**DocSpot – Online Doctor Appointment Booking System**

**1. Introduction**

**Project Title:** DocSpot – Online Doctor Appointment Booking System

**Project Version:** v1.0.0

**Team Member:**

* Yogeshwaran B

**2. Project Overview**

**Purpose**

DocSpot is a web-based healthcare platform that simplifies doctor-patient interactions by enabling patients to search for doctors, view profiles, schedule appointments, upload medical documents, and manage their health records online. Simultaneously, doctors can manage their schedules, handle appointment requests, upload prescriptions, and communicate with patients. Admins oversee doctor verification, monitor system activity, and manage platform integrity.

**Goals**

* Streamline the appointment booking process for patients
* Enable doctors to efficiently manage their schedules and patient interactions
* Provide secure storage and access to medical documents
* Implement role-based access control for multi-user security
* Deliver real-time updates for appointment status and notifications

**Key Features**

* **User Authentication & Authorization:** JWT-based secure login with bcrypt password hashing
* **Doctor Discovery:** Search, filter, and view doctor profiles by specialization, location, and ratings
* **Appointment Booking:** Schedule, reschedule, and cancel appointments with real-time slot availability
* **Secure Document Management:** Upload and download medical reports, prescriptions, and documents
* **Patient & Doctor Dashboards:** View upcoming/past appointments, manage profiles, track medical history
* **Admin Dashboard:** Approve/reject doctor registrations, monitor platform metrics, and manage system users
* **Real-Time Notifications:** Email and in-app notifications for appointment confirmations and updates
* **Responsive Design:** Mobile-friendly interface using Bootstrap for seamless experience across devices

**3. Architecture**

**3.1 Frontend Architecture ([React.js](http://react.js))**

The frontend is built using React.js with a component-based architecture, providing a responsive and interactive user interface.

**Key Technologies:**

* **React.js:** UI library for building dynamic components
* **Bootstrap & CSS:** Responsive styling and layout management
* **Axios:** HTTP client for API communication with the backend
* **React Context API:** State management for authentication and global data
* **React Router:** Client-side routing for multi-page navigation

**Core Modules:**

* **Authentication Module** – Login, registration, password reset pages
* **Doctor Browsing Module** – Search, filter, and view doctor profiles
* **Appointment Management Module** – Book, reschedule, view appointments
* **Document Upload Module** – Upload and manage medical files
* **User Dashboard Module** – Profile management and appointment history
* **Admin Dashboard Module** – Doctor verification, system monitoring, user management

**3.2 Backend Architecture (Node.js & [Express.js](http://express.js))**

The backend implements RESTful APIs using Node.js and Express.js with a clear separation of concerns.

**Key Technologies:**

* **Node.js:** JavaScript runtime for server-side development
* **Express.js:** Web framework for building scalable APIs
* **JWT (JSON Web Tokens):** Token-based authentication and authorization
* **bcrypt:** Password hashing and encryption for secure credential storage
* **Middleware:** Custom middleware for authentication, validation, and error handling
* **Multer:** File upload handling for medical documents

**Core Modules:**

* **Authentication API** – Register, login, token generation, password reset
* **User Management API** – CRUD operations for user profiles (patients, doctors, admins)
* **Doctor Management API** – Doctor profile creation, updates, availability management
* **Appointment API** – CRUD operations for appointment scheduling and status management
* **Document API** – Secure file upload, storage, and access control
* **Admin API** – Doctor verification, monitoring, and system management

**3.3 Database Architecture (MongoDB)**

MongoDB stores all application data using a document-oriented approach with three primary collections.

**Collections:**

**Users Collection**

Stores patient and admin account information.

{

\_id: ObjectId,

name: String,

email: String (unique),

passwordHash: String (bcrypt-hashed),

role: Enum("user", "admin"),

phone: String,

gender: String,

dateOfBirth: Date,

bloodGroup: String,

address: String,

city: String,

createdAt: Date,

updatedAt: Date

}

**Doctors Collection**

Stores doctor profiles, specialization, and availability.

{

\_id: ObjectId,

userId: ObjectId (reference to Users),

name: String,

email: String (unique),

specialization: String,

experienceYears: Number,

qualifications: String,

fees: Number,

location: String,

city: String,

availableSlots: [

{

day: String,

startTime: String,

endTime: String

}

],

status: Enum("pending", "approved", "rejected"),

documents: {

licenseUrl: String,

uploadedAt: Date

},

ratings: Number,

createdAt: Date,

updatedAt: Date

}

**Appointments Collection**

Tracks appointment records between patients and doctors.

{

\_id: ObjectId,

patientId: ObjectId (reference to Users),

doctorId: ObjectId (reference to Doctors),

appointmentDate: Date,

timeSlot: String,

reason: String,

status: Enum("pending", "confirmed", "completed", "cancelled"),

documents: [

{

documentId: ObjectId,

type: String ("report", "prescription"),

url: String,

uploadedBy: String,

uploadedAt: Date

}

],

notes: String,

createdAt: Date,

updatedAt: Date

}

**Database Interactions:**

* All CRUD operations validate user roles before accessing sensitive data
* Indexes on frequently queried fields (email, doctorId, patientId) improve query performance
* References between collections maintain data consistency and enable relationship queries

**4. Setup Instructions**

**4.1 Prerequisites**

Before setting up DocSpot, ensure you have the following installed:

* **Node.js** (v14 or higher) – Download from <https://nodejs.org/>
* **MongoDB** (v4.4 or higher) – Download from <https://www.mongodb.com/try/download/community> or use MongoDB Atlas for cloud database
* **Git** – For version control
* **npm or yarn** – Package manager (comes with Node.js)

**4.2 Installation Steps**

**Step 1: Clone the Repository**

git clone https://github.com/yourusername/docspot.git

cd docspot

**Step 2: Backend Setup**

# Navigate to backend directory

cd backend

# Install dependencies

npm install

# Create .env file in backend root directory

# Add the following environment variables:

# MONGO\_URI=mongodb+srv://username:password@cluster.mongodb.net/docspot

# JWT\_SECRET=your\_jwt\_secret\_key

# PORT=5000

# SMTP configuration - replace placeholders with your SMTP provider details

SMTP\_HOST=smtp.gmail.com

SMTP\_PORT=587

SMTP\_USER=email

SMTP\_PASS=password 16 characters

# Admin email where notifications are sent

ADMIN\_EMAIL=email

# Frontend URL used in email links

FRONTEND\_URL=http://localhost:5173

npm start

**Step 3: Frontend Setup**

# Navigate to frontend directory (in a new terminal)

cd frontend

# Install dependencies

npm install

# Create .env file in frontend root directory

# Add the following:

VITE\_API\_BASE\_URL=http://localhost:5000/api

# Start the frontend development server

npm start

**Step 4: Verify Installation**

* Backend should run on http://localhost:5000
* Frontend should run on http://localhost:3000
* MongoDB connection should be established (check console for confirmation)

**5. Folder Structure**

**5.1 Client (Frontend) Structure**

frontend/

├── public/

│ ├── index.html

│ └── favicon.ico

├── src/

│ ├── assets/ # Images, icons, media files

│ ├── components/

│ │ ├── landing/

│ │ │ ├── CTASection.jsx

│ │ │ ├── FeatureSection.jsx

│ │ │ ├── Footer.jsx

│ │ │ ├── HeroSection.jsx

│ │ │ └── WhySection.jsx

│ │ ├── AppNavbar.jsx # Global navigation bar

│ │ ├── BookCard.jsx # Appointment/doctor card component

│ │ └── Navbar.css # Navigation styling

│ ├── context/

│ │ └── AuthContext.jsx # Global authentication state

│ ├── pages/

│ │ ├── AdminDashboard.jsx # Admin panel for doctor verification

│ │ ├── AdminDashboard.css

│ │ ├── BookDetails.jsx # Appointment details page

│ │ ├── BookDetails.css

│ │ ├── Cart.jsx # Appointment cart (if applicable)

│ │ ├── Checkout.jsx # Payment/confirmation page

│ │ ├── Home.jsx # Landing/home page

│ │ ├── Landing.jsx # Initial landing

│ │ ├── Login.jsx # Login page

│ │ ├── Orders.jsx # User's appointment history

│ │ ├── Register.jsx # Registration page

│ │ └── [other pages]

│ ├── services/

│ │ └── api.js # Axios instance and API calls

│ ├── App.jsx

│ ├── App.css

│ ├── index.jsx

│ ├── index.css

│ ├── main.jsx # Entry point for Vite

│ └── .gitignore

├── .env # Environment variables

├── package.json

├── package-lock.json

├── vite.config.js # Vite configuration (if using Vite)

└── README.md

**Frontend Component Hierarchy:**

* **Landing Components:** Hero, Features, CTA, Footer sections for marketing
* **Navbar:** Global navigation with conditional rendering based on user role
* **Dashboard Pages:** Separate pages for patient dashboard, doctor profile, and admin controls
* **Auth Pages:** Login and registration forms with validation
* **Document Upload:** File handling for reports and prescriptions

**5.2 Server (Backend) Structure**

backend/

├── config/

│ └── db.js # MongoDB connection configuration

├── controllers/

│ ├── adminController.js # Admin verification and monitoring logic

│ ├── appointmentController.js # Appointment CRUD operations

│ ├── authController.js # Authentication logic (register, login)

│ ├── doctorController.js # Doctor profile management

│ └── userController.js # User profile operations

├── middleware/

│ ├── authMiddleware.js # JWT verification and role checking

│ └── uploadDoctorPhoto.js # File upload handler

├── models/

│ ├── Appointment.js # Appointment schema

│ ├── Doctor.js # Doctor schema

│ └── User.js # User schema

├── routes/

│ ├── adminRoutes.js # Admin endpoints

│ ├── appointmentRoutes.js # Appointment endpoints

│ ├── authRoutes.js # Auth endpoints

│ ├── doctorRoutes.js # Doctor endpoints

│ └── userRoutes.js # User endpoints

├── scripts/ # Database seeding or utility scripts

├── uploads/

│ └── doctors/ # Uploaded doctor documents

├── utils/

│ ├── generateToken.js # JWT token generation

│ └── mailer.js # Email service (notifications)

├── .env # Environment variables

├── .gitignore

├── app.js # Express app setup

├── server.js # Main server entry point

├── package.json

├── package-lock.json

└── README.md

**Backend Organization:**

* **Controllers:** Business logic separated from route handlers
* **Models:** Mongoose schemas defining data structure
* **Routes:** HTTP endpoints organized by feature (auth, appointments, doctors, admin)
* **Middleware:** Authentication checks, validation, and file uploads
* **Utils:** Reusable functions like token generation and email sending
* **Uploads:** Secure storage for user-uploaded medical documents

**6. Running the Application**

**6.1 Development Mode**

**Terminal 1: Backend**

cd backend

npm start

# Expected output: "Server running on port 5000"

# MongoDB connection: "Connected to MongoDB"

**Terminal 2: Frontend**

cd frontend

npm start

# Expected output: "Compiled successfully!"

# Open http://localhost:3000 in your browser

**6.2 Access the Application**

After both servers are running:

* **Frontend:** Open http://localhost:3000
* **API Base URL:** http://localhost:5000
* **Test Accounts:**
  + Patient: patient.test@docspot.com / patient123
  + Doctor: doctor.test@docspot.com / doctor123
  + Admin: admin@docspot.com / admin123

**6.3 Building for Production**

**Frontend Build**

cd frontend

npm run build

# Creates optimized build in 'dist' folder

**Backend Deployment**

* Set environment variables in production environment
* Use process managers like PM2 for Node.js
* Deploy MongoDB to MongoDB Atlas or managed service

**7. API Documentation**

**7.1 Authentication Endpoints**

**1. Authentication Endpoints**

**Register User**

POST /api/auth/register  
Content-Type: application/json

Request Body:

{

"name": "John Doe",

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

"password": "securePassword123",

"role": "user"

}

Response (201 Created):

{

"success": true,

"message": "User registered successfully",

"user": {

"\_id": "userId",

"name": "John Doe",

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

"role": "user"

},

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

}

**Login**

POST /api/auth/login  
Content-Type: application/json

Request Body:

{

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

"password": "securePassword123"

}

Response (200 OK):

{

"success": true,

"message": "Login successful",

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",

"user": {

"\_id": "userId",

"name": "John Doe",

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

"role": "user"

}

}  
  
**2. Doctor Endpoints**

Get All Doctors (with filters)

GET /api/doctors?specialization=Cardiology&city=Mumbai&sort=rating  
Authorization: Bearer token

Response (200 OK):

{

"success": true,

"count": 5,

"doctors": [

{

"\_id": "doctorId",

"name": "Dr. Rajesh Kumar",

"specialization": "Cardiology",

"fees": 500,

"location": "Heart Care Hospital",

"city": "Mumbai",

"ratings": 4.8,

"status": "approved"

}

]

}

**Get Doctor Profile**

GET /api/doctors/:doctorId  
Authorization: Bearer token

Response (200 OK):

{

"success": true,

"doctor": {

"\_id": "doctorId",

"name": "Dr. Rajesh Kumar",

"specialization": "Cardiology",

"qualifications": "MD, DM Cardiology",

"experienceYears": 12,

"fees": 500,

"location": "Heart Care Hospital",

"city": "Mumbai",

"availableSlots": [

{

"day": "Monday",

"startTime": "09:00",

"endTime": "17:00"

}

],

"ratings": 4.8

}

}

**Get User Appointments**

GET /api/appointments/user/:userId  
Authorization: Bearer token

Response (200 OK):

{

"success": true,

"appointments": [

{

"\_id": "appointmentId",

"doctorId": {

"\_id": "doctorId",

"name": "Dr. Rajesh Kumar",

"specialization": "Cardiology"

},

"appointmentDate": "2025-12-15",

"timeSlot": "10:00-10:30",

"status": "confirmed"

}

]

}

**Cancel Appointment**

PUT /api/appointments/:appointmentId/cancel  
Authorization: Bearer token

Response (200 OK):

{

"success": true,

"message": "Appointment cancelled successfully",

"appointment": {

"\_id": "appointmentId",

"status": "cancelled"

}

}

**Document Endpoints**

**Upload Document**

POST /api/appointments/:appointmentId/upload  
Authorization: Bearer token  
Content-Type: multipart/form-data

Request Body:

file: [binary file]

documentType: "report" | "prescription"

Response (201 Created):

{

"success": true,

"message": "Document uploaded successfully",

"document": {

"documentId": "docId",

"type": "report",

"url": "/uploads/documents/file\_1733558400000.pdf",

"uploadedAt": "2025-12-06T10:00:00Z"

}

}

**5. Admin Endpoints**

Get Pending Doctor Approvals

GET /api/admin/doctors/pending  
Authorization: Bearer admin\_token

Response (200 OK):

{

"success": true,

"pendingDoctors": [

{

"\_id": "doctorId",

"name": "Dr. Priya Sharma",

"specialization": "Gynecology",

"status": "pending",

"documents": {

"licenseUrl": "/uploads/doctors/license\_1733558400000.pdf"

}

}

]

}

**Approve Doctor**

PUT /api/admin/doctors/:doctorId/approve  
Authorization: Bearer admin\_token  
Content-Type: application/json

Request Body:

{

"status": "approved"

}

Response (200 OK):

{

"success": true,

"message": "Doctor approved successfully",

"doctor": {

"\_id": "doctorId",

"status": "approved"

}

}

**8. Authentication & Authorization**

**8.1 JWT-Based Authentication**

DocSpot uses JSON Web Tokens (JWT) for stateless authentication.

**Token Generation Flow:**

* User registers or logs in with email and password
* Backend verifies credentials and hashes password using bcrypt
* On successful verification, a JWT token is generated containing user ID and role
* Token is returned to frontend and stored in browser's localStorage or cookie
* Client includes token in Authorization header for subsequent requests

**Token Structure:**

Header: { "alg": "HS256", "typ": "JWT" }

Payload: {

"userId": "user\_id",

"role": "user|doctor|admin",

"iat": 1733558400,

"exp": 1733644800

}

Signature: HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), SECRET\_KEY)

**Token Expiration:** 24 hours (configurable via JWT\_EXPIRE in .env)

8.2 Password Security

Passwords are hashed using bcrypt with a salt round of 10:

// Hashing during registration

const saltedPassword = await bcrypt.hash(password, 10);

// Verification during login

const isPasswordValid = await bcrypt.compare(inputPassword, saltedPassword);

**8.3 Role-Based Access Control (RBAC)**

Three roles with specific permissions:

**Patient (user):**

* Search and view doctor profiles
* Book, reschedule, and cancel own appointments
* Upload personal medical reports
* View own dashboard and medical history

**Doctor:**

* View and manage own profile and availability
* View appointments assigned to them
* Upload prescriptions and notes
* Update appointment status

**Admin:**

* Approve/reject doctor registrations
* View all users, doctors, and appointments
* Monitor system activity and generate reports
* Deactivate suspicious accounts

**8.4 Middleware Protection**

Protected routes are wrapped with middleware that:

* Verifies JWT token in Authorization header
* Decodes token to extract user ID and role
* Checks if user role has permission for the endpoint
* Rejects request with 401 (Unauthorized) or 403 (Forbidden) if validation fails

**Example Middleware:**

const authMiddleware = (req, res, next) => {

const token = req.headers.authorization?.split(' ')[1];

if (!token) return res.status(401).json({ error: "No token provided" });

try {

const decoded = jwt.verify(token, process.env.JWT\_SECRET);

req.userId = decoded.userId;

req.userRole = decoded.role;

next();

} catch (err) {

res.status(401).json({ error: "Invalid token" });

}

};

const adminOnly = (req, res, next) => {

if (req.userRole !== 'admin') {

return res.status(403).json({ error: "Admin access required" });

}

next();

};

**9. User Interface**

**9.1 Key Pages & Features**

**1. Landing Page**

* Hero section with CTA buttons ("Book Appointment", "Register as Doctor")
* Features overview highlighting key benefits
* Testimonials from users
* Call-to-action sections for registration

**2. Doctor Search & Browsing**

* Search bar with filters (specialization, location, availability, rating)
* Doctor cards displaying name, photo, specialization, fees, and ratings
* Sorting options (by rating, fees, nearest)
* Doctor detail page with full profile, availability slots, and reviews

**3. Appointment Booking**

* Doctor selection and date/time slot picker
* Appointment confirmation with details
* Booking confirmation page with appointment ID
* Real-time slot availability updates

**4. Patient Dashboard**

* Upcoming appointments list with status
* Past appointments with option to leave reviews
* Medical records and uploaded documents
* Profile management section

**5. Doctor Dashboard**

* Daily/weekly schedule view
* Appointment list with patient details
* Option to accept, reject, or mark appointments as completed
* Profile and availability management

**6. Admin Dashboard**

* Pending doctor approvals with verification documents
* System statistics (total users, doctors, appointments)
* User and doctor management with deactivation options
* Search and filter for monitoring

**7. Authentication Pages**

* Responsive login form with email and password fields
* Registration form with role selection (patient/doctor)
* Password reset flow via email verification
* Form validation with clear error messages

**9.2 Responsive Design**

DocSpot uses Bootstrap and CSS media queries for responsive layouts:

* **Desktop:** Full navigation, multi-column layouts
* **Tablet:** Optimized grid layouts, collapsible menus
* **Mobile:** Stack layouts, hamburger navigation, touch-friendly buttons

**10. Testing**

**10.1 Testing Strategy**

DocSpot employs comprehensive testing across multiple layers:

**Unit Testing**

* Test individual components, utility functions, and services
* Focus on authentication, validation, and business logic
* Tools: Jest, React Testing Library (frontend)

**Integration Testing**

* Test API endpoints with various input scenarios
* Verify correct database interactions
* Test authentication flow and role-based access
* Tools: Postman, Mocha (backend)

**End-to-End Testing**

* Test complete user workflows (registration → booking → confirmation)
* Test across browsers and devices
* Verify real-time notifications and updates
* Tools: Cypress, Selenium

**Performance Testing**

* Monitor page load times and API response times
* Test with simulated high traffic scenarios
* Tools: Apache JMeter, Lighthouse

**10.2 Test Coverage**

**Functional Testing (Features Tested):**

* User registration with email validation
* Login with correct/incorrect credentials
* JWT token generation and validation
* Doctor search and filtering
* Appointment booking with slot availability checks
* Document upload validation (file type, size)
* Admin doctor approval workflow
* Real-time appointment status updates
* Password reset via email
* Profile updates for patients and doctors

**Security Testing:**

* Test unauthorized access to admin endpoints
* Verify JWT expiration and refresh
* Test SQL injection prevention in queries
* Verify sensitive data encryption in transit (HTTPS)

**Validation Testing:**

* Empty field handling
* Invalid email format rejection
* Password strength requirements
* File upload validation (extensions, size limits)
* Date/time validation for appointments

**10.3 Running Tests**

# Frontend tests

cd frontend

npm test # Run unit tests

npm run test:coverage # Generate coverage report

# Backend tests

cd backend

npm test # Run all tests

npm run test:auth # Test authentication

npm run test:api # Test API endpoints

**11. Screenshots / Demo**

Demo LInk :

<https://drive.google.com/file/d/1LCDjJWaCJJZS_wX9idC0YlOUYHwj9Lwk/view?usp=sharing>

**12. Known Issues**

Current Limitations

* **Email Notifications**
  + Email service requires proper SMTP configuration
  + Gmail requires app-specific password, not regular password
  + **Workaround:** Use SendGrid or Mailgun for reliable email delivery
* **File Storage**
  + Currently stores files in local server directory
  + Not suitable for production with multiple servers
  + **Recommendation:** Migrate to AWS S3 or MongoDB GridFS for scalability
* **Real-Time Updates**
  + Uses polling instead of WebSockets for real-time data
  + Higher server load with many concurrent users
  + **Future:** Implement Socket.io for true real-time communication
* **Doctor Availability**
  + Availability slots are static (not auto-generated based on busy time)
  + Manual time slot management can be error-prone
  + **Future:** Implement calendar integration with automatic conflict detection
* **Payment Integration**
  + Not yet integrated with payment gateway (Stripe, Razorpay)
  + Appointments can be booked without payment confirmation
  + **Future:** Add payment processing before appointment confirmation
* **Document Security**
  + Uploaded documents accessible via direct URL
  + No encryption for stored files
  + **Improvement:** Implement access tokens and document encryption

**Reported Bugs**

| **Bug ID** | **Description** | **Status** | **Severity** |
| --- | --- | --- | --- |
| BG-001 | Appointment slot shows as available after booking | Open | High |
| BG-002 | Doctor cannot update availability slots | In Progress | Medium |
| BG-003 | Email notifications not sending for some users | Open | Medium |

**13. Future Enhancements**

Phase 2 Features

1. **Payment Integration**

* Integrate Razorpay or Stripe for secure payment processing
* Payment status tracking in appointments
* Invoice generation and download

1. **Real-Time Communication**

* Implement Socket.io for live chat between patient and doctor
* Video consultation feature using WebRTC
* Screen sharing for remote consultations

1. **Advanced Scheduling**

* Integration with Google Calendar and Outlook
* Automatic conflict detection
* Smart scheduling suggestions based on patient history

1. **Analytics & Reporting**

* Patient appointment history and trends
* Doctor performance metrics (completion rate, ratings)
* System-wide analytics dashboard for admins
* Revenue tracking for payment transactions

1. **Mobile Application**

* Native iOS and Android apps using React Native
* Push notifications for appointment reminders
* Offline appointment viewing

1. **AI-Powered Features**

* Doctor recommendation engine based on symptoms
* Appointment slot optimization
* Chatbot for FAQ support and appointment guidance

1. **Enhanced Security**

* Two-factor authentication (2FA)
* Role-based encryption for sensitive documents
* HIPAA compliance for healthcare data
* Audit logging for all system activities

1. **Telemedicine**

* Prescription generation and digital signature
* Integration with pharmacy for prescription delivery
* Lab test integration and result sharing

1. **Scalability Improvements**

* Caching layer (Redis) for frequently accessed data
* Database sharding for large appointment volumes
* CDN integration for static assets
* Microservices architecture for independent scaling

1. **User Experience**

* Dark mode support
* Multi-language support (localization)
* Accessibility improvements (WCAG compliance)
* Progressive Web App (PWA) capabilities