JEPPIAAR
ENGINEERING COLLEGE

NM1042-MERN Stack Powered by Mongo DB: Online Complaint Registration and Management System

A PROJECT REPORT

SUBMITTED BY

HARSHAD H - 310821104036

ABINAVV A B - 310821104002

JAFAR SHARIF J - 310821104037

BALAGANESH S - 310821104015

IN PARTIAL FULFILLMENT FOR THE AWARD OF THE DEGREE OF

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

JEPPIAAR ENGINEERING COLLEGE

ANNA UNIVERSITY : CHENNAI - 600025

NOVEMBER 2024

ANNA UNIVERSITY : CHENNAI 600025

BONAFIDE CERTIFICATE

Certified that this Project report **"Online Complaint Registration and Management System"** is the bonafide work of **"Balaganesh S (310821104015)"** who carried out the project work for Naan Mudhalvan.

NM ID - A90AED740804E9487E3457881CAD4C39

MENTOR **HEAD OF THE DEPARTMENT**

Date: _____ Internal: _____

External: _____

Contents

List of Figures

Executive Summary

The Online Complaint Registration and Management System (OCRMS) is a software solution designed to streamline the process of lodging, tracking, and resolving complaints. Built with the MERN stack (MongoDB, Express, React.js, Node.js), OCRMS centralizes complaint handling, ensuring efficiency, regulatory compliance, and improved user satisfaction. It is ideal for organizations aiming to enhance issue resolution and safety management in line with industry standards.

OCRMS incorporates several key features:

- **User Registration**: Users can create accounts to securely submit complaints and track their progress.
- **Complaint Submission**: Users can provide detailed descriptions of their issues, along with supplementary information like name, address, and relevant attachments such as documents or images.
- **Tracking and Notifications**: Real-time updates allow users to monitor complaint statuses and receive notifications via email or SMS for every significant action taken, such as assignment or resolution.
- **Agent Interaction**: Users can directly communicate with the assigned agent to discuss and resolve complaints effectively.
- **Intelligent Complaint Routing**: The system ensures complaints are routed to the appropriate personnel or department based on predefined criteria, optimizing resource allocation.
- **Security and Confidentiality**: OCRMS prioritizes data security and compliance with privacy regulations by employing robust measures like user authentication, data encryption, and access controls.

Description

OCRMS is an intuitive and scalable solution for handling complaints, designed with the end-user in mind. It simplifies the registration and resolution of grievances while fostering communication between users and service agents. With features such as automatic notifications, intelligent complaint routing, and state-of-the-art security protocols, it ensures timely and effective complaint resolution.

Scenario

Consider the case of John, a customer who recently purchased a defective product online. He uses OCRMS to file a complaint and have his issue resolved efficiently:

1. **User Registration and Login**:
    - John visits the OCRMS website and registers by providing his name, email, and a secure password.
    - After verifying his account through email, John logs into the system and accesses the user dashboard.
2. **Complaint Submission**:
    - John submits a detailed complaint by filling out a form, attaching images of the defect, and providing necessary details such as the product purchase date and contact information.
    - Upon submission, he receives a confirmation message.
3. **Tracking and Notifications**:
    - John navigates to the "My Complaints" section to monitor the progress of his complaint.
    - He also receives email notifications whenever his complaint status is updated, such as when it is assigned to an agent or resolved.
4. **Interaction with Agent**:
    - A service agent, Sarah, is assigned to John's case. Using OCRMS's built-in messaging feature, Sarah contacts John to discuss the issue in detail.

- John and Sarah collaborate to find a solution, with Sarah assuring prompt action.

5. **Resolution and Feedback**:

   - After investigation, the company offers John a replacement or refund.
   - John is notified of the resolution and provides feedback on his experience, commending the prompt and professional service.

6. **Admin Management**:

   - The platform administrator oversees all registered complaints, assigning them to agents based on workload and expertise.

Introduction

In modern governance and customer service, addressing grievances efficiently is vital for maintaining trust and satisfaction. Traditional complaint systems often suffer from inefficiencies, poor transparency, and lack of communication, leading to delays and user frustration.

The Online Complaint Registration and Management System (OCRMS) is designed to address these issues. By leveraging modern technologies, OCRMS streamlines complaint registration, management, and resolution, ensuring transparency, accountability, and real-time communication throughout the process.

1.1 Objectives

The primary objectives of OCRMS are as follows:

- **To create a user-friendly complaint registration platform:** The system aims to offer an intuitive interface that enables users to easily lodge complaints, provide necessary details, and interact with the assigned agents.
- **To streamline complaint management for authorities:** OCRMS optimizes the workflow for handling complaints by intelligently assigning them to appropriate personnel or departments and facilitating better resource utilization.
- **To provide real-time updates to users about complaint status:** The system ensures transparency by allowing users to track the status of their complaints and receive notifications about any updates or resolutions.

This system addresses the limitations of existing complaint handling mechanisms while improving efficiency, fostering trust, and ensuring a higher level of user satisfaction. OCRMS is designed to be scalable and adaptable, making it suitable for a variety of industries and organizations.

Methodology

2.1 Approach

The Online Complaint Registration and Management System (OCRMS) is designed using the MERN stack, a collection of powerful, open-source JavaScript technologies. This stack is ideal for developing modern, dynamic, and scalable web applications. By leveraging MongoDB, Express.js, React.js, and Node.js, OCRMS provides an efficient, robust, and extensible solution for managing user complaints. The following subsections detail each of the core components of the MERN stack:

- **MongoDB**: MongoDB is a highly scalable NoSQL database that stores data in flexible, JSON-like documents. Its schema-less structure allows for rapid prototyping and quick adjustments during development. Unlike traditional relational databases, MongoDB is particularly well-suited for managing semi-structured or unstructured data. This flexibility helps OCRMS easily accommodate changing data requirements as the system evolves. Furthermore, MongoDB's built-in support for horizontal scaling ensures that the system can efficiently handle an increase in data as the application grows, making it suitable for large-scale complaint management.

- **Express.js**: Express.js is a minimal yet highly extensible framework for Node.js, making it an ideal choice for building web applications and APIs. It simplifies the creation of robust server-side logic, enabling the seamless processing of HTTP requests, managing routes, and integrating middleware. Express.js allows the creation of RESTful APIs, which facilitate communication between the frontend and backend, making it a core component of OCRMS for handling user requests such as submitting complaints, updating complaint statuses, and managing user profiles. Additionally, Express's middleware system allows the system to integrate important features like request logging,

data validation, and error handling with minimal effort.

- **React.js**: React.js is a powerful JavaScript library that specializes in building complex user interfaces, especially single-page applications (SPAs). React's declarative nature and component-based architecture allow developers to build interactive and reusable UI components. Each component in React encapsulates its own logic and UI, ensuring maintainability and reusability. React's Virtual DOM mechanism improves performance by updating only the parts of the UI that have changed, leading to faster rendering times. In OCRMS, React.js is utilized to create a dynamic and responsive frontend where users can register complaints, track their status, interact with agents, and receive real-time updates.

- **Node.js**: Node.js is an open-source, cross-platform runtime environment that allows for the execution of JavaScript code outside of the browser. It is built on Google Chrome's V8 JavaScript engine and uses an event-driven, non-blocking I/O model, making it lightweight and efficient. This architecture is well-suited for building scalable network applications that require high throughput and real-time communication. OCRMS leverages Node.js to handle backend processes, including server-side logic, API integration, and real-time data handling. Its asynchronous nature enables OCRMS to handle many user requests concurrently, ensuring smooth operations even under high traffic conditions.

2.2 Methods

2.2.1 System Architecture

The architecture of OCRMS is designed using a client-server model, where the frontend (client-side), backend (server-side), and database are clearly separated, allowing for better scalability, modularity, and ease of maintenance. The architecture ensures smooth communication between all components, enabling a responsive and efficient application that can handle a high volume of user interactions. Below is a detailed breakdown of the OCRMS architecture:

- **Frontend (Client-Side):** The frontend of OCRMS is the user-facing interface, built using React.js. It provides users with a dynamic and responsive experience, allowing

them to register complaints, track complaint statuses, communicate with agents, and view their complaint history. The key technical highlights of the frontend include:

  - **Component-Based Architecture:** React's component-based structure allows for the development of reusable UI components such as forms, modals, dashboards, and tables. This promotes consistency across the application and reduces redundancy in code.
  - **UI/UX Design:** The frontend incorporates modern design principles and responsive UI frameworks such as Bootstrap and Material UI to ensure a seamless user experience across a variety of devices, including desktops, tablets, and smartphones.
  - **State Management:** React's state management system ensures that the application maintains and updates its data dynamically. Using state management libraries such as Redux or React's built-in hooks allows for better control over the flow of data between components, leading to an optimized user experience.
  - **Real-Time Features:** React is integrated with WebSockets, specifically Socket.IO, to enable real-time communication features. These include live updates for complaint statuses, direct communication with agents, and instant notifications for new messages or updates.
  - **RESTful API Integration:** The frontend communicates with the backend through RESTful APIs. Axios, a promise-based HTTP client, facilitates asynchronous communication between the frontend and backend for submitting complaints, fetching data, and updating complaint statuses.

- **Backend (Server-Side):** The backend of OCRMS is built with Node.js and Express.js, forming the backbone of the system's logic and data processing. The backend handles user authentication, communication between the client and the database, and the execution of business logic. Key features of the backend include:

  - **RESTful API Design:** The backend exposes a series of RESTful APIs that enable the frontend to interact with the system. These APIs facilitate CRUD operations on entities like complaints, users, and agents. The endpoints are designed to be secure, efficient, and easily extendable.
  - **Middleware Architecture:** Express.js allows the use of middleware functions to handle various server-side operations, such as request validation, user authentica-

tion, logging, and error handling. Middleware functions enhance security, ensure data integrity, and streamline server processing.

  - **Authentication and Authorization:** To ensure secure access to the application, the backend employs JSON Web Tokens (JWT) for stateless authentication. This ensures that only authorized users (e.g., regular users, agents, or administrators) can access the appropriate system features. Additionally, role-based access control (RBAC) is implemented to restrict access based on user roles.
  - **Real-Time Interaction:** Socket.IO enables bi-directional communication between the frontend and backend, allowing the system to push real-time updates to users. This is critical for notifying users about changes in complaint statuses, new messages, or agent availability.

- **Database (Data Storage):** MongoDB serves as the database for OCRMS, offering flexibility, scalability, and ease of use. It stores data in collections of JSON-like documents, which can represent dynamic entities such as users, complaints, and agents. Key aspects of the database design include:

  - **Data Model:** OCRMS uses MongoDB collections to store documents for various entities, including users, complaints, agents, and administrators. Each document contains key fields necessary for the operation of OCRMS, such as complaint description, user details, complaint status, and timestamps.
  - **Indexing and Query Optimization:**MongoDB supports indexing, which enhances the speed of data retrieval. Indexes are created for frequently queried fields, such as complaint status, user ID, and timestamps, allowing for faster searches and efficient data access.
  - **Data Integrity:** Mongoose, an ODM (Object Data Modeling) library for MongoDB, is used to define schemas and enforce data validation rules. Mongoose ensures that the data stored in the database adheres to the required structure, which maintains the integrity of the application data.
  - **Scalability:** MongoDB is horizontally scalable, allowing it to handle large volumes of data as the application grows. The system can be distributed across multiple servers using sharding, ensuring high availability and fault tolerance.

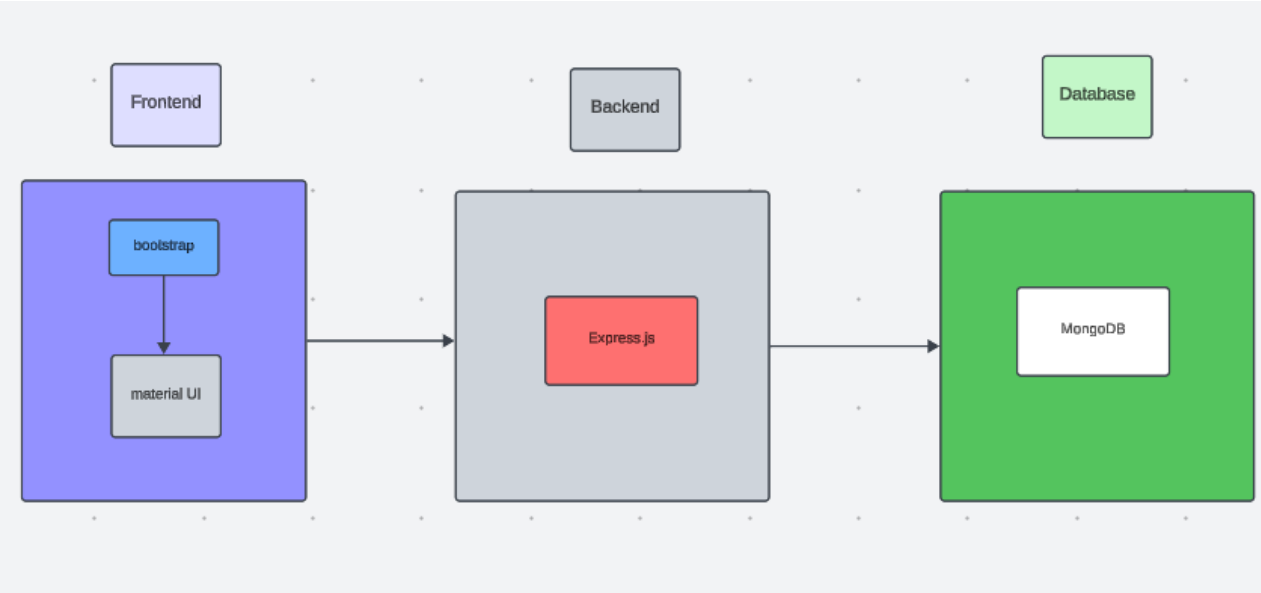


Figure 2.1: System Architecture Diagram

2.2.2 ER Diagram

The Entity-Relationship (ER) Diagram illustrates the relationships and dependencies between core entities in OCRMS, such as users, complaints, agents, and administrators. Each entity is modeled with attributes that define the data to be stored, and the relationships between entities specify how data is linked. For example:

- **Users** are related to **Complaints** through a one-to-many relationship, where each user can file multiple complaints.
- **Agents** are linked to **Complaints** in a many-to-one relationship, as each complaint can be handled by a single agent.
- **Administrators** can manage both **Users** and **Complaints**, with full access to the system for managing complaint resolutions and user accounts.

This ER diagram helps in understanding the structure of the database and how different entities interact within OCRMS.

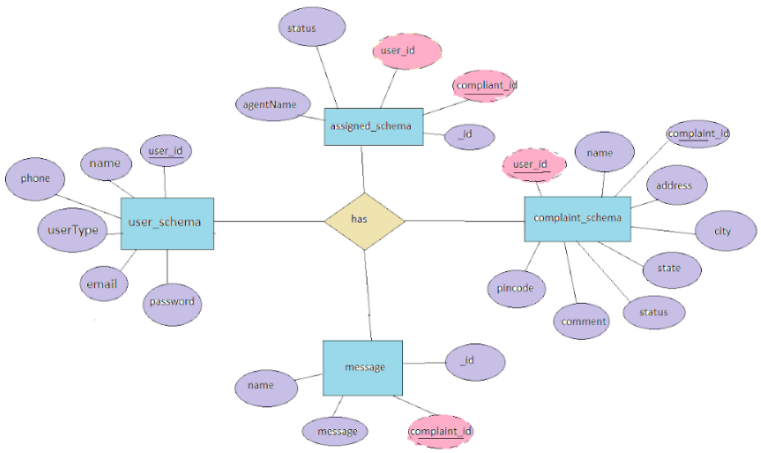


Figure 2.2: Entity-Relationship Diagram

2.2.3 Project Structure

The OCRMS project is organized into three primary components, each serving a distinct function in the system:

Frontend Setup

- **Frontend:**
  - The React.js codebase, structured into functional and class components, ensures that the application is modular, maintainable, and scalable.
  - The frontend includes pages for registering complaints, tracking status, user profiles, and live chat features.
  - State management and data fetching are handled using React's hooks and Axios to ensure efficient communication with the backend and real-time updates.
- **Frontend Setup:** The frontend is built using the following steps:

– **Initial Setup & Libraries:**

Before diving into the creation of UI components, we first need to install the required libraries and set up the basic project structure.

```
npx create-react-app customer-care-registry
cd customer-care-registry
```

– **Required Libraries:**

Install libraries that will help in building the user interface and handling various tasks such as routing, state management, and API calls. Key libraries include:

```
npm install axios react-router-dom redux react-redux material-ui
@mui/icons-material
```

* `axios`: For making HTTP requests to communicate with the backend API.
* `react-router-dom`: For handling navigation between different pages in the app.
* `redux` and `react-redux`: For state management across the application.
* `material-ui`: A UI component library to help design a modern and responsive interface.

- **Backend:**
  - Built using Node.js and Express.js, the backend exposes a suite of RESTful APIs for handling user requests such as complaint registration, status updates, and user management.
  - The backend is also responsible for authentication and authorization using JWT, ensuring secure access.
- **Database:**
  - MongoDB stores user profiles, complaint details, and agent information in collections, ensuring fast and scalable data access.
  - The database schema is defined using Mongoose to enforce data integrity and consistency.

2.3 Technical Workflow

The following sequence of actions illustrates the technical workflow of OCRMS, integrating the frontend, backend, and database layers:

1. A user accesses the frontend and submits a complaint or views their complaint status.
2. The frontend sends an authenticated request to the backend via Axios.
3. The backend processes the request, applies business logic, and communicates with MongoDB to fetch or update data.
4. The backend responds with the requested data, and real-time updates (such as changes in complaint status) are pushed to the frontend using Socket.IO.
5. Data security is ensured through encryption protocols, authentication, and access control policies.

This technical workflow ensures that OCRMS is both user-friendly and scalable, capable of handling high traffic and providing real-time, efficient complaint management.

Structure and Components:

3.1 Frontend Implementation

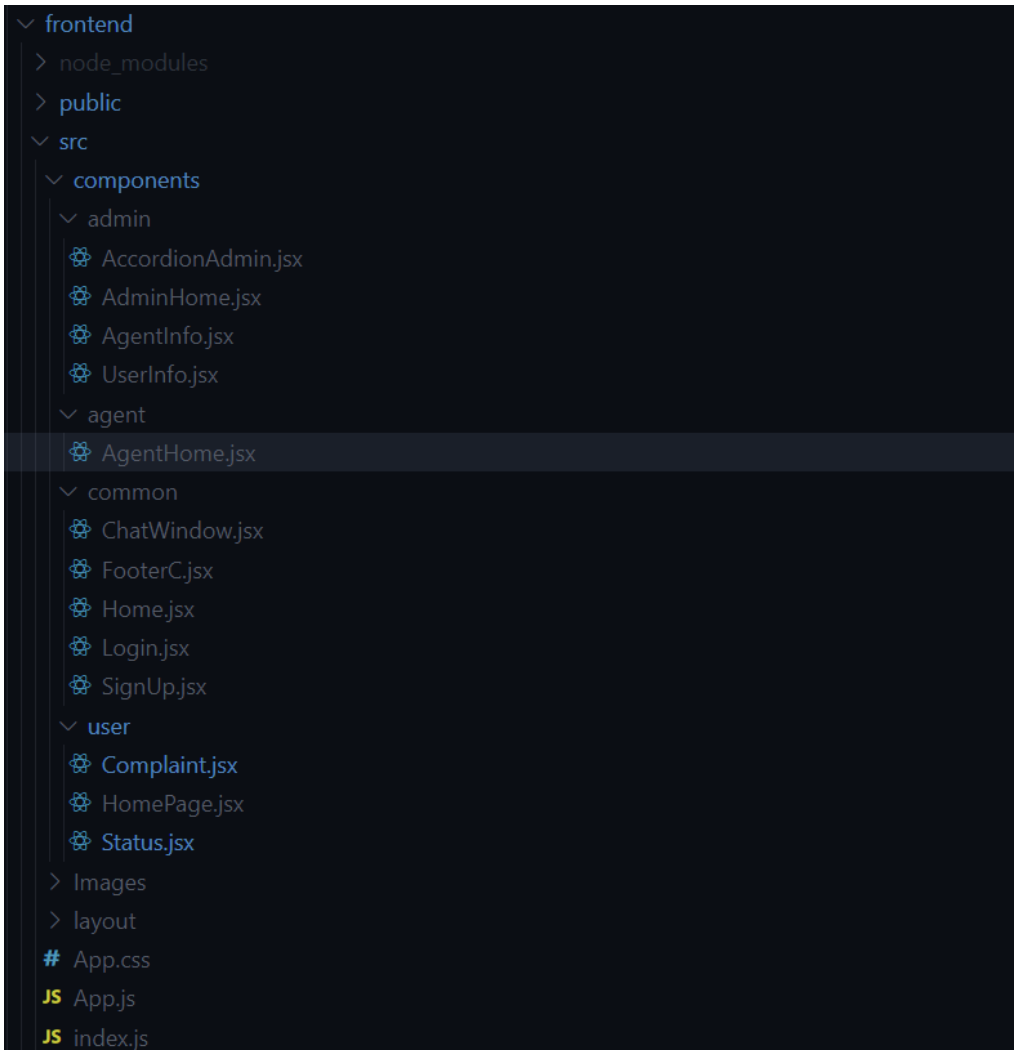


Figure 3.1: Folder Structure

3.2 Admin Implementation

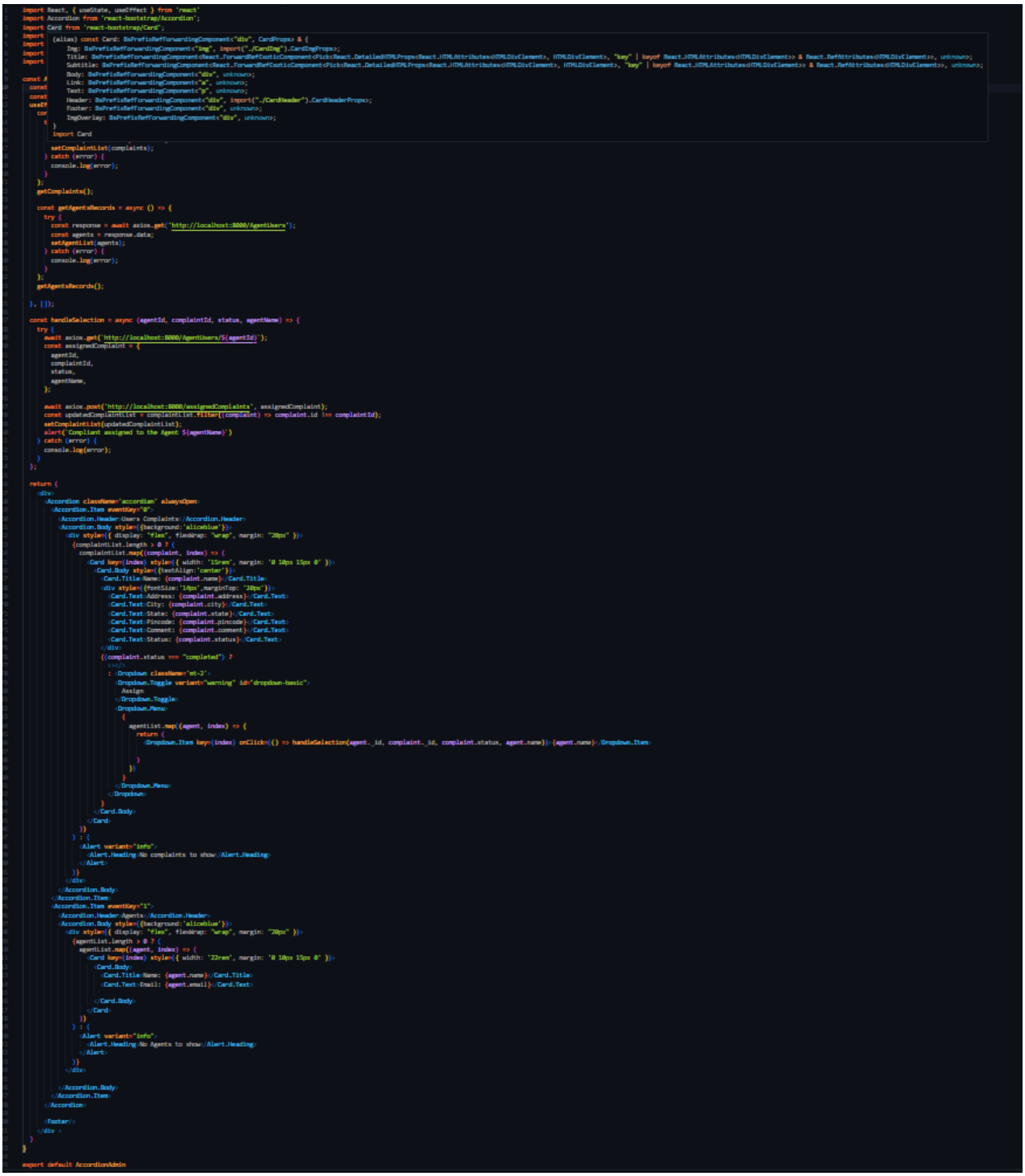


Figure 3.2: Admin Accordion component

```jsx
const AdminHome = () => {
  const navigate = useNavigate();
  const [activeComponent, setActiveComponent] = useState('dashboard');

  const [userName, setUserName] = useState('');


  useEffect(() => {
    const getData = async () => {
      try {
        const user = JSON.parse(localStorage.getItem('user'));
        if (user) {
          const { name } = user;
          setUserName(name);
        } else {
          navigate('/');
        }
      } catch (error) {
        console.log(error);
      }
    };
    getData();
  }, [navigate]);



  const handleNavLinkClick = (componentName) => {
    setActiveComponent(componentName);
  };

  const LogOut = () => {
    localStorage.removeItem('user');
    navigate('/');
  };

  return (
    <>
      <Navbar className="text-white" bg="dark" expand="lg">
        <Container fluid>
          <Navbar.Brand className="text-white" href="#">
            Hi Admin {userName}
          </Navbar.Brand>
          <Navbar.Toggle aria-controls="navbarScroll" />
          <Navbar.Collapse id="navbarScroll">
            <Nav className="text-white me-auto my-2 my-lg-0" style={{ maxHeight: '100px' }} navbarScroll>
              <NavLink
                className={`nav-link text-light ${activeComponent === 'dashboard' ? 'active' : ''}`}
                onClick={() => handleNavLinkClick('dashboard')}
              >
                Dashboard
              </NavLink>
              <NavLink
                className={`nav-link text-light ${activeComponent === 'UserInfo' ? 'active' : ''}`}
                onClick={() => handleNavLinkClick('UserInfo')}
              >
                User
              </NavLink>
              <NavLink
                className={`nav-link text-light ${activeComponent === 'Agent' ? 'active' : ''}`}
                onClick={() => handleNavLinkClick('Agent')}
              >
                Agent
              </NavLink>
            </Nav>
            <Button onClick={LogOut} variant="outline-danger">
              Log out
            </Button>
          </Navbar.Collapse>
        </Container>
      </Navbar>
      <div className="content">
        {activeComponent === 'Agent' ? <AgentInfo /> : null}
        {activeComponent === 'dashboard' ? <AccordionAdmin /> : null}
        {activeComponent === 'UserInfo' ? <UserInfo /> : null}
      </div>
    </>
```

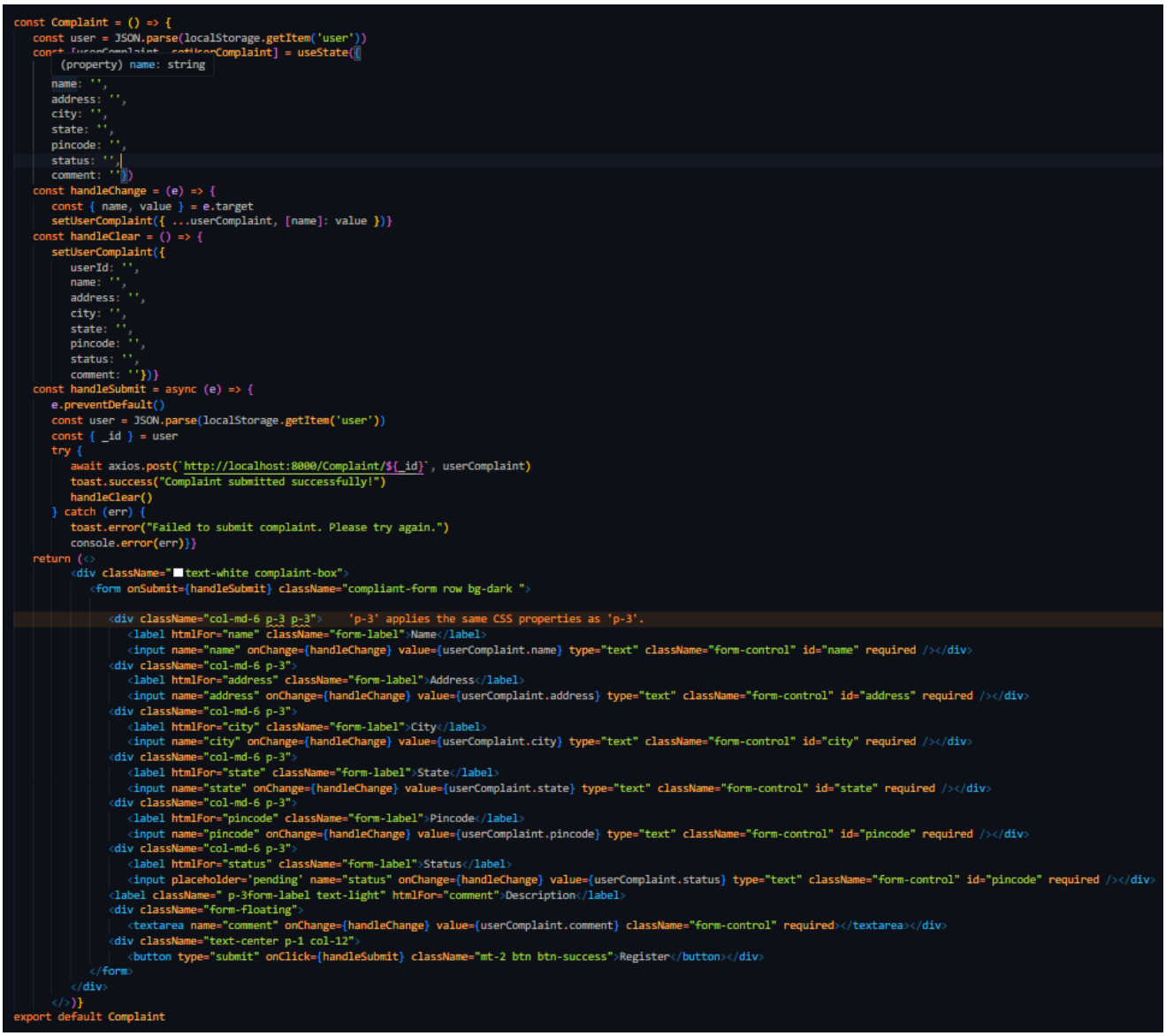
Figure 3.3: Admin Home component

3.3 Agent Implementation

```
const AgentHome = () => {
   const style = {marginTop: '66px',}
   const navigate = useNavigate();
   const [userName, setUserName] = useState('');
   const [toggle, setToggle] = useState({})
   const [agentComplaintList, setAgentComplaintList] = useState([]);
   useEffect(() => {
      const getData = async () => {
         try {const user = JSON.parse(localStorage.getItem('user'));
         if (user) {const { _id, name } = user;setUserName(name);
               const response = await axios.get(`http://localhost:8000/allcomplaints/${_id}`);
               const complaints = response.data;
               setAgentComplaintList(complaints);
            } else {navigate('/');}
         } catch (error) {console.log(error);}};
      getData();}, [navigate]);

   const handleStatusChange = async (complaintId) => {
      try {
         await axios.put(`http://localhost:8000/complaint/${complaintId}`, { status: 'completed' });
         setAgentComplaintList((prevComplaints) =>
            prevComplaints.map((complaint) =>complaint._doc.complaintId === complaintId ? { ...complaint, _doc: { ...complaint._doc, status: 'completed' } } : complaint));
      } catch (error) {console.log(error);}};

   const handleToggle = (complaintId) => {
      setToggle((prevState) => ({
         ...prevState,
         [complaintId]: !prevState[complaintId],}));};
   const LogOut = () => {localStorage.removeItem('user');navigate('/');};
   return (
      <div className="body">
         <Navbar className=" text-white" bg="dark" expand="lg">
            <Container fluid>
               <Navbar.Brand className=" text-white">
                  Hi Agent {userName}
               </Navbar.Brand>
               <Navbar.Toggle aria-controls="navbarScroll" />
               <Navbar.Collapse id="navbarScroll">
                  <Nav className=" text-white me-auto my-2 my-lg-0" style={{ maxHeight: '100px' }} navbarScroll>
                     <NavLink style={{ textDecoration: 'none' }} className=" text-white">
                        View Complaints
                     </NavLink>
                  </Nav>
                  <Button onClick={LogOut} variant="outline-danger">
                     Log out
                  </Button>
               </Navbar.Collapse>
            </Container>
         </Navbar>
         <div className="container" style={{ display: 'flex', flexWrap: 'wrap', margin: '20px' }}>
            {agentComplaintList && agentComplaintList.length > 0 ? (
               agentComplaintList.map((complaint, index) => {
                  const open = toggle[complaint._doc.complaintId] || false;
                  return (
                     <Card key={index} style={{ width: '18rem', margin: '15px' }}>
                        <Card.Body>
                           <Card.Title><b>Name:</b> {complaint.name}</Card.Title>
                           <Card.Text><b>Address:</b> {complaint.address}</Card.Text>
                           <Card.Text><b>City:</b> {complaint.city}</Card.Text>
                           <Card.Text><b>State:</b> {complaint.state}</Card.Text>
                           <Card.Text><b>Pincode:</b> {complaint.pincode}</Card.Text>
                           <Card.Text><b>Comment:</b> {complaint.comment}</Card.Text>
                           <Card.Text><b>Status:</b> {complaint._doc.status}</Card.Text>

                           {complaint.status !== 'completed' && (
                              <Button onClick={() => handleStatusChange(complaint._doc.complaintId)} variant="primary">
                                 Status Change
                              </Button>
                           )}
                           <Button onClick={() => handleToggle(complaint._doc.complaintId)}
                              aria-controls={`collapse-${complaint._doc.complaintId}`}
                              aria-expanded={!open} className='mx-3' variant="primary">
                              Message
                           </Button>
                           <div>
```

Figure 3.4: agent home

3.4 User Implementation

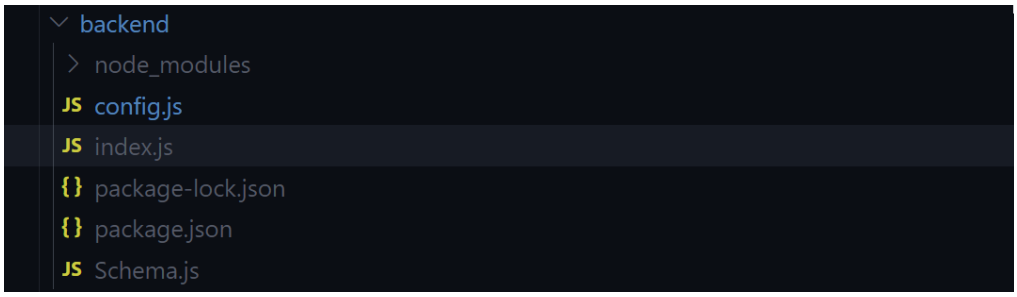


```
const Complaint = () => {
   const user = JSON.parse(localStorage.getItem('user'))
   const [userComplaint, setUserComplaint] = useState({
      (property) name: string
      name: '',
      address: '',
      city: '',
      state: '',
      pincode: '',
      status: '',
      comment: ''})
   const handleChange = (e) => {
      const { name, value } = e.target
      setUserComplaint({ ...userComplaint, [name]: value })}
   const handleClear = () => {
      setUserComplaint({
         userId: '',
         name: '',
         address: '',
         city: '',
         state: '',
         pincode: '',
         status: '',
         comment: ''})}
   const handleSubmit = async (e) => {
      e.preventDefault()
      const user = JSON.parse(localStorage.getItem('user'))
      const { _id } = user
      try {
         await axios.post(`http://localhost:8000/Complaint/${_id}`, userComplaint)
         toast.success("Complaint submitted successfully!")
         handleClear()
      } catch (err) {
         toast.error("Failed to submit complaint. Please try again.")
         console.error(err)}}
   return (<>
         <div className="text-white complaint-box">
            <form onSubmit={handleSubmit} className="compliant-form row bg-dark ">

               <div className="col-md-6 p-3 p-3">    'p-3' applies the same CSS properties as 'p-3'.
                  <label htmlFor="name" className="form-label">Name</label>
                  <input name="name" onChange={handleChange} value={userComplaint.name} type="text" className="form-control" id="name" required /></div>
               <div className="col-md-6 p-3">
                  <label htmlFor="address" className="form-label">Address</label>
                  <input name="address" onChange={handleChange} value={userComplaint.address} type="text" className="form-control" id="address" required /></div>
               <div className="col-md-6 p-3">
                  <label htmlFor="city" className="form-label">City</label>
                  <input name="city" onChange={handleChange} value={userComplaint.city} type="text" className="form-control" id="city" required /></div>
               <div className="col-md-6 p-3">
                  <label htmlFor="state" className="form-label">State</label>
                  <input name="state" onChange={handleChange} value={userComplaint.state} type="text" className="form-control" id="state" required /></div>
               <div className="col-md-6 p-3">
                  <label htmlFor="pincode" className="form-label">Pincode</label>
                  <input name="pincode" onChange={handleChange} value={userComplaint.pincode} type="text" className="form-control" id="pincode" required /></div>
               <div className="col-md-6 p-3">
                  <label htmlFor="status" className="form-label">Status</label>
                  <input placeholder='pending' name="status" onChange={handleChange} value={userComplaint.status} type="text" className="form-control" id="pincode" required /></div>
               <label className=" p-3form-label text-light" htmlFor="comment">Description</label>
               <div className="form-floating">
                  <textarea name="comment" onChange={handleChange} value={userComplaint.comment} className="form-control" required></textarea></div>
               <div className="text-center p-1 col-12">
                  <button type="submit" onClick={handleSubmit} className="mt-2 btn btn-success">Register</button></div>
            </form>
         </div>
      </>)}
export default Complaint
```

Figure 3.5: User Complaint Component

3.5 Backend Implementation

Backend Structure:



3.5.1 Configuration file

This file is responsible for establishing and managing the connection to the MongoDB database. It contains the necessary configurations to ensure seamless communication between the application and the database, enabling data storage and retrieval for the complaint management system.

3.5.2 Source file

The index.js file serves as the core of the application, this file defines the main API routes and application logic. It handles incoming requests, processes business logic, and communicates with the database to facilitate functions such as complaint registration, user authentication, and status updates.

3.5.3 Schema file

This file contains the database schemas, which define the structure of the data in MongoDB. It includes the schemas for different entities such as users, complaints, assigned complaints, and messages, ensuring data consistency and integrity within the system.

3.6 API Endpoints

The API supports the following endpoints:

3.6.1 Register a New Complaint

Endpoint: `POST /complaint/:id`

Description: Register a new complaint.

Request Body:

```
{
  "userId": "637b3342dc8f242d0c8b4567",
  "name": "John Doe",
  "address": "123 Main Street",
  "city": "Chennai",
  "state": "TN",
  "pincode": 600064,
  "comment": "Issue with product delivery",
  "status": "Pending"
}
```

Response:

```
{
  "_id": "638e2345f23d7428a01e1234",
  "userId": "637b3342dc8f242d0c8b4567",
  "name": "John Doe",
  "address": "123 Main Street",
  "city": "Chennai",
  "state": "TN",
  "pincode": 600064,
  "comment": "Issue with product delivery",
  "status": "Pending",
  "__v": 0
}
```

If the user ID does not exist in the database::

```
{
```

```
  "error": "User not found"
}
```

If there's an issue saving the complaint to the database:

```
{
  "error": "Failed to register complaint"
}
```

3.6.2 Retrieve Complaint Details by ID

Endpoint: `GET /complaints/:id`

Description: Retrieve details of a complaint by its ID.

Response:

```
{
    "complaint_id": "67890",
    "user_id": "12345",
    "complaint_title": "Defective Product",
    "complaint_description": "The product I received was damaged.",
    "status": "Assigned to Agent",
    "created_at": "2024-11-20T12:00:00Z",
    "updated_at": "2024-11-20T14:00:00Z"
}
```

3.6.3 Update Complaint Status by ID

Endpoint: `PUT /complaints/:id`

Description: Update the status of a complaint.

Request Body:

```
{
    "status": "Resolved"
}
```

Response:

```
{
    "message": "Complaint status updated to Resolved"
}
```

3.6.4 Send a Message to the Complaint Agent

Endpoint: `POST /complaints/:id/messages`

Description: Send a message regarding the complaint to the assigned agent.

Request Body:

```
{
    "message": "Can you provide an update on the resolution?"
}
```

Response:

```
{
    "message": "Message sent successfully"
}
```

3.6.5 User Login

Endpoint: `POST /users/login`

Description: Authenticate a user with their credentials.

Request Body:

```
{
    "email": "user@example.com",
    "password": "securePassword123"
}
```

Response:

```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIy
lKxwRJSMeKKF2QT4fwpMeJf36POk6yV_adQssw5c"
}
```

3.7 Database Implementation

The backend of the Online Complaint Registration and Management System uses MongoDB as its database, with Mongoose for schema-based data modeling. Below are the details of the MongoDB connection and the schemas used in the system.

3.7.1 MongoDB Connection Configuration

The application connects to MongoDB using Mongoose. The connection string provided is to a MongoDB Atlas cluster, which is a cloud-based database service. The connection is configured as follows:

```
const mongoose = require("mongoose");
mongoose .connect( "mongodb+srv://balaganesh102004:¡password¿@cluster0.
nsgbb.mongodb.net/?retryWrites=truew=majorityappName=Cluster0",
useNewUrlParser: true, useUnifiedTopology:true )
.then(() => console.log("Connected to MongoDB"); )
.catch((err) => console.error("Failed to connect to MongoDB:", err); );
```

This configuration ensures that the application connects to the MongoDB database and handles potential connection errors. The `useNewUrlParser` and `useUnifiedTopology` options are used to avoid deprecation warnings in Mongoose.

3.7.2 User Schema

The `UserSchema` defines the structure for storing user data. The schema includes fields for the user's name, email, password, phone number, and user type. Here is the user schema definition:

```
const userSchema = mongoose.Schema(
name:  type: String, required: 'Name is required' ,
email:  type: String, required: 'Email is required' ,
password:  type: String, required: 'Password is required' ,
phone:  type: Number, required: 'Phone is required' ,
userType:  type: String, required: 'UserType is required' ,
,  timestamps: true, );
const UserSchema = mongoose.model("user_schema", userSchema);
```

This schema is used to store and validate user information. The `timestamps: true` option adds `createdAt` and `updatedAt` fields to each document.

3.7.3 Complaint Schema

The `ComplaintSchema` defines the structure for storing complaints submitted by users. It includes fields for the complaint's associated user, name, address, city, state, pincode, comment, and status. Below is the complaint schema:

```
const complaintSchema = mongoose.Schema(
userId: type: mongoose.Schema.Types.ObjectId,
required: true, ref: "user_schema",
name: type: String, required: true ,
address:  type: String, required: true ,
city:  type: String, required: true ,
state:  type: String, required: true ,
pincode:  type: Number, required: true ,
comment:  type: String, required: true ,
status:  type: String, required: true , );
const ComplaintSchema = mongoose.model("complaintschema", com-
plaintSchema);
```

The `userId` field is a reference to the `UserSchema`, establishing a relationship between users and their complaints.

3.7.4 Assigned Complaint Schema

The `AssignedComplaint` schema is used to store complaints assigned to agents. It includes fields for the agent's ID, complaint ID, status, and agent name. Below is the assigned complaint schema:

```
const assignedComplaint = mongoose.Schema(
agentId: type: mongoose.Schema.Types.ObjectId, required: true, ref: "user-
Schema" ,
complaintId: type: mongoose.Schema.Types.ObjectId, required: true, ref:
"complaintschema" ,
status: type: String, required: true , agentName: type: String, required:
true , );
const AssignedComplaint = mongoose.model("assignedcomplaint", as-
signedComplaint);
```

The `agentId` field references the `UserSchema`, linking the assigned complaint to an agent.

3.7.5 Message Schema

The `MessageSchema` defines the structure for storing messages sent between users and agents regarding complaints. It includes fields for the sender's name, the message content, and a reference to the assigned complaint. Below is the message schema:

```
const messageSchema = new mongoose.Schema( name: type: String, re-
quired: 'Name is required',
message: type: String, required: 'Message is required',
complaintId: type: mongoose.Schema.Types.ObjectId, ref: "assignedcom-
plaint" ,
timestamps: true );
const MessageSchema = mongoose.model('message', messageSchema);
```

The `complaintId` field references the `AssignedComplaint` schema, linking messages to a specific complaint.

3.7.6 Schema Overview

The use of Mongoose schemas ensures that the data structure is consistent and validated throughout the application. Each schema is designed to capture the necessary information for managing users, complaints, assigned agents, and messages. By leveraging MongoDB's flexibility and Mongoose's schema validation, the system is able to maintain data integrity while offering scalability and ease of querying.

3.8 Schema Models Export

The schemas are exported from the `schema.js` file to be used in other parts of the application. The following code snippet shows the exported models:

```
module.exports = UserSchema, ComplaintSchema, AssignedComplaint,
MessageSchema, ;
```

These models can now be imported into other files for querying, updating, and manipulating data.

Conclusion

4.1 Summary

The **Online Complaint Registration and Management System (OCRMS)** streamlines complaint registration and resolution, reducing manual effort and enhancing transparency. With features like real-time updates, user authentication, and complaint prioritization, the platform ensures efficiency and user satisfaction. Built on the scalable MERN stack, OCRMS is a versatile and reliable solution suitable for diverse applications.

4.2 Recommendations

To further enhance the OCRMS and meet evolving user demands, future iterations of the system could incorporate the following improvements:

* **Multi-language Support:** Adding multi-language functionality would make the platform more accessible and user-friendly for diverse audiences.
* **AI-based Categorization and Routing:** Integrating AI can automate complaint classification and routing, improving efficiency and response times.
* **Enhanced Security Features:** Implementing measures like two-factor authentication and data encryption would strengthen user data protection.

Incorporating these enhancements will make OCRMS more inclusive, efficient, and secure, elevating it to a cutting-edge complaint management solution for diverse applications.

Supplementary Materials

A.1 ER Diagram

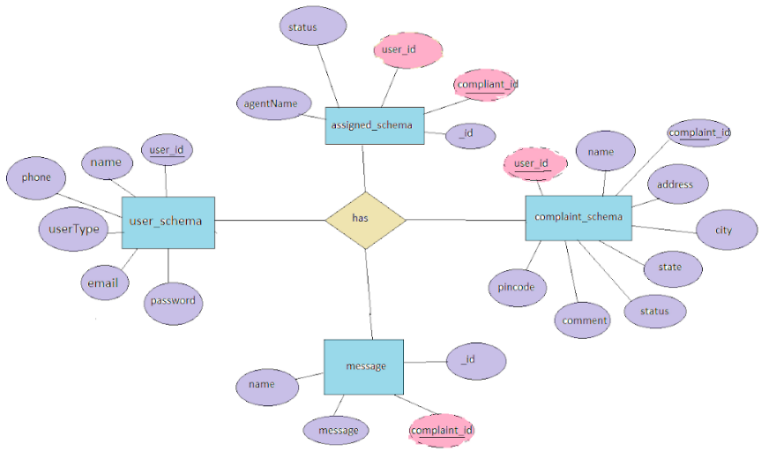


Figure A.1: Entity-Relationship Diagram

A.2 Output Screenshots

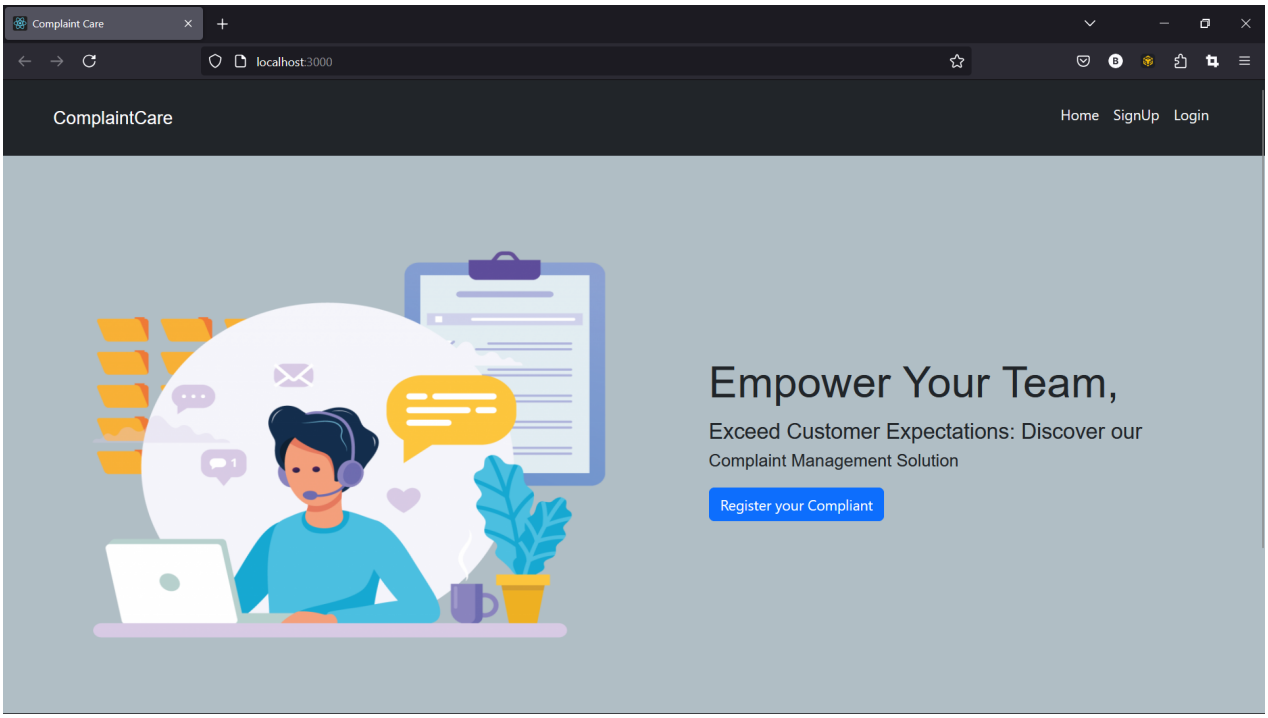


Figure A.2: Landing page

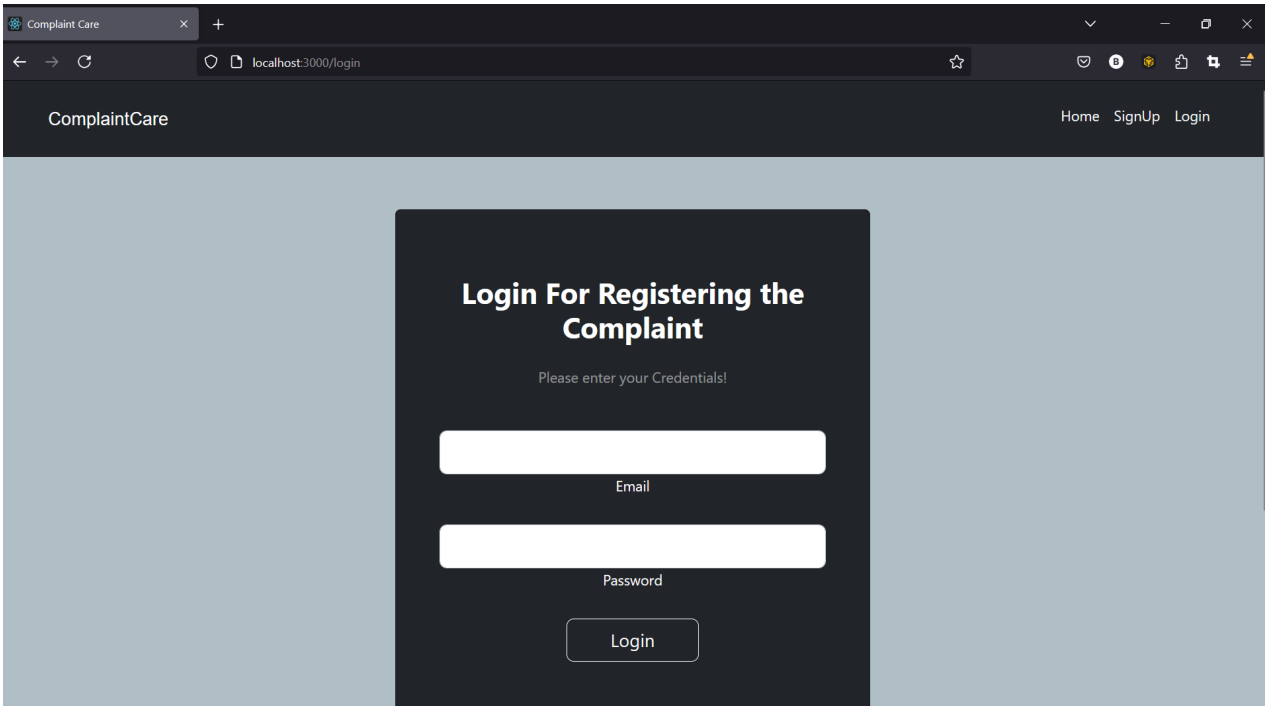


Figure A.3: Login page

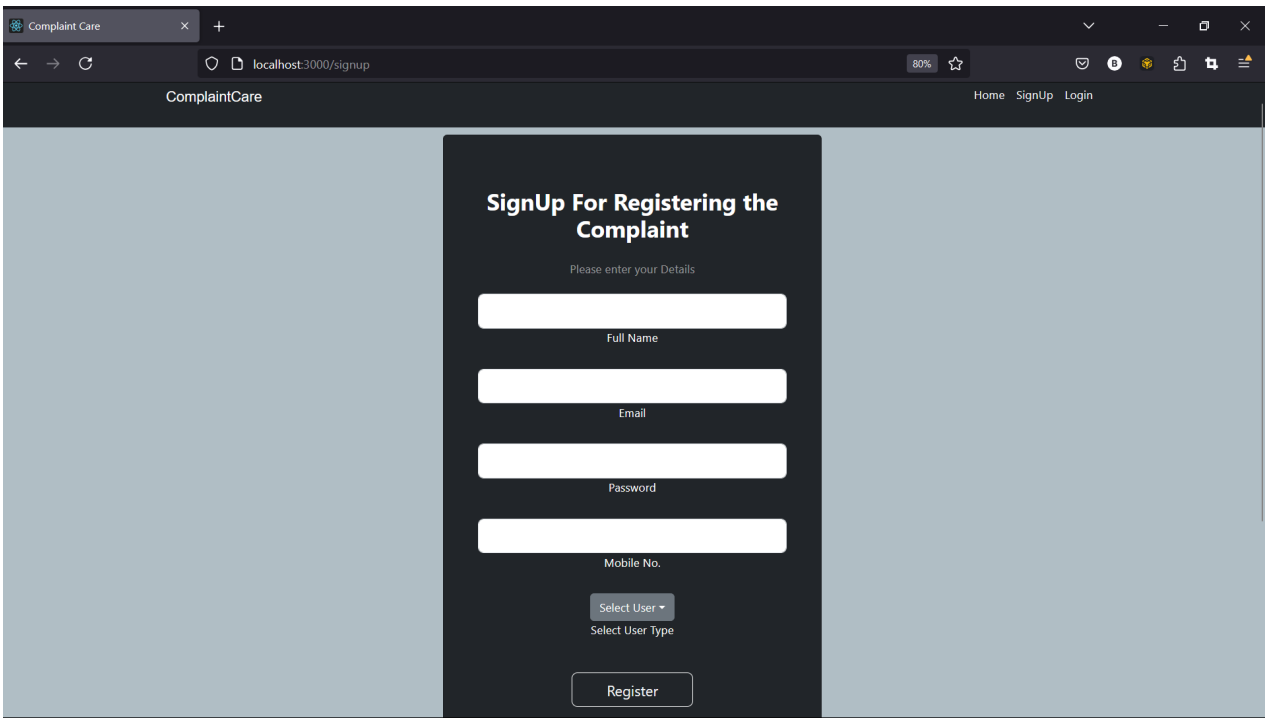


Figure A.4: Signup page

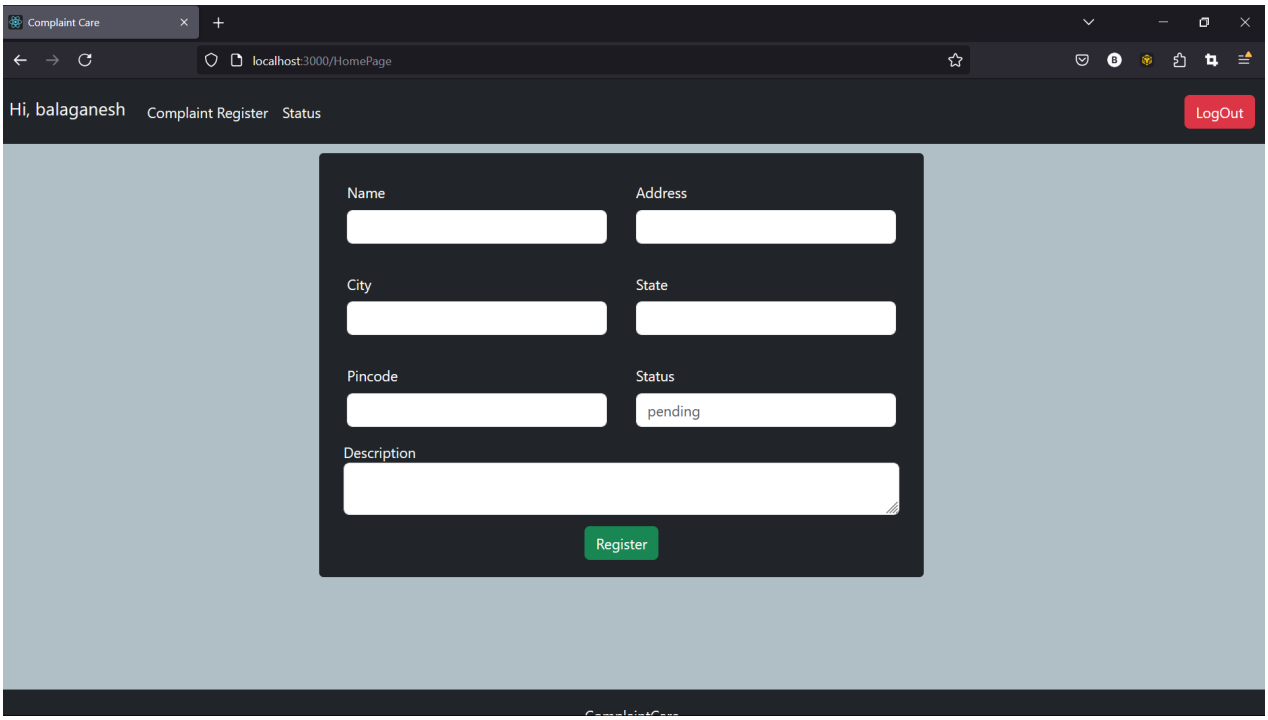


Figure A.5: Complaint Page page

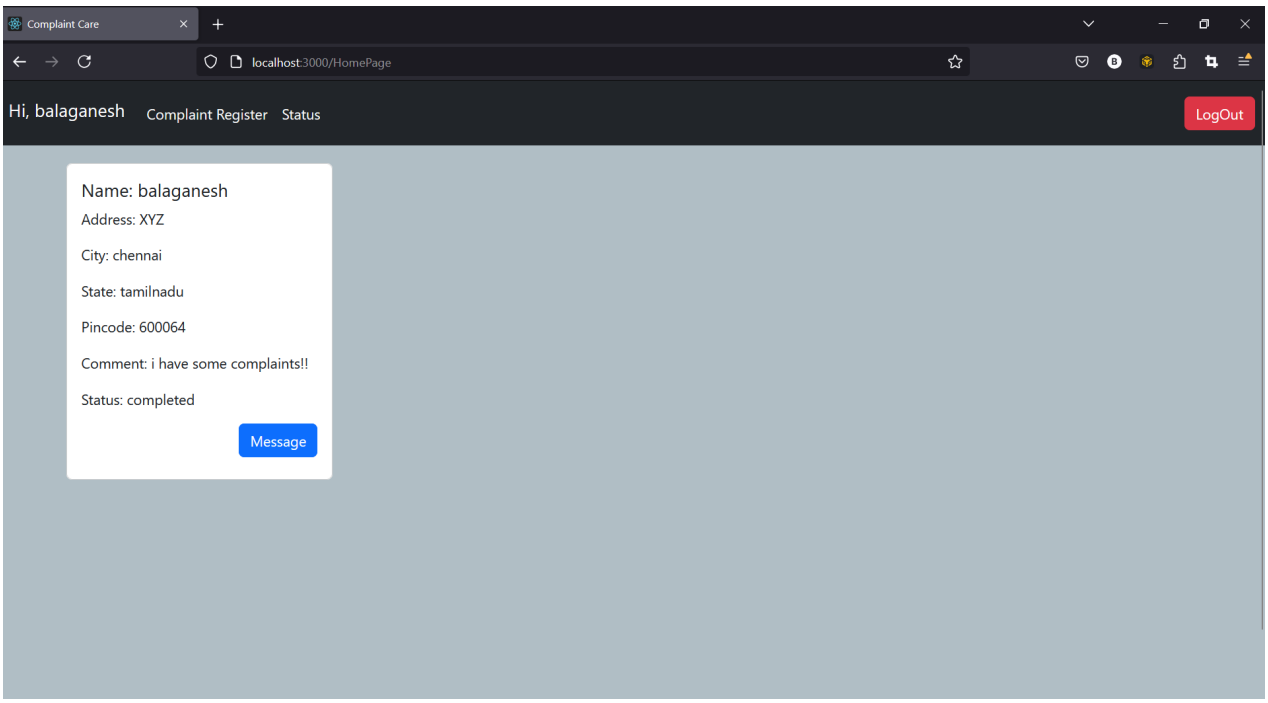


Figure A.6: Complaint Status page

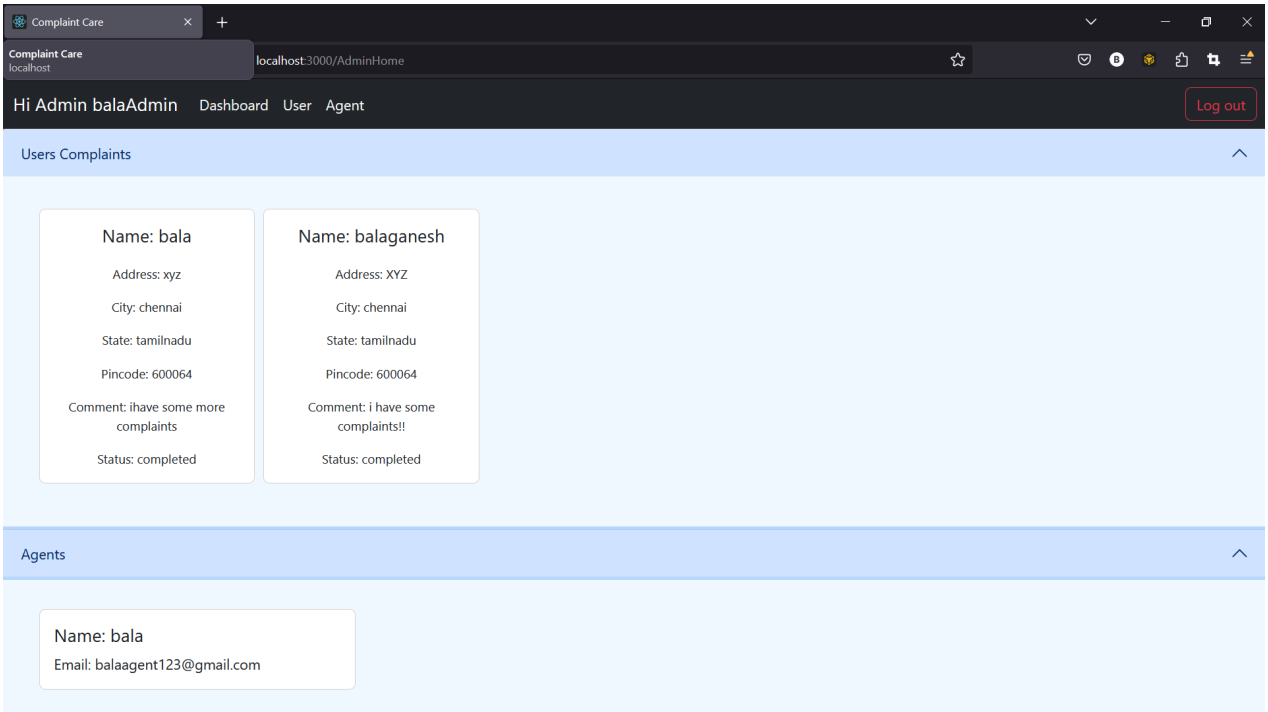


Figure A.7: Admin Dashboard page

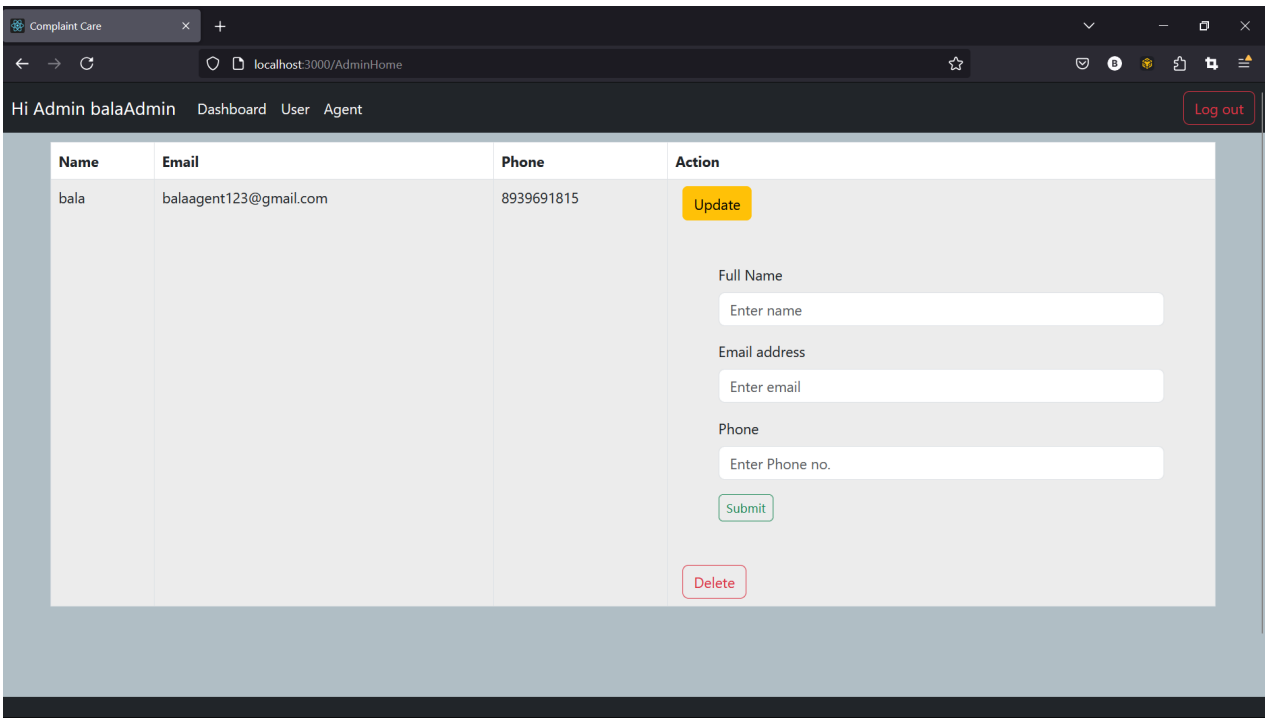


Figure A.8: Admin operations page

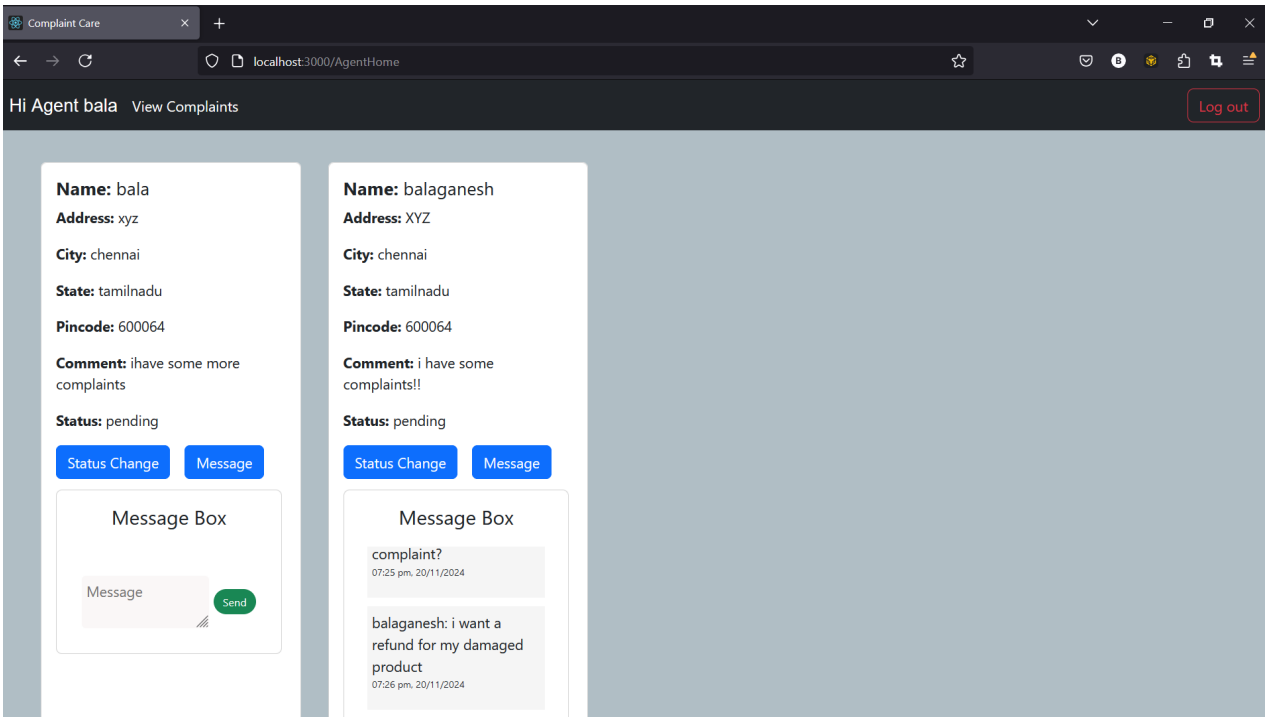


Figure A.9: Agent Dashboard page