Final year project report

Hospital Management System

A MERN application with role-based portals and conflict-safe appointment booking

Document	Sample project report, published as a free download
Prepared by	Yaseen Nurmahammad Khatib, Senior Full-Stack AI Engineer
Published at	DefendIt — campus.streamerosai.com/sample-project-report/
Project page	campus.streamerosai.com/hospital-management-mern/
Stack	React, Node.js, Express, MongoDB, Mongoose, JWT, Tailwind CSS
Date	October 2026

About this sample. This report shows the structure and depth of the project report delivered with the Academic and Mentored tiers for this project. It describes the system exactly as the project page describes it. The college title page, certificate, declaration and acknowledgement are left out because they belong to your institution; the delivered report carries them in your department's format. It is published to be read, not submitted: a report earns marks only when it matches the code you submit and you can defend both.

Table of contents

Abstract 2

1. Introduction 3

1.1 Background 3
1.2 Problem statement 3
1.3 Objectives 3
1.4 Scope 3

2. Literature survey and existing system 4

2.1 Typical student implementations 4
2.2 Proposed system 4

3. Software requirements specification 5

3.1 Functional requirements 5
3.2 Non-functional requirements 5
3.3 Hardware and software requirements 6
3.4 Use cases 6

4. System design 7

4.1 Architecture 7
4.2 Data model 7
4.3 Data flow 8
4.4 The appointment engine 9
4.5 API design 10
4.6 Design decisions and their justification 10

5. Implementation 12

5.1 Project structure 12
5.2 Authentication and role guards 12
5.3 Appointment booking and conflict detection 13
5.4 Prescriptions and billing 14
5.5 Front end 14
5.6 Deployment 14

6. Testing 15

6.1 Strategy 15
6.2 Test cases 15
6.3 Known limitations 16

7. Results 17

8. Conclusion and future scope 18

References 19

Appendix A: Viva questions on this project 20

Abstract

Hospital management is the most-attempted final year project in Indian colleges, and most submissions stop at a set of forms over a database. This project builds the system the way it should be built: separate portals for patients, doctors and administrators behind a single login; JSON Web Token authentication with role guards enforced on every API route rather than only in the user interface; appointment booking that detects slot collisions at the database level instead of overwriting them; and prescriptions and billing tied to real MongoDB relations through the appointment they belong to.

The system is implemented on the MERN stack — MongoDB, Express, React and Node.js — with Mongoose for schema validation, bcrypt for password hashing and Tailwind CSS for the interface. A doctor's availability is stored as weekly working hours divided into fixed slots. A unique compound index on doctor and slot start time means a second booking for the same slot is rejected by the database itself, even when two patients confirm at the same moment; the API turns that rejection into a clear slot-taken message. Slots are stored in UTC and displayed in Indian Standard Time so a server in another region does not shift appointments.

The report covers the requirements specification, the architecture and data model with diagrams, the implementation of the authentication layer and the appointment engine with code listings, a test case table built from the numbered functional requirements, and the limitations and future scope. The marks in this project are in the schema design and the authentication flow, and the report spends its depth there.

Keywords: hospital management system, MERN stack, MongoDB, Express, React, Node.js, JWT, role-based access control, appointment scheduling, unique index.

1. Introduction

1.1 Background

A hospital's daily work is appointments. Patients want a slot with a particular doctor; doctors have fixed working hours; the front desk must not promise the same slot twice; after the visit a prescription has to be recorded and a bill raised. Paper registers and spreadsheets handle this badly: double bookings are common, a patient's history is scattered, and nobody can say how busy a department was last week without counting by hand.

A web application solves the visibility problem easily. The harder problem, and the one this project is built around, is correctness under concurrency: two patients pressing confirm on the same slot at the same time, and a user calling the API directly to read records that are not theirs.

1.2 Problem statement

Design and implement a hospital management system in which patients book appointments against a doctor's real availability, the system refuses a double booking under concurrent requests, each role sees only the data it is entitled to, doctors record prescriptions against a visit, bills are generated from the visit, and administrators manage doctors, departments and daily load — with the access rules enforced by the server, not by the interface.

1.3 Objectives

1. Provide three role-based portals — patient, doctor, admin — behind one login.
2. Authenticate with JWT and enforce the role required by every API route in server middleware.
3. Generate bookable slots from a doctor's weekly schedule and detect slot conflicts per doctor.
4. Guarantee that a slot cannot be double-booked, including under simultaneous requests.
5. Let doctors write prescriptions against a completed visit, and let patients see their history.
6. Generate a bill from the consultation fee and listed services, with a printable invoice.
7. Give administrators doctor onboarding, department management and a per-department, per-day load view.
8. Provide search and filtering across patients, doctors and appointments.

1.4 Scope

In scope: the three portals, authentication and authorisation, appointment scheduling with conflict detection, prescriptions, billing with printable invoices, the admin dashboard, and search.

Out of scope: online payment collection (bills are recorded and printed, not paid through the system), pharmacy inventory, laboratory results, insurance claims, SMS or email reminders, and multi-branch hospitals. These are listed under future scope in chapter 8.

2. Literature survey and existing system

2.1 Typical student implementations

The common hospital management project is a set of CRUD screens — add patient, add doctor, add appointment — over a single flat table. Three weaknesses recur. First, appointment booking is a plain insert: the application may check for a clash before inserting, but two requests can both pass the check and both insert, so a double booking is possible exactly when the system is busiest. Second, authorisation lives in the interface: a page is hidden from a role, but the API behind it answers anyone who calls it. Third, records are not related: a prescription stores a patient name as text rather than a reference, so history cannot be queried reliably.

Commercial hospital information systems solve these problems at a scale and cost outside a final year project, and their internals are not available to study. The relevant published material is instead the documentation of the building blocks: MongoDB's treatment of unique indexes as a correctness guarantee, Express's middleware model for cross-cutting checks, the JSON Web Token specification, and the bcrypt password scheme (see References).

2.2 Proposed system

Table 1: Existing approach versus the proposed system

Concern	Typical submission	This system
Double booking	Check-then-insert in application code; a race window remains.	Unique compound index on (doctor, slotStart); the database rejects the second insert atomically.
Authorisation	Pages hidden per role; API routes unprotected.	requireAuth and requireRole middleware on every route; queries filtered by the signed-in user's id.
Records	Names stored as text in each table.	Six related collections with Mongoose references; history is a query, not a search.
Passwords	Stored in plain text or reversible encoding.	bcrypt hash with per-password salt; the plain password is never stored.
Time	Local server time stored as-is.	Slots stored in UTC, displayed in IST.

3. Software requirements specification

Requirements are numbered so each one can be traced to a test case in chapter 6. The layout follows the practice recommended in IEEE Std 830-1998.

3.1 Functional requirements

Table 2: Functional requirements

ID	Requirement
FR-01	A user can register as a patient and sign in with email and password.
FR-02	An administrator can onboard a doctor with a department, consultation fee and weekly schedule.
FR-03	On sign-in the system issues a JWT carrying the user id and role.
FR-04	Every API route verifies the JWT and checks the role it requires before the handler runs.
FR-05	The system generates bookable slots from a doctor's weekly working hours and slot length.
FR-06	A patient can view a doctor's free slots for a date and book one.
FR-07	A slot already booked for that doctor cannot be booked again, including when two requests arrive together.
FR-08	A patient can cancel a booked appointment before its start time.
FR-09	A doctor can see their own appointments for a day and mark a visit completed.
FR-10	A doctor can write a prescription — medicines with dose and duration, and notes — against a completed visit.
FR-11	A patient can view their appointment history and the prescriptions linked to it.
FR-12	Completing a visit generates a bill from the consultation fee plus listed services, viewable and printable as an invoice.
FR-13	An administrator can manage departments and view appointments per department and per day.
FR-14	Patients, doctors and appointments can be searched and filtered by name, department, date and status.

3.2 Non-functional requirements

Table 3: Non-functional requirements

ID	Requirement
NFR-01	Security: passwords are stored as bcrypt hashes; tokens are signed with a server-side secret; role checks are enforced on the server.
NFR-02	Data isolation: a patient can read only their own records and a doctor only their own patients, enforced by filtering every query on the signed-in user's id.
NFR-03	Integrity: required fields and enumerated values (for example appointment status) are validated at the model layer.
NFR-04	Consistency: appointment times are stored in UTC and rendered in IST.
NFR-05	Usability: each portal shows only the actions available to its role; forms validate input before submission and the API validates it again.

ID	Requirement
NFR-06	Deployability: the front end builds to static files and the API runs on a free Node.js host with a MongoDB Atlas free-tier cluster.

3.3 Hardware and software requirements

Table 4: Hardware and software requirements

Item	Requirement
Development machine	Any laptop able to run Node.js and a browser; 8 GB RAM is comfortable.
Runtime	Node.js 18 or later, npm.
Database	MongoDB 6 or later, locally or on MongoDB Atlas.
Front end	React 18, React Router, Tailwind CSS.
Back end	Express, Mongoose, jsonwebtoken, bcrypt.
Browser	Any current Chromium, Firefox or Safari release.

3.4 Use cases

Table 5: Principal use cases

Use case	Actor	Main flow
Book appointment	Patient	Choose department and doctor; pick a date; the free slots are listed; confirm a slot; the appointment is created with status booked, or the slot is reported as taken.
Complete visit and prescribe	Doctor	Open today's list; mark a visit completed; enter medicines with dose and duration and notes; the prescription is linked to the appointment and a bill is generated.
View history	Patient	Open history; appointments are listed newest first with the linked prescription and bill for each completed visit.
Onboard doctor	Admin	Create the doctor's user account, department, fee and weekly schedule; the doctor appears in booking immediately.
Review daily load	Admin	Pick a date; appointments are aggregated per department and per doctor.

4. System design

4.1 Architecture

The system is a three-tier web application. The React client is a single-page application with a route tree per role; it holds the JWT in memory and sends it as a Bearer header on every request. The Express API is the only component that talks to the database, and it is where every access rule lives. MongoDB stores six related collections through Mongoose models.

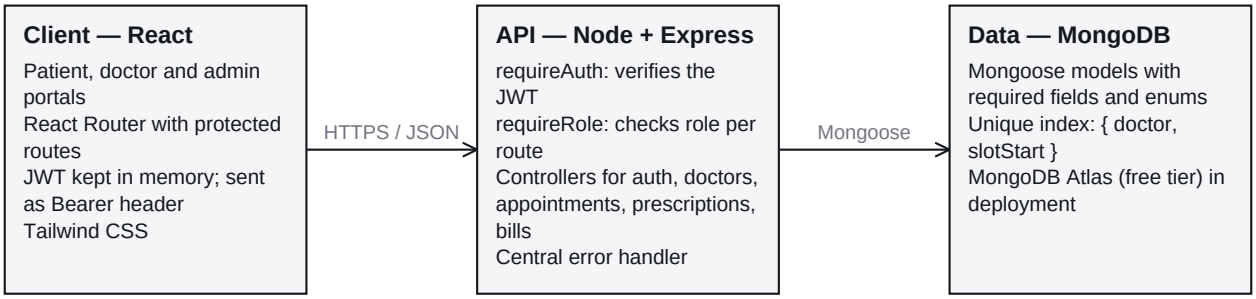


Figure 1: System architecture

One login serves three roles. The JWT carries the role; React renders the patient, doctor or admin portal from it; and Express middleware enforces the same role on every API route, so the interface hiding a button is never the only protection.

4.2 Data model

Six collections. A user carries the credentials and the role; a doctor or patient document extends the user with role-specific fields. Appointments reference a doctor and a patient and carry the slot start time and status. A prescription and a bill each reference the appointment they belong to, so a patient's history is every prescription on their completed appointments — a query, not a text search.

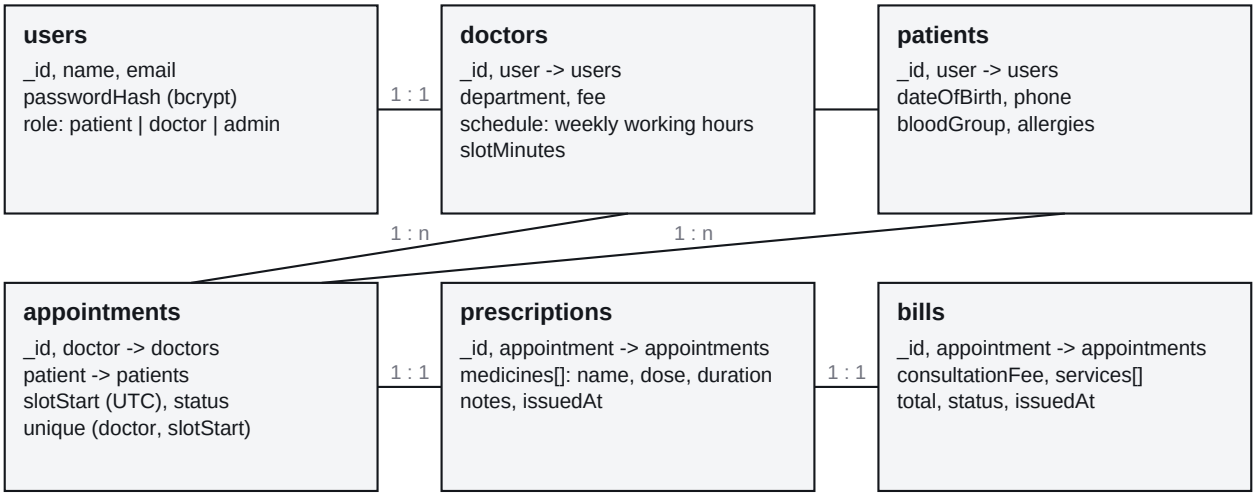


Figure 2: Data model — collections and references

Table 6: Collections and validation rules

Collection	Fields	Validation
users	name, email (unique), passwordHash, role	role is an enum: patient, doctor, admin
doctors	user, department, fee, schedule (weekday -> start, end), slotMinutes	fee is a non-negative number; schedule times are HH:MM
patients	user, dateOfBirth, phone, bloodGroup, allergies	phone is 10 digits
appointments	doctor, patient, slotStart, status, createdAt	status enum: booked, completed, cancelled; unique (doctor, slotStart)
prescriptions	appointment, medicines [name, dose, duration], notes, issuedAt	at least one medicine; appointment must be completed
bills	appointment, consultationFee, services [name, amount], total, status, issuedAt	total is computed server-side, never accepted from the client

4.3 Data flow

At the context level, three external entities interact with one process that reads and writes a single data store.

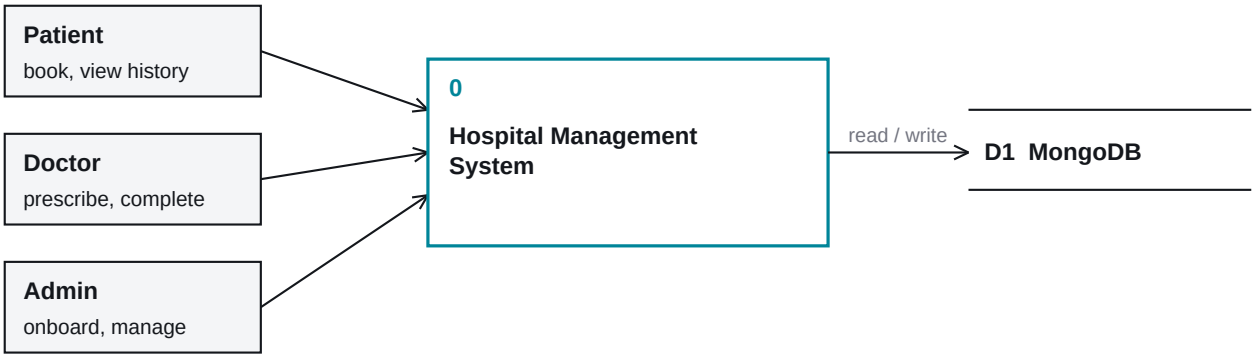


Figure 3: Level 0 data flow diagram

At level 1 the process splits into authentication, appointment scheduling, clinical records (prescriptions) and billing, with the admin dashboard reading aggregates from the appointments collection. The delivered report carries the level 1 diagram; it is omitted here for length.

4.4 The appointment engine

A doctor's availability is stored as weekly working hours — for example Monday 10:00 to 13:00 — and a slot length in minutes. Slots for a date are generated on request by walking that window in slot-length steps, then removing any slot that already has a booked appointment for that doctor. The generation is deterministic, so the client and server always agree on which instants are valid slot starts.

```
function freeSlots(doctor, date):
    window  = doctor.schedule[weekday(date)]        # e.g. 10:00-13:00, or none
    if window is none: return []
    slots   = every instant from window.start to window.end step doctor.slotMinutes
    taken   = slotStart of appointments where doctor = doctor and date(slotStart) = date
                                            and status != cancelled
    return [s for s in slots if s not in taken]
```

Listing: Slot generation, in pseudocode

Removing taken slots before showing them handles the normal case. It does not handle the race: two patients can load the same free list and both confirm. The guarantee comes from the database. A unique compound index on doctor and slotStart means the second insert fails with a duplicate-key error, and the API turns that error into HTTP 409 with a slot-taken message. The sequence below is the whole mechanism.

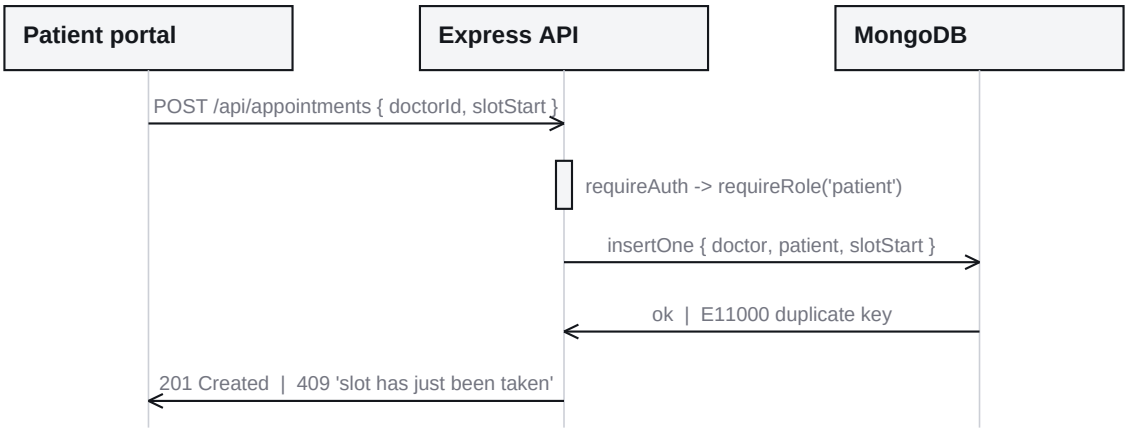


Figure 4: Booking an appointment — the conflict is resolved by the database

4.5 API design

Table 7: API routes

Method and path	Role	Purpose
POST /api/auth/register	public	Create a patient account.
POST /api/auth/login	public	Verify the password hash and issue a JWT.
GET /api/doctors?department=	any	List doctors, optionally by department.
GET /api/doctors/:id/slots?date=	patient	Free slots for a doctor on a date.
POST /api/appointments	patient	Book a slot; 409 if it has just been taken.
DELETE /api/appointments/:id	patient	Cancel before start time; only the owner may cancel.
GET /api/appointments/mine	patient	The signed-in patient's history.
GET /api/appointments/today	doctor	The signed-in doctor's appointments for a day.
PATCH /api/appointments/:id/complete	doctor	Mark the visit completed and generate the bill.
POST /api/prescriptions	doctor	Write a prescription against a completed appointment.
GET /api/bills/:id	patient, admin	View a bill; the patient must own it.
POST /api/admin/doctors	admin	Onboard a doctor.
GET /api/admin/load?date=	admin	Appointments per department and doctor for a day.

4.6 Design decisions and their justification

- MongoDB rather than a relational database. The records are naturally document-shaped — a prescription with a list of medicines, a bill with a list of services — and Mongoose gives schema validation at the model layer. For heavy cross-record reporting or strict transactions across many tables a relational database would be a reasonable alternative, and that trade-off is stated rather than hidden.
- A unique index rather than an application-level check. Checking for a clash and then inserting leaves a gap where two requests both pass the check. The index makes the second insert fail atomically, which application code cannot guarantee.
- Role checks in middleware rather than in the interface. Anyone can call the API directly with a

tool like Postman. Middleware that reads the role from a verified token is the real protection; hiding pages is only convenience.

- Filtering every query by the signed-in user's id rather than by an id sent from the browser. A patient's history endpoint takes no patient id parameter at all; the server derives it from the token.
- UTC storage with IST display. A server running in another region would otherwise shift every appointment by five and a half hours.
- Bill totals computed on the server. The client sends service names; the server looks up amounts and sums them, so a total can never be tampered with.

5. Implementation

5.1 Project structure

```
hospital-management/
  client/                       React application (Vite)
    src/routes/                 patient/, doctor/, admin/ route trees
    src/components/             shared UI: Protected, SlotPicker, InvoiceView
    src/lib/api.js              fetch wrapper that attaches the Bearer token
  server/
    src/models/                 User, Doctor, Patient, Appointment, Prescription, Bill
    src/middleware/auth.js      requireAuth, requireRole
    src/controllers/            auth, doctors, appointments, prescriptions, bills, admin
    src/routes/                 one router per resource
    src/lib/slots.js            slot generation
    src/app.js                  Express app, central error handler
    seed.js                     sample departments, doctors and patients
  .env.example                  MONGODB_URI, JWT_SECRET, PORT
```

Listing: Repository layout

5.2 Authentication and role guards

Registration hashes the password with bcrypt and a per-password salt; the plain password is never stored. Login compares the hash and, on success, signs a token carrying the user id and role. Two middleware functions then protect every route.

```
// server/src/middleware/auth.js
const jwt = require("jsonwebtoken");

function requireAuth(req, res, next) {
  const header = req.headers.authorization || "";
  const token = header.startsWith("Bearer ") ? header.slice(7) : null;
  if (!token) return res.status(401).json({ error: "Not signed in" });
  try {
    // payload: { id, role, patientId?, doctorId? }
    req.user = jwt.verify(token, process.env.JWT_SECRET);
    return next();
  } catch {
    return res.status(401).json({ error: "Session expired" });
  }
}

function requireRole(...roles) {
  return (req, res, next) =>
    roles.includes(req.user.role)
      ? next()
      : res.status(403).json({ error: "Forbidden" });
}

module.exports = { requireAuth, requireRole };
```

Listing: JWT verification and role guard middleware

Every protected route composes the two, and every query inside the handler is filtered by the id the token carries, never by an id from the request:

```
// server/src/routes/appointments.js
router.get(
  "/mine",
  requireAuth,
  requireRole("patient"),
  async (req, res) => {
    const list = await Appointment.find({ patient: req.user.patientId })
      .populate("doctor", "name department fee")
      .sort({ slotStart: -1 });
    res.json(list);
  },
);
```

Listing: A patient's history is filtered by the token, not by a parameter

5.3 Appointment booking and conflict detection

The model declares the unique compound index. Mongoose creates it when the model is first used; in production it is created once with createIndexes.

```
// server/src/models/Appointment.js
const { Schema, model } = require("mongoose");

const appointmentSchema = new Schema(
  {
    doctor:    { type: Schema.Types.ObjectId, ref: "Doctor",  required: true },
    patient:   { type: Schema.Types.ObjectId, ref: "Patient", required: true },
    slotStart: { type: Date, required: true },                  // stored in UTC
    status:    { type: String, enum: ["booked", "completed", "cancelled"], default: "booked" },
  },
  { timestamps: true },
);

// The database, not the application, refuses a second booking for a slot.
appointmentSchema.index({ doctor: 1, slotStart: 1 }, { unique: true });

module.exports = model("Appointment", appointmentSchema);
```

Listing: Appointment model with the unique compound index

The controller validates that the requested instant is one of the doctor's generated slots, then inserts. A duplicate-key error (code 11000) is the signal that another booking won the race.

```
// server/src/controllers/appointments.js
async function book(req, res, next) {
  const { doctorId, slotStart } = req.body;
  const start = new Date(slotStart);
  const valid = await isGeneratedSlot(doctorId, start);   // from lib/slots.js
  if (!valid) return res.status(400).json({ error: "Not a bookable slot" });
  try {
    const appt = await Appointment.create({
      doctor: doctorId,
      patient: req.user.patientId,
      slotStart: start,
    });
    return res.status(201).json(appt);
  } catch (err) {
    if (err.code === 11000) {
      return res.status(409).json({ error: "That slot has just been taken" });
    }
    return next(err);                                    // central error handler
  }
}
```

Listing: Booking: the duplicate-key error becomes a 409

A cancelled appointment keeps its row with status cancelled, so the unique index would block re-booking that slot. The slot generator therefore treats cancelled appointments as free, and cancelling an appointment deletes the row rather than only changing its status, with the cancellation recorded on the patient's record. The delivered report discusses this trade-off in full.

5.4 Prescriptions and billing

Marking a visit completed is a single endpoint that does two things in order: it sets the appointment status to completed, and it creates a bill whose consultation fee is copied from the doctor's record at that moment, so a later fee change does not alter an old bill. The doctor then writes the prescription against the completed appointment; the model refuses a prescription for an appointment that is still booked. The patient's portal shows the prescription and the bill under the visit, and the invoice view renders a printable page with the hospital header, line items and total.

5.5 Front end

The React application keeps the token and decoded role in a context. A Protected component wraps each route tree and redirects to the login page when the role does not match; this is convenience, not security, which is why chapter 4 places the real check on the server. The slot picker calls the free-slots endpoint for the chosen date and renders IST times from the UTC instants the API returns. Forms validate required fields and formats before submission; the API validates again.

5.6 Deployment

The React build is static files served from a free static host. The API runs on a free Node.js host with MONGODB_URI and JWT_SECRET set as environment variables, and the database is a MongoDB Atlas free-tier cluster with the API host's address allow-listed. The client's API base URL is set at build time. The delivered report includes the exact steps and the live URL of the deployed instance.

6. Testing

6.1 Strategy

Three levels. Unit tests cover slot generation and the role middleware in isolation. API tests run against a test database and exercise each route with each role, including the roles that must be refused. Manual system tests walk the three portals end to end on seeded data. The table below maps test cases to the functional requirements in chapter 3; the delivered report carries the actual result column from the student's own run, which is why only the expected result is given here.

6.2 Test cases

Table 8: Test cases mapped to requirements

ID	Req.	Scenario and input	Expected result
TC-01	FR-01	Register with a valid email and an 8-character password.	201; user stored with a bcrypt hash, not the password.
TC-02	FR-01	Register with an email that already exists.	409 with a clear message; no duplicate user.
TC-03	FR-03	Login with the correct password.	200 with a JWT whose payload carries id and role.
TC-04	FR-04	Call GET /api/appointments/today with a patient's token.	403 Forbidden.
TC-05	FR-04	Call any protected route with no token or an expired token.	401.
TC-06	FR-05	Request slots for a doctor on a weekday outside their schedule.	Empty list.
TC-07	FR-06	Book a listed free slot.	201; the slot disappears from the free list.
TC-08	FR-07	Book the same slot for the same doctor a second time.	409 'That slot has just been taken'.
TC-09	FR-07	Send two booking requests for the same slot concurrently (script).	Exactly one 201 and one 409; one appointment in the database.
TC-10	FR-06	Book an instant that is not a generated slot start.	400 'Not a bookable slot'.
TC-11	FR-08	Cancel another patient's appointment by id.	403 or 404; the appointment is unchanged.
TC-12	FR-10	Write a prescription against an appointment still in status booked.	400; prescription refused.
TC-13	FR-12	Complete a visit for a doctor with fee 500 and one service of 200.	Bill created with total 700, computed on the server.
TC-14	NFR-02	Request GET /api/bills/:id for a bill owned by another patient.	403; no bill data returned.
TC-15	NFR-04	Book a 10:00 IST slot; read it back from the database.	Stored as 04:30 UTC; displayed as 10:00 IST.
TC-16	FR-14	Filter appointments by department and date in the admin portal.	Only matching appointments listed.

ID	Req.	Scenario and input	Expected result
TC-17	NFR-05	Submit the registration form with a letter in the phone field.	Client-side validation message; API also rejects it with 400.

6.3 Known limitations

- Cancelling deletes the appointment row so the slot can be re-booked; an audit of cancellations is kept on the patient record rather than in the appointments collection.
- Slot generation assumes one continuous working window per weekday; split shifts need a schedule with multiple windows.
- Bills are recorded and printed, not paid online.
- The free-tier API host sleeps when idle, so the first request after a pause is slow; this is a hosting limit, not a code limit.

7. Results

Against the objectives in section 1.3:

1. Three portals behind one login — delivered. The same credentials route a user to the patient, doctor or admin tree based on the role in the token.
2. Server-enforced roles — delivered. Every route composes requireAuth and requireRole; TC-04 and TC-05 exercise the refusals.
3. Slot generation and conflict detection — delivered. Free slots are generated from the weekly schedule; booked ones are removed.
4. No double booking under concurrency — delivered by the unique compound index; TC-09 exercises the race with two simultaneous requests.
5. Prescriptions against completed visits, visible in the patient's history — delivered.
6. Bills from fee plus services with a printable invoice — delivered; totals are computed on the server.
7. Admin onboarding, departments and daily load — delivered.
8. Search and filtering — delivered.

The delivered report for a purchased build includes screenshots of each portal on seeded data: the patient's slot picker with taken slots greyed out, the 409 message when a slot is lost, the doctor's day list and prescription form, the printable invoice, and the admin load view. They are described rather than pasted in this sample so that nothing shown here can be mistaken for a submission.

8. Conclusion and future scope

The project set out to build the most familiar final year topic the way it should be built, and the difference is in two places a demo never shows: the database refusing a double booking that application code cannot reliably prevent, and the server refusing a request the interface would never have made. Those two mechanisms, with a data model whose references make history a query rather than a search, are what make the system defensible rather than merely runnable.

Concrete extensions, each building on the existing layers:

- Multiple working windows per day in the doctor schedule, extending the slot generator without changing the index.
- Online bill payment through a test-mode payment gateway, writing a payment record against the bill.
- Reminder notifications by email or SMS from a scheduled job over the appointments collection.
- A pharmacy module that decrements stock from the medicines on a prescription.
- Per-branch scoping by adding a branch reference to doctors and appointments.

References

1. MongoDB Inc. MongoDB Manual: Unique Indexes. https://www.mongodb.com/docs/manual/core/index-unique/
2. OpenJS Foundation. Express — Using middleware. https://expressjs.com/en/guide/using-middleware.html
3. Mongoose documentation: Schemas, validation and indexes. https://mongoosejs.com/docs/guide.html
4. Meta Open Source. React documentation. https://react.dev/
5. M. Jones, J. Bradley and N. Sakimura. JSON Web Token (JWT). RFC 7519, IETF, May 2015.
6. N. Provos and D. Mazieres. A Future-Adaptable Password Scheme. Proceedings of the USENIX Annual Technical Conference, 1999.
7. OWASP Foundation. OWASP Top Ten. https://owasp.org/www-project-top-ten/
8. IEEE. IEEE Std 830-1998, Recommended Practice for Software Requirements Specifications.
9. Tailwind Labs. Tailwind CSS documentation. https://tailwindcss.com/docs

Appendix A: Viva questions on this project

The questions a panel asks about this system, with the answers the design supports. These are the same questions listed on the project page.

How do you prevent two patients booking the same slot?

A unique compound index on doctor id and slot start time. The database rejects the second insert atomically, which a check-then-insert in application code cannot guarantee.

Why enforce roles on the server if the UI already hides pages?

Anyone can call the API directly with a tool like Postman. Middleware that checks the role from the verified JWT is the real protection; hiding pages is only convenience.

How is a prescription linked to the patient?

Through the appointment. The prescription references the appointment, which references both doctor and patient, so a patient's history is every prescription on their completed appointments.

Why MongoDB for hospital data?

The records are naturally document-shaped, such as a prescription with a list of medicines. For heavy cross-record reporting or strict transactions across many tables, a relational database would be a reasonable alternative, and that trade-off is worth stating.

How are passwords stored?

Hashed with bcrypt and a per-password salt. The plain password is never stored, and login compares the hash.

What happens if the server runs in a different time zone?

Nothing changes for the user. Slots are stored in UTC and converted to IST for display, so the stored instant is the same wherever the server runs.