High-Load Booking System (Full Stack)

Objective

Design and implement a **production-grade event booking system** that is resilient to race conditions and overselling. The system must include a RESTful backend API with transactional booking logic and a polished client-side interface with proper state management and error handling.

The core challenge: When 10 concurrent booking requests arrive for an event with only 2 remaining tickets, exactly 2 bookings must succeed and 8 must be gracefully rejected. No overselling. No crashes. No ambiguous states.

Tech Stack

Layer	Technology	Notes
Backend	NestJS (TypeScript)	Modular architecture, Dependency Injection
Database	PostgreSQL	ACID transactions, row-level locking
ORM	Prisma	Schema-first, migrations, type safety
Frontend	Next.js 14+ (App Router)	Server Components, Route Handlers where appropriate
Styling	TailwindCSS	Utility-first, responsive design
State Management	Zustand **or** Redux Toolkit	Candidate's choice — justify the decision in README
Auth	JWT (Access + Refresh tokens)	Secure token storage, middleware guards

Part 1: Backend (API & Concurrency)

1.1 Project Setup

- Initialize a NestJS project with TypeScript strict mode enabled.
- Configure Prisma with PostgreSQL as the datasource.
- Set up environment-based configuration using `@nestjs/config` (`.env` file for DB credentials, JWT secrets, etc.).
- Implement global exception filters and validation pipes (`class-validator` , `class-transformer`).
- Enable CORS for the frontend origin.

1.2 Database Schema (Prisma)

Design and implement the following schema (at minimum):

```
model User {
  id        String   @id @default(uuid())
  email     String   @unique
  password  String   // bcrypt hashed
  name      String
  bookings  Booking[]
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

model Event {
  id              String   @id @default(uuid())
  title           String
  description     String
  date            DateTime
  venue           String
  totalTickets    Int
  remainingTickets Int
  price           Float
  bookings        Booking[]
  createdAt       DateTime @default(now())
  updatedAt       DateTime @updatedAt
}

model Booking {
  id        String        @id @default(uuid())
  user      User          @relation(fields: [userId], references: [id])
  userId    String
  event     Event         @relation(fields: [eventId], references: [id])
  eventId   String
  status    BookingStatus @default(CONFIRMED)
  createdAt DateTime      @default(now())

  @@unique([userId, eventId]) // One booking per user per event
}

enum BookingStatus {
  CONFIRMED
  CANCELLED
}
```

Run `npx prisma migrate dev` and include migration files in the repository.

Seed data: Create a seed script (`prisma/seed.ts`) that populates:

- At least 3 users (with hashed passwords).
- At least 5 events with varying `totalTickets` (including one event with exactly 2 tickets for testing the race condition).

1.3 Authentication Module

`POST /auth/register`

Request Body:

```
{
  "email": "user@example.com",
  "password": "securePassword123",
  "name": "John Doe"
}
```

Requirements:

- Validate email format and password strength (minimum 8 characters, at least one number and one letter).
- Hash password using `bcrypt` with salt rounds >= 10.
- Return a clear error if the email is already registered (HTTP 409 Conflict).
- On success, return the created user (without password) and a JWT token pair.

Success Response (201):

```json
{
  "user": {
    "id": "uuid",
    "email": "user@example.com",
    "name": "John Doe"
  },
  "accessToken": "eyJhbG...",
  "refreshToken": "eyJhbG..."
}
```

`POST /auth/login`

Request Body:

```json
{
  "email": "user@example.com",
  "password": "securePassword123"
}
```

Requirements:

- Validate credentials against the database.
- Return HTTP 401 Unauthorized with a generic message ("Invalid credentials") on failure — do not reveal whether the email exists.
- On success, return the same response structure as registration.

Token Configuration:

- Access token: expires in 15 minutes.
- Refresh token: expires in 7 days.
- JWT secret must be loaded from environment variables.

`POST /auth/refresh`

Request Body:

```json
{
  "refreshToken": "eyJhbG..."
}
```

- Validate the refresh token and issue a new access/refresh token pair.

- Invalidate the old refresh token (implement token rotation).

Auth Guard

- Implement a global `JwtAuthGuard` using NestJS guards.
- All endpoints except `/auth/login`, `/auth/register`, and `/auth/refresh` must require a valid access token in the `Authorization: Bearer <token>` header.
- The guard should extract and attach the user payload to the request object.

1.4 Events Module

`GET /events`

Requirements:

- Return a paginated list of events.
- Each event must include the current `remainingTickets` count.
- Support optional query parameters:
  - `page` (default: 1)
  - `limit` (default: 10, max: 50)
  - `search` (filter by title, case-insensitive)
  - `sortBy` (date, price, title — default: date)
  - `sortOrder` (asc, desc — default: asc)

Success Response (200):

```
{
  "data": [
    {
      "id": "uuid",
      "title": "Tech Conference 2025",
      "description": "Annual technology conference...",
      "date": "2025-03-15T10:00:00Z",
      "venue": "Convention Center",
      "totalTickets": 100,
      "remainingTickets": 42,
      "price": 49.99
    }
  ],
  "meta": {
    "total": 25,
    "page": 1,
    "limit": 10,
    "totalPages": 3
  }
}
```

`GET /events/:id`

- Return a single event by ID with all details.
- Return HTTP 404 if not found.

1.5 Booking Module — THE CORE CHALLENGE

`POST /book`

Request Body:

```
{
  "eventId": "uuid"
}
```

> The `userId` must be extracted from the JWT token, NOT from the request body.

Critical Implementation Requirements:

1. **Transaction Isolation:** The entire booking operation must be wrapped in a Prisma interactive transaction with `Serializable` isolation level (or use `SELECT ... FOR UPDATE` with `RepeatableRead`).
2. **Artificial Delay:** Inside the transaction, after reading the current ticket count but before writing the booking, add the following delay to simulate real-world latency (e.g., payment processing):

   ```
   await new Promise(resolve => setTimeout(resolve, 1000));
   ```

3. **Concurrency Safety:** The implementation must guarantee that if an event has N remaining tickets and M simultaneous booking requests arrive (where M > N), exactly N bookings succeed and (M - N) are rejected.
4. **Booking Logic (pseudocode):**

   ```
   BEGIN TRANSACTION (Serializable)
     1. Lock and read the event row
     2. Check: remainingTickets > 0?
        – If NO: throw ConflictException("No tickets available")
     3. Check: user already has a booking for this event?
        – If YES: throw ConflictException("Already booked")
     4. await delay(1000)  // Artificial delay
     5. Decrement event.remainingTickets by 1
     6. Create Booking record with status CONFIRMED
   COMMIT TRANSACTION
   ```

5. **Error Responses:**
   - `409 Conflict` — No tickets available (race condition rejection).
   - `409 Conflict` — User already has a booking for this event.
   - `404 Not Found` — Event does not exist.
   - `401 Unauthorized` — Missing or invalid token.

Success Response (201):

```
{
  "booking": {
    "id": "uuid",
    "eventId": "uuid",
    "userId": "uuid",
    "status": "CONFIRMED",
```

```
    "createdAt": "2025-01-15T12:00:00Z"
  },
  "event": {
    "id": "uuid",
    "title": "Tech Conference 2025",
    "remainingTickets": 1
  }
}
```

`GET /bookings`

- Return all bookings for the authenticated user.
- Include event details in the response.

`DELETE /bookings/:id`

- Cancel a booking (set status to `CANCELLED`).
- Increment `remainingTickets` on the event (also within a transaction).
- Only the booking owner can cancel.

1.6 Concurrency Test Script

Create a standalone test script (`scripts/test-concurrency.ts`) that:

1. Registers or logs in as 10 different users.
2. Identifies an event with exactly 2 remaining tickets.
3. Fires 10 simultaneous `POST /book` requests (using `Promise.all`).
4. Logs results: how many succeeded (expected: 2), how many failed (expected: 8).
5. Verifies the event's `remainingTickets` is 0 after completion.
6. Prints a PASS/FAIL summary.

Example output:

```
=== Concurrency Test ===
Event: "Limited Concert" (2 tickets available)
Sending 10 simultaneous booking requests...

Results:
  Successful bookings: 2
  Rejected (no tickets): 8
  Remaining tickets after test: 0

TEST PASSED: No overselling detected
```

Part 2: Frontend (Next.js)

2.1 Project Setup

- Initialize a Next.js 14+ project with the App Router.
- Configure TailwindCSS.
- Set up the chosen state management solution (Zustand or Redux Toolkit).

- Create an Axios (or fetch wrapper) instance with:
  - Base URL from environment variable.
  - Automatic `Authorization` header injection from stored token.
  - Response interceptor for 401 errors (auto-redirect to login or silent token refresh).

2.2 Authentication Pages

Login Page (`/login`)

- A clean, centered login form with email and password fields.
- Client-side validation (required fields, email format).
- Display server-side errors inline (e.g., "Invalid credentials").
- On success:
  - Store the access token securely. **Preferred:** `httpOnly` cookie set by the backend. **Acceptable:** in-memory store (Zustand/Redux) with refresh logic. **Not acceptable:** `localStorage` for access tokens.
  - Redirect to the Event Dashboard.
- "Don't have an account? Register" link.

Registration Page (`/register`)

- Form with name, email, password, and confirm password fields.
- Client-side validation:
  - Name: required, minimum 2 characters.
  - Email: required, valid format.
  - Password: minimum 8 characters, at least one letter and one number.
  - Confirm password: must match.
- On success, redirect to the Event Dashboard (auto-login).

Route Protection

- Implement middleware or a layout-level check that redirects unauthenticated users to `/login`.
- Redirect authenticated users away from `/login` and `/register`.

2.3 Event Dashboard (`/events`)

Event List

- Display events in a responsive grid (cards).
- Each event card shows:
  - Title
  - Date (formatted, e.g., "March 15, 2025 at 10:00 AM")
  - Venue
  - Price (formatted with currency symbol)

- Remaining tickets with visual indicator:
  - Green badge: > 50% remaining
  - Yellow badge: 10-50% remaining
  - Red badge: < 10% remaining
  - Gray badge with "Sold Out": 0 remaining
- "Book Now" button (disabled if sold out)

Search and Sort

- A search input that filters events by title (debounced, 300ms).
- Sort dropdown (by date, price, title).

Real-Time Updates

Implement one of the following (state your choice and reasoning):

- **Option A: Polling** — Re-fetch event data every 5 seconds.
- **Option B: Optimistic Updates** — After a booking attempt, immediately update the local state and reconcile with server response.
- **Option C: WebSockets** — Real-time push updates when ticket counts change. (Bonus — not required, but impressive.)

2.4 Booking Flow — UX Excellence

When the user clicks "Book Now":

1. **Loading State:**
   - The button shows a spinner and text changes to "Booking..."
   - The button is disabled to prevent double-clicks.
   - Optionally, show a subtle progress indicator or skeleton.
2. **Success State:**
   - Display a success toast/notification: "Successfully booked [Event Title]!"
   - Update the remaining tickets count on the card (decrement by 1).
   - Optionally, change the button to "Booked ✓" (disabled, green).
3. **Error State — Tickets Sold Out (Race Condition):**
   - Display an error toast/notification: "Sorry, tickets for [Event Title] are no longer available."
   - Update the card to show "Sold Out" status.
   - The button becomes disabled.
   - **Important:** The UI must NOT crash, show a raw error, or display a generic "Something went wrong" message. The user must understand exactly what happened.

4. **Error State — Already Booked:**
   - Display an info toast: "You have already booked this event."
   - Change the button to "Booked ✓".
5. **Error State — Network/Server Error:**
   - Display an error toast: "Failed to complete booking. Please try again."
   - The button returns to its default "Book Now" state.

2.5 My Bookings Page (`/bookings`)

- Display a list of the user's bookings.
- Each booking shows: event title, date, venue, booking status, booking date.
- "Cancel Booking" button with confirmation dialog.
- After cancellation, update the event's remaining tickets accordingly.

2.6 UI/UX Polish

- Responsive design: works on mobile (375px), tablet (768px), and desktop (1280px+).
- Loading skeletons for data fetching states.
- Empty states with helpful messaging (e.g., "No events found" or "You haven't booked any events yet").
- Consistent color scheme and typography.
- Accessible (proper ARIA labels, keyboard navigation, focus management).

Part 3: Delivery Requirements

3.1 Repository Structure

Option A — Monorepo (Preferred):

```
booking-system/
├── apps/
│   ├── api/           # NestJS backend
│   │   ├── src/
│   │   │   ├── auth/
│   │   │   ├── events/
│   │   │   ├── bookings/
│   │   │   ├── common/
│   │   │   └── main.ts
│   │   ├── prisma/
│   │   │   ├── schema.prisma
│   │   │   ├── seed.ts
│   │   │   └── migrations/
│   │   └── test/
│   └── web/           # Next.js frontend
│       ├── app/
│       │   ├── (auth)/
│       │   │   ├── login/
│       │   │   └── register/
│       │   ├── (dashboard)/
│       │   │   ├── events/
│       │   │   └── bookings/
│       │   └── layout.tsx
│       ├── components/
│       ├── lib/
│       ├── store/
│       └── styles/
├── scripts/
│   └── test-concurrency.ts
├── docker-compose.yml
├── .env.example
└── README.md
```

Option B — Two Separate Repositories:

- `booking-api` — NestJS backend.
- `booking-web` — Next.js frontend.
- Each with its own README and Docker setup.

3.2 Docker Compose

Provide a `docker-compose.yml` that spins up:

- PostgreSQL (with health check).
- Backend API (depends on PostgreSQL, runs migrations and seeds on startup).
- Frontend (depends on Backend).

The entire system must start with a single command:

```
docker-compose up --build
```

3.3 Environment Variables

Provide a `.env.example` file:

```
# Database
DATABASE_URL=postgresql://postgres:postgres@localhost:5432/booking_db

# JWT
JWT_ACCESS_SECRET=your-access-secret-here
JWT_REFRESH_SECRET=your-refresh-secret-here

# Backend
API_PORT=3001

# Frontend
NEXT_PUBLIC_API_URL=http://localhost:3001
```

3.4 README

The README must include:

1. **Project Overview** — Brief description of what this is.
2. **Tech Stack** — List all technologies used.
3. **Architecture Decisions** — Why you chose Zustand/Redux, how you handle the race condition, token storage strategy.

4. **Getting Started:**
   - Prerequisites (Node.js, Docker, etc.).
   - Installation steps.
   - Running with Docker Compose.
   - Running locally (without Docker).
5. **API Documentation** — List all endpoints with request/response examples.
6. **Concurrency Test** — How to run the test script and what to expect.
7. **Known Limitations / Future Improvements** — Honest assessment.

Evaluation Criteria

Criteria	Weight	Description
Concurrency Handling	30%	The booking endpoint MUST prevent overselling under concurrent load. This is the most critical requirement.
Code Quality	20%	Clean architecture, proper typing, consistent naming, separation of concerns, SOLID principles.
API Design	15%	RESTful conventions, proper HTTP status codes, meaningful error messages, input validation.
Frontend UX	15%	Smooth booking flow, proper loading/error/success states, responsive design.
State Management	10%	Proper use of Zustand/Redux, clean store design, efficient re-renders.
DevOps & Documentation	10%	Working Docker setup, comprehensive README, seed scripts, test script.

Instant Disqualification

The following will result in an automatic rejection:

- Overselling occurs during the concurrency test (the core requirement fails).
- The application does not start with `docker-compose up --build`.
- No JWT authentication (endpoints are unprotected).
- The frontend crashes or shows raw error JSON on booking failure.

Bonus Points (Not Required)

- Unit tests for the booking service (especially the transactional logic).
- E2E tests (Playwright or Cypress) for the booking flow.

- Rate limiting on the booking endpoint.
- WebSocket-based real-time ticket count updates.
- CI/CD pipeline configuration (GitHub Actions).
- API documentation with Swagger/OpenAPI.
- Dark mode toggle.

Hints and Guidance

On Solving the Race Condition

There are several valid approaches to prevent overselling in PostgreSQL:

1. **Pessimistic Locking (Recommended for this task):**
   Use `SELECT ... FOR UPDATE` within a transaction to lock the event row before reading the ticket count. Other transactions attempting to read the same row will wait.

   ```
   await prisma.$transaction(async (tx) => {
     const event = await tx.$queryRaw`
       SELECT * FROM "Event" WHERE id = ${eventId} FOR UPDATE
     `;
     // Check tickets, delay, book...
   });
   ```

2. **Serializable Isolation Level:**
   Use Prisma's transaction with serializable isolation. This will cause conflicting transactions to fail with a serialization error, which you must catch and handle.

   ```
   await prisma.$transaction(async (tx) => {
     // Normal Prisma queries inside
   }, { isolationLevel: 'Serializable' });
   ```

3. **Optimistic Locking with Version Field:**
   Add a `version` column to the Event model. On update, check that the version hasn't changed. If it has, retry or reject.

Pick **one** approach, implement it correctly, and explain your choice in the README.

On Token Storage

The most secure approach for a browser-based application:

- Access token: store in memory (JavaScript variable / state management store).
- Refresh token: store in an `httpOnly`, `Secure`, `SameSite=Strict` cookie.
- This prevents XSS attacks from accessing tokens.

If you choose a simpler approach (e.g., storing in `localStorage`), acknowledge the security trade-off in your README.

On State Management Choice

Zustand is a good choice if:

- You prefer minimal boilerplate.
- The state is relatively simple (auth state, event list, bookings).
- You want fine-grained subscriptions without extra selectors.

Redux Toolkit is a good choice if:

- You want a well-established pattern with clear conventions.
- You plan to use RTK Query for data fetching/caching.
- You prefer explicit action/reducer patterns for complex state.

Good luck. We look forward to reviewing your implementation.