EasyBooking Endterm Project Report

Course: Advanced Databases

Project Type: Web Application (Backend + Frontend + MongoDB)

Nurkassymov Nauryzbay and Asylzhan Bakytzhan

SE-2423

Abstract

This report describes the design and implementation of EasyBooking, a full-stack web application for hotel discovery and reservation management. The project was developed to satisfy the endterm requirements of the Advanced Databases (NoSQL) course. The system includes a modular Node.js/Express backend, a MongoDB data layer with both referenced and embedded models, and a frontend that integrates with REST APIs. The application supports authentication, role-based authorization, complete CRUD operations, advanced update/delete operators, multi-stage aggregation pipelines, indexing strategy, and basic analytics for administrative users.

The final implementation exceeds the minimum endpoint and page requirements for a one-student project. In addition to core requirements, the project includes API version aliasing, structured technical documentation, and an OpenAPI specification. The result is a practical NoSQL-oriented system with real business logic and clear mapping to grading criteria.

1. Introduction and Objectives

The main goal of this project was to build a practical NoSQL-based web application that demonstrates understanding of advanced MongoDB features and backend engineering. The functional domain selected was hotel booking, because it naturally requires multiple related entities, access control, transactional-style business rules, and analytical reporting.

The project objectives were:

1. Build a working web application with backend and frontend.
2. Use MongoDB as the primary database.
3. Implement full CRUD flows for business entities.
4. Demonstrate advanced MongoDB update and delete operators.

5. Build at least one meaningful aggregation endpoint.

6. Apply authentication and authorization.

7. Provide API and database documentation.

8. Apply indexing and explain optimization decisions.

The project is implemented as a multi-page website with a REST backend. It supports guest browsing, user reservations, and administrator management/analytics.

2. Requirements Coverage Summary

The implemented solution matches and extends the required criteria:

- Web application format: completed (backend + frontend).
- MongoDB as core DB: completed.
- REST API endpoints: implemented 17 endpoints (requirement for one student is at least 8).
- Full CRUD: implemented for hotels and bookings.
- Aggregation endpoint: implemented in analytics module.
- Advanced update/delete operators: implemented with $set, $inc, $push, $pull, and positional $.
- Referenced and embedded data models: both implemented.
- Authentication/authorization: implemented with sessions and role checks.
- Frontend pages: well above minimum 4 pages.
- Documentation: README, REPORT, and OpenAPI provided.
- Indexing and optimization: single and compound indexes created at startup.

3. System Architecture

The project uses a modular layered architecture:

- src/config for environment/session configuration.
- src/models for database operations and aggregation.
- src/controllers for business logic and request handling.
- src/routes for API and web route definitions.
- src/middlewares for auth, async handling, and centralized error logic.
- src/services for startup maintenance tasks (index creation).

- views for HTML templates.
- public for static CSS/JS files.

Request Flow

1. A client sends a request to either a web route (/) or API route (/api/*).
2. Session middleware and auth middleware attach current user context.
3. Controllers validate input and apply business rules.
4. Models execute MongoDB commands.
5. Response is returned as JSON (API) or rendered HTML (web).

This structure improves maintainability and allows each layer to remain focused on one responsibility.

4. Database Design (NoSQL Modeling)

The data model uses four collections: users, hotels, bookings, and contact_requests.

4.1 Users Collection

Fields:

- _id
- email (unique)
- passwordHash
- role (user or admin)
- createdAt, updatedAt

Purpose: authentication and authorization source.

4.2 Hotels Collection

Fields:

- hotel information (title, description, location, address)
- pricing and availability (price_per_night, available_rooms)
- metadata (amenities, imageUrl)
- rating metrics (rating, ratingVotes, ratingTotal)
- embedded rating events (recentRatings[])
- creator and timestamps

This collection combines static listing data and dynamic rating data.

4.3 Bookings Collection

Fields:

- hotelId (reference to hotels)
- userId (reference to users)
- reservation details (checkIn, checkOut, guests, status, notes)
- embedded audit history (statusHistory[])
- timestamps

statusHistory stores internal state changes as embedded documents. This is useful because status updates are tightly coupled to the parent booking and are usually read with it.

4.4 Contact Requests Collection

Fields:

- customer contact fields
- status
- timestamps

Used for support inquiry form submissions.

4.5 Referenced + Embedded Approach

The project intentionally uses both patterns:

- Referenced: for cross-entity relationships (bookings to users and hotels).
- Embedded: for event-like history arrays where locality and parent-context reads are beneficial.

This hybrid approach demonstrates practical NoSQL design tradeoffs.

5. REST API and Business Logic

The API is grouped by domain and supports full functional flows.

5.1 Endpoint Set

Implemented endpoints include:

- Auth session endpoint
- Hotels endpoints (list/get/create/update/delete/rate/amenities patch)

- Bookings endpoints (list/get/create/update/delete/status patch/history patch/history delete)
- Analytics endpoint (admin)
- Version alias support via /api/v1/*

5.2 CRUD Functionality

CRUD is complete for major entities:

- Hotels: create, read (single/list), update, delete.
- Bookings: create, read (single/list), update, delete.

5.3 Business Logic Rules

Important domain rules:

- Guests can browse hotels but cannot book/rate without login.
- Users can manage only their own bookings.
- Admin users can manage hotels and all bookings.
- Booking status changes append history records.
- Rating updates adjust counters and preserve recent rating events.
- Analytics endpoint is restricted to admin role.

This ensures the application is not only CRUD-based but behavior-driven.

6. Advanced MongoDB Operations

The project explicitly uses advanced operators required by the grading rubric.

6.1 $set

Used in regular updates for hotels/bookings/user role changes and metadata updates.

6.2 $inc

Used in hotel rating flow to atomically increase ratingVotes and ratingTotal.

6.3 $push

Used to append:

- rating events into hotels.recentRatings
- status events into bookings.statusHistory

6.4 $pull

Used to remove:

- selected amenities from hotel amenities array
- specific booking history entries

6.5 Positional Operator $

Used in status history annotation:

- update specific embedded entry fields such as comment and annotation metadata.

These operators are used in real business operations, not artificial examples.

7. Aggregation Framework Implementation

A multi-stage aggregation endpoint was implemented in analyticsModel:

GET /api/analytics/overview?months=N

Pipeline stages include:

- $match on recent booking window.
- $lookup join with hotels.
- $unwind joined arrays.
- $addFields for parsed dates, nights, flags, and estimated amount.
- $facet to produce multiple outputs in one query:
- summary totals
- monthly trend
- top hotels
- city breakdown
- $group, $sort, $limit inside facet branches.

This endpoint provides practical admin insight:

- total, confirmed, cancelled bookings
- estimated revenue
- top-performing hotels
- monthly performance trend

The aggregation is meaningful and directly tied to booking business analysis.

8. Indexing and Optimization Strategy

Indexes are created automatically on startup.

Implemented indexes include:

- users.email unique index
- hotel indexes for location, price, compound filter/sort path
- booking indexes for ownership/status/createdAt queries
- additional compound booking indexes for reporting patterns
- contact request index by createdAt

Optimization goals:

1. Speed up common list filters and sorting.
2. Reduce scan cost on owner/admin booking dashboards.
3. Improve analytics query performance over recent periods.
4. Keep data access stable under larger dataset growth.

The index plan is aligned with real query patterns used by controllers and models.

9. Authentication, Authorization, and Security

Security features implemented:

- Password hashing with bcryptjs.
- Session-based auth (express-session) with Mongo-backed session storage.
- Role-based middleware (user vs admin).
- Owner/resource access checks for booking privacy.
- Input validation for hotels, bookings, registration, and contact requests.
- Safe redirect pattern to prevent open redirects.
- Centralized error and 404 handling.
- Environment-based secrets and runtime config in .env.

These protections ensure both access control and data quality.

10. Frontend and API Integration

Frontend is implemented as server-rendered HTML pages with supporting browser scripts.

Key pages:

- Home
- Hotels list/details/new/edit
- Bookings list/details/new/edit
- Login/Register
- Contact/About/Terms/Privacy
- Admin Analytics

User actions are available through buttons such as:

- Browse hotels, view details
- Book, edit, delete booking
- Rate hotel
- Add/edit/delete hotel (admin)
- Load analytics (admin)

API integration is explicitly demonstrated on the analytics page, which fetches data from /api/analytics/overview and renders dynamic metrics/tables in the browser.

11. Documentation Deliverables

The project includes:

- readme.md with architecture, schema, API, indexing, and setup.
- REPORT.md as structured defense report.
- openapi.yaml for machine-readable endpoint documentation.

This directly supports submission and defense requirements.

12. Challenges and Solutions

Challenge 1: Native bcrypt startup issue on newer Node runtime

A native bcrypt binary mismatch caused startup failures on Node v24.

- Solution: switched to bcryptjs, preserving hashing logic while avoiding native binary dependency.

Challenge 2: Startup failure on empty database namespace

When collections were missing, index inspection failed.

- Solution: added namespace-missing handling in startup maintenance before index cleanup.

Challenge 3: Empty hotel list confusion

No hotels were visible initially because production data seeding was not automatic.

- Solution: role-based admin creation flow and hotel creation UI were used to populate data intentionally.

These changes improved reliability and made first-run behavior clearer.

13. Conclusion

EasyBooking successfully fulfills the endterm objective of demonstrating advanced MongoDB and backend engineering in a real web application. The project includes complete CRUD flows, advanced update/delete operators, meaningful aggregation analytics, proper data modeling with references and embedded documents, role-based security, indexing, and clear documentation.