Short answer: Choose a modular **monolith-first** design with clear **modules** and a pragmatic **tech choices** stack; explicitly state what won’t be built now (non-goals) and what’s optional if time allows (stretch goals).

**Architecture choice**

* One deployable Spring Boot service organized into domain-focused modules (users, events, bookings, analytics) to keep development simple and fast.
* Stateless REST APIs, layered structure (controller → service → repository), and transactions at the service layer for correctness.
* Strong consistency via a relational database; caching is read-focused and never used to decide availability or capacity.

**Modules**

* Users: registration, login, roles (USER, ADMIN), profile basics.
* Events: CRUD, listing, details, capacity, time/venue.
* Bookings: create (idempotent), cancel, list history; concurrency-safe capacity updates.
* Analytics (admin): totals, capacity utilization, popular events, cancellations overview.

**Tech choices**

* Spring Boot: Web, Data JPA (Hibernate), Validation, Security, Actuator.
* Database: PostgreSQL (or MySQL) with Flyway migrations; Redis for caching hot reads.
* Docs & ops: springdoc-openapi (Swagger UI), Docker, simple CI, container deploy (Render/Railway/Heroku/AWS).

**Data model basics**

* Core tables: users, events, bookings, with foreign keys and NOT NULLs for integrity.
* Indexes on event\_id, user\_id, event time; unique constraint to prevent duplicate active booking per user-event.
* Capacity is derived from event settings and active booking count; canceled bookings do not consume capacity.

**Concurrency rules**

* Optimistic locking on Event (or a separate EventInventory) using a version field to avoid overselling.
* Booking create/cancel inside a single transaction: check capacity → insert/update → commit, with bounded retries on version conflict.
* Idempotency key for “create booking” to return the same result on repeated requests and avoid duplicates.

**API surface**

* Public: list events (pagination/filters), event details, create booking (idempotent), cancel booking, list user bookings.
* Admin: create/update/delete events, analytics summaries; protected by role-based access.
* Consistent HTTP codes: 201 created, 200 success, 400/422 validation, 401/403 auth, 404 not found, 409 conflict (no capacity).

**Caching and performance**

* Cache “upcoming events” and “event details” with short TTL; invalidate on event updates.
* Never cache booking writes or availability checks; rely on DB + locking for truth.
* Add essential indexes; consider read replicas or rollups for analytics only if needed.

**Security**

* Spring Security with JWT (or session) and roles USER/ADMIN; input validation on all endpoints.
* Rate limits (at gateway or simple filter) for booking endpoints to reduce abuse during spikes.
* Avoid exposing internal errors; return clear, stable error messages and codes.

**Deployment**

* Containerize with Docker; use environment variables for DB, JWT secrets, cache.
* Health and readiness via Actuator; simple log-based monitoring.
* One-click deploy to a managed platform; share the live base URL and Swagger UI.

**Testing**

* Unit tests for services and validators; repository tests for queries and constraints.
* Integration tests for booking flows under conflict (optimistic lock) and idempotency behavior.
* Short load tests focused on “last seat” contention to prove no overselling.

**Non-goals**

* No microservices split, service mesh, or complex distributed tooling at this stage.
* No full frontend; use Swagger UI and a few sample scripts to demo flows.
* No payments, complex seat maps, or multi-tenant sharding in the initial scope.

**Stretch goals**

* Waitlist: join when full, auto-promotion on cancellations, idempotent promotion flows.
* Seat-level booking: per-seat inventory with the same concurrency guarantees.
* Notifications: email/push on waitlist promotion or event updates.
* Advanced analytics: daily booking trends, top events, cancellation rates with time windows.

**Why this works**

* Delivers a stable, correct core quickly by keeping decisions simple and central.
* Preserves future options through modular boundaries and clear domain seams.
* Aligns with the challenge: correctness under concurrency, clean APIs, basic analytics, and a live deployment.