specified roles
- **Required Fields**: users (array of {user_id, role})
- **Output**:
    - Success: Updated permissions list
    - Error: Permission denied or invalid request

Update Event

Endpoint: `PUT /api/events/{id}`

- **Functionality**: Updates an event if the user has appropriate permissions
- **Fields**: Any event fields to be updated
- **Important**: All updates must be tracked with version history
- **Output**:
    - Success: Updated event object
    - Error: Permission denied or validation error

Changelog with Diff

Endpoint: `GET /api/events/{id}/diff/{versionId1}/{versionId2}`

- **Functionality**: Returns a detailed diff between two versions
- **Output**:
    - Success: JSON object with field-by-field differences
    - Error: Version not found or permission denied

Advanced Implementation Challenges

1. **Conflict Resolution**: Implement strategies for handling concurrent edits
2. **Temporal Queries**: Support querying events at a specific point in time
3. **Performance Optimization**: Optimize for handling large numbers of events and users
4. **Audit Trail**: Maintain a comprehensive audit trail of all system operations
5. **Data Synchronization**: Implement efficient mechanisms for clients to sync changes

Data Model Considerations

Design an efficient schema that supports:

1. User authentication and roles
2. Event data with support for recurring patterns
3. Permissions with granular access control
4. Version history with change tracking
5. Efficient querying for events by date range, user access, etc.

Evaluation Criteria

1. **Architecture Design**: Is the system well-structured with proper separation of concerns?
2. **Code Quality**: Is the code clean, maintainable, and following best practices?
3. **Performance**: How efficiently does the system handle the required operations?
4. **Security**: Are proper security measures implemented?
5. **Testing**: Is the code well-tested with appropriate test cases?
6. **Documentation**: Is the API well-documented and easy to understand?
7. **Error Handling**: Are errors handled gracefully with meaningful responses?

Submission Requirements

1. Source code in a GitHub repository
2. Postman collection or swagger for testing the API

Time Limit

3 days from receipt of the challenge

Notes for Candidates

- Focus on demonstrating good software engineering practices rather than implementing all features.
- Make deliberate architectural decisions and document your reasoning.
- Consider edge cases and how your system would scale.
- Feel free to use libraries but justify major dependencies.
- The changelog diff feature is particularly important—implement it thoughtfully.
- The requirements are fairly straightforward; all aspects of architectural design, schema design and coding practices are left to the developer.