**Online Complaint Registration and Management System**

**INTRODUCTION**

**Project Title:**

Online Complaint Registration and Management System

**Team Members:**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **NAME** | **ROLE** | **RESPONSIBILITIES** |
| **1.** | **PRASATH M** | **Full-Stack**  **Developer** | **Responsible for overall development,Including front-end, back-end, server-side logic, and database design** |
| **2.** | **PRIYANGA K** | **Backend**  **Developer** | **Responsible for Express.js setup and API development.** |
| **3.** | **PREM KUMAR B** | **Frontend**  **Developer** | **Responsible for UI/UX design using React, Material UI, and Bootstrap.** |
| **4.** | **PRAVEEN BHARATHI A** | **Database**  **Administrator** | **Responsible for MongoDB setup and ensuring data integrity.** |

**Project Overview**

The **Online Complaint Registration and Management System** is a comprehensive solution designed to streamline the process of registering, tracking, and resolving complaints for individuals or organizations. By leveraging modern web technologies, the system provides a centralized platform that ensures efficient complaint handling, enhancing customer satisfaction and operational efficiency.

**Purpose:**

The system's primary goal is to digitize the complaint management process, providing users with real-time updates and seamless communication with assigned agents. It eliminates manual processes, ensures transparency, and improves response times.

**Key Features:**

* User Registration and Authentication**:** Secure access for users to register and log complaints.
* Complaint Management: Users can submit detailed complaints, attach relevant files, and track resolution progress.
* Real-time Updates: Notifications are sent to users for complaint status changes and agent interactions.
* Agent Interaction: Built-in messaging system for users to communicate with assigned agents for clarification or updates.
* Admin Oversight: Comprehensive admin dashboard to monitor complaints, assign tasks, and manage platform operations.

**Technology Stack:**

* **Frontend:** React.js with Material-UI and Bootstrap for a responsive, user-friendly interface.
* **Backend:** Node.js and Express.js for server-side logic and API management.
* **Database:** MongoDB for scalable and efficient data storage.

**System Benefits:**

1. **Enhanced Efficiency:** Reduces manual effort and automates key complaint management processes.
2. **Transparency:** Provides users with real-time updates on complaint progress, fostering trust.
3. **Scalability:** Built with modern technologies to handle growing numbers of users and complaints.
4. **Security:** Implements robust measures such as encrypted data storage and secure authentication protocols.

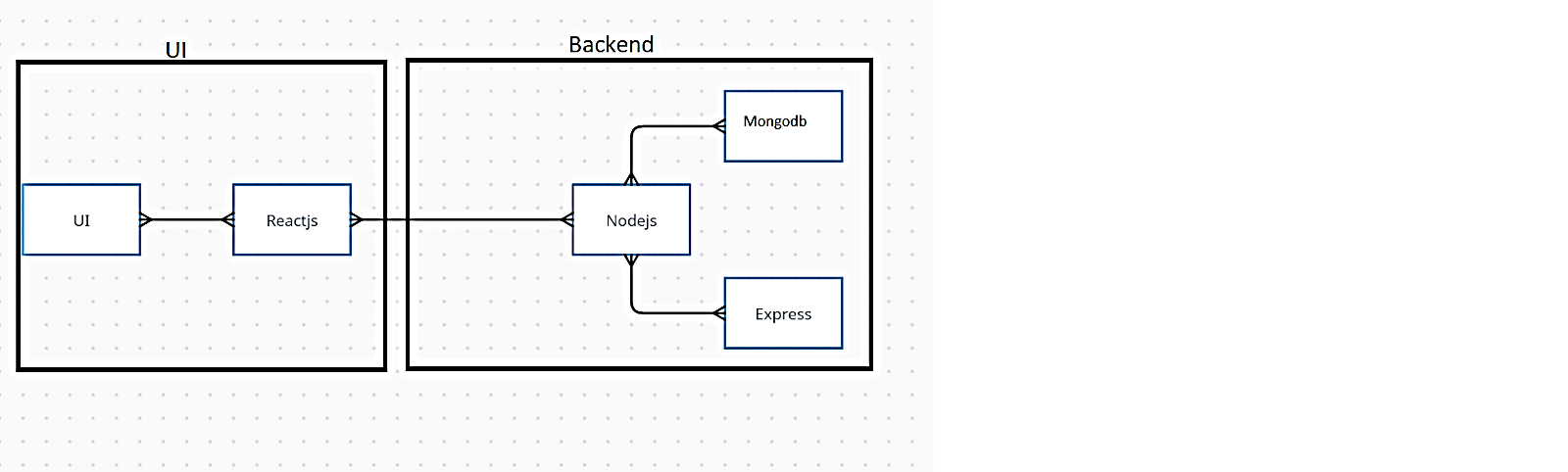
**Use Case Example:**

**Scenario:** John, a customer, encounters a defective product. Using the platform, he:

1. **Registers and logs in** to the system.
2. **Submits a complaint** with details and attaches photos of the defect.
3. **Tracks updates** as the complaint is assigned to an agent.
4. **Communicates** with the agent through the chat system to resolve the issue.
5. Receives a **notification** of the resolution and provides feedback.

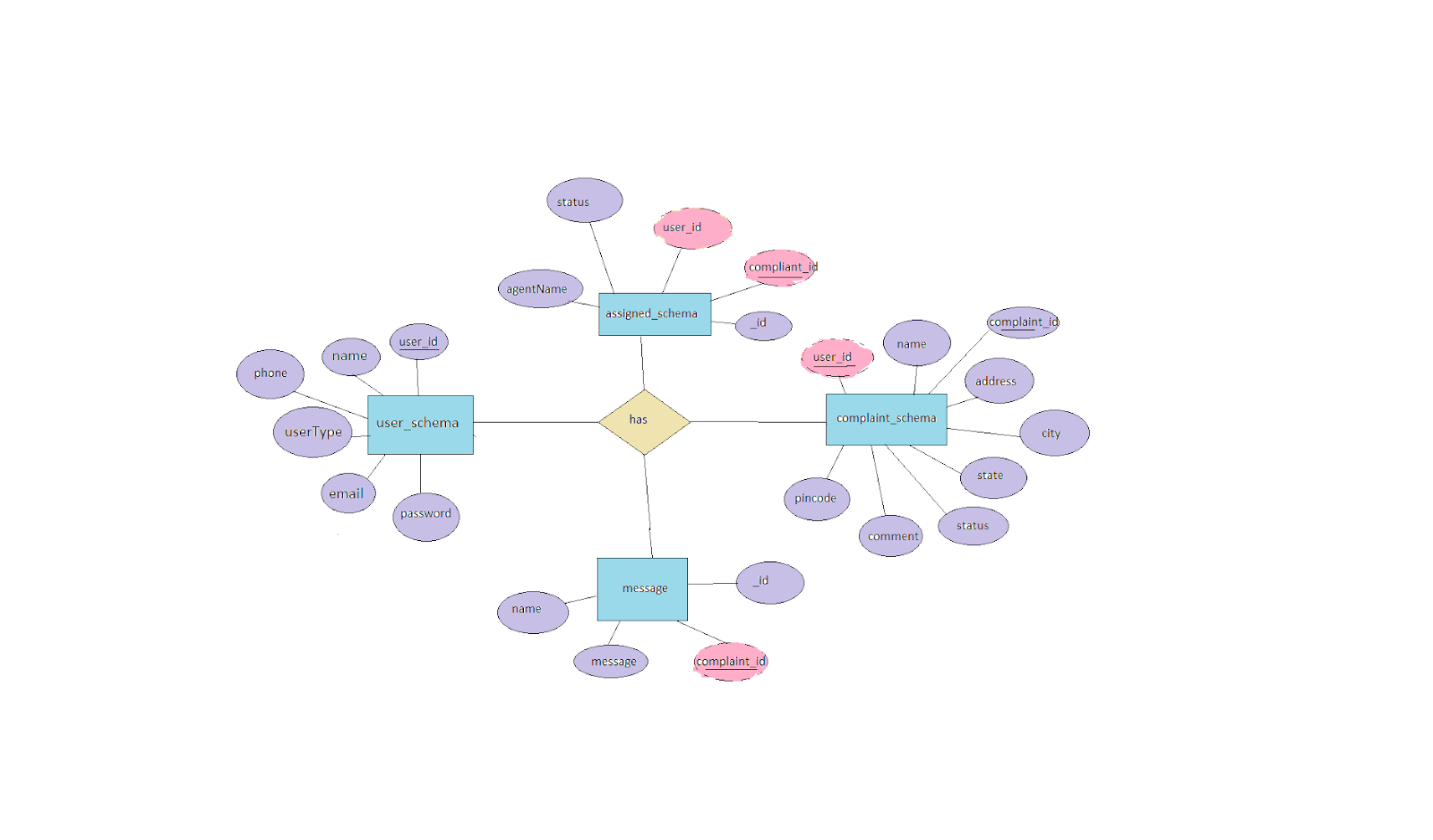
This project empowers users to manage complaints efficiently, ensures high data security, and promotes transparency in complaint handling, making it an essential tool for organizations aiming to improve customer service and streamline internal processes.

**ARCHITECTURE**



* The technical architecture of our online complaint registration and management app follows a client-server model, where the frontend serves as the client and the backend acts as the server. The frontend encompasses not only the user interface and presentation but also incorporates the axios library to connect with backend easily by using RESTful Apis.
* The frontend utilizes the bootstrap and material UI library to establish real-time and better UI experience for any user whether it is agent, admin or ordinary user working on it.
* On the backend side, we employ Express.js frameworks to handle the server-side logic and communication.
* For data storage and retrieval, our backend relies on MongoDB. MongoDB allows for efficient and scalable storage of user data, including user profiles, for complaints registration, etc. It ensures reliable and quick access to the necessary information during registration of user or any complaints.
* Together, the frontend and backend components, along with socket.io, Express.js, WebRTC API, and MongoDB, form a comprehensive technical architecture for our video conference app. This architecture enables real-time communication, efficient data exchange, and seamless integration, ensuring a smooth and immersive video conferencing experience for all users.

**ER DIAGRAM:**



This is the er diagram of the project which shows the relationship between user and the agent.

**SETUP INSTRUCTIONS**

**PRE-REQUISITES:**

Here are the key prerequisites for developing a full-stack application using Node.js, Express.js, MongoDB, and React.js:

**Node.js and npm:**

Node.js is a powerful JavaScript runtime environment that allows you to run JavaScript code on the server-side. It provides a scalable and efficient platform for building network applications.

Install Node.js and npm on your development machine, as they are required to run JavaScript on the server-side.

Download: https://nodejs.org/en/download/

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

Express.js:

Express.js is a fast and minimalist web application framework for Node.js. It simplifies the process of creating robust APIs and web applications, offering features like routing, middleware support, and modular architecture.

Install Express.js, a web application framework for Node.js, which handles server-side routing, middleware, and API development.

Installation: Open your command prompt or terminal and run the following command:

 npm install express

MongoDB:

MongoDB is a flexible and scalable NoSQL database that stores data in a JSON-like format. It provides high performance, horizontal scalability, and seamless integration with Node.js, making it ideal for handling large amounts of structured and unstructured data.

Set up a MongoDB database to store your application's data.

Download: https://www.mongodb.com/try/download/community

Installation instructions: https://docs.mongodb.com/manual/installation/

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.

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. To Connect the Database with Node JS go through the below provided link:

https://www.section.io/engineering-education/nodejs- mongoosejs-mongodb/

Front-end Framework: Utilize Reactjs to build the user-facing part of the application, including entering complaints, status of the complaints, and user interfaces for the admin dashboard.

For making better UI we have also used some libraries like material UI and boostrap.

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>

To run the existing Video Conference App project downloaded from GitHub:

Follow below steps:

Clone the Repository:

* Open your terminal or command prompt.
* Navigate to the directory where you want to store the e-commerce app.
* Execute the following command to clone the repository:

git clone: <https://github.com/awdhesh-student/complaint-registery.git>

Install Dependencies:

• Navigate into the cloned repository directory:

cd complaint-registry

• Install the required dependencies by running the following commands:

cd frontend

npm install

cd ../backend

npm install

Start the Development Server:

• To start the development server, execute the following command:

npm start, The online complaint registration and management app will be accessible at <http://localhost:3000>

**FOLDER STRUCTURE**

**Client (React Frontend):** The frontend is built using React.js and follows a modular structure to ensure scalability and maintainability.

The first image is of frontend part which is showing all the files and folders that have been used in UI development

**Server (Node.js Backend):** The second image is of Backend part which is showing all the files and folders that have been used in backend development

**Running the Application**

To run the application locally, follow the steps below to start both the frontend and backend servers:

**Frontend:**

Navigate to the client directory:

**cd client**

Install the dependencies (if not already done):

**npm install**

Start the frontend server:

**npm start**

The application will be accessible at:

arduino

**http://localhost:3000**

**Backend:**

Navigate to the server directory:

**cd server**

Install the dependencies (if not already done):

**npm install**

Start the backend server:

**npm start**

By default, the backend will run on:

arduino

**http://localhost:5000**

**API Documentation**

**BASE URL:**

http://localhost:5000/api

**1. User Registration & Authentication**

POST /auth/register

* Description: Register a new user.
* Request Body:

{

"name": "John Doe",

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

"password": "password123"

}

* Response:

{

"message": "User registered successfully",

"user": {

"id": "user\_id",

"name": "John Doe",

"email": "john@example.com"

}

}

POST /auth/login

* Description: Login an existing user.
* Request Body:

json

{

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

"password": "password123"

}

* Response:

{

"message": "Login successful",

"token": "jwt\_token\_here"

}

**2. Complaint Management**

POST /complaints

* Description: Create a new complaint.
* Request Body:

json

{

"userId": "user\_id",

"title": "Defective Product",

"description": "The product I received is defective.",

"productImage": "base64encodedimage",

"address": "123 Main St, City, State",

"city": "CityName",

"state": "StateName",

"pincode": "123456"

}

* Response:

{

"message": "Complaint created successfully",

"complaint": {

"id": "complaint\_id",

"userId": "user\_id",

"title": "Defective Product",

"description": "The product I received is defective.",

"status": "Pending"

}

}

GET /complaints/

* Description: Get details of a specific complaint.
* URL Parameters:

id (string): Complaint ID.

* Response:

{

"complaint": {

"id": "complaint\_id",

"userId": "user\_id",

"title": "Defective Product",

"description": "The product I received is defective.",

"status": "Pending",

"createdAt": "2024-11-01T10:00:00Z"

}

}

GET /complaints

* Description: Get all complaints for a user.
* Query Parameters:
  + userId (string): User ID to filter complaints by user.
* Response:

{

"complaints": [

{

"id": "complaint\_id",

"title": "Defective Product",

"status": "Resolved"

},

{

"id": "complaint\_id\_2",

"title": "Late Delivery",

"status": "Pending"

}

]

}

PUT /complaints/

* Description: Update a complaint (e.g., status).
* URL Parameters:
  + id (string): Complaint ID.
* Request Body:

{

"status": "Resolved",

"assignedTo": "agent\_id"

}

* Response:

json

Copy code

{

"message": "Complaint status updated successfully",

"complaint": {

"id": "complaint\_id",

"status": "Resolved",

"assignedTo": "agent\_id"

}

}

**3. Chat/Interaction Between User and Agent**

POST /messages

* Description: Send a message in the chat between user and agent regarding a complaint.
* Request Body:

{

"userId": "user\_id",

"complaintId": "complaint\_id",

"message": "I have more details regarding the issue."

}

* Response:

{

"message": "Message sent successfully",

"chat": {

"id": "message\_id",

"userId": "user\_id",

"complaintId": "complaint\_id",

"message": "I have more details regarding the issue.",

"timestamp": "2024-11-16T14:00:00Z"

}

}

GET /messages/

* Description: Get all messages related to a specific complaint.
* URL Parameters:

complaintId (string): Complaint ID.

* Response:

{ "messages": [

{

"id": "message\_id\_1",

"userId": "user\_id",

"message": "I have more details regarding the issue.",

"timestamp": "2024-11-16T14:00:00Z"

},

{

"id": "message\_id\_2",

"userId": "agent\_id",

"message": "Please provide your contact details.",

"timestamp": "2024-11-16T14:05:00Z"

}

]

}

**4. Admin Management**

GET /admin/complaints

* Description: Get all complaints for the admin.
* Response:

{

"complaints": [

{

"id": "complaint\_id\_1",

"userId": "user\_id",

"status": "Pending"

},

{

"id": "complaint\_id\_2",

"userId": "user\_id\_2",

"status": "Resolved"

}

]

}

POST /admin/assignComplaint

* Description: Admin assigns a complaint to an agent.
* Request Body:

json

Copy code

{

"complaintId": "complaint\_id",

"agentId": "agent\_id"

}

"message": "Complaint assigned to agent successfully.

**Authentication**

**Authentication** is the process of verifying the identity of a user. In this system, a user must be registered and logged in using their credentials (email and password) to access protected routes (e.g., complaint creation, viewing complaints, etc.).

**1. Registration (Sign-up)**

* Users provide their details (name, email, password) during registration.
* **Passwords are hashed** using a secure hashing algorithm (e.g., bcrypt) before storing them in the database to ensure security.
* After successful registration, the user’s details (excluding the password) are saved in the database.

**2. Login**

* After registration, users can log in with their credentials (email and password).
* The system verifies the credentials:
  + **Email**: It is checked against the database to see if the user exists.
  + **Password**: The system hashes the entered password and compares it with the stored hashed password in the database.
* Upon successful verification, the system generates a **JWT token** (which acts as the authentication token).
* This token is returned to the client (e.g., frontend), which stores it (usually in **localStorage** or as an **HTTP-only cookie**).

**3. JWT (JSON Web Token)**

* A **JWT** is a secure and compact way of representing user identity.
* The token is **signed** and **encoded** to ensure that the data cannot be tampered with.
* The token contains user-specific information (e.g., user ID, role), and is used to authenticate the user for subsequent requests to protected routes.

**Example JWT structure**:

json

Copy code

{

"sub": "user\_id", // User ID

"name": "John Doe", // User's Name

"role": "user", // User's Role (user, admin)

"iat": 1613148304, // Issued at time (Unix timestamp)

"exp": 1613151904 // Expiry time (Unix timestamp)

}

**4. JWT Usage**

* After logging in, the frontend stores the token (usually in **localStorage** or **HTTP-only cookie**) to include in the headers of all subsequent requests.
* When making requests to protected API routes, the frontend sends the Authorization header with the JWT token:

bash

Copy code

Authorization: Bearer <JWT\_TOKEN>

* The backend verifies the token in the request header. If the token is valid, the backend allows access to the protected route.

**Authorization**

**Authorization** refers to the process of determining what actions an authenticated user is allowed to perform, based on their role or permissions.

**1. Role-Based Access Control (RBAC)**

In this system, we implement **role-based access control** (RBAC), where each user is assigned a specific role (e.g., "user", "admin"). Different roles have different levels of access.

* **User Roles**:
  + **User**: Regular users can register complaints, view their complaints, and interact with agents.
  + **Admin**: Admins have more privileges, such as managing all complaints, assigning complaints to agents, and interacting with any user.

**2. Role Management**

* Each user has a **role** assigned when they register (e.g., a user might be assigned the "user" role by default, or "admin" can manually assign roles).
* During authentication, the user's role is included in the **JWT payload** (as shown in the token structure above).
* The backend verifies the role by inspecting the JWT token’s payload, ensuring that the user has the appropriate role to access a given route.
* **3. Access Control for Routes**
* **Protected Routes**: Routes like complaint creation, viewing all complaints, or chat messages require the user to be authenticated via JWT.
* **Admin Routes**: Only users with an **admin** role can access routes that involve managing all complaints, assigning agents, etc.

**Examples**:

* **User Route** (accessible by "user" role):
  + GET /complaints - View user’s own complaints.
  + POST /complaints - Register a new complaint.
* **Admin Route** (accessible by "admin" role):
  + GET /admin/complaints - View all complaints.
  + POST /admin/assignComplaint - Assign complaints to agents.

The server performs role checks before allowing access to a specific route:

* The **JWT token** is decoded, and the user’s role is checked.
* If the role matches the required permission for the route, the user is authorized to proceed. Otherwise, an error message is returned (e.g., **403 Forbidden**).

**Token Expiry and Refresh**

**1. Token Expiry**

* JWT tokens have an **expiry time** (exp) set when they are created. After this time, the token becomes invalid.

**2. Refresh Token**

* If a user’s token expires, they can use a **refresh token** (issued alongside the JWT) to request a new JWT without having to log in again. The refresh token has a longer expiry and can be stored in the browser.

**Refresh Token Workflow**:

1. When a user logs in, both a JWT and a refresh token are issued.
2. The **JWT** is used to authenticate API requests until it expires.
3. Once the JWT expires, the frontend uses the **refresh token** to request a new JWT from the backend.
4. The backend verifies the refresh token, and if valid, issues a new JWT.

**Session Management:**

* For this system, **sessions are not directly used** (because JWTs are stateless), but an optional session-based authentication method could be used in combination with JWT for added security (e.g., storing the JWT in a session cookie).
* If session management were to be added:
  + A **session ID** would be stored on the server and associated with the user’s JWT.
  + The session ID would be stored in a cookie, and the server would use it to manage the user’s authenticated session.

**Security Considerations:**

* **Password Hashing**: Use a strong hashing algorithm like **bcrypt** to securely store passwords.
* **HTTPS**: Always use **HTTPS** to protect tokens and other sensitive data in transit.
* **JWT Storage**: Store JWTs securely in **localStorage** or **sessionStorage**. Avoid storing them in regular cookies unless using **HTTP-only cookies** to prevent XSS attacks.
* **Token Expiry**: JWT tokens should have a short expiry time to limit potential abuse.
* **Refresh Tokens**: Refresh tokens should be stored securely, and care should be taken to prevent abuse by using proper validation and rotating tokens.

**Authentication and Authorization Flow Summary**

1. **User Registration**:
   * User provides their information (email, password) and is registered. Passwords are hashed, and a new user is saved.
2. **Login**:
   * User logs in by providing their email and password. The system validates the credentials and generates a JWT token with the user's role and other details.
3. **JWT Token**:
   * The frontend stores the JWT token and includes it in the Authorization header of future API requests.
4. **Authorization**:
   * Backend checks the role of the user using the JWT token to ensure the user has access to the requested route.
5. **Token Expiry & Refresh**:
   * When the JWT expires, the frontend uses a refresh token to get a new JWT without requiring the user to log in again.

**User Interface**

The **Online Complaint Registration and Management System** has a well-designed and user-friendly interface, ensuring seamless navigation for users, agents, and administrators. Below are the major sections of the UI with descriptions:

**1. Login Page**

* **Purpose:** Allows users, agents, and admins to authenticate themselves using their credentials.
* **Features:**
  + Email and password fields.
  + "Forgot Password" link for recovery.
  + "Sign Up" option for new users.
* **Design:** A clean and responsive layout using Material-UI components.

**2. Registration Page**

* **Purpose:** Enables new users to register on the platform.
* **Features:**
  + Input fields for name, email, password, and contact details.
  + Validation messages for incorrect or missing input.
  + Terms and Conditions checkbox.

**3. User Dashboard**

* **Purpose:** Displays complaint details and allows users to interact with the system.
* **Features:**
  + View the status of registered complaints.
  + Submit a new complaint with an easy-to-use form.

**4.Admin Dashboard**

* **Purpose:** Provides an overview of all system activities for administrators.
* **Features:**
  + List of all registered complaints with their statuses.
  + User and agent management tools.
  + Complaint assignment functionality.
  + Reports and analytics on complaint trends.

**5. Agent Interface**

* **Purpose:** Enables agents to manage and resolve complaints assigned to them.
* **Features:**
  + List of assigned complaints with filtering and sorting options.
  + Chat system to communicate with users.
  + Ability to update complaint statuses.

**6. Complaint Submission Page**

* **Purpose:** Allows users to submit new complaints.
* **Features:**
  + Input fields for complaint title, description, and supporting documents.
  + Dropdowns for category and urgency level.
  + Confirmation message upon successful submission.

**7. Real-time Chat Window**

* **Purpose:** Facilitates direct communication between users and agents.
* **Features:**
  + Chat history for each complaint.
  + Input box for messages.
  + Real-time notifications for new messages.

**Design Principles**

* **Responsive Design:** Optimized for desktops, tablets, and mobile devices.
* **Consistent Theme:** Uses Material-UI for a unified look and feel.

**Testing Strategy and Tools**

Testing is essential to ensure the functionality, reliability, and security of the system. The strategy includes unit tests, integration tests, end-to-end (E2E) tests, security testing, performance testing, and user acceptance testing (UAT) to validate both individual components and the overall system.

**1. Unit Testing**

**Objective:** Verify individual functions or methods in isolation to ensure correctness.  
**Tools Used:**

* **Jest**: A popular JavaScript testing framework for unit tests.
* **Mocha & Chai**: Alternative tools for testing and assertions in Node.js.

**2. Integration Testing**

**Objective:** Test the interaction between different system components (e.g., APIs, database).  
**Tools Used:**

* **Supertest**: Used for testing REST APIs with HTTP assertions.
* **Database Mocks**: Tools like mock-knex or mongoose simulate database interactions to test components that rely on a database.

**3. End-to-End (E2E) Testing**

**Objective:** Simulate real user interactions to test the entire system, from front-end to back-end.  
**Tools Used:**

* **Cypress**: A tool for automating browser interactions to test the whole application.
* **Selenium & Playwright**: Other alternatives for browser-based E2E testing.

**4. Security Testing**

**Objective:** Identify vulnerabilities such as SQL injection, XSS, and CSRF.  
**Tools Used:**

* **OWASP ZAP**: A security testing tool to identify vulnerabilities.
* **Burp Suite**: A tool for intercepting web traffic and testing for security flaws.

**5. Performance Testing**

**Objective:** Assess the system's ability to handle large traffic loads and high user volumes.  
**Tools Used:**

* **JMeter**: A load testing tool for measuring API performance.
* **Artillery**: A modern tool for simulating high traffic and load.

**6. Continuous Integration and Automation**

**Objective:** Automate tests to run with every code change to maintain high-quality code.  
**Tools Used:**

* **GitHub Actions / Jenkins**: Continuous integration tools that automate the testing process to ensure smooth deployments.

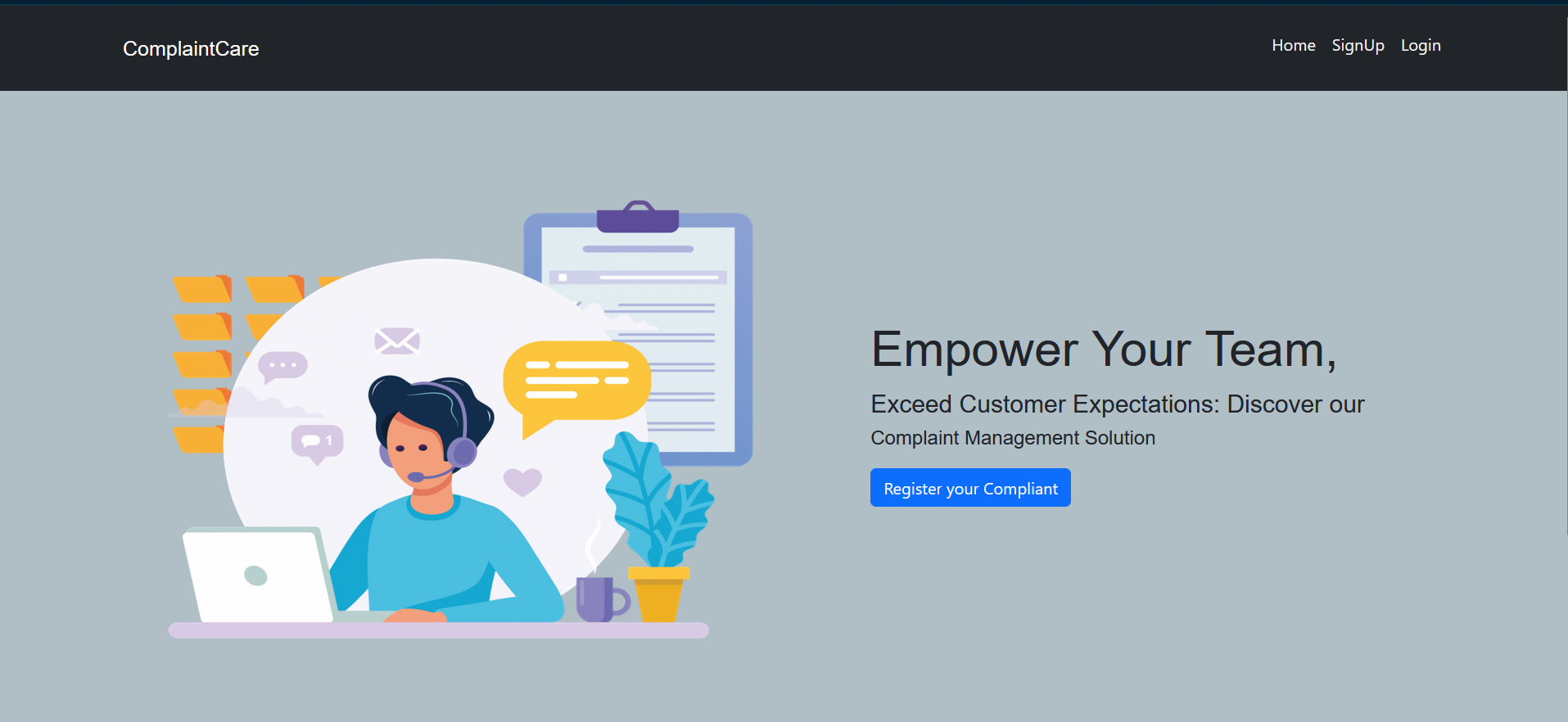
**7. User Acceptance Testing (UAT)**

**Objective:** Validate that the system meets the business requirements and user expectations.  
**Tools Used:**

* **TestRail**: A test management tool for tracking UAT progress.
* **Manual Testing**: Users test the application in real-world scenarios to ensure it functions as expected.

**Screenshots or Demos**

* Landing Page



* Login Page



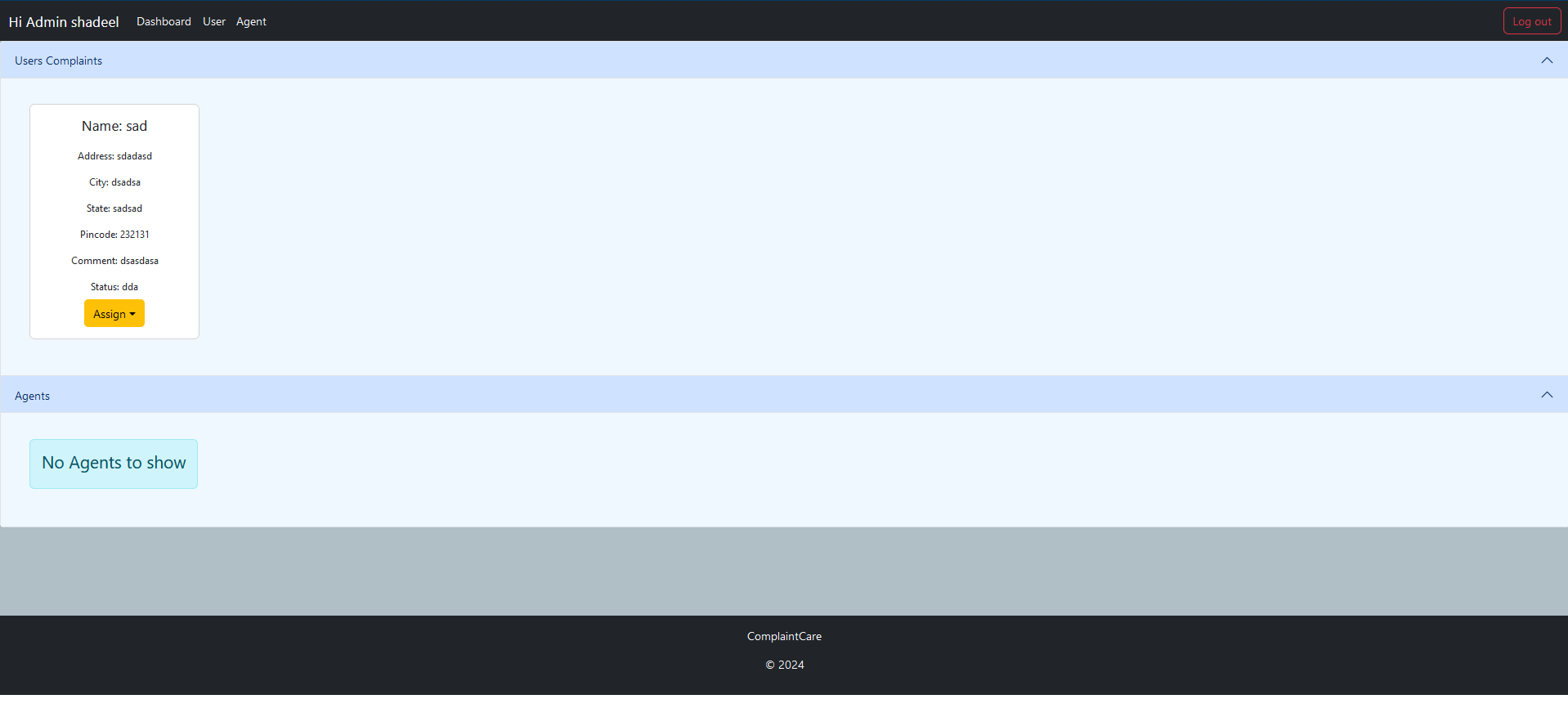
* Registration Page



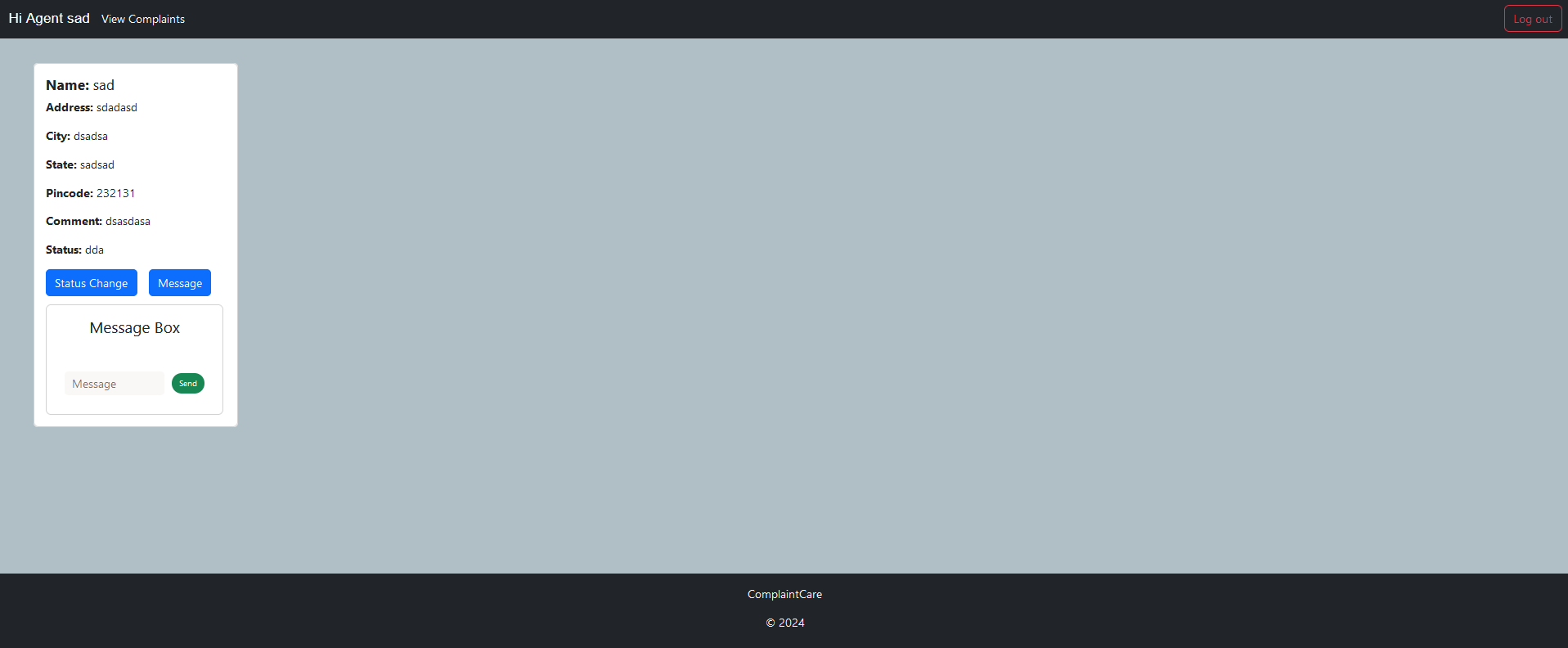
* Common Dashboard For Complaint



* Admin Dashboard



* Agent Dashboard



**Known Issues**

While the system is designed to be robust and reliable, there are a few known bugs or limitations that developers and users should be aware of:

1. **Session Timeout for JWT:**

**Issue:** The JWT expiration mechanism occasionally leads to a delay in session expiration after a user logs out.

**Impact:** Users may remain logged in for a brief period even after logging out, causing potential confusion or security concerns.

**Current Status:** The issue is under investigation, and a fix is scheduled for the next update.

**Database Synchronization:**

**Issue:** In certain circumstances, the database schema does not sync immediately with the application during deployment.

**Impact:** This can lead to temporary discrepancies, such as missing data or mismatched fields.

**Current Status:** Manual schema migration is required temporarily until the automation script is updated.

**File Upload Size Limitation:**

**Issue:** The system restricts file uploads to 5 MB, but larger files may occasionally cause unexpected errors.

**Impact:** Users attempting to upload larger files may experience failures.

**Current Status:** The limit will be increased in the next version to handle larger files (up to 10 MB).

**Slow Load Times on Admin Dashboard:**

**Issue:** The admin dashboard sometimes experiences slow load times when displaying a large number of complaints.

**Future Enhancements**

While the current system meets the needs of users, there are several potential features and improvements that could further enhance its functionality:

1. **Role-Based Permissions Management:**
   * **Enhancement:** Introduce a more granular permission management system to allow admins to assign specific rights to users based on their roles (e.g., view-only, editing rights, etc.).
   * **Benefit:** More control over user interactions and finer control over data access.
2. **Mobile App Integration:**
   * **Enhancement:** Develop a mobile application for iOS and Android that allows users to access the system on-the-go.
   * **Benefit:** Users will be able to submit complaints, track their status, and communicate with agents via mobile devices, increasing accessibility and engagement.
3. **Multi-Language Support:**
   * **Enhancement:** Add support for multiple languages to the system to cater to a global audience.
   * **Benefit:** The application will be accessible to a broader user base, improving usability for non-English speakers.
4. **Enhanced Reporting & Analytics:**
   * **Enhancement:** Implement advanced reporting features with data visualizations to provide detailed insights into complaints, resolution times, agent performance, and more.
   * **Benefit:** This will empower admins and users with better data to make informed decisions.
5. **AI-Powered Complaint Classification:**
   * **Enhancement:** Incorporate machine learning algorithms to automatically classify and prioritize complaints based on urgency, keywords, and historical data.
   * **Benefit:** More efficient routing and quicker responses to critical complaints.
6. **Real-Time Notifications:**
   * **Enhancement:** Add real-time notifications for users and admins about complaint status changes, new assignments, or messages from agents.
   * **Benefit:** Ensures users and administrators are always up to date, improving the overall user experience and responsiveness.
7. **Integration with External Systems:**
   * **Enhancement:** Develop integrations with third-party CRM and helpdesk systems to streamline complaint management across multiple platforms.
   * **Benefit:** Simplifies data synchronization and improves operational efficiency.
8. **Improved File Upload Support:**
   * **Enhancement:** Increase the file upload limit and support multiple file types (e.g., images, documents, etc.).
   * **Benefit:** Allows users to submit more comprehensive documentation for their complaints, improving the system’s flexibility.