

DOCTOR BOOKING USING MERN

INTRODUCTION

Introducing Doctor Booker, the cutting-edge digital platform poised to revolutionize the way you book a doctor using online. With doctor booker, your experience will reach unparalleled levels of convenience and efficiency.

In a **Doctor Booker** application, one of the key features is the ability to display and manage different types of doctors with varying specialties. This feature allows patients to easily find and book appointments with the right medical professionals based on their needs.

A **Doctor Booker** app is designed to facilitate the process of booking appointments with healthcare professionals. The app allows users to choose from a wide range of **doctors with different specialties**, making it easier for patients to find the right care based on their medical requirements.

Doctors in the system can be categorized by their **specializations**, which might include:

- **Psychiatrists,**
- **Dermatologists,**
- **Genral surgeon,**
- **Cardiologist,**
- **Endocrinologist,**
- **Physiotheraphy** and so on.

Each **doctor profile** will include essential details such as:

1. **Name:** The doctor's full name.
2. **Specialization:** What medical field the doctor specializes in.
3. **Availability:** The doctor's schedule of available appointment slots.
4. **Experience and Qualifications:** The doctor's professional background, certifications, and years of practice.
5. **Ratings & Reviews:** Patient feedback that helps new patients make informed decisions.
6. **Consultation Fees:** The cost for booking an appointment with the doctor.
7. **Profile Picture:** A photo of the doctor (optional, but helps personalize the experience

The core goal of **Doctor Booker** is to create a seamless experience for patients to **find, select, and book appointments** with doctors that match their needs. To achieve this, the application needs a well-organized system to display these doctors based on various filters, including specialty, location, ratings, and availability.

The **Doctor Booker** application allows patients to easily navigate through a list of doctors with various specialties and find the right healthcare provider for their needs. By categorizing doctors based on specialization, location, and patient reviews, you can build an intuitive platform that helps patients make informed decisions and efficiently book appointments with their preferred doctors. Through this system, doctors can also manage their schedules and patient appointments, while admins can oversee and maintain the integrity of the system.

SCENARIO:

Let's walk through a practical scenario for a **Doctor Booking Application** built with the **MERN** stack (MongoDB, Express, React, Node.js). This scenario will cover the flow of a patient booking an appointment, the interactions between different components of the system, and the technologies used to support it.

Scenario Overview:

A **patient** wants to book an appointment with a **dermatologist**. The system should allow the patient to:

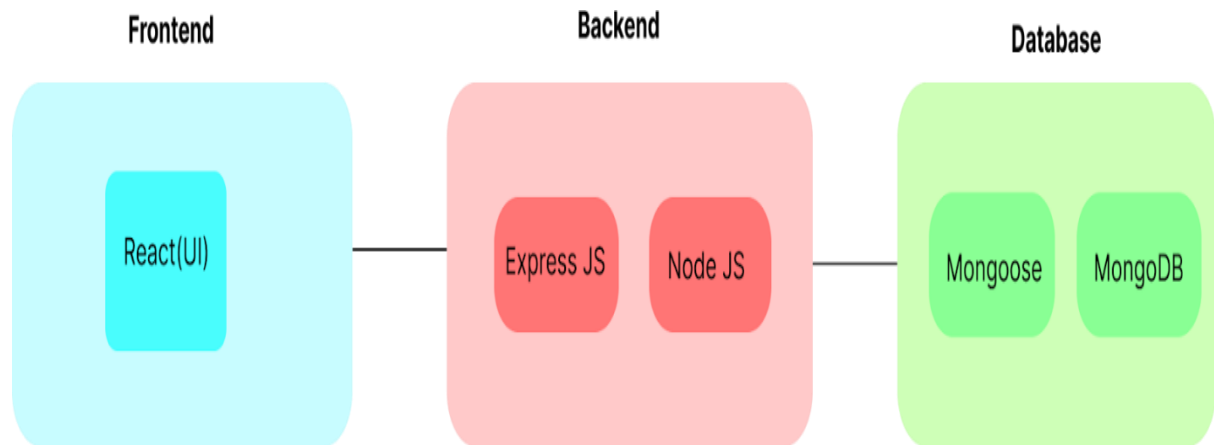
1. **Search for doctors** based on their specialization and availability.
2. **View doctor profiles** with their qualifications, ratings, and available time slots.
3. **Book an appointment** with the chosen doctor at a time that fits their schedule.
4. **Receive notifications** about their appointment status and any changes.

The **doctor** should also be able to:

- **Manage their profile:** Include specialization, availability, and consultation fees.
- **Set their availability:** Choose specific dates and times when they are free to accept appointments.
- **View appointments:** Keep track of the upcoming patient consultations.

The **admin** will manage the entire system, including verifying doctor profiles, handling patient complaints, and ensuring the platform's smooth operation.

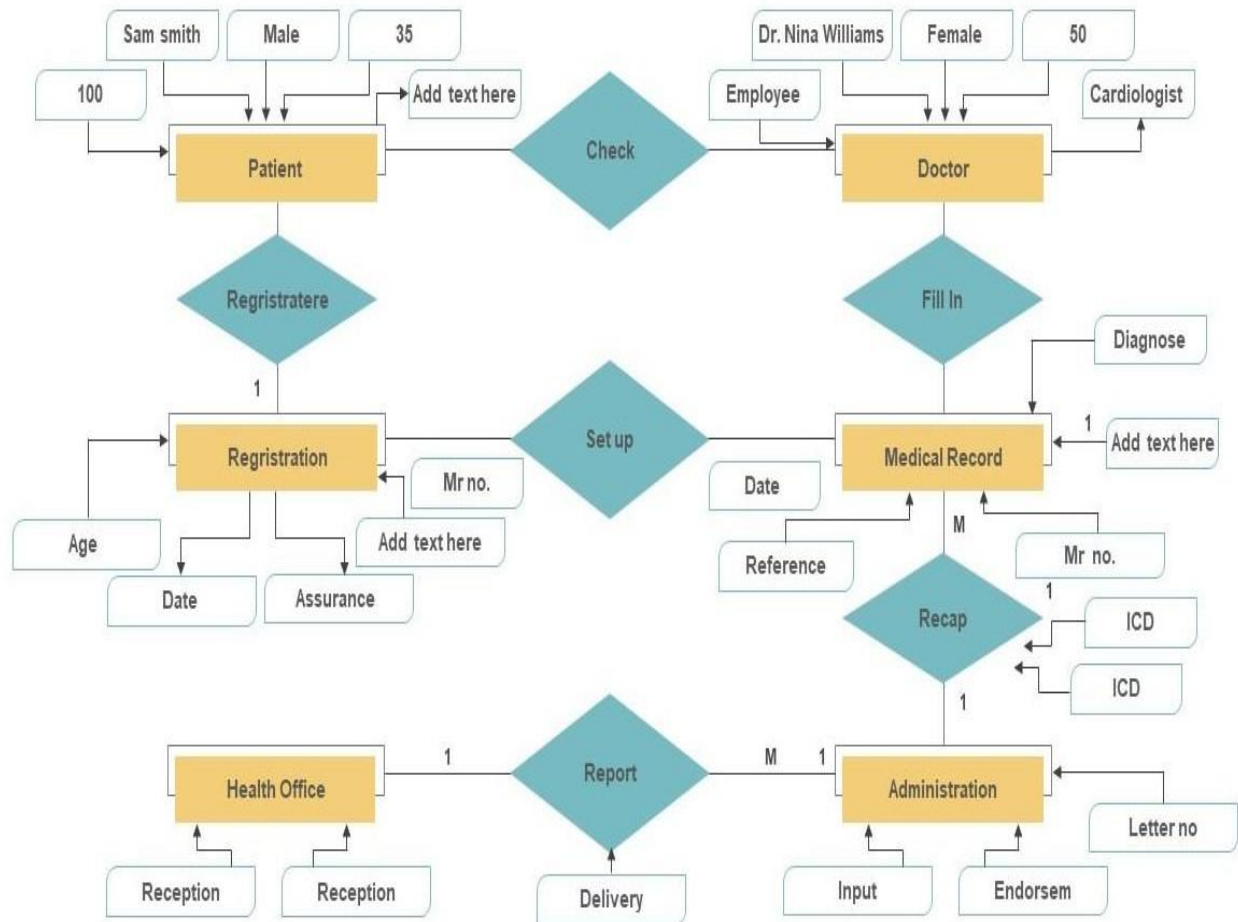
TECHNICAL ARCHITECTURE:



In this architecture diagram:

- The frontend is represented by the "Frontend" section, including user interface components such as User Authentication, application, Profile, Admin dashboard, etc.,
- The backend is represented by the "Backend" section, consisting of API endpoints for Users, application, conformation, etc., It also includes Admin Authentication and an Admin Dashboard.
- The Database section represents the database that stores collections for Users, Admin, application, and conformation

ER DIAGRAM:



Creating an **Entity Relationship Diagram (ERD)** for a **Doctor Booker** application built using the **MERN stack** helps to visualize the relationships between different entities (such as doctors, patients, appointments, etc.) in the system.

In this scenario, the key entities include:

1. **Doctor**: Represents a medical professional who offers consultations (either in-person or telemedicine).
2. **Patient**: Represents a user who books appointments with doctors.
3. **Appointment**: Represents an appointment booked by a patient with a doctor.
4. **Review**: Represents reviews left by patients for doctors after consultations.
5. **Admin**: Manages the overall system, including doctor and patient profiles, appointments, etc.
6. **Payment**: Represents the payment for appointments (could be linked to a third-party payment system like Stripe or PayPal).

FEATURES:

1. User Authentication and Authorization

- **User Sign-up and Login:** Users (patients) can sign up or log in to the platform. This can be achieved using **JWT (JSON Web Token)** or **Passport.js** for session management.
- **Role-based Access:** Different roles for users (patients) and doctors (e.g., admin, doctor, patient) to manage access control.

2. Doctor Profiles

- **Doctor Listing:** Users can view a list of available doctors, their specializations, experience, and ratings.
- **Doctor Profiles:** Each doctor has a profile with detailed information, including availability, consultation fees, specialization, bio, and patient reviews.
- **Search and Filters:** Ability to search for doctors by name, specialization, availability.

3. Appointment Booking System

- **Available Time Slots:** Doctors can set their availability (time slots) through a simple UI, and patients can view available slots for each doctor.
- **Booking an Appointment:** Patients can select a time slot and book an appointment with a doctor.
- **Booking Confirmation:** After booking, the patient receives a confirmation email/SMS with appointment details. The doctor also receives a notification of the new appointment.
- **Appointment Reminders:** Automated reminders via email/SMS/Push Notifications for upcoming appointments.

4. Admin Panel

- **Manage Doctors:** Admins can manage doctor profiles, including adding/removing doctors, updating availability, and verifying credentials.
- **Manage Appointments:** Admin can view all upcoming appointments, manage conflicts, and reassign doctors if needed.
- **Analytics:** Admin can view statistical data, such as the number of appointments per day, doctor ratings, and patient feedback.

5. Patient Dashboard

- **View Appointments:** Patients can view upcoming and past appointments, including status (confirmed, completed, or canceled).
- **Cancel/Reschedule Appointments:** If allowed, patients can cancel or reschedule appointments directly through their dashboard.
- **Health Records:** Optionally, patients can upload or access their medical records if integrated with a health system.

6. Doctor Dashboard

- **Manage Availability:** Doctors can set their availability, modify time slots, or mark themselves as unavailable.
- **View Appointments:** Doctors can see their upcoming appointments, along with patient details (name, contact info, and reason for the visit).
- **Patient History:** If implemented, doctors can access the patient's past appointment history and medical notes (via a secure system).

7. Notifications and Alerts

- **Real-time Notifications:** Use WebSockets (via **Socket.io**) or push notifications to send updates about appointment status, cancellations, or reminders.
- **Email and SMS Notifications:** Integration with email and SMS services (like **SendGrid** or **Twilio**) to send confirmations, reminders, and alerts.

8. Payment Integration

- **Online Payment:** Patients can pay consultation fees online through integrated payment gateways like **Stripe**, **Razorpay**, or **PayPal**.
- **Invoice Generation:** After a successful payment, patients can receive an invoice for their consultation.

9. Ratings and Reviews

- **Patient Reviews:** After an appointment, patients can rate their doctor and leave feedback (e.g., stars, comments).
- **Doctor Rating:** Doctors can be rated based on various factors like professionalism, punctuality, and communication.

11. Search and Filter Options

- **Search Doctors:** Allow patients to search doctors based on specialization, location, language, or ratings.
- **Filter by Availability:** Allow patients to filter available doctors by specific time slots or dates.

12. Data Security and Privacy

- **HIPAA Compliance:** For handling sensitive medical data, ensure encryption for medical records and patient data (using **bcrypt** for password hashing).
- **Secure Payments:** Use SSL encryption for payment transactions and personal data storage.

13. Mobile-Friendly Design

- **Responsive UI:** The app should be designed to work seamlessly across desktop and mobile devices.
- **Mobile App (Optional):** You could also create a mobile version using **React Native** for iOS/Android, allowing patients and doctors to book/manage appointments on the go.

14. Calendar Integration

- **Sync with Google Calendar:** Patients and doctors can sync their appointments with Google Calendar or other calendar applications to track appointments more efficiently.

PREREQUISITES:

To develop a full-stack for doctor booking using React JS, Node.js, and MongoDB, there are several prerequisites you should consider. Here are the key prerequisites for developing such an application:

Node.js and npm: Install Node.js, which includes npm (Node Package Manager), on your development machine. Node.js is required to run JavaScript on the server side.

- Download: <https://nodejs.org/en/download/>

Installation instructions: <https://nodejs.org/en/download/package-manager/>

MongoDB: Set up a MongoDB database to store appointment and booking information. Install MongoDB locally or use a cloud-based MongoDB service.

- Download: <https://www.mongodb.com/try/download/community>
- Installation instructions: <https://docs.mongodb.com/manual/installation/>

Express.js: Express.js is a web application framework for Node.js. Install Express.js to handle server-side routing, middleware, and API development.

- Installation: Open your command prompt or terminal and run the following command: **npm install express**

React.js: React.js is a popular JavaScript library for building user interfaces. It enables developers to create interactive and reusable UI components, making it easier to build dynamic and responsive web applications. To install React.js, a JavaScript library for building user interfaces, follow the installation guide:

<https://reactjs.org/docs/create-a-new-react-app.html>

HTML, CSS, and JavaScript: Basic knowledge of HTML for creating the structure of your app, CSS for styling, and JavaScript for client-side interactivity is essential.

Database Connectivity: Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose to connect your Node.js server with the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations.

Front-end Framework: Utilize Angular to build the user-facing part of the application, including product listings, booking forms, and user interfaces for the admin dashboard.

Version Control: Use Git for version control, enabling collaboration and tracking changes throughout the development process. Platforms like GitHub or Bitbucket can host your repository.

- Git: Download and installation instructions can be found at: <https://git-scm.com/downloads>

Development Environment: Choose a code editor or Integrated Development Environment (IDE) that suits your preferences, such as Visual Studio Code, Sublime Text, or WebStorm.

- Visual Studio Code: Download from <https://code.visualstudio.com/download>

- Sublime Text: Download from <https://www.sublimetext.com/download>
- WebStorm: Download from <https://www.jetbrains.com/webstorm/download>

Install Dependencies:

- Navigate into the repository directory:
cd client-App-MERN
- Install the required dependencies by running the following command:
npm install

Start the Development Server:

- To start the development server, execute the following command:
npm run dev or npm run start
- The e-commerce app will be accessible at <http://localhost:3000> by default. You can change the port configuration in the .env file if needed.

Access the App:

- Open your web browser and navigate to <http://localhost:3000>.
 - You should see the flight booking app's homepage, indicating that the installation and setup were successful.
- You have successfully installed and set up the doctor booking using mern app on your local machine. You can now proceed with further customization, development, and testing as needed.

USER & ADMIN FLOW:

1. User Flow:

- Users start by registering for an account.
- After registration, they can log in with their credentials.
- Once logged in, they can check for the available products in the platform.
- Users can add the products they wish to their book and confirm doctor.

- They can then proceed by entering address and payment details.
- After ordering, they can check them in the profile section.

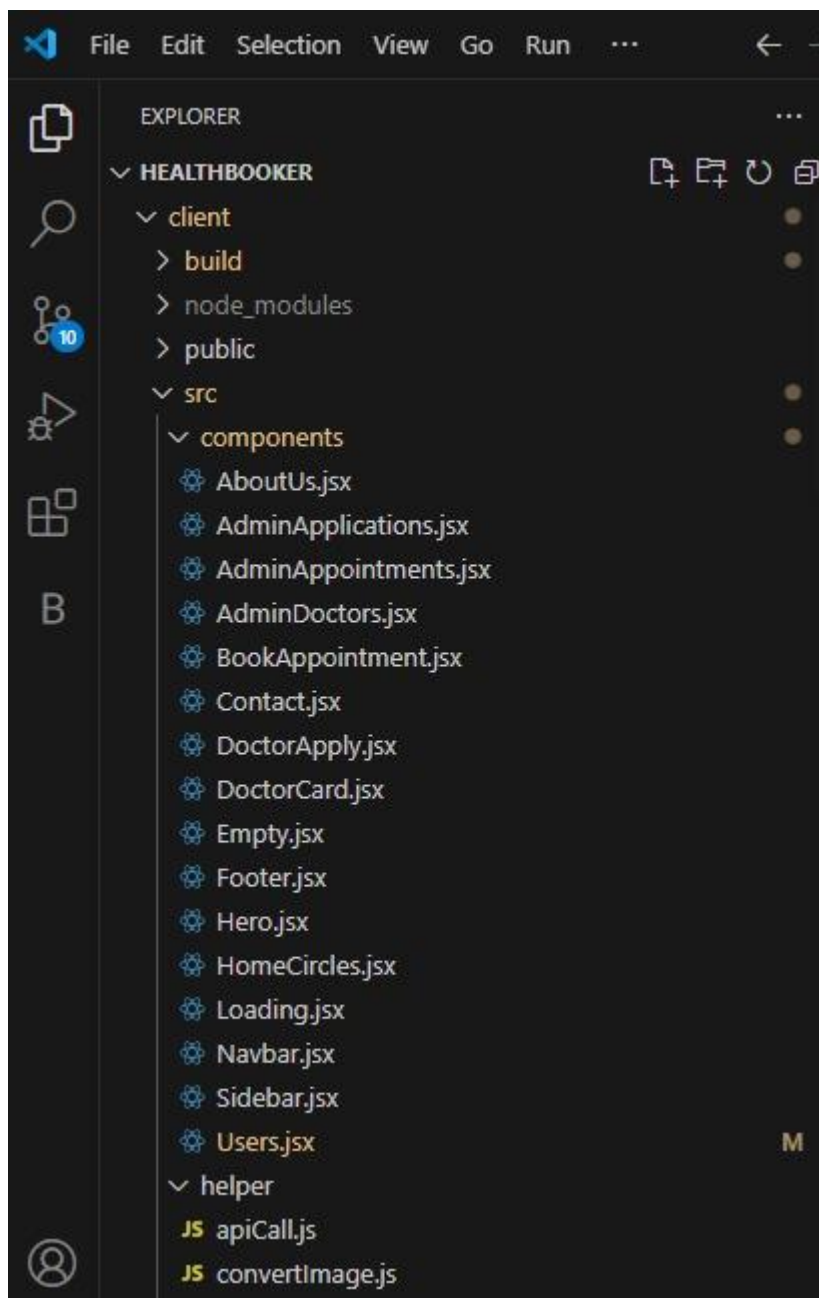
2. Restaurant Flow:

- Restaurants start by authenticating with their credentials.
- They need to get approval from the admin to start listing the products.
- They can add/edit the food items.

3. Admin Flow:

- Admins start by logging in with their credentials.
- Once logged in, they are directed to the Admin Dashboard.
- Admins can access the users list, products, orders, etc.

PROJECT STRUCTURE



This structure assumes a React app and follows a modular approach. Here's a brief explanation of the main directories and files:

- src/components: Contains components related to the application such as, register, login, home, etc.,
- src/pages has the files for all the pages in the application.

PROJECT SETUP AND CONFIGURATION:

Install required tools and software:

- Node.js.

Reference Article: <https://www.geeksforgeeks.org/installation-of-node-js-on-windows/>

- Git.

Reference Article: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

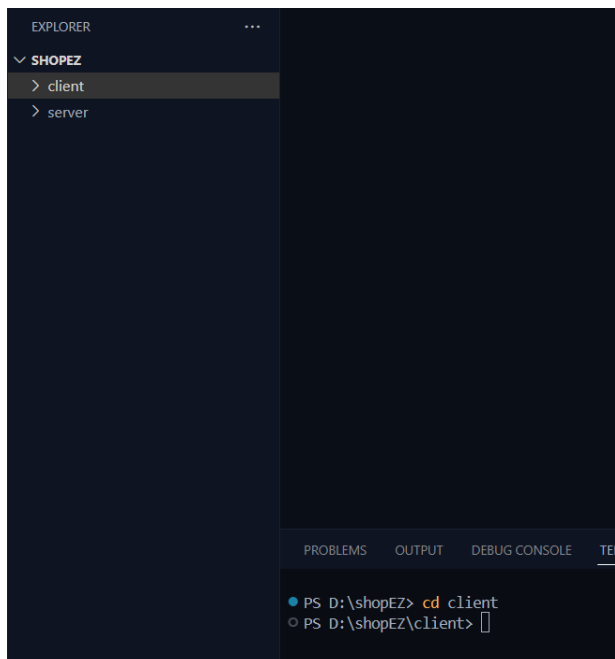
Create project folders and files:

- Client folders.
- Server folders

Referral Video Link:

https://drive.google.com/file/d/1uSMbPIAR6rfAEMcb_nLZAZd5QIjTpnYQ/view?usp=sharing

Referral Image:



DATABASE DEVELOPMENT:

Create database in cloud video link:-

<https://drive.google.com/file/d/1CQil5KzGnPvkVOPWTLP0h-Bu2bXhq7A3/view>

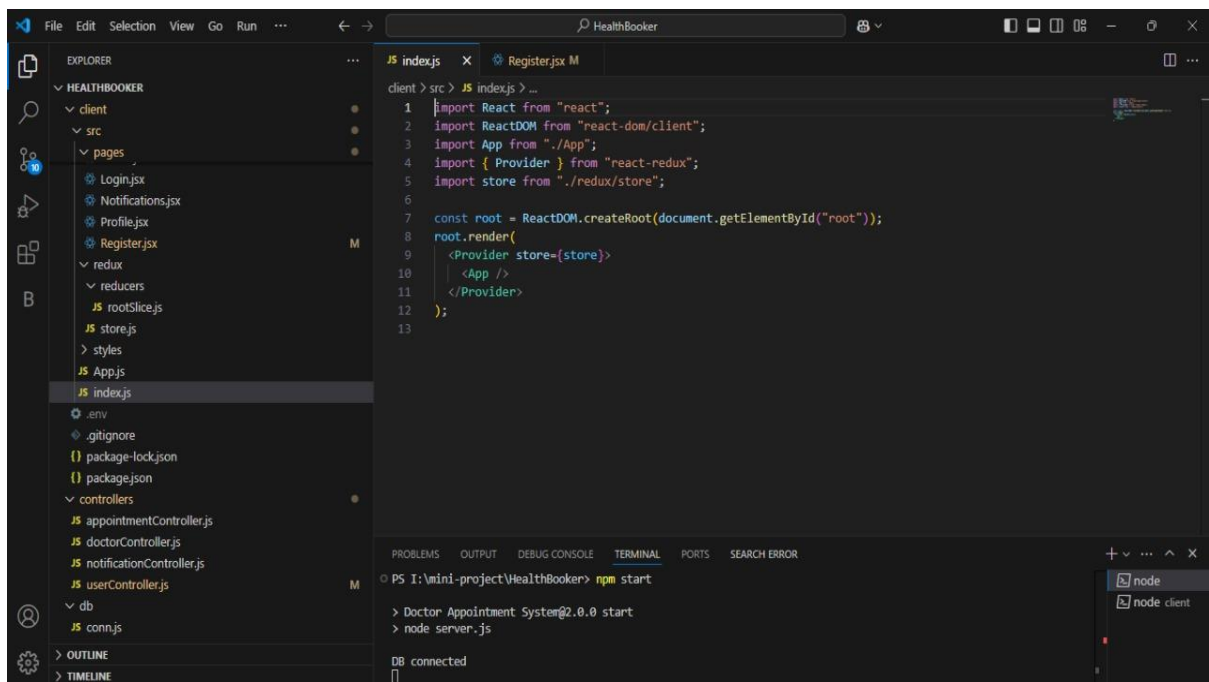
- Install Mongoose.
- Create database connection.

Reference Video of connect node with mongoDB database:

https://drive.google.com/file/d/1cTS3_EOAAvDctkibG5zVikrTdmoyY2Ag/view?usp=sharing

Reference Article: <https://www.mongodb.com/docs/atlas/tutorial/connect-to-your-cluster/>

Reference Image:



Schema use-case:

1. User Schema (Patient)

The User schema represents the patients who book appointments with doctors. Patients need to have basic personal information, authentication details, and a history of their bookings.

Use Case:

- **Registration/Login:** Users sign up/login using their email and password.
- **Appointments:** appointments field references the appointments the user has made.

2. Doctor Schema

The Doctor schema represents the doctors available for booking. It includes their professional information, availability, and ratings.

Use Case:

- **Profile Management:** Doctors manage their profiles, including specialties, bio, fee, and availability.
- **Availability Management:** Doctors set their weekly availability with time slots for appointments.
- **Appointments:** appointments field references all the appointments for that doctor.

3. Appointment Schema

The Appointment schema represents the booking of an appointment between a patient and a doctor. It includes information such as the date, time, status, and payment details.

Use Case:

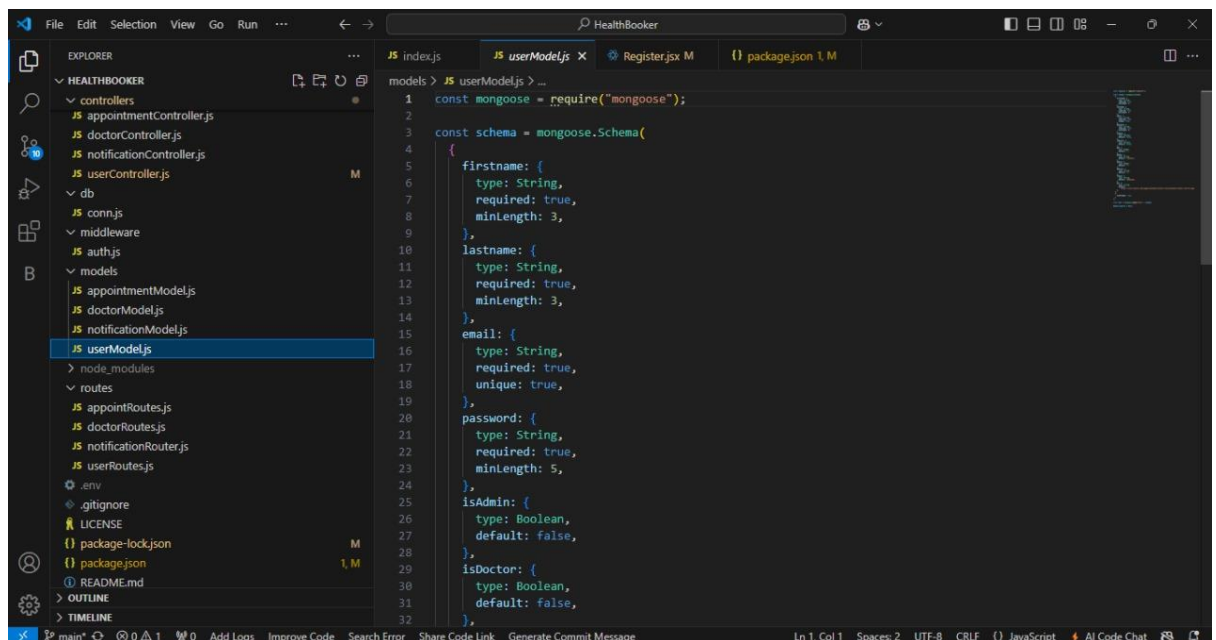
- **Appointment Booking:** The patient books an appointment, and the doctor confirms it.
- **Appointment Status:** The status of the appointment changes as the appointment progresses (from pending to confirmed, completed, or canceled).
- **Payment Status:** Handles the payment status (unpaid or paid) and the associated consultation fee.

4. Review Schema

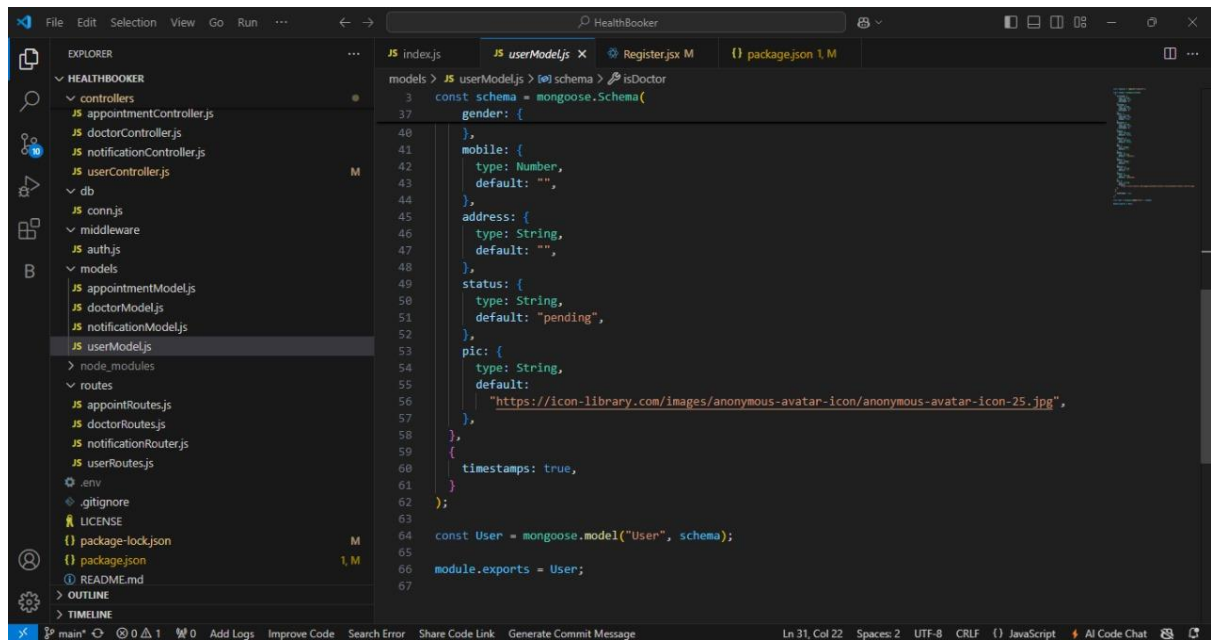
The Review schema is used for patients to rate and provide feedback about the doctor after the appointment

Use Case:

- **Ratings and Reviews:** After completing an appointment, patients can rate the doctor (1–5 stars) and leave feedback.
- **Doctor Ratings:** Average ratings are stored on the doctor profile, influencing their reputation.
- **Reference image:**



```
1 const mongoose = require("mongoose");
2
3 const schema = mongoose.Schema(
4   {
5     firstname: {
6       type: String,
7       required: true,
8       minLength: 3,
9     },
10    lastname: {
11      type: String,
12      required: true,
13      minLength: 3,
14    },
15    email: {
16      type: String,
17      required: true,
18      unique: true,
19    },
20    password: {
21      type: String,
22      required: true,
23      minLength: 5,
24    },
25    isAdmin: {
26      type: Boolean,
27      default: false,
28    },
29    isDoctor: {
30      type: Boolean,
31      default: false,
32    },
33  }
34 );
```



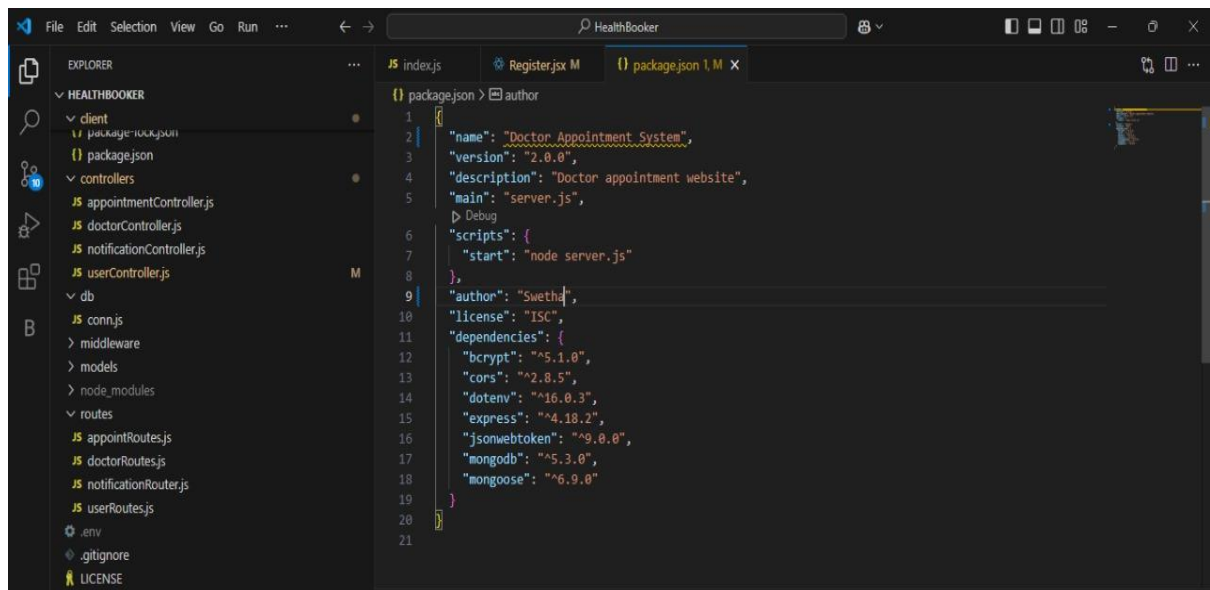
BACKEND DEVELOPMENT:

Set Up Project Structure:

- Create a new directory for your project and set up a package.json file using the npm init command.
- Install necessary dependencies such as Express.js, Mongoose, and other required packages.

Reference Video: <https://drive.google.com/file/d/19df7NU-gQK3D06wr7ooAfJYIQwnemZoF/view?usp=sharing>

Reference Image:

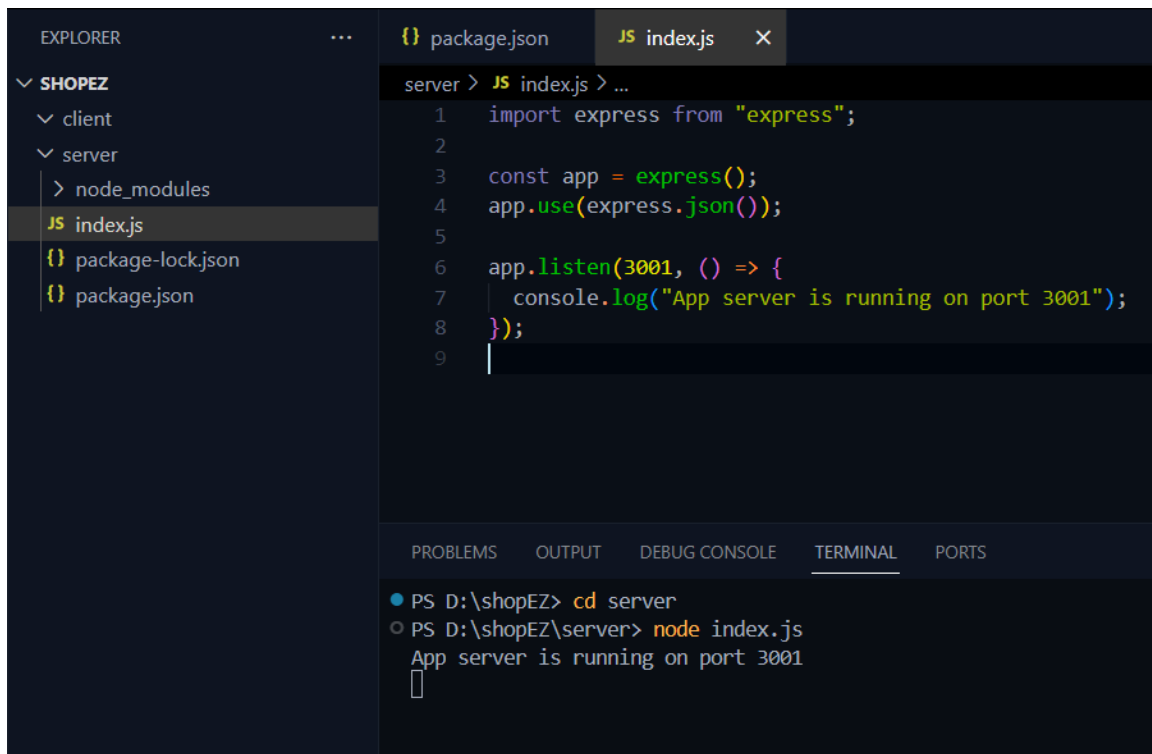


1. Setup express server:

- Create index.js file.
- Create an express server on your desired port number.
- Define API's

Reference Video: https://drive.google.com/file/d/1-uKMIcrok_ROHyZl2vRORggrYRio2qXS/view?usp=sharing

Reference Image:



```
server > JS index.js > ...
1  import express from "express";
2
3  const app = express();
4  app.use(express.json());
5
6  app.listen(3001, () => {
7    console.log("App server is running on port 3001");
8  });
9  |
```

```
PS D:\shopEZ> cd server
PS D:\shopEZ\server> node index.js
App server is running on port 3001
```

2. Database Configuration:

- Set up a MongoDB database either locally or using a cloud-based MongoDB service like MongoDB Atlas or use locally with MongoDB compass.
- Create a database and define the necessary collections for admin, users, doctor profile, conformation

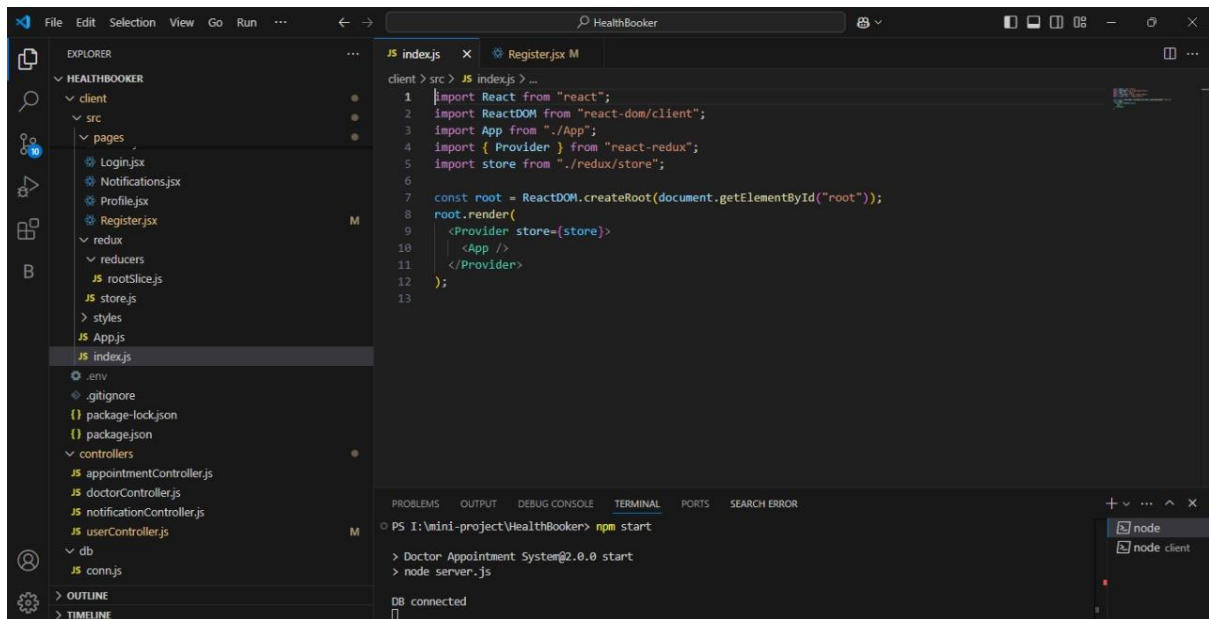
Reference Video of connect node with mongoDB database:

https://drive.google.com/file/d/1cTS3_EOAAvDctkibG5zVikrTdmoy2Ag/view?usp=sharing

Reference Article:

<https://www.mongodb.com/docs/atlas/tutorial/connect-to-your-cluster/>

Reference Image:



3. Create Express.js Server:

- Set up an Express.js server to handle HTTP requests and serve API endpoints.
- Configure middleware such as body-parser for parsing request bodies and cors for handling cross-origin requests.

Reference Video: https://drive.google.com/file/d/1-uKMIcrok_ROHyZl2vRORggrYRio2qXS/view?usp=sharing

Reference Image:

```
server > JS index.js > ...
1  import express from "express";
2
3  const app = express();
4  app.use(express.json());
5
6  app.listen(3001, () => {
7    console.log("App server is running on port 3001");
8  });
9
```

```
PS D:\shopEZ> cd server
PS D:\shopEZ\server> node index.js
App server is running on port 3001
```

4. Define API Routes:

- Create separate route files for different API functionalities such as users , authentication and validation.
- Define the necessary routes for listing doctors, handling user registration and login , managing dates, etc.
- Implement route handlers using Express.js to handle requests and interact with the database.

5. Implement Data Models:

- Define Mongoose schemas for the different data entities like users, applications ,book doctor (or)cancel , conformation
 - Create corresponding Mongoose models to interact with the MongoDB database.
 - Implement CRUD operations (Create, Read, Update, Delete) for each model to perform database operations.

6. User Authentication:

- Create routes and middleware for user registration, login, and logout.
- Set up authentication middleware to protect routes that require user authentication.

7. Handle new products and Orders:

- Create routes and controllers to handle new product listings, including fetching products data from the database and sending it as a response.
- Implement functionality by creating routes and controllers to handle requests, including validation and database updates.

8. Admin Functionality:

- Implement routes and controllers specific to admin functionalities such as adding doctors, managing user , etc.
- Add necessary authentication and authorization checks to ensure only authorized admins can access these routes.

9. Error Handling:

- Implement error handling middleware to catch and handle any errors that occur during the API requests.
- Return appropriate error responses with relevant error messages and HTTP status codes.

FRONTEND DEVELOPMENT:

1. Setup React Application:

- Create a React app in the client folder.
- Install required libraries
- Create required pages and components and add routes.

2.Design UI components:

- Create Components.
- Implement layout and styling.
- Add navigation.

3.Implement frontend logic:

- Integration with API endpoints.
- Implement data binding.

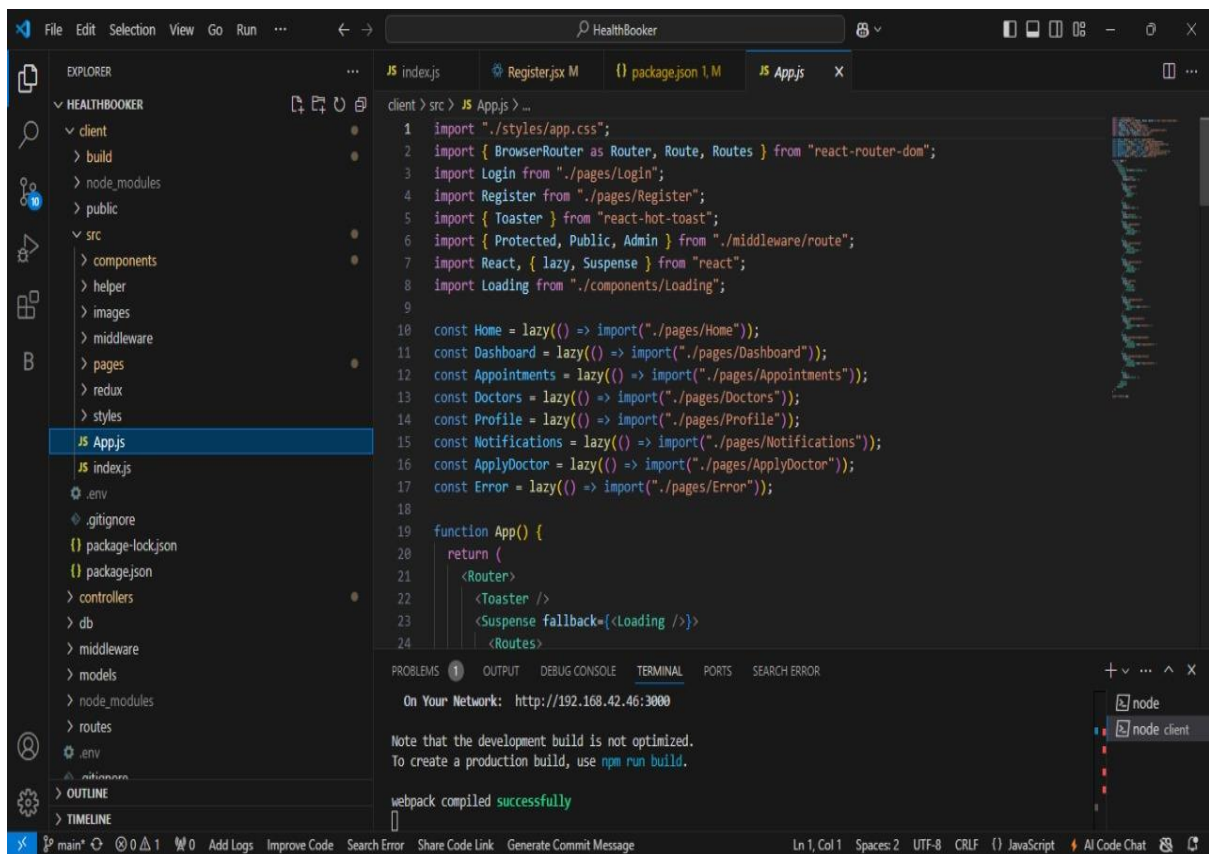
Reference Video Link:

<https://drive.google.com/file/d/1EokogagcLMUGiIluwHGYQo65x8GRpDcP/view?usp=sharing>

Reference Article Link:

https://www.w3schools.com/react/react_getstarted.asp

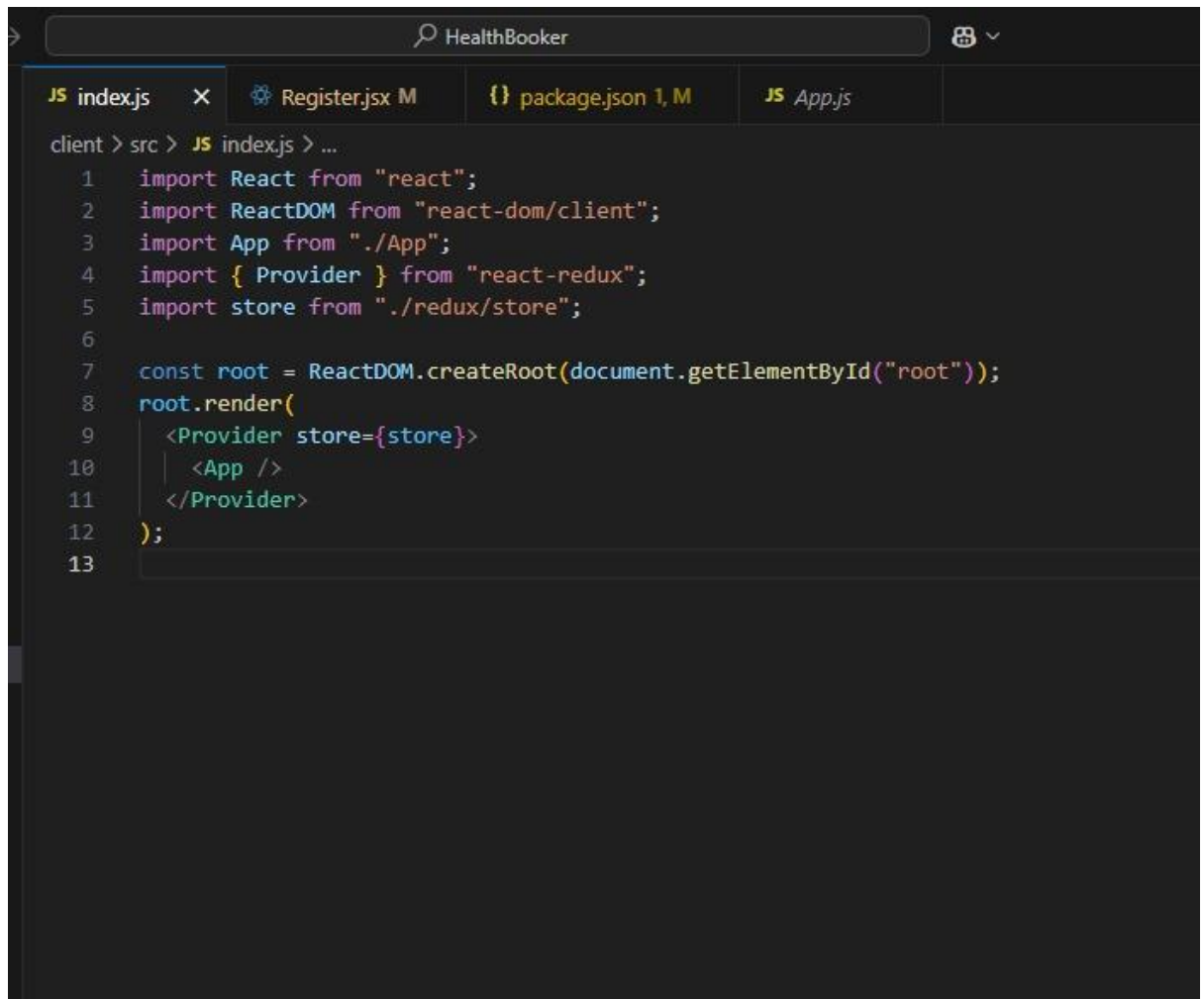
Reference Image:



CODE EXPLANATION

Server setup:

Let us import all the required tools/libraries and connect the database.



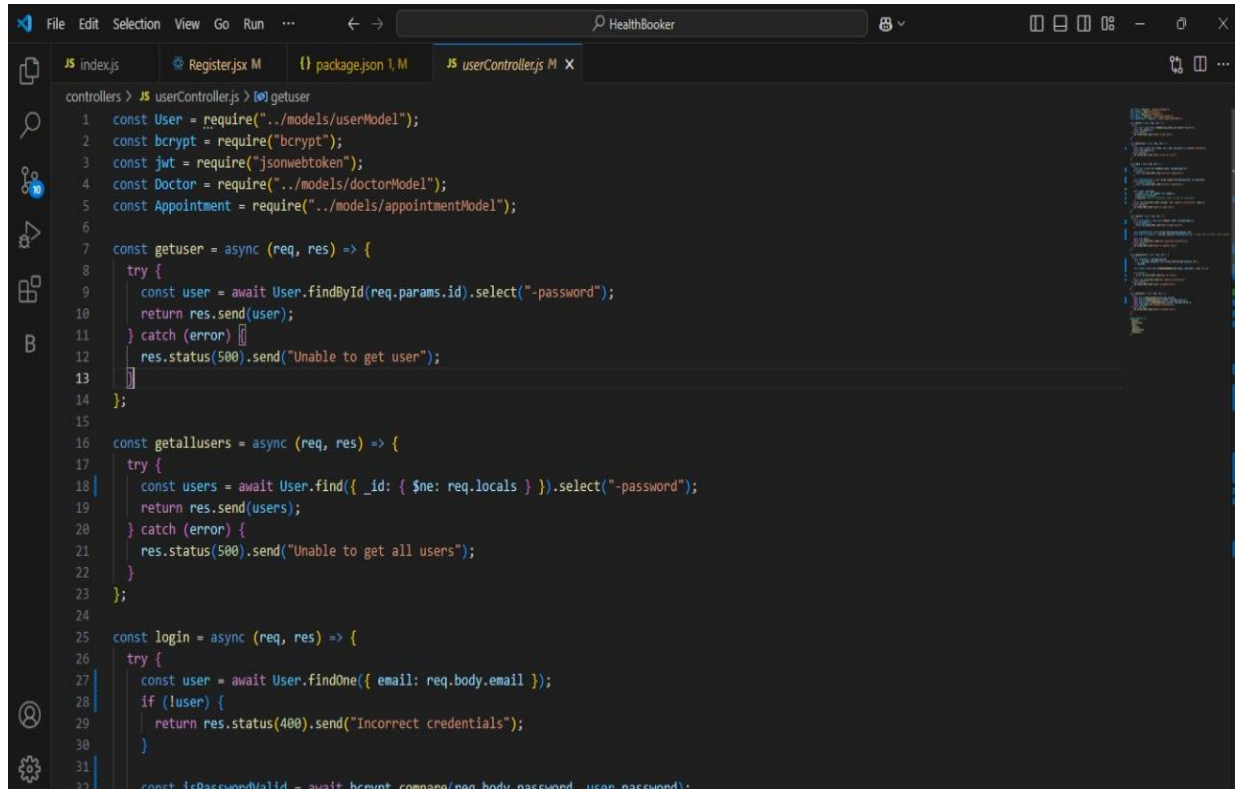
The screenshot shows a code editor with a dark theme. At the top, there's a search bar with the text "HealthBooker" and a dropdown menu. Below the search bar, there are four tabs: "index.js" (selected), "Register.jsx M", "package.json 1, M", and "App.js". The main editor area shows the following code:

```
client > src > JS index.js > ...
1  import React from "react";
2  import ReactDOM from "react-dom/client";
3  import App from "./App";
4  import { Provider } from "react-redux";
5  import store from "./redux/store";
6
7  const root = ReactDOM.createRoot(document.getElementById("root"));
8  root.render(
9    <Provider store={store}>
10     <App />
11   </Provider>
12 );
13
```

USER AUTHENTICATION:

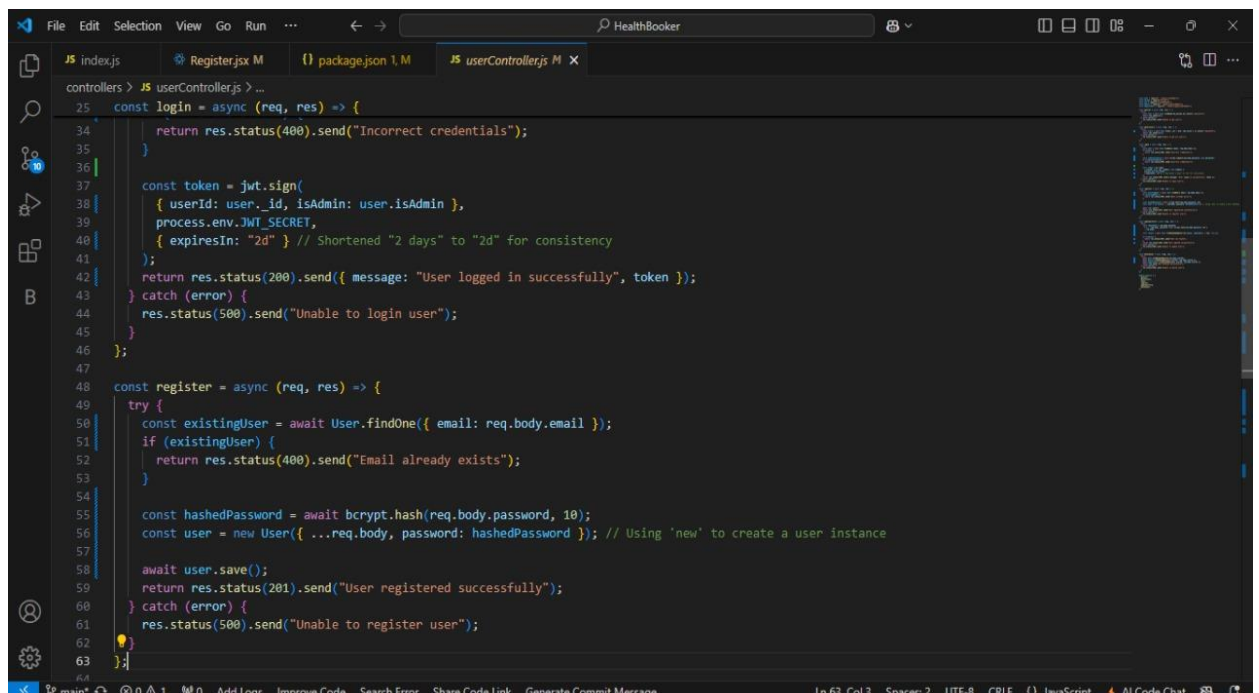
Backend:

Now, here we define the functions to handle http requests from the client for authentication.



```
File Edit Selection View Go Run ... HealthBooker
JS index.js Register.js M {} package.json 1, M JS userController.js M X
controllers > JS userController.js > getuser
1  const User = require("../models/userModel");
2  const bcrypt = require("bcrypt");
3  const jwt = require("jsonwebtoken");
4  const Doctor = require("../models/doctorModel");
5  const Appointment = require("../models/appointmentModel");
6
7  const getuser = async (req, res) => {
8    try {
9      const user = await User.findById(req.params.id).select("-password");
10     return res.send(user);
11   } catch (error) {
12     res.status(500).send("Unable to get user");
13   }
14 };
15
16 const getAllusers = async (req, res) => {
17   try {
18     const users = await User.find({ _id: { $ne: req.locals } }).select("-password");
19     return res.send(users);
20   } catch (error) {
21     res.status(500).send("Unable to get all users");
22   }
23 };
24
25 const login = async (req, res) => {
26   try {
27     const user = await User.findOne({ email: req.body.email });
28     if (!user) {
29       return res.status(400).send("Incorrect credentials");
30     }
31
32     const isPasswordValid = await bcrypt.compare(req.body.password, user.password);
```

Frontend:

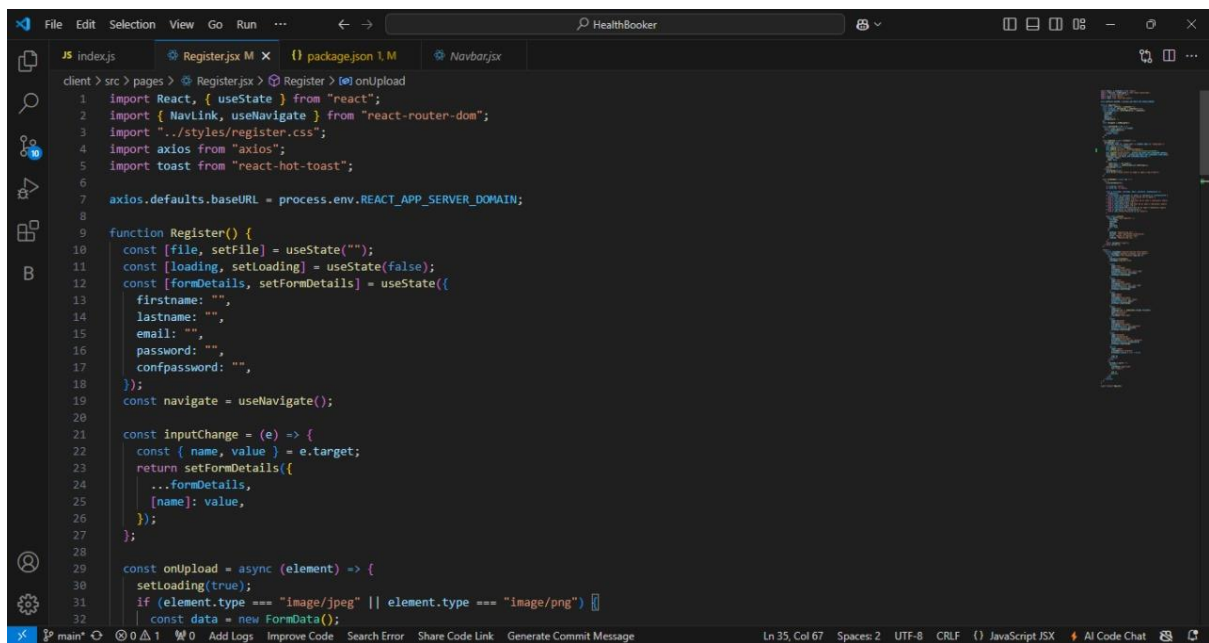


```
File Edit Selection View Go Run ... HealthBooker
JS index.js Register.js M {} package.json 1, M JS userController.js M X
controllers > JS userController.js > ...
25 const login = async (req, res) => {
34   return res.status(400).send("Incorrect credentials");
35 }
36
37 const token = jwt.sign(
38   { userId: user._id, isAdmin: user.isAdmin },
39   process.env.JWT_SECRET,
40   { expiresIn: "2d" } // Shortened "2 days" to "2d" for consistency
41 );
42 return res.status(200).send({ message: "User logged in successfully", token });
43 } catch (error) {
44   res.status(500).send("Unable to login user");
45 }
46 };
47
48 const register = async (req, res) => {
49   try {
50     const existingUser = await User.findOne({ email: req.body.email });
51     if (existingUser) {
52       return res.status(400).send("Email already exists");
53     }
54
55     const hashedPassword = await bcrypt.hash(req.body.password, 10);
56     const user = new User({ ...req.body, password: hashedPassword }); // Using 'new' to create a user instance
57
58     await user.save();
59     return res.status(201).send("User registered successfully");
60   } catch (error) {
61     res.status(500).send("Unable to register user");
62   }
63 };
64
```

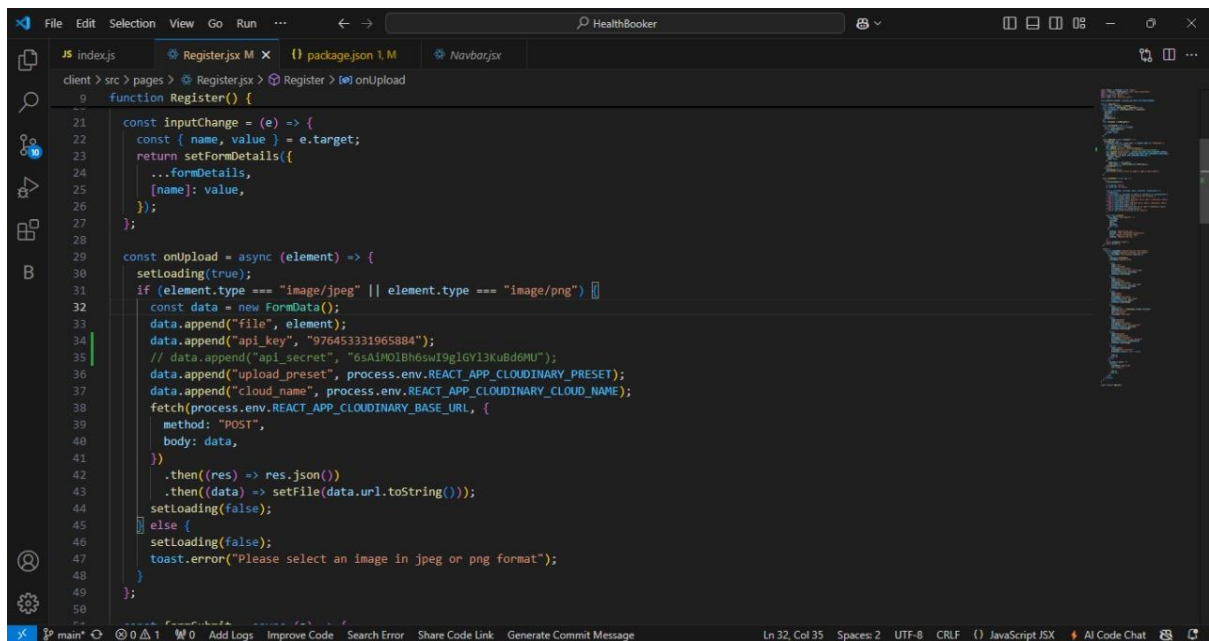

LOGIN:

```
const logoutFunc = () => {  
  dispatch(setUserInfo({}));  
  localStorage.removeItem("token");  
  navigate("/login");  
};
```

REGISTER:



```
File Edit Selection View Go Run ... HealthBooker  
JS index.js Register.jsx M X package.json 1. M Navbar.jsx  
client > src > pages > Register.jsx > Register > onUpload  
1 import React, { useState } from "react";  
2 import { NavLink, useNavigate } from "react-router-dom";  
3 import "../styles/register.css";  
4 import axios from "axios";  
5 import toast from "react-hot-toast";  
6  
7 axios.defaults.baseURL = process.env.REACT_APP_SERVER_DOMAIN;  
8  
9 function Register() {  
10   const [file, setFile] = useState("");  
11   const [loading, setloading] = useState(false);  
12   const [formDetails, setFormDetails] = useState({  
13     firstname: "",  
14     lastname: "",  
15     email: "",  
16     password: "",  
17     confirmPassword: "",  
18   });  
19   const navigate = useNavigate();  
20  
21   const inputChange = (e) => {  
22     const { name, value } = e.target;  
23     return setFormDetails({  
24       ...formDetails,  
25       [name]: value,  
26     });  
27   };  
28  
29   const onUpload = async (element) => {  
30     setloading(true);  
31     if (element.type === "image/jpeg" || element.type === "image/png") {  
32       const data = new FormData();
```



```
32       const data = new FormData();  
33       data.append("file", element);  
34       data.append("api_key", "976453331965884");  
35       // data.append("api_secret", "6sAJW01Bh6suI9glGV13Ku8d6MU");  
36       data.append("upload_preset", process.env.REACT_APP_CLOUDINARY_PRESET);  
37       data.append("cloud_name", process.env.REACT_APP_CLOUDINARY_CLOUD_NAME);  
38       fetch(process.env.REACT_APP_CLOUDINARY_BASE_URL, {  
39         method: "POST",  
40         body: data,  
41       })  
42         .then((res) => res.json())  
43         .then((data) => setFile(data.url.toString()));  
44       setloading(false);  
45     } else {  
46       setloading(false);  
47       toast.error("Please select an image in jpeg or png format");  
48     }  
49   };  
50 }  
51  
52 export default Register;
```

All User:

Frontend:

In the home page, we'll fetch all the products available in the platform along with the filters.

```
8
9  function Register() {
10    const [file, setFile] = useState("");
11    const [loading, setLoading] = useState(false);
12    const [formDetails, setFormDetails] = useState({
13      firstname: "",
14      lastname: "",
15      email: "",
16      password: "",
17      confpassword: "",
18    });
19    const navigate = useNavigate();
20
21    const inputChange = (e) => {
22      const { name, value } = e.target;
23      return setFormDetails({
24        ...formDetails,
25        [name]: value,
26      });
27    };

```

Backend:

```
47
48  const register = async (req, res) => {
49    try {
50      const existingUser = await User.findOne({ email: req.body.email });
51      if (existingUser) {
52        return res.status(400).send("Email already exists");
53      }
54
55      const hashedPassword = await bcrypt.hash(req.body.password, 10);
56      const user = new User({ ...req.body, password: hashedPassword }); // Using 'new' to create
57
58      await user.save();
59      return res.status(201).send("User registered successfully");
60    } catch (error) {
61      res.status(500).send("Unable to register user");
62    }
63  };

```

UPDATE PROFILE:

Frontend:

Here, we can add the product to the cart and later can buy them.

```
13 function Profile() {
14   const { userId } = jwt_decode(localStorage.getItem("token"));
15   const dispatch = useDispatch();
16   const { loading } = useSelector((state) => state.root);
17   const [file, setFile] = useState("");
18   const [formDetails, setFormDetails] = useState({
19     firstname: "",
20     lastname: "",
21     email: "",
22     age: "",
23     mobile: "",
24     gender: "neither",
25     address: "",
26     password: "",
27     confpassword: "",
28   });
29 }
```

Backend:

```
65 const updateprofile = async (req, res) => {
66   try {
67     const updateData = req.body.password
68       ? { ...req.body, password: await bcrypt.hash(req.body.password, 10) }
69       : req.body;
70
71     const result = await User.findByIdAndUpdate(req.locals, updateData, { new: true });
72
73     if (!result) {
74       return res.status(404).send("User not found");
75     }
76     return res.status(200).send("User updated successfully");
77   } catch (error) {
78     res.status(500).send("Unable to update user");
79   }
80 };
```

LOGIN:

Frontend:

```
13 function Login() {
14   const dispatch = useDispatch();
15   const [formDetails, setFormDetails] = useState({
16     email: "",
17     password: "",
18   });
19   const navigate = useNavigate();
20
21   const inputChange = (e) => {
22     const { name, value } = e.target;
23     return setFormDetails({
24       ...formDetails,
25       [name]: value,
26     });
27   };
28 }
```

Backend:

```
25 const login = async (req, res) => {
26   try {
27     const user = await User.findOne({ email: req.body.email });
28     if (!user) {
29       return res.status(400).send("Incorrect credentials");
30     }
31
32     const isPasswordValid = await bcrypt.compare(req.body.password, user.password);
33     if (!isPasswordValid) {
34       return res.status(400).send("Incorrect credentials");
35     }
36
37     const token = jwt.sign(
38       { userId: user._id, isAdmin: user.isAdmin },
39       process.env.JWT_SECRET,
40       { expiresIn: "2d" } // Shortened "2 days" to "2d" for consistency
41     );
42     return res.status(200).send({ message: "User logged in successfully", token });
43   } catch (error) {
44     res.status(500).send("Unable to login user");
45   }
46 };
```

NOTIFICATION:

Frontend:

```
12 const Notifications = () => {
13   const [notifications, setNotifications] = useState([]);
14   const dispatch = useDispatch();
15   const { loading } = useSelector((state) => state.root);
16
17   const getAllNotif = async (e) => {
18     try {
19       dispatch(setLoading(true));
20       const temp = await fetchData(`/notification/getallnotifs`);
21       dispatch(setLoading(false));
22       setNotifications(temp);
23     } catch (error) {}
24   };
25
26   useEffect(() => {
27     getAllNotif();
28   }, []);
```

Backend:

```
controllers > JS notificationController.js > ...
1  const Notification = require("../models/notificationModel");
2
3  const getallnotifs = async (req, res) => {
4    try {
5      const notifs = await Notification.find({ userId: req.locals });
6      return res.send(notifs);
7    } catch (error) {
8      res.status(500).send("Unable to get all notifications");
9    }
10 };
11
12 module.exports = {
13   getallnotifs,
14 };
15
```


APPLY FOR DOCTOR:

Frontend:

```
const Doctors = () => {
  const [doctors, setDoctors] = useState([]);
  const dispatch = useDispatch();
  const { loading } = useSelector((state) => state.root);

  const fetchAllDocs = async () => {
    dispatch(setLoading(true));
    const data = await fetchData(`/doctor/getalldoctors`);
    setDoctors(data);
    dispatch(setLoading(false));
  };

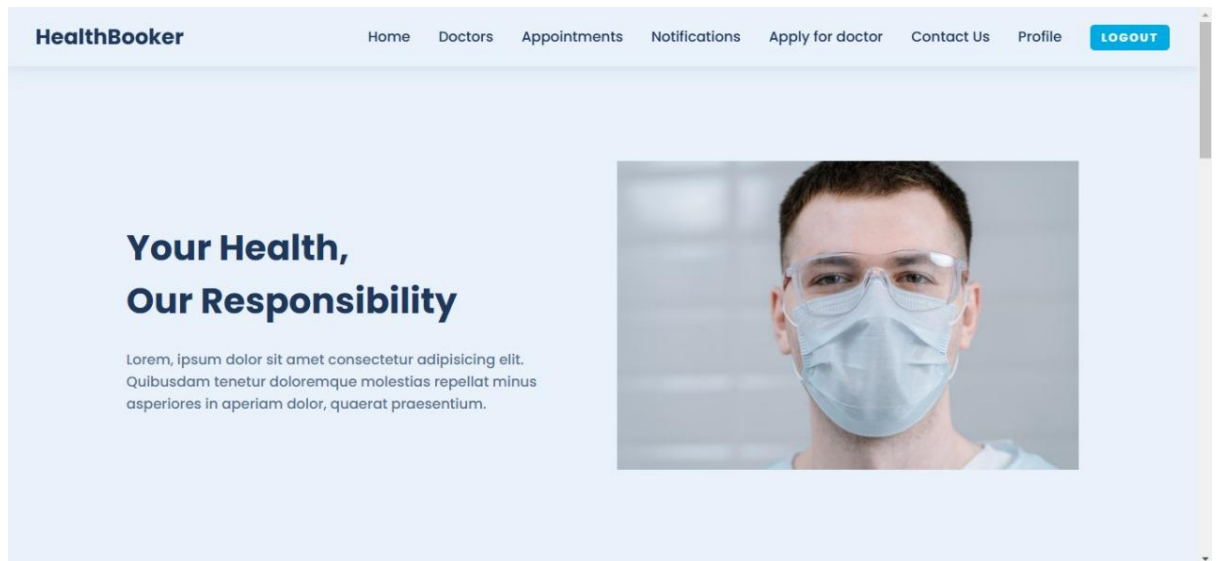
  useEffect(() => {
    fetchAllDocs();
  }, []);
}
```

Backend:

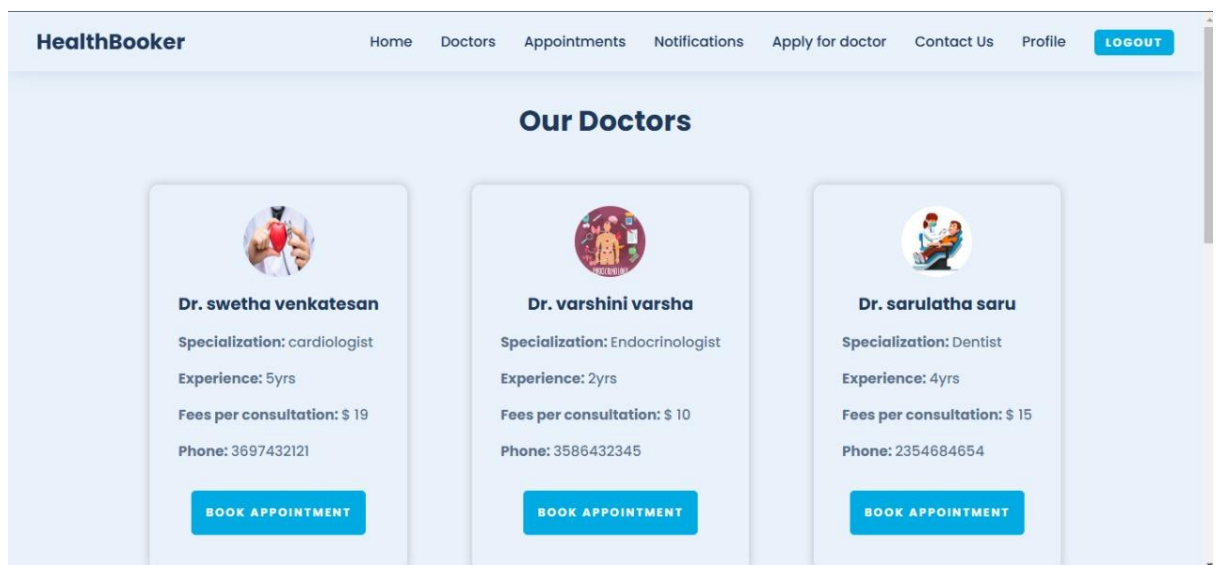
```
6  const getalldoctors = async (req, res) => {
7    try {
8      let docs;
9      if (!req.locals) {
10       docs = await Doctor.find({ isDoctor: true }).populate("userId");
11     } else {
12       docs = await Doctor.find({ isDoctor: true })
13         .find({
14           _id: { $ne: req.locals },
15         })
16         .populate("userId");
17     }
18
19     return res.send(docs);
20   } catch (error) {
21     res.status(500).send("Unable to get doctors");
22   }
23   };
```

Demo UI images:

Landing page:



DOCTORS:



HealthBooker

Home

Doctors

Appointments


Notifications

Apply for doctor

Contact Us

Profile

LOGOUT



Dr. algin algin


Specialization: Psychiatrist

Experience: 3yrs

Fees per consultation: \$ 12

Phone: 7843212315

BOOK APPOINTMENT



Dr. lavanya suji


Specialization: physiotherapy

Experience: 3yrs

Fees per consultation: \$ 16

Phone: 3688632343

BOOK APPOINTMENT



Dr. doctor venky

Specialization: General surgery

Experience: 10yrs

Fees per consultation: \$ 15

Phone: 1255487566

BOOK APPOINTMENT

APPOINTMENTS:

HealthBooker

Home

Doctors

Appointments

Notifications

Apply for doctor

Contact Us

Profile

LOGOUT

Your Appointments

S.No	Doctor	Patient	Appointment Date	Appointment Time	Booking Date	Booking Time	Status	Action
1	swetha venkatesan	reshma vadivel	2024-11-11	11:01	2024-11-09	06:32:49	Completed	COMPLETE

NOTIFICATION BAR:

HealthBooker

Home

Doctors

Appointments

Notifications

Apply for doctor

Contact Us

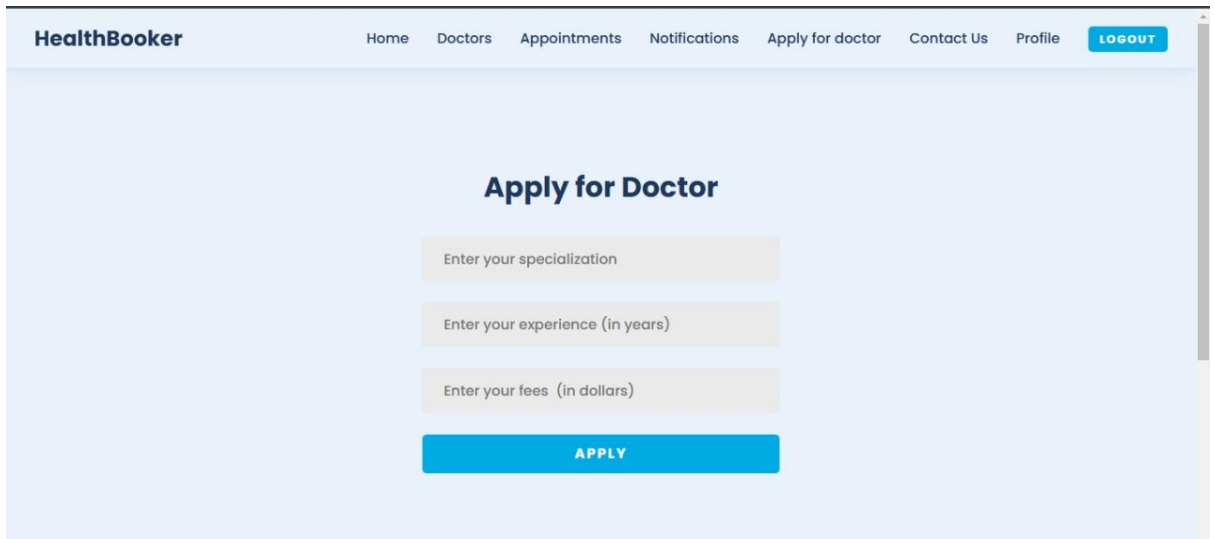
Profile

LOGOUT

Your Notifications

S.No	Content	Date	Time
1	You have an appointment with reshma vadivel on 2024-11-11 at 11:01	2024-11-09	06:31:58

APPLY FOR DOCTOR:



The screenshot shows the 'Apply for Doctor' page of the HealthBooker application. The page has a light blue background. At the top, there is a navigation bar with the following links: Home, Doctors, Appointments, Notifications, Apply for doctor, Contact Us, Profile, and a blue button labeled 'LOGOUT'. The main heading is 'Apply for Doctor'. Below it, there are three input fields for 'Enter your specialization', 'Enter your experience (in years)', and 'Enter your fees (in dollars)'. At the bottom, there is a blue button labeled 'APPLY'.

HealthBooker

Home Doctors Appointments Notifications Apply for doctor Contact Us Profile **LOGOUT**

Apply for Doctor

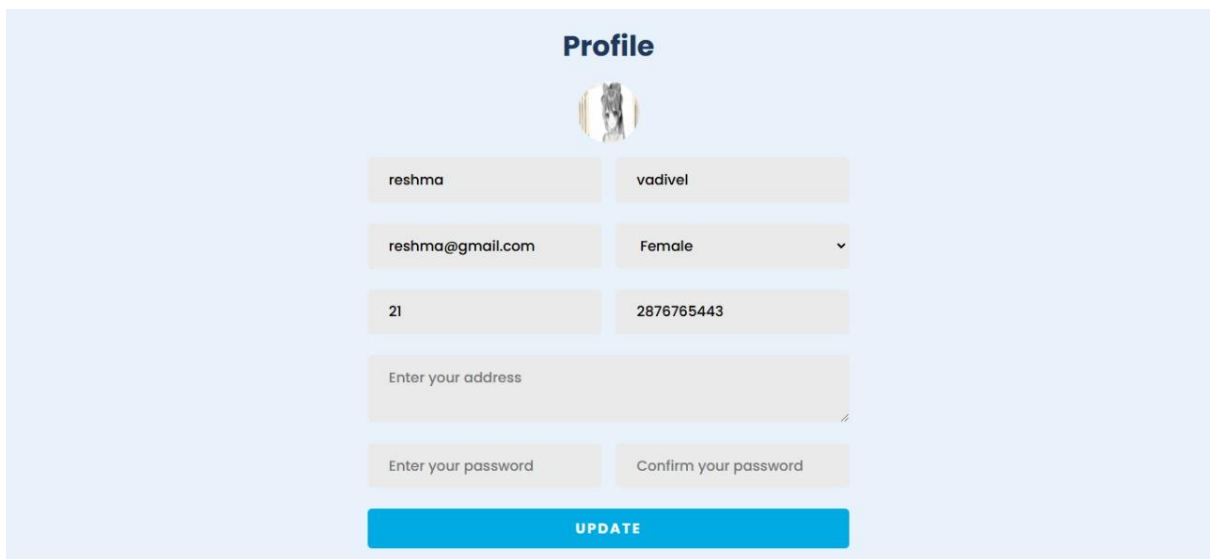
Enter your specialization

Enter your experience (in years)

Enter your fees (in dollars)


APPLY

USERPROFILE:



The screenshot shows the 'Profile' page of the HealthBooker application. The page has a light blue background. At the top, there is a heading 'Profile' and a circular profile picture placeholder. Below the profile picture, there are several input fields for user information: 'reshma', 'vadivel', 'reshma@gmail.com', 'Female' (with a dropdown arrow), '21', '2876765443', 'Enter your address', 'Enter your password', and 'Confirm your password'. At the bottom, there is a blue button labeled 'UPDATE'.

Profile



reshma vadivel

reshma@gmail.com Female ▾

21 2876765443

Enter your address

Enter your password Confirm your password

UPDATE

ADMIN DASHBOARD:

Home

Users

Doctors

Appointments

Applications

Profile

Logout

All Users

S.No	Pic	First Name	Last Name	Email	Mobile No.	Age	Gender	Is Doctor	Remove
1		swetha	venkatesan	swetha@gmail.com	3697432121	21	female	Yes	REMOVE
2		varshini	varsha	varshini@gmail.com	3586432345	21	female	Yes	REMOVE
3		sarulatha	saru	saru@gmail.com	2354684654	21	female	Yes	REMOVE
4		algin	algin	algin@gmail.com	7843212315	21	male	Yes	REMOVE
5		lavanya	suji	lavanya@gmail.com	3688632343	21	female	Yes	REMOVE
6		reshma	vadivel	reshma@gmail.com	2876765443	21	female	No	REMOVE
7		doctor	venky	venky@gmail.com	1255487566	50	male	Yes	REMOVE

ACCEPT APPOINTMENTS:

Home

Users

Doctors

Appointments

Applications

Profile

All Users

Doctor	Patient	Appointment Date	Appointment Time	Booking Date	Booking Time	Status
swetha venkatesan	reshma vadivel	2024-11-11	11:01	2024-11-09	06:32:49	Completed
varshini varsha	doctor venky	2024-11-12	14:42	2024-11-09	08:12:38	Completed

** THANK YOU **