System Design Document — Ticket Booking System (Backend)

Project: Ticket Booking System — High-Concurrency Backend
Tech stack: Node.js, Express, Sequelize ORM, PostgreSQL
Prepared for: Modex Assessment
Author: Ramya R
REG NO: RA2211003050056

1. Executive Summary

This document describes a backend system that supports creating shows/trips and booking seats in a concurrency-safe way. Key features:

- Admin: create shows/trips (name, startTime, total seats)
- Users: list shows and book one or more seats
- Booking lifecycle: PENDING → CONFIRMED or FAILED
- Concurrency-safe booking using DB transactions and row-level locks (prevent overbooking)
- Booking expiry worker: pending bookings auto-fail after 2 minutes and seats are restored
- Designed to be horizontally scalable with recommended production enhancements (caching, queueing, read replicas)

Assumptions: bookings are per-show (no seat-number tracking), payments/confirmations are external and out of scope. The system uses PostgreSQL as the single authoritative data store.

3. Data Model

Shows

- id (PK, integer)
- name (string)
- startTime (timestamp)
- seats (integer) — remaining seats
- createdAt, updatedAt

Bookings

- id (PK, integer)
- movieName (string) — denormalized
- userName (string)
- seats (integer)
- showTime (timestamp)

- status (enum/string): PENDING, CONFIRMED, FAILED
- expiresAt (timestamp, nullable)
- createdAt, updatedAt

Indexes:

- Bookings(status, expiresAt) — for fast expiry queries
- Shows(id) primary key
- Bookings(id) primary key

Notes:

- Denormalizing movieName and showTime into Booking keeps Booking stable even if show metadata later changes.
- For future seat-level booking (specific seat numbers), a new Seats table would be required.

4. API Endpoints (summary)

- GET /shows — list shows
- POST /shows — create show { name, startTime, seats }
- POST /bookings — create booking { showId, userName, numberOfSeats } → returns PENDING booking with expiresAt = now + 2min
- POST /bookings/:id/confirm — confirm pending booking → CONFIRMED
- GET /bookings — list bookings

5. Concurrency Control & Atomicity

Problem: multiple concurrent requests should not allow total seats to go negative (no overbooking).

Implemented solution (current):

- Use a DB transaction.
- SELECT ... FOR UPDATE on Shows row to acquire row-level lock.
- Verify show.seats >= seatsRequested.
- Decrement show.seats and save inside transaction.
- Insert Booking with status = PENDING and expiresAt.
- Commit transaction.

This ensures atomic check-and-update — only one transaction can modify the seats column at a time for a given show row.

Notes:

- This relies on Postgres row-level locks which work correctly across concurrent application instances.
- The approach scales horizontally because transactions and database locking provide global consistency.

6. Booking Expiry Mechanism

Goal: automatically mark stale PENDING bookings as FAILED and return seats.

Implementation:

- Background worker (setInterval every 30 seconds) queries:
- SELECT * FROM Bookings WHERE status = 'PENDING' AND expiresAt <= now();
- For each expired booking:
    - Start DB transaction.
    - Lock the corresponding Show row with FOR UPDATE (by name or id).
    - Increment show.seats by booking.seats.
    - Set booking.status = 'FAILED', booking.expiresAt = NULL.
    - Commit.

Rationale: the same locking pattern prevents race conditions between new bookings and expiry restoration.

7. Caching Strategy

Where caching helps

- GET /shows responses (read-heavy)
- Frequently requested show metadata

Recommended cache design

- Use **Redis** as an LRU cache.
- Cache GET /shows results for short TTL (e.g., 5–15s) to reduce DB read pressure while maintaining near-real-time updates.
- Invalidate cache on show update or booking confirm/fail (publish cache invalidation event after transaction commits).

9. Database Scaling & Reliability

Short-term (small load):

- Single Postgres primary with daily backups.

Medium-term:

- Add read replicas for heavy read traffic.
- Use connection pooling.
- Use managed Postgres provider (Render/Railway/AWS RDS).

Large-scale:

- Shard by geography or show group (for example, city_id or show_id mod N).
- Move time-series / analytics to separate data warehouse.
- Use distributed cache (Redis cluster).

High availability:

- Use managed Postgres with automated failover.
- Use WAL archiving & point-in-time recovery.

10. Observability & Monitoring

- **Logging:** structured logs (winston/pino), include trace IDs for requests.
- **Metrics:** Prometheus metrics for request latency, DB connections, transactions, expiry successes/failures.
- **Alerts:** CPU, DB replication lag, error rates.
- **Tracing:** OpenTelemetry for distributed tracing when using microservices.

11. Security & Operational Considerations

- **Env vars:** store DB credentials & secrets in platform secret store.
- **Input validation & rate limiting:** validate and rate-limit booking endpoints to mitigate abuse.
- **Backups:** regular backups and recovery drills.
- **Access controls:** restrict database access and rotate credentials.