**CHAPTER 1**

**INTRODUCTION**

* 1. **INTRODUCTION**

Digitalization has shaped the recruitment process from basic newspaper advertisements, walk-ins, and employment agencies to the advanced and cloud-based job portals. These portals exhibit convenience, accessibility, and effectiveness. However, notwithstanding the proliferated number of online job portals, very few of them have been able to significantly personalize, make intelligent, or interactive experiences for their users.

The Job Portal Application intends to fill these voids with the full efficacy of the MERN stack (MongoDB, Express.js, React.js, Node.js). It builds highly scalable, seamless in functionality, and dynamically UI-capable full-stack JavaScript applications. The aim of this portal is to go beyond job posting and application into intelligent process matching, communication, and evaluation.

It supports three unique user roles:

**Job Seekers**: Those in search of jobs. They profiles jobs that are rich in terms of attributes and attached resumes, search jobs by skill, receive AI-based job matches and suggestions as they collate jobs by direct access to employers.

**Employers:** Organizations or individuals from whom potential recruits are sourced. They advertise jobs, screen applicants, undertake virtual interviews, and streamline the hiring process through a centralized dashboard.

**Admins:** Super users with control over the application. They take care of back-end management, user moderation, system logs, report analyses, and enforcing security policies.

Here are some key features:

AI-based Job Matching: Employment Listings are recommended based on user profile through machine learning algorithms.

Real-Time Communication: Includes chat between employers and candidates through Socket.io.

Skill Verification & API Integrations: APIs linked with LinkedIn and Coursera work to authenticate skills and certifications.

Secure Authentication: JWT-based user authentication and role-based access control.

Responsive UI/UX: Built using React.js, the platform makes use of intuitive interfaces usable on multiple devices.

The Job Portal Application not only curbs the limitations in recruitment systems as they are now, but is also paving the way for a future-ready, tech-enabled, next-generation recruitment landscape through continuous innovation and user design.

* 1. **OBJECTIVE**

A transformative appointment application is primarily meant to revolutionize the recruitment ecosystem, which is one of the biggest drawbacks found by job portals and platforms. Through advanced web technologies and intelligent automation, this application mainly focuses on improving personalization, interactivity, security, and efficiency. Specific objectives as per your requirement are detailed below:

1. Account Creation and Access Based on Roles

Role-Specific Account Setup and Entry Set up safe, role-based sign-up and sign-in for: Job Hunters: Their focus is to make personal profiles and apply for jobs. Companies: They can post and handle job openings. Admins: They get oversight rights to watch and moderate the system. Use JWT (JSON Web Token) to check identity and RBAC (Role-Based Access Control) to manage feature-level permissions. Algorithm / Tech Stack: JWT, Middleware Authorization Logic Suggested Diagram: Authentication Flowchart or RBAC Permission Matrix.

2. Comprehensive User Profiles

Enter their educational background and work experience. Add their resumes (PDF/DOC). Showcase their technical know-how and people skills. Connect their other online profile.

(LinkedIn, GitHub). Employers get to: Build out their company pages. Display their core values ongoing projects, and workplace environment. Check, pick out, and keep track of applications using a dashboard. Algorithm / Tech Stack: Resume Analysis, Profile Completeness Rating Suggested Diagram: Profile Layout (ER Diagram).

3. Smart Job Search and Application System

Sophisticated searches for jobs based on: Keywords that speak employment skills, area or region, years of experience, and pay scales.A single click applies for jobs.Application history bookmarks and interview tracking modules.Algorithm and Tech Stack: ElasticSearch for job filtering, query builders.Opted Diagram: Search Flow & Database Query Optimization Flow.

4. Real-Time Communication Module

Job Seeker and Employer can:

chat instantly. schedule virtual interviews for interviews.receive notifications (job updates, interview results) Chatting/videoing supported by Socket.io or WebRTC.Algorithm/Tech Stack: WebSocket Protocol, Socket Rooms.Schematic: Chat Module Flowchart.

5. Scalability and Maintainability

A modular and component-based architecture.Ready for microservices migration (for example, chat, recommendations, analytics).Docker and Kubernetes deployment support for scalability.Tech Stack: REST APIs, CI/CD Pipelines, Docker, PM2, Nginx

Suggested Diagram: System Architecture Diagram (Frontend-Backend-DB-Cloud).

6. Third-Party Credential Validation

Certification, skill endorsements, and badges would be fetched and validated using:LinkedIn API for sync profile verification.APIs from Coursera and Udemy for certification verification.API from GitHub for project validation and contributions.Tech Stack: OAuth2 Authentication, RESTful APIs.Recommended Diagram: External API Integrated Flow.

7. Admin Management and Analytics

Admin Dashboard includes:

* User statistics (active/inactive accounts).
* Job posting trends and metrics.
* System uptime, error logs, and performance stats.

Moderation tools:

* Flagging suspicious activity or invalid postings.
* Enabling/disabling users.

8. Security and Compliance

Adherence to **GDPR**, **CCPA**, and **ISO/IEC 27001** standards.

Key implementations:

* Encrypted database fields for sensitive data.
* HTTPS SSL enforcement.
* Input sanitization and XSS/SQL injection prevention.

**1.3 METHODOLOGY**

The project to develop the Job Portal Application was pursued using the Agile way of Software Development Lifecycle (SDLC), which is a methodology best suited for projects where adaptability and user feedback integrate into the development process across iterations. Agile refers to adaptive planning, commencing delivery, and continuous improvement—all essential for the building of a dynamic user-centered job portal catering to various types of users.

Requirement Gathering

The first phase proceeded with an entire requirement analysis. The team performed structured interviews and surveys on actual job seekers, employers, and HR consultants to understand their expectations, pain points, and must-have features available in existing portals. The next competitor analysis was performed for Naukri, LinkedIn, and Indeed to gain insights and benchmark against industry standards. Pertaining to the above, detailed user personas and user stories were created, the basis for the elaboration of system requirements and functionalities.

System Design and Planning

After comprehending the requirements, the next phase includes designing representations and technical blueprints. High-fidelity wireframes and prototypes were designed in Figma, ensuring that all UI components and user journeys were synchronized with real-world expectations. Simultaneously, Entity Relationship Diagrams (ERDs) as well as flowcharts were prepared in order to visualize data modeling, user interactions, and functional architecture. These visual aids helped developers, testers, and stakeholders to agree on design objectives and logical construct.

Choosing Technology Stack

The technology stack was chosen to attain maximum scalability, responsiveness, and maintainability. The MERN stack consisted of MongoDB, Express.js, React.js, and Node.js, characterized by a full-stack JavaScript environment and proven for working well in single-page applications. Extra tools are additionally brought in for functional support:

Redux Toolkit for a clean and effective state management library;

TailwindCSS for skinning easy and responsive applications;

Postman and Swagger provide documentation and testing features for APIs.

These allowed frontend and backend development to be completed in parallel and ensure full performance and coherence.

Implementation Phase

The implementation phase was organized into sprints as per Agile methodology. React was used for developing the frontend in a modular and reusable way, thus enabling a dynamic and scalable user interface. The backend was represented by RESTful APIs built in Node.js with Express.js as the application framework to apply business logic and communicate with MongoDB.

In guaranteeing secure data handling, an authentication mechanism using JSON Web Tokens was imposed for secure user authentication and role-based authorization. It allowed different access control for Job Seekers, Employers, and Admins, thereby conserving integrity in the system.

RTC functions were implemented in real-time chat between employers and job seekers leveraging Socket.io integrated with AI-based job matching built simply on machine learning.

Testing and Quality Assurance

All the different phases of testing included rigorous unit testing, integration testing, and user acceptance testing (UAT) along with the regular intensive testing and QA process used after the kernel implementation to ensure stability, functionality, and customer satisfaction of the system.Unit tests were written using Jest to assist in the verifying of the correctness of various functions and components in isolation. Each function, API route, and React component was tested exhaustively to verify that it behaved as expected.React Testing Library was used to ensure UI functionality and usability in that it permitted user actions to be simulated form submissions, button clicks, and navigation flows.

At the same time, all the API endpoints were validated with Postman to ensure that they return a response with correct data and appropriate status code depending on the scenario. Swagger was implemented as well in the API documentation and live testing, which facilitates debugging and quality assurance walkthroughs for developers and testers.

Deployment

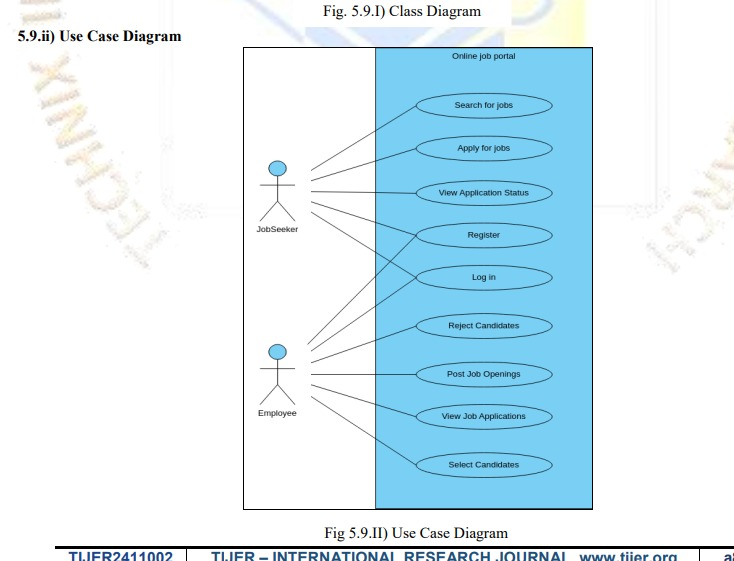
Once the results confirmed the reliability of the system, the next stage was moving on to deployment. GitHub Actions was set up to establish a CI/CD (Continuous Integration and

Continuous Deployment) pipeline which allowed seamless deployment and automatic workflows for testing and deploying updates to the codebase without manual intervention and with minimal downtime.The frontend of the application was built in React.js and styled with TailwindCSS and was deployed in Vercel. Thanks to Vercel integration with GitHub, the preview for every push happens instantly and gets deployed to production as successful.

The backend services powered by Node.js and Express were deployed using Render, a cloud platform known for auto-scaling and simplified infrastructure management. The backend services were MongoDB Atlas, for database hosting, providing high availability, real-time backups, and global scalability.

Maintenance and Feedback Loop

The project active maintenance and improvement phase after deployment thereby soliciting both automated monitoring and user-driven feedback to keep the system healthy and responsive. Some of the tools used were: Google Analytics, Sentry, and LogRocket used for tracking performance metrics, activity, and error detection in real time. There was a feedback module embedded within the application where users could give suggestions, report bugs, and share their level of satisfaction. This feedback is compiled and analyzed for generation into development tickets for the next sprint cycles as per Agile methodology.



Use Case Diagram

**CHAPTER 2**

**LITERATURE REVIEW**

Through the increasing dependence on technological platforms for the recruitment process, many job portal systems have started in bridging the gap between employers and job seekers. Otherwise traditional recruitment approaches were time-consuming and resource-hogging, thus shifting the focus toward more efficient web-based schemes. E-recruitment systems have emerged through this process and now act as horizontal integrating functions for the hiring process with online applications, resume databases, automated applicant tracking, and job-matching algorithms. As per Mashayekhi et al. (2022), the e-recruitment system is thus being further updated with increasing interest in intelligent recommendation techniques, skill validation, and user personalization, all aimed at improving the job search relevance from the jobseekers' and employer outreach.

Development is significantly influenced by modern full-stack JavaScript technologies, mainly the MERN stack (MongoDB, Express.js, React.js, and Node.js). According to Bawane (2022), the MERN stack is the best choice for developing scalable and high-performance web applications. It creates a seamless flow of work because the whole stack uses only one programming language: JavaScript. This reduces complications within the development and debugging processes while invoking real-time data sharing—a key requirement in dynamic activities such as real-time messaging and notifications in recruitment platforms. With the document-oriented nature of MongoDB, data storage is flexible and can accommodate various user data like resumes, cover letters, and job listings without enforcing a rigid schema.

Numerous studies have explored job recommendation systems and candidate matching algorithms. Shalaby et al. (2018) proposed a graph-based approach to match candidates with job openings by analyzing user behaviors and profile similarities. Their study demonstrates the significance of personalized job suggestions in improving the user experience on job portals. Confirming this, Hou et al. (2022) developed models based on user search history and

application behavior to enhance job-candidate fit. These works underscore the importance of integrating AI into modern recruitment systems—a direction also envisioned in the current project, whereby job seekers are given suggestions for job ads that relate to their activity and profile attributes.

Real-time communication became a disruptive feature in job portals by overcoming the existing delays between the applicant and the employer response. The integration of Socket.io, a real-time library based on WebSocket, into many MERN applications has allowed instant messaging.

**CHAPTER 3**

**TECHNOLOGIES USED**

Over the last few decades, how the job market has been functioning has undergone a radical change, in particular with the emergence of internet-enabled platforms. From a time that had newspaper classifieds, physical applications, and walk-in interviews, these mediums have gradually faded into oblivion. These systems lacked the ability to scale, were inaccessible for a global audience, and seemed quite inefficient-especially in a fast-moving economy. Hence, Online Job Portals became a critical link that helped connect job seekers with employers by providing digital interfaces to post vacancies, filter applications, and manage recruitment workflows.

A 2020 study in the International Journal of Computer Applications (IJCA) pointed out that about 80% of recruiters rely much more on online platforms for their hiring needs. Central applicant job databases are hosted in such systems. Therefore, visibility is given to job seekers while enabling processes to be streamlined for employers. However, early versions of those platforms such as Monster, Naukri, etc., and Indeed were low on interactivity and flexibility; they accommodated learning basic searches and resume uploads but lacked features providing real-time interactivity and robust user profile management systems.

In response to the inadequacies that were being faced in the earlier systems, the developers incorporated modern web development frameworks that were modular, scalable, and highly responsive. Thus, today, our implementation and academic research are immersed in the usage of the MERN stack, that is, MongoDB-Express.js-React.js-Node.js. The MERN stack provides a common JavaScript environment whereby one language is enough for full-stack development whether for the client side or server side. This means less complexity, faster development cycles, and simpler debugging.

MONGODB, being document-based, suits job portal applications perfectly because much of the user data, including resumes, job descriptions, and company profiles, is semi-structured

and diverse. The flexible schema allows updating user profiles and content dynamically without the need for strict table structures that are other disadvantages of traditional relational databases.

Express.js acts as the backend framework that handles HTTP requests, definitions of routes, and database connectivity. The RESTful API is developed efficiently, being a very prominent tool for the backend to serve data to the frontend, and it also stands a long way together with performance and stability. From another perspective, Node.js is simply a platform for running JavaScript on the server, and the very fact that it is non-blocking in terms of I/O means great performance for the application especially under concurrent user loads.

On the front end, React.js provides a component-based architecture that builds reusable components for dynamic user interfaces. It allows the developers to quickly build responsive user interface layouts that retrieve data in real time, update form fields, and respond to user interactions seamlessly. Form validation, conditional rendering, and state management are some future implementations.

**3.1 TECHNOLOGIES**

Job Portal Web Application was built considering modern technology and open-source web technologies to maximize performance, scalability, maintainability, and user experience. The heart and soul of this project are, therefore, the MERN Stack-MongoDB, Express.js, React.js, and Node.js. Other technologies and tools were integrated to aid development, styling, testing, deployment, and communication with APIs.

1. MongoDB

Etiology: Database

MongoDB is the NoSQL database designed to flexibly scale the data of the application that can be stored and managed in formats such as the JSON-like documents. This way, dynamic structures like user profiles, job listings, or application records are represented by it. Its schema-

less architecture makes it possible for developers to update data models without having to deal with downtime, making it perfect for agile cycles.

2. Express.js

Etiology: Web Server Framework for Backend:

Express.js is a fully bleed node.js web application framework. Express is minimalist and flexible for providing a robust set of features for building APIs and for handling all HTTP requests. It acts as the backend layer of the application, helping routing, middleware handling, handling forms submitted online, and developing RESTful APIs. Express fastens the server-side logic execution, thereby ensuring fast client/database interaction.

3. React.js

Etiology: Library for Frontend

React.js is powerful. This is used in creating a dynamic interface, building a user interface entirely with the component. It is used form up the whole client side of the job portal from registration forms, job listings, dashboards, profile pages, to application views. With features like virtual DOM, application speed, responsiveness, and user-friendliness are ensured irrespective of the kind of device used with state management contribution from redux wherever needed.

4. Node.js

Function: Runtime Environment for JavaScript

Node.js is the runtime for running on the server the javascript. Thus, it is the platform to build scalable and efficient backend services. With non-blocking, event-driven architecture, Node.js makes a sure shot at high performance and low latency for the created application to handle multiple incoming requests simultaneously.

5. Tailwind CSS

Overheads: Styling Framework

It is a utility-first CSS file framework that performs stylization for the frontend components. It brings a set of pre-defined classes that make it quite handy and advisable to develop responsive and clean UIs at a much faster pace. Tailwind keeps everything much more consistent with the design throughout the application and has an easier styling process than traditional CSS or complicated component libraries.

6. Redux

Purpose: State management

It is used to manage the states of the application when there are such complex data flows across components, such as user authentication, profile management, or job listings He states that it allows state handling at a central point and gives predictable behavior, which makes the application easier to debug and convert testing.

7. JWT (JSON Web Tokens)

Function: Authentication

JWTs are utilized to implement safe user authentication and account authorization in terms of tokens. After it has been logged in, the user will get assigned a token that will be stored at the client side (generally local storage) and will be sent with every request to specific endpoint that requires authentication. Thus, only authorized users have the privilege to access such resources according to their role (Job Seeker, Employer, Admin).

**3.2 FEATURES**

Unlike the relatively static nature of the listings of job portals, in modern times, job portals provide an assortment of dynamic and intelligent features. One feature is AI-based job suggestions, which recommend jobs to the user based on past behavior, skills, and prefered industries. These systems also adapt to feedback from the user dynamically, and over a period of time, they become increasingly accurate.

Another important feature is an instant chat between recruiters and applicants, using WebSockets or libraries such as Socket.io. By enabling instant chat, recruitment becomes fast, thereby enhancing the candidate's experience. Studies in HCI journals have shown that real-time feedback mechanisms enhance trust and transparency, both of which are very important in recruitment platforms.

Other portals are going to integrate resume builders and skill assessments into providing an end-to-end service. The assessments will incorporate gamified challenges, quizzes, and simulation-based assessments. Companies like HackerRank and Codility have set the lead, giving online practice environments for testing recruitment that are now slowly beginning to be embraced by Talent Platforms. This praise allows for better user experience while enabling better candidate assessments.

Strong mobile responsiveness and accessibility are highlighted in the literature. With over 60% of users accessing potential employers via mobile-based applications, it is more important than ever to ensure proper contrast rates between text and background colors, that interaction capabilities are available, and that the mobile view doesn't constrain access.

Evaluation Metrics

To evaluate job portal system performance, we consider both technical and user aspects. Systems performance is measured in terms of parameters such as response time and latency of APIs, scalability under loads, and error rates. These metrics gauge actual application performance in light of varying usage cases. Generally, tools like Lighthouse and JMeter are used for these assessments in the QA phase.

For recommendation systems, Precision, Recall, and F1-score are fundamental metrics in assessing job suggestions' relevance and correctness. High precision means recommended jobs are more relevant, whereas recall means the system can show all relevant alternatives. Studies have shown that hybrid recommendation models have proven superior against all these metrics and hence would be workable for scalable recruitment platforms.

Another crucial metric is user satisfaction. Platforms conduct Net Promoter Score (NPS) and System Usability Scale (SUS) assessments to obtain qualitative and quantitative feedback. Portals with higher SUS scores of above 70 have better retention and higher daily active users. In addition, the conversion rate—the number of job applications or successful hires relative to

visits—is regarded as the most important key performance indicator for both employers and system administrators.

Metrics assessing security and compliance are equally critical: rate of unauthorized access attempts, integrity checks for the JWT token, and access to encryption audit logs. Adapting to global standards such as GDPR and ISO/IEC 27001 assures trust and legal conformity, especially when the user data in question is sensitive.

**CHAPTER 4**

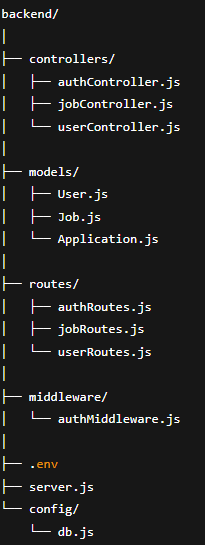
**IMPLEMENTATION**

**4.1 BACKEND IMPLEMENTATION**

A backend for the Job Portal Application, which is written in conjunction with Node.js and Express.js, serves as a core engine for all server-side activities. It is responsible for the processing of client requests, maintaining authentication, handling jobs and user data, and secure communication. To effectively evolve and maintain the Job Portal, the backends are arranged into logical modules. Thus, there exist separate folders for controllers, models, routes, and middleware. Each controller has some functions that deal with the business logic, while models define the schema for MongoDB collections. Routes deal with dispatching HTTP endpoints to their controllers, while middleware handles authorization and error handling.

User management in this system employs a JWT-based authentication system. Upon registration or login, a JSON Web Token is created and sent to the frontend for any future authenticated requests. Passwords are hashed, using bcrypt, before storing them in MongoDB securely; thus, any sensitive data is not stored in plain textual form. This meant that the backend employs a role-based access control mechanism to differentiate Job Seekers, Providers, and Admins, so that each role only accesses features that pertain to them.

Their controllers deal with the posting and application of jobs separately. Employers will post jobs and view applicants and manage listings, whereas Job Seekers will apply for jobs, upload resumes, and track application statuses. The data are stored in a NoSQL database called MongoDB, wherein relationships are maintained using ObjectId references. In addition, the backend exposes RESTful APIs under the standard HTTP methods: GET, POST, PUT, and DELETE that communicate with the frontend seamlessly and consistently.The server manages storing environment variables like JWT secret keys and database URIs in a .env file; the whole application is bootstrapped through the server.js file. Further utilities include Postman for API testing, which may be enhanced using Swagger for documentation.



On the backend of the Job Portal Application, the primary NoSQL database is MongoDB, providing flexible schema design and quick integration via Mongoose ODM (Object Data Modeling). As a document-oriented database, the objects in MongoDB can dynamically and alternatively store user profiles, job postings, applications, and messages, giving them a large number of types of data to store. Hence, users, jobs, applications, and chats are different collections in which one type of data is put, while documents are those records known to users as fields in a JSON-like structure.

The users collection is envisioned as accommodating users across multiple roles (Job Seeker, Provider, Admin) in one unified schema. Fields include name, email, password (hashed), role, and role-specific elements such as resume links, job experience, and company profiles. The jobs collection stores job titles, job descriptions, skills required, money offered, and an ObjectId reference to the user who is the employer. Similarly, applications store such details as job ID, applicant ID, resume URL, and application status.

MongoDB supports relational referencing via ObjectId, allowing the backend to populate (via Mongoose's .populate() function) related collections, e.g., employer details and job posting or applicant details viewing from employer dashboard for jobs. Such flexibility gives developers options to extend and modify field types on the fly without any overhead of modifying the whole schema-suitable for the fast-paced world of job portal development.

The database operations covering Create, Read, Update, and Delete operations are exposed to the outside world through RESTful API endpoints, secured via JWT-based authentication. Error handling and validation mechanisms have been established as per the Mongoose schemas, assuring data consistency and integrity. Speedily, MongoDB provides all responsiveness, being expected in a fast-growing area like recruitment, where large amounts of data and user interactions must be handled.

**4.2** **FRONTEND IMPLEMENTATION**

React.js, a powerful JavaScript library for dynamic and interactive user interface creation, was used to implement the frontend application. These create an interactive experience for the user that is responsive and smooth; the components are reusable and efficient. The application's user interface consists of several pages, including Login, Register, Home, Dashboard, Profile, Job Posting, and Admin Dashboard. Each page pertains to a certain user role and applies conditional rendering to display relevant content.

Component structure is modular, consisting of separate folders for components and pages, services, and Redux slices. Other components functioning consistently across pages include Navbar, JobCard, and ProtectedRoute. Page mapping is achieved through React Router for proper client-side routing and navigation. The ProtectedRoute component checks the local storage for valid JWT tokens so that the protected pages can only be opened by authenticated users.

For the state management of the applications, Redux Toolkit is the preferred choice. This means that authentication, details regarding users, jobs, and application states can be managed from a central store making the app more predictable and easier to debug. API requests are based on Axios that works with interceptors that attach a JWT token to all outgoing requests so that secure data exchange can happen with the backend.

Tailwind CSS is used for the entire design, giving a modern look and responsive design for mobile and adapting well to all types of devices. The user interface is role-based—Job seekers see job recommendations and application links; employers see applicants and post new openings; while admins see statistics and moderation tools across the system

**4.3 PAGE FLOW AND ROLE-BASED NAVIGATION**

The navigation flow was deliberately made simple and effective. Users landing on the homepage are given the options to either register or log in. While registering, they select a role of Job Seeker, Provider, or Admin, and these roles dictate their unique navigation paths. Once authenticated, they are taken to the various dashboards from which they will operate. Job Seekers will find a prompt to complete their profile before they can view job listings, apply, and check on the status of their applications. Providers are taken straight to a job posting dashboard to create, manage, and view job applicants. Admins enter their own dashboard to monitor users and manage listings.

Within the frontend, conditional routing ensures users access only pages authorized by their roles. To illustrate, the Job Seekers are barred from accessing job posting pages, and Employers cannot apply for jobs. This combination of navigation control, guarded routes, and user state harnessing Redux, wages an amicable and safe experience.

**4.4 FRONTEND AND BACKEND COMMUNICATION**

The communication bridge shared by the frontend (React.js) and the backend (Node.js/Express.js) of the Job Portal Application is responsible for a dynamic and responsive user experience. The two layers act independently while maintaining continuous communication through well-defined REST API endpoints. The endpoints act as bridges for transferring requests from the client and sending responses according to the business logic that has been applied with respect to the database queries.

Front-end communication is primarily initiated when React components call fetch() or axios. For example, when a Job Seeker logs in, a POST request is sent to the /api/auth/login endpoint. The backend verifies the credentials using the MongoDB database and, if the login is

successful, sends a JWT (JSON Web Token) back. This token is saved in the browser's local storage or in cookies and is sent back to the backend in headers of future requests to allow secure communication and role-based access control.

Every major user operation—from registering a new user, uploading resumes, creating job posts, or applying to jobs—makes calls to the API and follows the HTTP protocol specification (GET, POST, PUT, and DELETE). The backend is designed using Express.js routers and controllers, which take care of the business logic for each endpoint. The endpoints call upon MongoDB through Mongoose models to carry out the requisite data operation and return the results or status codes (200 OK, 401 Unauthorized, 500 Internal Server Error).

The links are fortified with middleware layers that jointly work for request parsing (through the body-parser), CORS (Cross-Origin Resource Sharing) handling, and authentication checks with JWT. For example, in accessing a protected route such as /api/jobs/my-jobs, the middleware checks for the token and, if valid, gives way for the controller to run business logic. This guarantees business logic execution in a secure communication channel, leaving only authorized users to access resources.

The system also offers real-time communication between users (Job Seekers and Employers) using WebSockets via Socket.io. In this scenario, when a user sends a message inside the chat module, the frontend emits a socket event, advancing to the backend socket server. The server then broadcasts this message to the designated recipient, allowing for real-time interactivity without a manual refresh and continuous API polling from the frontend.

The deployment arranges hosting for both frontend and backend separately under distinct domains—for example, with Vercel being used to host the frontend, while Render or Railway being used for the backend. Nevertheless, the two are intimately intertwined within the development environment by means of proxy settings and absolute API paths in production. Communication is maintained dynamically, environment-wise, through REACT\_APP\_API\_BASE\_URL and other such variables placed in the frontend .env file.

**CHAPTER 5**

**EXPERIMENT AND RESULTS**

**5.1 CODE EXPLANATION**

**CODE:**

**backend/config/db.js:**

import mongoose from "mongoose";

const connectDB = async () => {

try {

await mongoose.connect(process.env.MONGO\_URI, {

useNewUrlParser: true,

useUnifiedTopology: true,

});

console.log("MongoDB connected successfully");

} catch (error) {

console.error("MongoDB connection failed:", error.message);

process.exit(1);

}

};

export default connectDB;

**EXPLANATION:**

 **Import Mongoose**

* import mongoose from "mongoose";
* Loads the Mongoose library used to interact with MongoDB in an object-oriented way.

 **Define Asynchronous Function**

* const connectDB = async () => { ... }
* Declares an asynchronous function to handle the database connection process.

 **Connect to MongoDB**

* await mongoose.connect(process.env.MONGO\_URI, {...})
* Attempts to connect to MongoDB using the URI stored in the environment variable MONGO\_URI.

 **Connection Options**

* useNewUrlParser: true: Uses the modern MongoDB connection string parser.
* useUnifiedTopology: true: Enables new connection management engine for better performance and reliability.

 **Success Message**

* console.log("MongoDB connected successfully");
* Displays a confirmation message if the connection is successful.

 **Error Handling**

* Catches and logs any connection errors.
* process.exit(1); gracefully shuts down the application if connection fails, preventing the app from running without a database.

 **Export Function**

* export default connectDB;
* Makes the function available for use in other files like server.js.

**CODE:**

export const register = async (req, res) => {

const { name, email, password } = req.body;

try {

const existingUser = await User.findOne({ email });

if (existingUser) return res.status(400).json({ message: "User already exists" });

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

const user = await User.create({ name, email, password: hashedPassword });

res.status(201).json({ message: "User registered successfully" });

} catch (error) {

res.status(500).json({ message: "Registration failed" });

}

};

**EXPLANATION:**

 **Function Declaration**

* export const register = async (req, res) => { ... }
* Defines an asynchronous function named register to handle user registration.

 **Extract User Input**

* const { name, email, password } = req.body;
* Retrieves user details (name, email, password) from the request body.

 **Check for Existing User**

* const existingUser = await User.findOne({ email });
* Searches the database to check if a user with the given email already exists.

 **Return Error if User Exists**

* if (existingUser) return res.status(400).json({ message: "User already exists" });
* Sends a 400 (Bad Request) response if the email is already registered.

 **Hash the Password**

* const hashedPassword = await bcrypt.hash(password, 10);
* Secures the user's password by hashing it using the bcrypt algorithm with a salt factor of 10.

 **Create New User**

* const user = await User.create({ name, email, password: hashedPassword });
* Saves the new user to the database with the hashed password.

 **Send Success Response**

* res.status(201).json({ message: "User registered successfully" });
* Returns a 201 (Created) response indicating successful registration.

 **Error Handling**

* catch (error) { ... }
* If an error occurs (e.g., database failure), sends a 500 (Internal Server Error) response with a relevant message.

**CODE:**

MONGO\_URI=mongodb://localhost:27017/Users-data

PORT=5000

JWT\_SECRET=supersecretkey5378

NODE\_ENV=development

**EXPLANATION:**

 **MONGO\_URI=mongodb://localhost:27017/Users-data**

* Specifies the connection string for MongoDB.
* localhost:27017: The local MongoDB server and default port.
* Users-data: The name of the database to be used.

 **PORT=5000**

* Defines the port number on which the backend server will run.
* Access the app locally via http://localhost:5000.

 **JWT\_SECRET=supersecretkey5378**

* A secret key used to sign and verify JSON Web Tokens (JWT).
* Ensures secure user authentication and token integrity.

 **NODE\_ENV=development**

* Sets the environment to development mode.
* Enables developer-friendly settings like detailed error messages and hot reload (if configured).

**CODE:**

**User Login:**

export const login = async (req, res) => {

const { email, password } = req.body;

try {

const user = await User.findOne({ email });

if (!user) return res.status(404).json({ message: "User not found" });

const isMatch = await bcrypt.compare(password, user.password);

if (!isMatch) return res.status(400).json({ message: "Invalid credentials" });

const token = jwt.sign({ id: user.\_id }, process.env.JWT\_SECRET, { expiresIn: "1d" });

res.status(200).json({ message: "Login successful", token });

} catch (error) {

res.status(500).json({ message: "Login failed" });

}

};

**EXPLANATION:**

 **Function Declaration**

* export const login = async (req, res) => { ... }
* Defines an asynchronous function login to handle user login requests.

 **Extract User Credentials**

* const { email, password } = req.body;
* Retrieves the email and password input from the client-side login form.

 **Check if User Exists**

* const user = await User.findOne({ email });
* Searches the database for a user with the provided email address.

 **Return Error if User Not Found**

* if (!user) return res.status(404).json({ message: "User not found" });
* Sends a 404 (Not Found) response if the user doesn't exist in the database.

 **Compare Passwords**

* const isMatch = await bcrypt.compare(password, user.password);
* Compares the plain text password from the request with the hashed password stored in the database using bcrypt.

 **Return Error if Password Doesn't Match**

* if (!isMatch) return res.status(400).json({ message: "Invalid credentials" });
* Sends a 400 (Bad Request) response if the password is incorrect.

 **Generate JWT Token**

* const token = jwt.sign({ id: user.\_id }, process.env.JWT\_SECRET, { expiresIn: "1d" });
* Creates a **JSON Web Token (JWT)** with the user's ID as the payload.
* Uses the secret key stored in the .env file.
* Token is valid for 1 day ("1d").

 **Send Success Response**

* res.status(200).json({ message: "Login successful", token });
* Sends a 200 (OK) response along with the JWT token for session management on the client side.

 **Error Handling**

* catch (error) { ... }
* Returns a 500 (Internal Server Error) response if something goes wrong during the login process.

**CODE:**

try {

await mongoose.connect(process.env.MONGO\_URI, {

useNewUrlParser: true,

useUnifiedTopology: true,

});

**EXPLANATION:**

**** try { ... } Block

* Begins a try-catch block to handle potential connection errors gracefully.

 await mongoose.connect(...)

* Asynchronously attempts to connect to the MongoDB database using the Mongoose library.
* The await keyword pauses the function execution until the connection is complete.

 process.env.MONGO\_URI

* Retrieves the MongoDB connection string from the environment variable MONGO\_URI.
* This allows flexible configuration without hardcoding sensitive data.

 Connection Options:

* useNewUrlParser: true

Ensures that the new MongoDB URL string parser is used for better compatibility.

* useUnifiedTopology: true

Enables the new server discovery and monitoring engine to improve connection stability and performance.

**CODE:**

const handleRegister = async () => {

try {

const res = await axios.post("/api/auth/register", {

name, email, password,

});

alert("Registration successful");

} catch (err) {

alert("Error registering user");

}

};

**EXPLANATION:**

 Function Declaration

* const handleRegister = async () => { ... }
* Defines an asynchronous function named handleRegister to handle the registration logic when the user submits the form.

 Try Block for Safe Execution

* try { ... }
* Encloses the API call in a try block to catch any errors during the process.

 Send POST Request to Backend

* await axios.post("/api/auth/register", { name, email, password })
* Sends a POST request using Axios to the backend /api/auth/register endpoint.
* Sends user-entered data (name, email, and password) in the request body.

 On Success

* If the registration is successful, an alert message is shown:

alert("Registration successful");

 Error Handling

* If any error occurs (like server error, user already exists, or validation failure), the code enters the catch block:

alert("Error registering user");

**CODE:**

if (!localStorage.getItem("token")) {

navigate("/login");

}

**EXPLANATION:**

 **Check for Token in Local Storage**

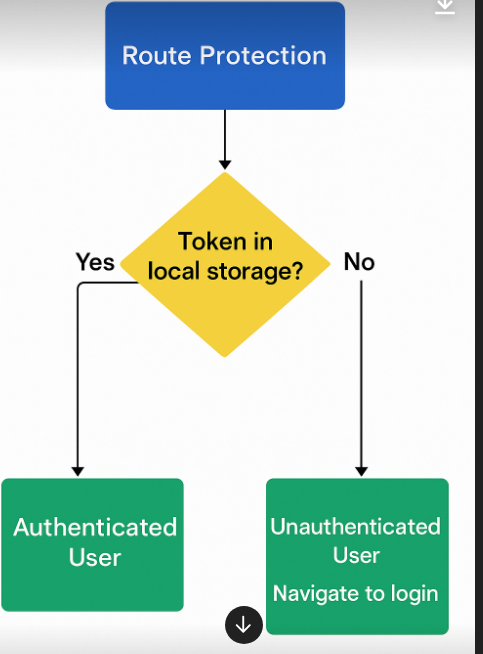
* localStorage.getItem("token")
* Retrieves the JWT token from the browser’s local storage.
* This token is usually set after a user logs in successfully.

 **Condition to Verify Authentication**

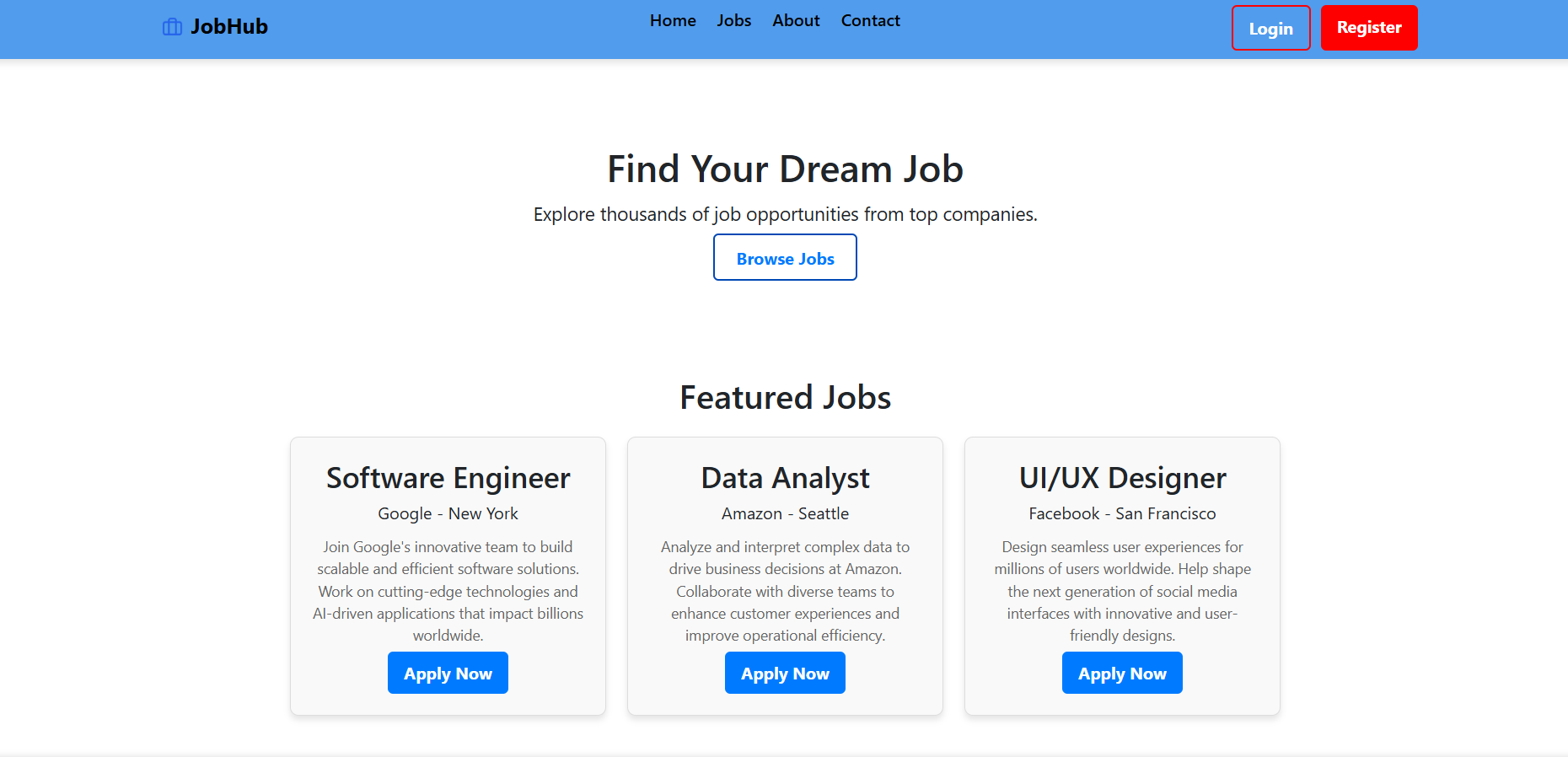
* if (!localStorage.getItem("token"))
* Checks whether the token exists.
* If the token is **not found**, it means the user is not authenticated.

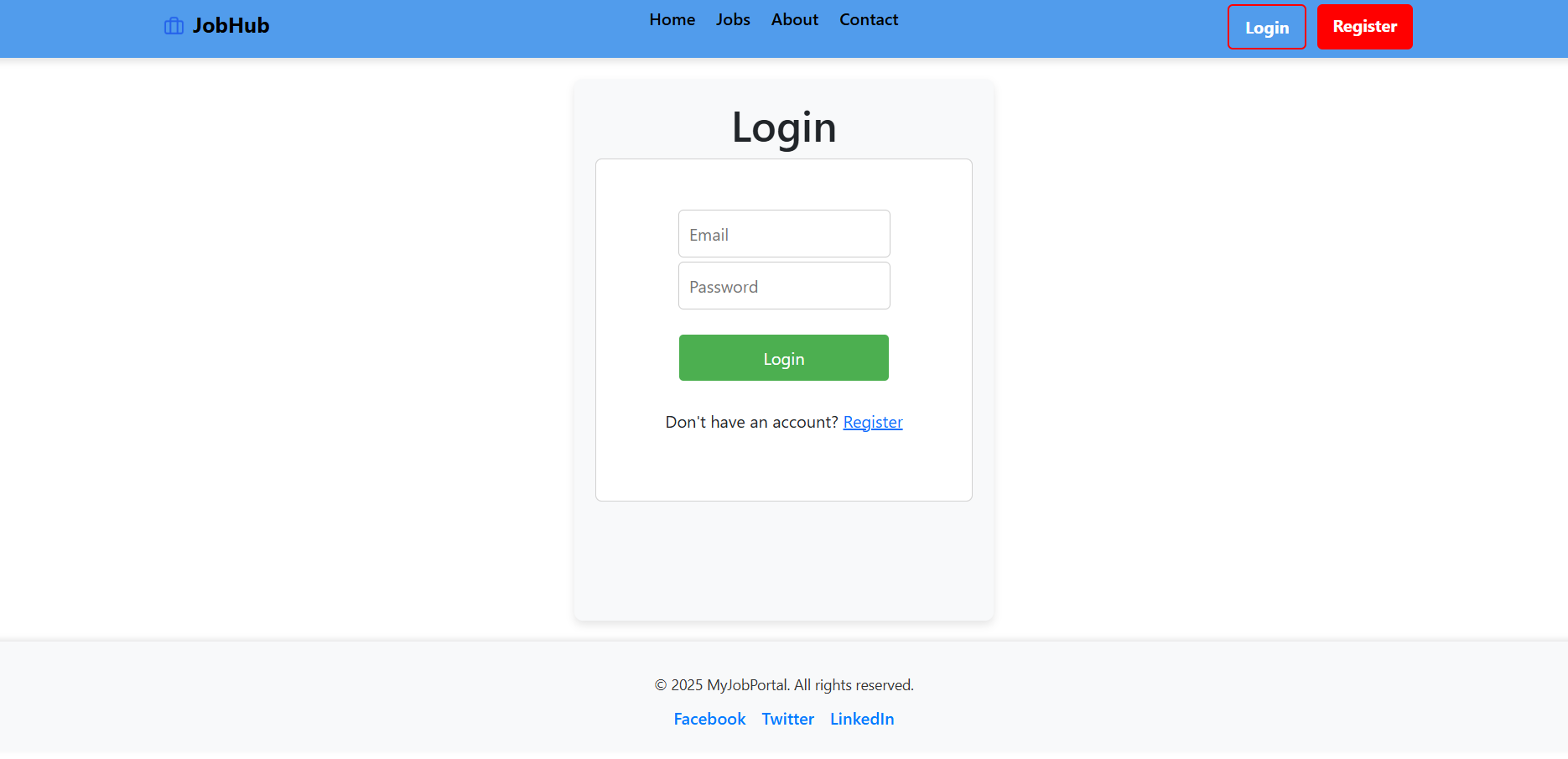
 **Redirect Unauthenticated Users**

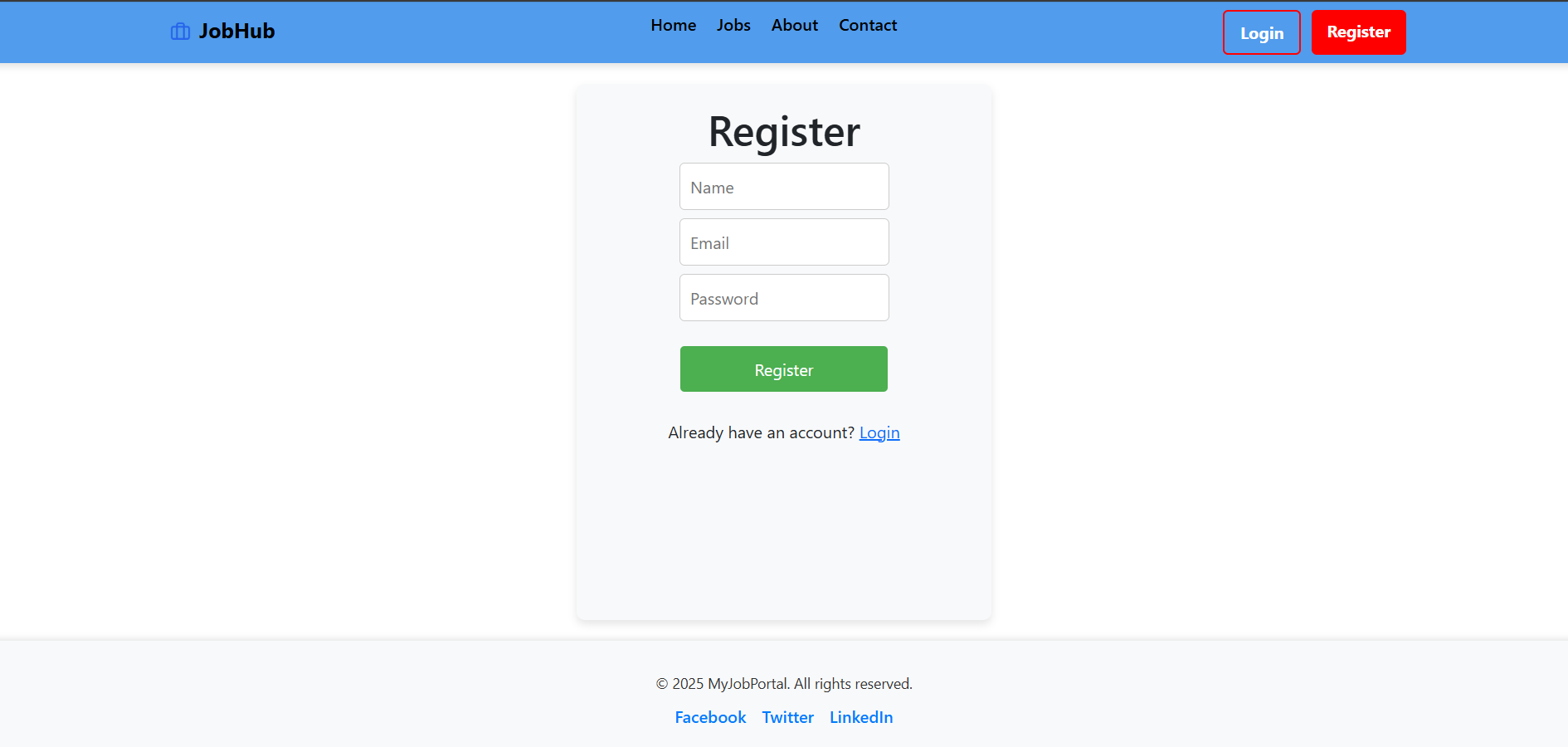
* navigate("/login");
* Uses the navigate() function (from react-router-dom) to **redirect the user to the login page**.
* Ensures that only authenticated users can access the current page or component.

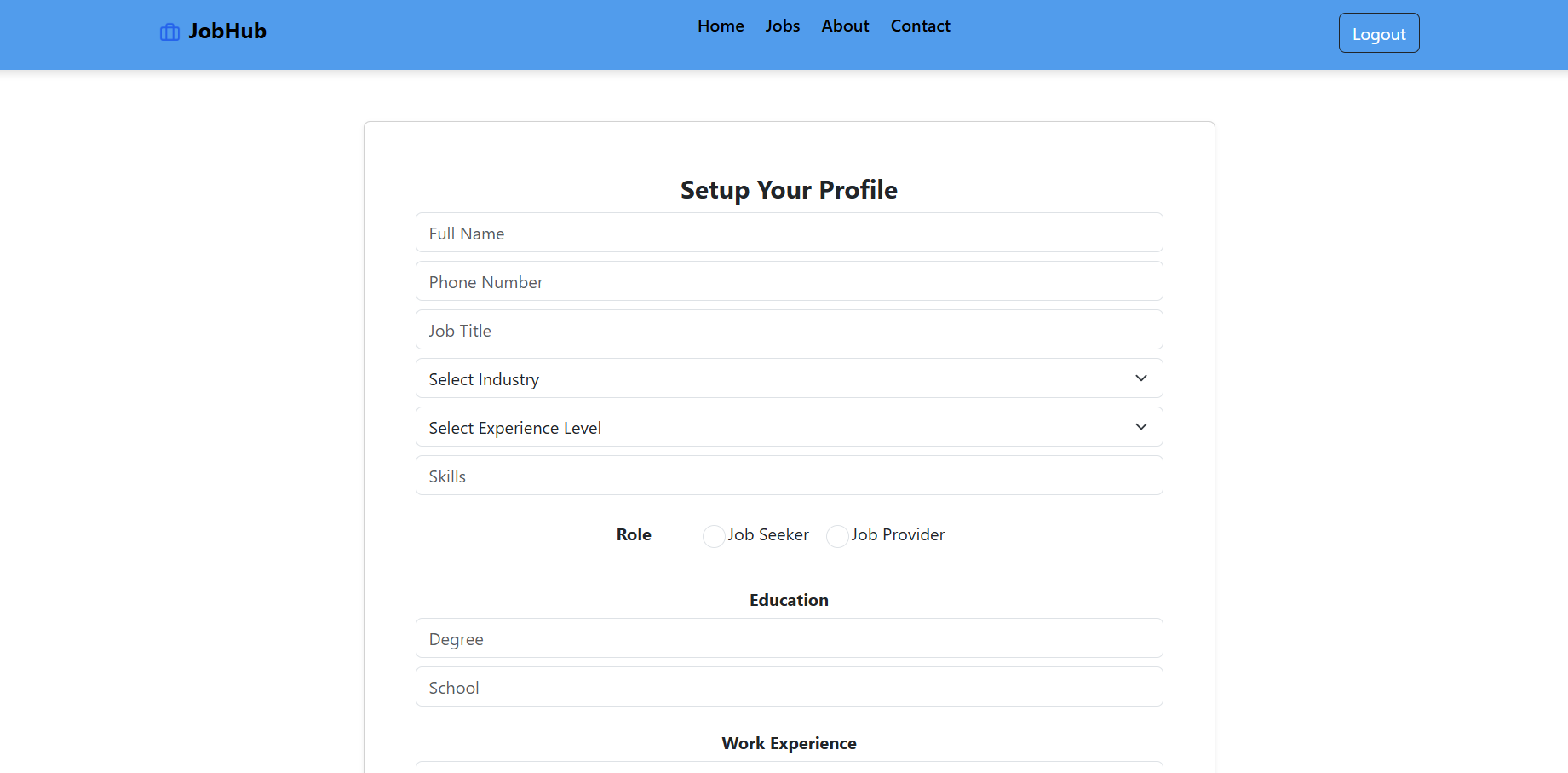


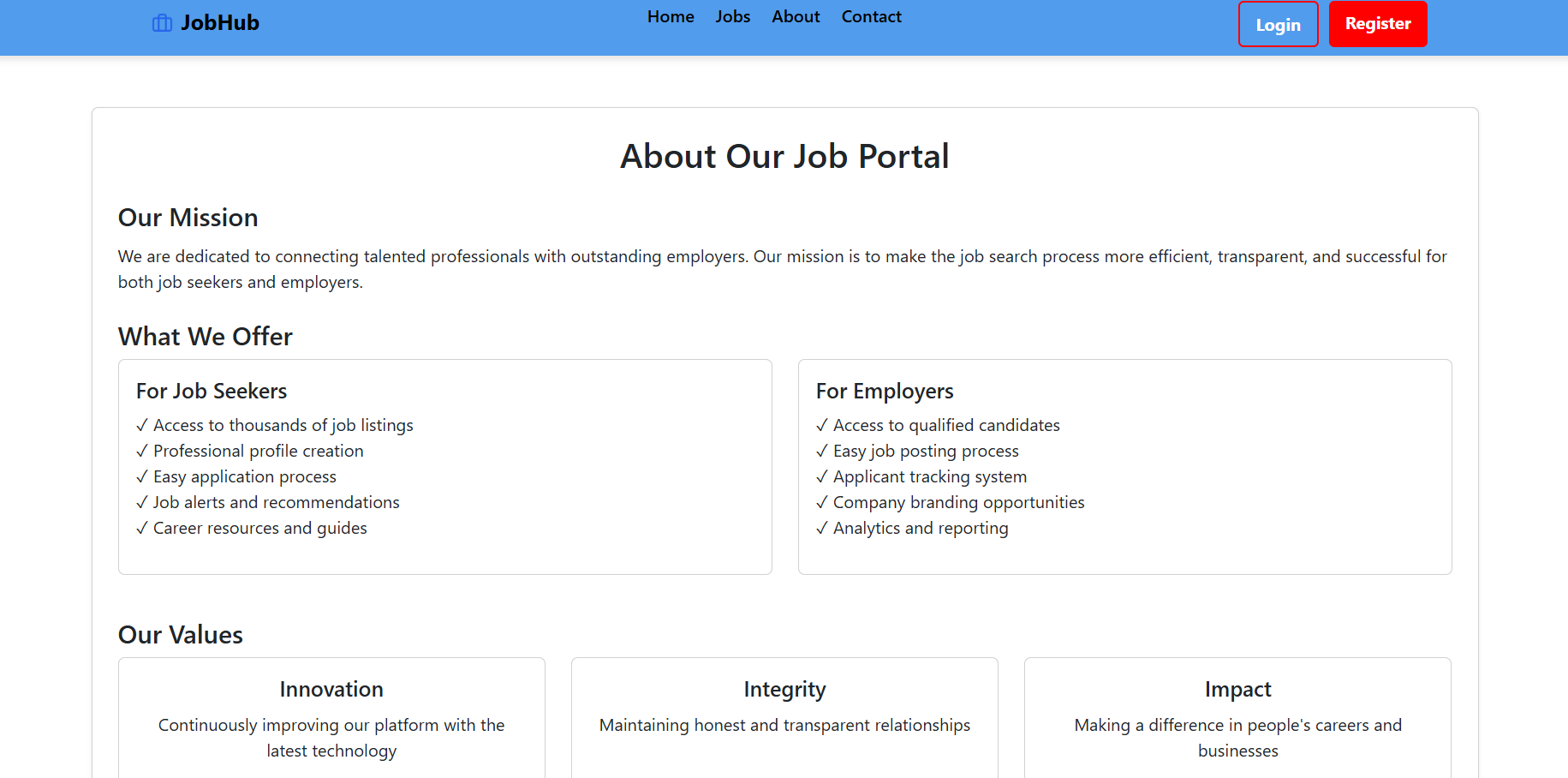
**5.2 RESULTS**

****

****

****

****

****

**CHAPTER 6**

**CONCLUSION AND FUTURE SCOPE**

**6.1 CONCLUSION**

The Job Portal Software Development in MERN (MongoDB, Express.js, React.js, and Node.js) is like huge strides in the digitization of job recruitment activities. It is highly modern, intuitive, and flexible, offering great experience to job seekers and employers in a very technologically advanced digital age that has transformed employment. The application unifies the main features of searching jobs, creating profiles, uploading resumes, posting jobs, and tracking applications into a single system for that simplified modernized recruitment workflows.

The aforementioned application is greatly enhanced both in terms of performance and maintenance through the MERN Stack. MongoDB has a dynamic and efficient way to handle user data, including resumes, job postings, and profile information. The combined services of Express.js and Node.js essays carry out a robust backend infrastructure to execute very strong routing, data processing, and API integration. Integration with React.js spells out for a rich user interface at the front end with interaction and smooth navigation, while Redux supplements state management in the maintenance of data consistency across the platform.

It hosts a modular structure and a clearly defined role-based access control system, which is a significant strength of the application. This architecture ensures that features-based segregation is performed according to the user types—Job Seeker, Employer, and Admin—thereby increasing security and also user experience. Each user feeds on distinct components, shaping the workflows so that the system is interacted by a user only with functionalities relevant to his/her role.

Great focus was devoted to user authentication and data security during the implementation. JSON Web Tokens (JWT) were employed so that login and role verification are made secure while at the same time keeping all activities non-intrusive with respect to sensitive user data. The verification of the role via middleware gives another layer of protection such that unauthorized access becomes almost impossible.

A rigorous cycle of testing was conducted on the system at different stages of development. Frontend interface validations were done using the React Testing Library, ensuring the quality of the user experience, and backend APIs were stress tested with Postman, evaluating their robustness and proper handling of data. These efforts have resulted in a highly stable and responsive build, ready for live deployment.

Finally, GitHub Actions worked for continuous integration and deployment (CI/CD) pipelines while Render and Vercel were for hosting the backend and frontend applications respectively. All these tools ensured rolling out new updates rapidly and with no downtime to support the agile development and frequent iteration.

In conclusion, it passes all the goals set at the start of the design and implementation of this Job Portal Application. The platform has proved itself as a robust, scalable, and user-friendly one that has a lot of potential for further modifications and ubiquity. This foundation opened the door to advanced enhancements and broader.

**5.2 FUTURE SCOPE**

Certainly, although the platform has successfully achieved the core objectives, there remains a promising future for enhancements and expansion. The project could offer even more value to users through the addition of more intelligent features in future versions. For instance, there could be a sophisticated job recommendation engine that intelligently matches users with jobs according to attributes such as skill tags, preferred locations, professional experience levels, and interaction patterns. By determining the nature of user activities with regard to preferences, the platform would be able to further refine such jobs suggestions.

Real-time chat facility enhancement is another area that could undergo tremendous transformation. Currently limited to text messaging, the system can evolve to support multimedia modes of communication. Very few features like CV previews, Voice messages, and File-attachments would greatly improve flow communication. Also, it can develop real-time notifications for new chat messages, changes in job application statuses, and system updates.

Another significant feature coming up is the video interview module. The inclusion of in-browser video conferencing tools, where the employer has an option to schedule, record, and evaluate interviews, can make remote hirings more streamlined. Google Calendar or Outlook can be synchronized with interview slots to add more convenience to both parties.

From the administrative part, a more comprehensive yet appealing analytics dashboard also does provide a look toward the future. Admin users may be provided with graphical tools showing user activity, job market trends, real-time traffic, system uptime, and job posting performance. These might serve as guidance for making improvements, identifying bottlenecks, and optimizing a system for growing demand.

Another topic is the improvement of the resume management system. The implementation of a resume parser, which would automatically extract qualifications, skills, job titles, and keywords, would enable highly focused candidate searches to be done by recruiters; most manual filtering will be reduced, and matching accuracy will improve.

The next solution would be to develop a dedicated mobile application for Android and iOS for more extensive reach and usability. With this native mobile exposure, users would be able to register, apply for jobs, receive alerts, and communicate anytime-and-anywhere. Push notifications could also be included to ensure people's real-time updates, thus improving responsiveness and retention.

There are also security enhancements that will be part of the future development. The JWT-based authentication implementation could be very well fortified by incorporating OAuth 2.0 for social logins, using LinkedIn or Google, making the dependence on the platform even stronger in terms of user trust and user convenience. Furthermore, compliance with GDPR and emerging international data protection regulations will guarantee that the platform meets international quality and ethical/legal practices in user data protection.

Advanced Job Recommendations:Advanced matching logic, which uses skill tags, location radius filters, and experience levels, is implemented.

Recommendation refinement through data analytics, taking into consideration user engagement.

Chat Upgrades:Real-time chat now supports multimedia (CV previews and audio messages).

New message and progress update notifications.

Video Interview Module:

A built-in video calling feature for remote interviewing integrated with calender facilities.

Admin Analytics Dashboard:

Visualization tools for advanced job statistics, user activity logs, and the general health of the system.

Resume Parsing and Keyword Indexing:

Logic needs to be incorporated to extract key skills, qualifications, and roles from uploaded resumes for better searching.

Mobile Application Development:To further enhance accessibility and expand user base, development of an Android/iPhone mobile app.

"More interactive educational component may also be taken into account."

**REFERENCES**

1. S. M. Raj and R. R. Rajamani, “A Job Portal with Smart Recommendation System,” *2021 International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM)*, Chennai, India, 2021, pp. 123–127. doi: 10.1109/ICSTM52236.2021.9505678
2. P. Gopal and M. S. Rajeswari, “Design and Development of Web-Based Job Portal Application,” *2020 International Conference on Intelligent Computing and Control Systems (ICICCS)*, Madurai, India, 2020, pp. 624–628. doi: 10.1109/ICICCS48265.2020.9121030
3. N. Singh and P. Sharma, “Web Based Employment Portal Using Angular and Node.js,” *2019 International Conference on Emerging Smart Computing and Informatics (ESCI)*, Pune, India, 2019, pp. 337–340. doi: 10.1109/ESCI.2019.8861882
4. R. S. Kale and S. B. Thorat, “Job Recommendation System Using Content-Based and Collaborative Filtering,” *2020 International Conference on Computer Communication and Informatics (ICCCI)*, Coimbatore, India, 2020, pp. 1–5. doi: 10.1109/ICCCI48352.2020.9104162
5. X. Wang, “Design and Implementation of Job Recruitment System Based on Cloud Platform,” *2019 IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, Chengdu, China, 2019, pp. 363–366. doi: 10.1109/ICCCBDA.2019.8721234