Event Ticket Booking System with Seat Lock

Project Overview

Build a backend system using NestJS + PostgreSQL that allows customers to book tickets for events with a 2-minute automatic seat lock system. When a customer selects seats, they are locked for 2 minutes. If not confirmed within time, seats become available again.

Required Modules

- Auth

- Users

- Events

- Seats

- Bookings

- SeatLock System (cron only, no API)

User Roles

Admin:

- Create events

- Add seat layout

- View bookings

Customer:

- View events

- View seats

- Book seats

- Confirm/cancel booking

- View booking history

Admin Seeding Requirement

System must start with one default admin user created using a migration or seed script:

email: admin@example.com

password: Admin@123 (hashed)

role: admin

API List (15–20 APIs)

AUTH (3):

1. POST /auth/login
2. POST /auth/refresh
3. GET /auth/me

EVENTS (3–4):

4. POST /events
5. GET /events
6. GET /events/:id
7. PATCH /events/:id (optional)

SEATS (3–4):

8. POST /events/:id/seats
9. GET /events/:id/seats
10. GET /events/:id/seats/available
11. PATCH /seats/:id (optional)

BOOKINGS (5–6):

12. POST /bookings
13. POST /bookings/:id/confirm
14. POST /bookings/:id/cancel
15. GET /bookings/my
16. GET /bookings/:id
17. GET /events/:id/bookings (optional)

Seat Lock System (Mandatory)

When booking is created, seats are locked for 2 minutes. Cron job runs every minute to detect expired locks. Expired locks release seats and booking becomes expired. Confirming booking makes seats permanently booked.

Functional Requirements Summary

- JWT Authentication with role guards

- Admin-only event creation & seating layout

- Cloudinary or AWS S3 for event banners

- Customers can view seats, book seats, confirm/cancel bookings

- Auto expire seat locks using cron job

Event Banner Upload Requirement

Event banners must be uploaded using Cloudinary (free plan) or AWS S3 Free Tier. Store only the image URL.

Swagger Documentation

Swagger documentation required at /api/docs with DTOs, request/response models, and validation.

Additional Notes & Coding Standards:

1. Code Quality & Reusability

- Write **reusable functions/services** instead of repeating logic.
- Keep controllers thin — business logic goes in **services only**.
- Use DTOs with **class-validator** for all inputs.

Clean Architecture & Project Organization

- Follow NestJS best practices:
    - controllers → request handling
    - services → business logic
    - modules → feature grouping
- Do not create unnecessary files or folders.
- Use meaningful names for variables, methods, DTOs, and services.

API Stability

- All APIs must work without runtime errors.
- Handle errors using **HttpException** and proper status codes.
- Validate request bodies strictly.

ORM Requirement

You may use **either one**:

✓ **Prisma** (recommended for speed & schema clarity)
or
✓ **Sequelize** (if preferred)

But NOT both. Select one and maintain consistency.

GitHub Rules

- You **must use a public GitHub repository**.
- Proper commit history is required:
    - No one-shot “big bang push”
    - No single commit with entire code
    - No “final commit only”
- Commit progressively