Room Booking - High Level Design

PAGE UNDER CONSTRUCTION

Introduction

Overview

This document describes the high level design of the a room-booking platform that enables users to register (login) , search for available rooms, make or cancel bookings and retrieve their booking history in real time using API-driven microservices, multi-region databases, caching, and strong concurrency controls to prevent double-booking

Scope

The l design aims to drill down the room-bookin architecture topic and elaborate on the technical details required for implementation/Integration.

Out of Scope

There is an external system responsible for inserting room data and availability into the database, as well as removing unavailable rooms.

In addition, a background job invalidates the cache based on updates from the real database.

Requirements

User

- Users can **register**, and **log in**,
- Users can **view their booking history**.

Room Search

- Search rooms by **date**, **location**, **capacity**, **price**, and **amenities**.
- Real-time **availability check**.

Booking

- Users can **create a booking** for a room.

- Users can **cancel a booking**.
- System must **prevent double-booking**.
- Users receive **confirmation notifications** (email/SMS)

Api

***User**:*

The user API must implement a “forgot password” feature as well.
The validation of the email format and password msut be handled in the UI (and enroce in BE we as well).

`POST /api/v1/users/register`

Headers

- `Content-Type: application/json`

body:

```
{
  "email": "yosefy2@hotmail.com",
  "password": "Lavi:99798e",
  "fullName": "Yaniv yosef",
  "phone": "+972500000000",
  "locale": "en-IL"

}
```

Responses

Status code

- 201 - successfully `register` created
  `{ "userId": "u_123456", "email": "yosefy2@hotmail.com", "createdAt": "2025-11-26T09:00:00Z", token:token}`
- 400 - missing or invalid body
- 409 - conflict - user already exit
- 500 - internal server error

`POST /api/v1/users/logIn`

Headers

- `Content-Type: application/json`

body:

```
{
  "email": "yosefy2@hotmail.com",
  "password": "Lavi:99798e",
 }
```

Responses

Status code

- 200 - successfully `logIn`
  `{ userId": "u_123456" , token:token }`
- 400 - missing or invalid body
- 500 - internal server error .

`GET /api/v1/users/{userId}/bookingsHistory`
`Query Parameters : startDate , endDate`

Headers

- `Content-Type: application/json`
- `Authorization: Bearer <access_token>`

Responses

Status code

- 200 - Success
  `{ bookingsHistory:bookingsHistory}`
- 404 - no user Id
- 500 - internal server error .

Booking:

`POST /api/v1/bookings`

Headers

- `Content-Type: application/json`
- `Authorization: Bearer <access_token>`

body:

```
{
"id" :"123",
  "roomId": "r_987",
  "userId": "u_123456",
  "checkin": "2025-12-20",
  "checkout": "2025-12-22",
  "guests": 2,
  "payment": {
    "method": "card",
    "paymentToken": "tok_visa_123"
  },
  "metadata": { "source": "web" }
}
```

Responses

Status code

- `201 Created` (booking confirmed)
  `{ "roomId": .....}`
- 400 - missing or invalid body
- `409 Conflict` (double-booking conflict)
- `422 Unprocessable Entity` (invalid dates)
- `429 Too Many Requests` (rate-limited)
- 500 internal server error

Note: Display a warning before calling this API to indicate that the cancellation complies with policy

POST /api/v1/bookings/{bookingId}/cancel

Headers

- `Content-Type: application/json`
- `Authorization: Bearer <access_token>`

body:

```
{
"reason" :"sick",
}
```

Responses

Status code

- `201 Created` (booking confirmed)
  `{ "roomId": .....}`
- 403 Forbidden (user does not own this booking)
- 404 Booking not found
- 409 Cannot cancel (e.g., past booking, outside cancellation window)
- 500 Internal server error
- 401 Unauthorized (invalid/missing token)

`search room`:

`GET /api/v1/rooms/search`

`Query Parameters : startDate , endDate <TBD>`

Headers

- `Content-Type: application/json`
- `Authorization: Bearer <access_token>`

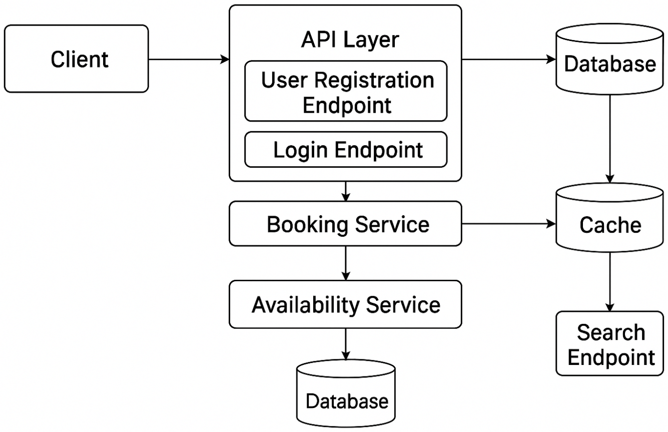
Responses

Status code

- `200 ok`
  {
  "rooms": [
  {
  ...}
-]}
- 400 - invalid body `Query Parameters`
-
- `429 Too Many Requests` (rate-limited)
- 500 internal server error

Components diagram

The below diagram demonstrates the Room-booking BE.



5) Database Schema (MongoDB)

Document-oriented, denormalized where useful, with careful use of references for transactional parts (e.g., bookings).
Includes indexing strategy and consistency considerations.

Collections & Schema

`users` Collection

Purpose: Authentication, profile, preferences.

Example document:

```
{ "_id": ObjectId("..."), "email": "user@example.com", "passwordHash": "...", "name": "John Doe", "phone": "+1-555-1234", "createdAt": ISODate("2025-01-01T10:00:00Z"), "updatedAt": ISODate("2025-01-05T08:00:00Z") }
```

Indexes:

- `{ email: 1 }` — **unique**

- `{ createdAt: -1 }` — listing & admin analytics

`rooms` Collection

Purpose: Static room info (hotel, type, capacity, location).
Stored separately from availability to reduce document growth.

```
{ "_id": ObjectId("..."), "hotelId": ObjectId("..."), "type": "Deluxe Suite", "capacity": 4, "price": 180, "location": { "type": "Point", "coordinates": [34.78, 32.07] }, "images": ["s3://bucket/1.jpg", "s3://bucket/2.jpg"], "amenities": ["wifi", "parking", "pool"], "createdAt": ISODate(), "updatedAt": ISODate() }
```

Indexes:

- `{ location: "2dsphere" }` — geo-search
- `{ hotelId: 1 }`
- `{ price: 1, capacity: 1 }`

`availability` Collection

Purpose: Fast lookup of available dates per room.

Two models possible choose per load requirements:

Option A — Date-based subdocument (recommended for fast queries):

```
{ "_id": ObjectId("..."), "roomId": ObjectId("..."), "dates": { "2025-06-20": "available", "2025-06-21": "reserved", "2025-06-22": "blocked" }, "updatedAt": ISODate() }
```

Mongo TTL or write-heavy warning: For very large calendars (years), we avoid huge documents.

Indexes:

- `{ roomId: 1 }`
- Or if using “1 doc per date”: `{ roomId: 1, date: 1 }`

`bookings` Collection

Purpose: Main transactional entity.
A booking is immutable after confirmation (eventually written to Kafka search index).

```
{ "_id": ObjectId("..."), "userId": ObjectId("..."), "roomId": ObjectId("..."), "hotelId": ObjectId("..."), "startDate": "2025-06-21", "endDate": "2025-06-24", "status": "confirmed", // pending | confirmed | canceled "paymentId": ObjectId("..."), "createdAt": ISODate(), "updatedAt": ISODate() }
```

Indexes:

- `{ userId: 1, createdAt: -1 }` — history view
- `{ roomId: 1, startDate: 1, endDate: 1 }` — conflict detection
- `{ status: 1 }` — operational filters / queueing

Concurrency Handling with MongoDB

A. Atomic Single-Document Updates (Recommended for per-day availability)

Model:

- Collection: `room_calendar`
- One document per `(room_id, date)` with fields:

```
{ "_id": ObjectId(), "room_id": "room123", "date": "2025-12-01", "is_available": true, "inventory": 5 }
```

Booking flow (atomic update):

```
const bookingResult = await db.collection('room_calendar').findOneAndUpdate( { room_id: roomId, date:
bookingDate, is_available: true }, { $set: { is_available: false } }, // or decrement inventory: { $inc: {
inventory: -1 } } { returnDocument: 'after' } ); if (!bookingResult.value) { throw new Error('Room not available')
; }
```

Guarantees:

- Atomic per document.
- Avoids double-booking for single-day per-room entries.

B. Multi-document Transactions (for multi-day bookings)

Model:

- Collection: `room_calendar` with one document per `(room_id, date)`.

Booking flow:

```
const session = client.startSession(); try { await session.withTransaction(async () => { const docs = await db.col
lection('room_calendar') .find({ room_id: roomId, date: { $gte: startDate, $lte: endDate } }) .toArray(); if
(docs.some(doc => !doc.is_available)) { throw new Error('Some dates not available'); } await db.collection('room_c
alendar').updateMany( { room_id: roomId, date: { $gte: startDate, $lte: endDate } }, { $set: { is_available: false
} }, { session } ); await db.collection('bookings').insertOne({ room_id: roomId, start_date: startDate, end_date:
endDate, user_id: userId, created_at: new Date() }, { session }); }); } finally { await session.endSession(); }
```

Guarantees:

- Strong consistency for multi-day bookings in replica sets or sharded clusters.
- Avoids double-booking across multiple documents.

5. Scalability Strategies

For Search

- Use **Elasticsearch / OpenSearch** cluster as before for full-text search, indexing room documents and precomputing availability snapshots.
- Sync from MongoDB via **change streams** for near-real-time updates instead of Debezium/Kafka.
- Use caches and CDN for static lists (popular queries).
- Use **geo-sharding** or **hashed sharding** in Elasticsearch indices if needed.

For Booking throughput

- Keep the **booking service stateless** and scale horizontally.
- For **write consistency**, MongoDB supports **multi-document transactions** in replica sets.
  - Option 1 (simpler): All writes go to a single primary (leader) in the primary region.
  - Option 2 (advanced): For multi-region setup, use **MongoDB global clusters** with **zone sharding** to route writes to the correct region while maintaining strong consistency via transactions.
- Use **MongoDB change streams** or a **message bus (Kafka, RabbitMQ)** for async tasks like notifications or analytics.

Caching

- Use **Redis** for hot-room details and per-room availability caches.
- Use short TTLs and proactive invalidation on booking events.
- Consider caching query results for frequent availability checks.

Multi-region

- Deploy services to multiple regions for low read latency and high availability.
- Option 1: Single-primary writes; reads from secondary replicas. Simple, consistent.
- Option 2: MongoDB global clusters with **zone sharding** for multi-region writes and strong consistency.

Autoscaling & resilience

- Use **HPA** for services; implement circuit breakers and health checks.
- Retry with jitter for transient errors.
- Bulkhead isolation: separate booking service from analytics to prevent noisy neighbors.
- Use feature flags and gradual rollout gates.

Monitoring & SLOs

- Track request latency percentiles, error rates, booking conflict rate, queue depth.
- Maintain **SLOs** with error budgets.
- Monitor MongoDB metrics: replication lag, transaction conflicts, operation throughput.