**Smart Traffic Automation**

**1. Project Title**

**Smart Traffic Automation - Booking-based Traffic Smoothing with Occupancy-Aware Routing**

Short description: a web app where users reserve road-usage slots on a simulated road graph. The backend (Node/Express) computes optimal routes (Dijkstra) using occupancy-adjusted weights, invokes a Python predictor for recommended speeds, and stores users, roads and bookings in MongoDB.

**2. Database Schema**

**Theory (what a “schema” means in MongoDB / NoSQL)**

* MongoDB is schema-flexible (schema-less): collections store BSON documents which can vary per document.
* Even though schema is flexible, it is best practice to **design a logical schema** (shape of documents, required fields, types) and to enforce it at the application layer (Mongoose schemas or validation rules).
* For performance, design around common queries — embed when read-mostly and related, reference for one-to-many or large arrays.
* Index frequently queried fields (e.g., email, slot, user) to speed reads.

**Our collection design (logical schema)**

We used three primary collections: users, roads, bookings. Below are Mongoose-style schemas and example JSON documents.

**Users collection (users)**

**Purpose:** authentication and role-based access.

Mongoose schema:

// models/User.js

const UserSchema = new Schema({

email: { type: String, required: true, unique: true },

passwordHash: { type: String, required: true }, // bcrypt

name: { type: String },

role: { type: String, enum: ['user','admin'], default: 'user' },

createdAt: { type: Date, default: Date.now },

lastLoginAt: { type: Date }

});

Sample document:

{

"\_id":"64a1abcd1234",

"email":"user@user.com",

"passwordHash":"$2b$10$abcdefg...",

"name":"Sanjay",

"role":"user",

"createdAt":"2025-09-15T09:12:00Z",

"lastLoginAt":"2025-09-20T10:20:00Z"

}

Indexes:

* Unique index on email: db.users.createIndex({email: 1},{unique: true})

Notes:

* Passwords are never stored in plaintext; we store bcrypt hashes.
* lastLoginAt is optional and updated on successful login.

**Bookings collection (bookings)**

**Purpose:** store each booking with computed path and recommended speed.

Mongoose schema:

// models/Booking.js

const BookingSchema = new Schema({

user: { type: Schema.Types.ObjectId, ref: "User", required: true },

userEmail: { type: String, required: true }, // denormalized for quick display

path: { type: [String], required: true }, // list of node ids, e.g., ["N1","N5","N9"]

slot: { type: Date, required: true },

status: { type: String, enum: ['pending','confirmed','cancelled'], default: 'confirmed' },

recommendedSpeed: { type: Number },

features: { type: Schema.Types.Mixed }, // e.g. path\_length, hour, occupancy\_score

createdAt: { type: Date, default: Date.now }

});

BookingSchema.index({ user: 1 });

BookingSchema.index({ slot: 1 });

Sample document:

{

"\_id":"64bk001",

"user":"64a1abcd1234",

"userEmail":"user@user.com",

"path":["N1","N5","N9"],

"slot":"2025-10-26T10:00:00Z",

"status":"confirmed",

"recommendedSpeed":38.0,

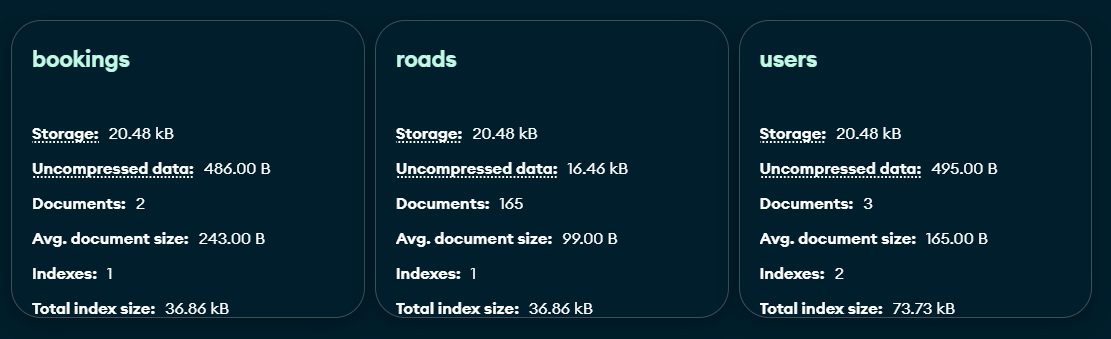
"features":{"path\_length":3,"hour":10,"is\_peak":1,"occupancy\_score":0.15},

"createdAt":"2025-09-15T09:12:00Z"

}

Indexes:

* bookings.user: fast retrieval of user history.
* bookings.slot: for slot-based occupancy queries.

**

*Fig 2.1 : Schema Collection*

**3. Number of Instances**

* In MongoDB context, **instances** often refers to running MongoDB server processes. Typical deployment options:
  + **Single-node (standalone)**: one mongod process — simplest for development/demo.
  + **Replica set**: multiple mongod processes forming a fault-tolerant cluster (primary + secondaries) — used in production for HA.
  + **Sharded cluster**: multiple shards for horizontal scalability — overkill for this project.

**What we used:**

* **Development/demo**: **single local MongoDB instance** (one mongod process) running on localhost:27017.
  + This is sufficient for the lab demonstration, seeding and experiments.

**4. Queries**

**Query types & Rationale**

* **Authentication login**: find a user by email to validate password. Use an index on email for O(log n) lookup.
* **Booking creation**: fetch open roads to build the graph (read many road documents). Use index on (from,to) for adjacency builds.
* **Occupancy calculation**: count bookings overlapping a slot for an edge — use aggregation pipeline to compute occupancy ratios.
* **User booking history**: find bookings filtered by user (ObjectId) sorted by slot or createdAt. Use index on user.
* **Admin listing**: list all bookings optionally paginated.

Below are the exact queries used in the project — both in **raw Mongo shell** and the corresponding **Mongoose/Node.js** code.

**A) Login — find user by email (Mongo shell)**

// mongo shell

use traffic

db.users.findOne({ email: "user@user.com" })

**Mongoose (Node.js)**

const user = await User.findOne({ email: req.body.email }).exec();

if (!user) return res.status(401).json({ error: "Invalid credentials" });

// then bcrypt.compare(req.body.password, user.passwordHash)

**Notes:** We use findOne to fetch the hashed password and compare using bcrypt on the server.

**B) Register — insert user (Mongo shell)**

db.users.insertOne({

email: "newuser@example.com",

passwordHash: "<bcrypt-hash>",

name: "New User",

role: "user",

createdAt: new Date()

});

**Mongoose**

const newUser = new User({ email, passwordHash, name, role: 'user' });

await newUser.save();

**C) Load roads for graph building (all open roads) — Mongo shell**

db.roads.find({ status: "open" }).toArray()

**Mongoose**

const roads = await Road.find({ status: "open" }).lean().exec();

**Why .lean()?** Returns plain JS objects (faster) because we don't need Mongoose documents for read-only graph building.

**D) Booking creation — store booking (Mongo shell)**

db.bookings.insertOne({

user: ObjectId("64a1abcd1234"),

userEmail: "user@user.com",

path: ["N1","N5","N9"],

slot: ISODate("2025-10-26T10:00:00Z"),

status: "confirmed",

recommendedSpeed: 38.0,

features: { path\_length: 3, hour: 10, is\_peak: 1 },

createdAt: new Date()

});

**Mongoose**

const b = new Booking({ user: req.user.\_id, userEmail: req.user.email, path, slot, recommendedSpeed, features });

await b.save();

**E) Booking history for a user — Mongo shell**

db.bookings.find({ user: ObjectId("64a1abcd1234") }).sort({ slot: -1 }).toArray()

**Mongoose**

const bookings = await Booking.find({ user: userId }).sort({ slot: -1 }).lean().exec();

**F) Admin: list all bookings (with pagination) — Mongo shell**

db.bookings.find({}).sort({ slot: -1 }).skip(0).limit(50).toArray()

**Mongoose**

const page = parseInt(req.query.page || 0, 10);

const perPage = 50;

const bookings = await Booking.find({}).sort({ slot: -1 }).skip(page \* perPage).limit(perPage).lean().exec();

**G) Occupancy check for an edge and slot window — Aggregation pipeline (important)**

We need to compute how many bookings occupy an edge (or node pair) in a given slot period. This helps compute occupancy\_ratio = bookings\_on\_edge / capacity.

**Assumptions**:

* Bookings store path as list of node ids (e.g., [N1,N5,N9]). To count bookings that use a specific edge (from A to B), we check whether path contains both nodes in consecutive order or just include either node (simpler). For accurate edge counting one can store edges array for each booking during creation (recommended optimization). Below is a pipeline that counts bookings whose path contains both from and to in the array (not necessarily consecutive — conservative estimate).

**Mongo shell pipeline (count bookings for edge from: "N1", to: "N5" in a time window)**

const from = "N1";

const to = "N5";

const start = ISODate("2025-10-26T09:30:00Z"); // slot window start

const end = ISODate("2025-10-26T10:30:00Z"); // window end

db.bookings.aggregate([

{ $match: {

slot: { $gte: start, $lt: end },

status: "confirmed",

path: { $all: [from, to] } // conservative: path includes both nodes

}},

{ $count: "bookingsUsingEdge" }

])

**Mongoose**

const countRes = await Booking.aggregate([

{ $match: { slot: { $gte: start, $lt: end }, status: "confirmed", path: { $all: [from,to] } } },

{ $count: "bookingsUsingEdge" }

]);

const bookingsUsingEdge = (countRes[0] && countRes[0].bookingsUsingEdge) || 0;

**More accurate edge detection (recommended):** When creating bookings, also store an edges array of pairs like ["N1-N5","N5-N9"]. Then you can match edges: "N1-N5" using simple $match.

**Example: add edges at booking creation (recommended)**:

// build edges from path

const edges = path.map((n,i) => i < path.length-1 ? `${path[i]}-${path[i+1]}` : null).filter(Boolean);

// save edges: edges

Then occupancy query:

db.bookings.countDocuments({

edges: "N1-N5",

slot: { $gte: start, $lt: end },

status: "confirmed"

})

This is faster and accurate.

**H) Toggle road status (admin action) — Mongo shell**

db.roads.updateOne({ \_id: ObjectId("64r23abcd") }, { $set: { status: "closed" } })

**Mongoose**

await Road.findByIdAndUpdate(roadId, { status: "closed" });

**I) Export bookings to CSV (example query)**

Use mongoexport or produce CSV from Node:

**mongoexport CLI**

mongoexport --db=traffic --collection=bookings --query='{}' --type=csv --fields=\_id,userEmail,slot,path,recommendedSpeed,status,createdAt --out=bookings\_export.csv

**Node (stream to CSV)** — use a library like json2csv after Booking.find().lean().

**5. Relationships**

**Theory (How relationships are modeled in MongoDB)**

* MongoDB supports **embedding** (store related data inside documents) and **referencing** (store ObjectId references).
* Use embedding for one-to-few tightly coupled data and where reads are frequent and require both sets. Use referencing for one-to-many large or unbounded arrays (e.g., user → bookings).
* For relational integrity, application code enforces referential behavior (MongoDB does not enforce foreign keys).

**Our design choices (implementation)**

* **User — Booking**: **one-to-many** implemented via **referencing**. Bookings store a user ObjectId that references users.\_id. We also **denormalized** userEmail inside the booking for quick display in admin views without needing a join.
  + Rationale: bookings can be many per user (unbounded), and storing them inside users would create unbounded growth on a single document — referencing is appropriate.
* **Roads — Bookings**: We **do not directly reference roads** in bookings. Instead, bookings store path as an ordered array of node IDs (and optionally edges as "A-B").
  + Rationale: bookings reference the route path rather than road documents directly. During occupancy computation we match edges against road identifiers (or path nodes) to determine how bookings influence occupancy.
* **Nodes** (if stored): nodes may be stored as separate documents; roads reference node ids. In our minimal setup we used node ids embedded in roads.from/roads.to and bookings.path.

**Relationship diagram (simple)**

* User (1) ---> (N) Booking
* Road used by Booking via path/edges (indirect relationship).

**Referential actions and integrity**

* When a **user is deleted**, we can either:
  + cascade-delete bookings (dangerous for audit), or
  + anonymize (userEmail set to deleted@example.com, user set to null) — recommended for audit and history retention.
* When a **road is removed**, we should avoid deleting historical bookings because past bookings are records. Mark road status: 'removed' or keep road metadata for historical analysis.