Markd - Technical Design Document

Table of Contents

1. Introduction
2. System Overview
3. Architecture
    - High-Level Architecture
    - Component Interactions
4. Backend Design
    - Technologies Used
    - Project Structure
    - API Design
    - Database Schema
    - Middleware
5. Frontend Design
    - Technologies Used
    - Project Structure
    - Routing
    - State Management
    - Key Components
6. Security Considerations
7. Deployment Plan
8. Unit Testing
9. Future Plans
10. Conclusion

Introduction and System Overview

Introduction

CareerSync is a modern job-finding platform designed to enable users to create, share, and discover jobs. The platform offers a seamless user experience with features like real-time updates, user authentication, job management, and job applying.

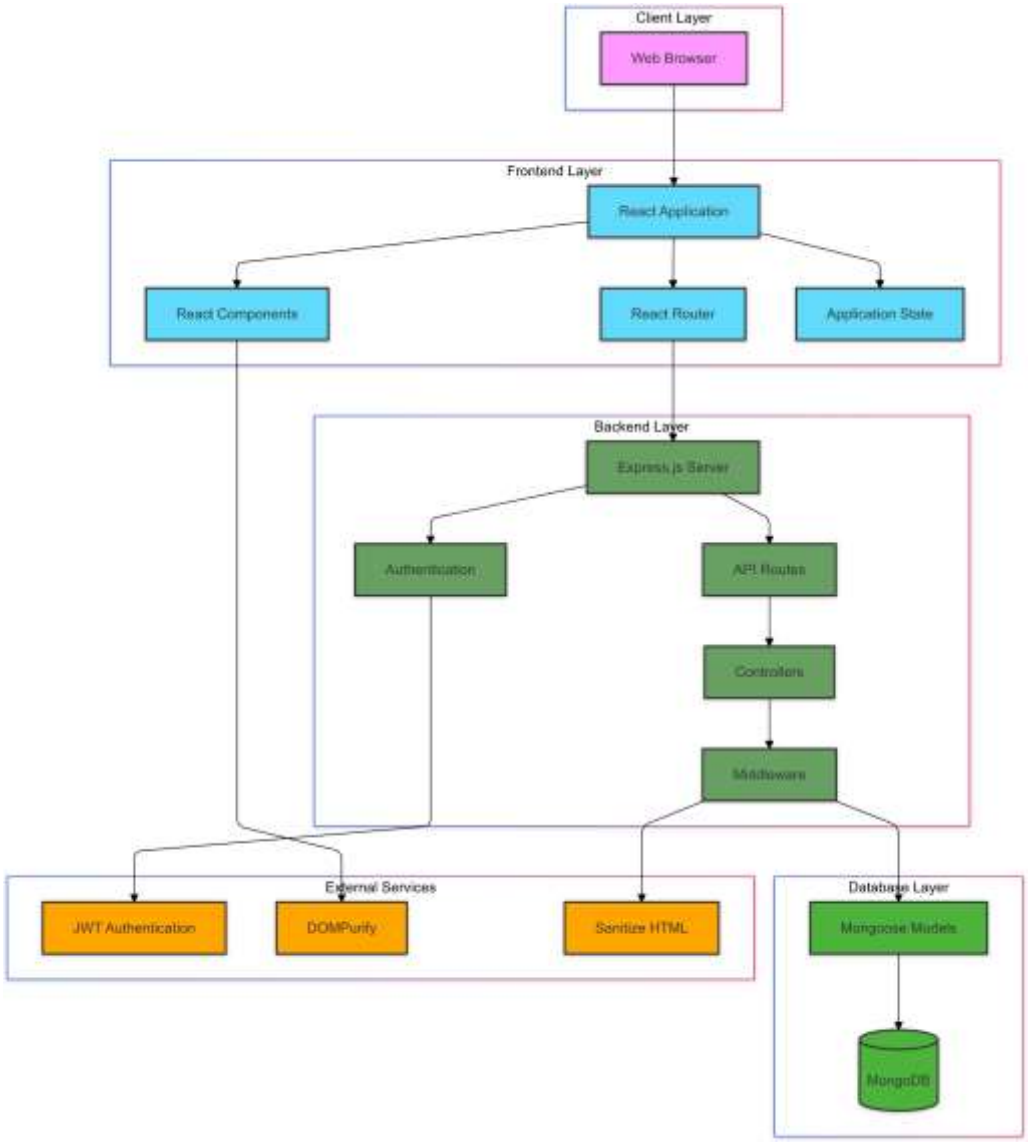
This design document provides a comprehensive overview of the project's architecture, components, technology stack, and design decisions. It aims to serve as a guide for developers, stakeholders, and contributors.

System Overview

CareerSync is built using the MERN stack (MongoDB, Express.js, React, Node.js), utilizing modern web development practices. The application is structured to provide scalability, maintainability, and a responsive user experience across devices.

Architecture

High-Level Architecture



The system follows a three-tier architecture, comprising:

1. **Frontend**: Developed with React.js, responsible for the client-side user interface and interactions.
2. **Backend API**: Built with Express.js and Node.js, handling server-side logic, API endpoints, authentication, and business logic.
3. **Database**: Utilizes MongoDB for storing user data, jobs, and related information.

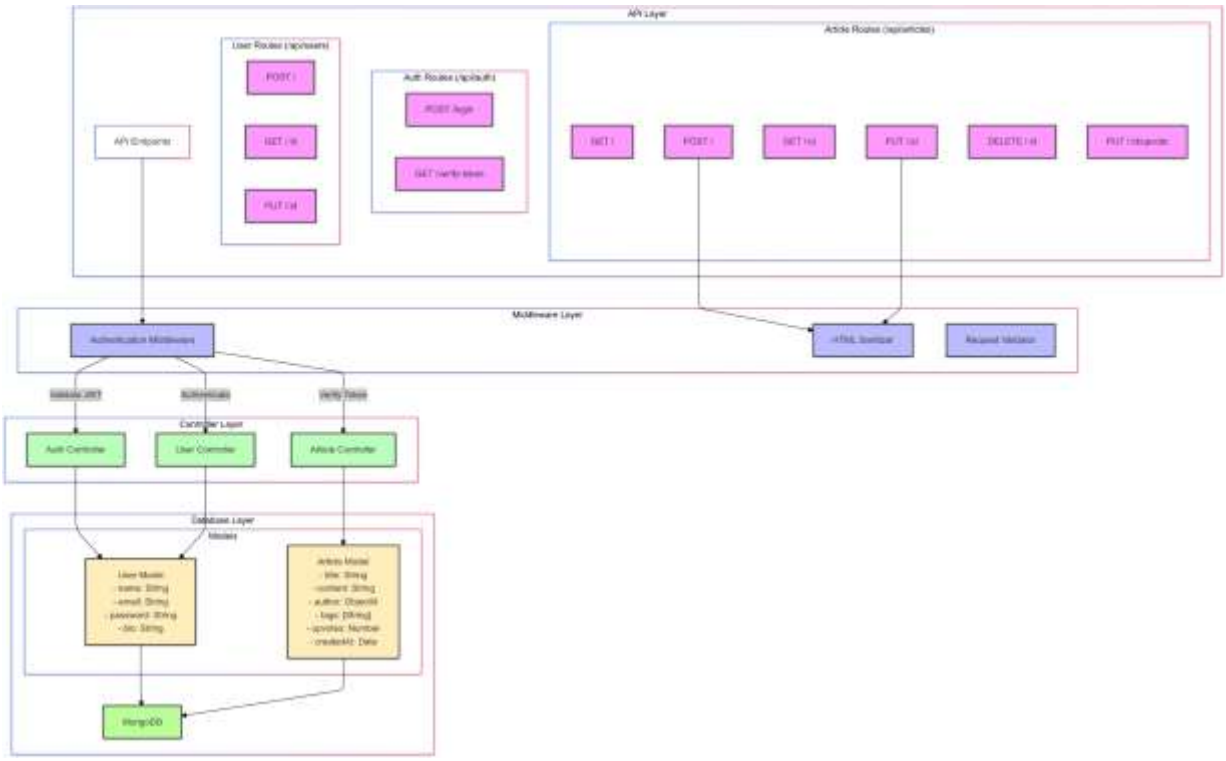
Backend Design

Technologies Used

- **Node.js**: JavaScript runtime environment.
- **Express.js**: Web application framework for building APIs.
- **MongoDB**: NoSQL database for data storage.
- **Mongoose**: ODM (Object Data Modeling) library for MongoDB.
- **JWT**: JSON Web Tokens for authentication.
- **bcrypt**: Library for hashing passwords.
- **Cors**: Allows cross-domain API requests.
- **Validator**: Ensures data format correctness.

Project Structure

- **index.js**: Entry point of the server application.
- **middleware/**: Contains middleware functions, including authentication.
- **models/**: Mongoose schemas for User, Job and Application models.
- **routes/**: Defines API endpoints for application, user, and jobs.



Backend Design - API, Database Schema and Middlewares

API Design

The backend exposes RESTful API endpoints categorized under:

- **Authentication (`/api/v1/user`)**
  - `POST /login`: User login and JWT token issuance.
  - `POST /register`: User Register and JWT token issuance.
  - `GET /logout`: Logs out the user by clearing the JWT token cookie.
  - `GET /getuser`: Retrieves the authenticated user's information.

- **Application (`/api/v1/application`)**
  - `POST /post`: Allows job seekers to submit a job application with a resume, cover letter, and personal details. Validates file type, uploads the resume to Cloudinary, and links the application to the job and employer.
  - `GET /employer/getall`: Enables employers to retrieve all job applications submitted for their job postings..
  - `GET /jobseeker/getall`: Allows job seekers to view all their submitted job applications.
  - `DELETE /delete/:id`: Allows job seekers to delete a specific job application by ID.

- **Jobs (`/api/v1/job`)**
  - `GET /getall`: Retrieves all active jobs that have not expired.
  - `POST /post`: Allows employers to post a new job with details like title, description, location, and salary. Validates required fields and salary constraints.
  - `GET /getmyjobs`: Allows employers to view all jobs posted by them.
  - `PUT /update/:id`: Enables employers to update specific job details by job ID.
  - `DELETE /delete/:id`: Allows employers to delete a specific job posting by job ID.
  - `DELETE /:id`: Retrieves the details of a specific job posting by job ID.

Database Schema

User Model

(UserSchema.js)

Fields:

- **name** (String, required): User's full name.
- **email** (String, required, unique): User's email address.
- **Phone** (Number, required): User Phone Number
- **password** (String, required): Hashed password.
- **role** (String, required): User Role.
- **CreatedAt** (Date, default=Date.now): User Creation Date

Indexes:

- Unique index on **email** for ensuring unique user emails.

Relations:

- A user can author multiple articles.

Notes:

- Passwords are hashed using bcrypt before being saved, as defined in userSchema.pre("save").

Application Model

(applicationSchema.js)

Fields:

- **name** (String, required): Name of the applicant.
- **email** (String, required): Email of the applicant.
- **coverLetter** (String, required): Cover Letter of the applicant.
- **phone** (Number, required): Number of the applicant.
- **address** (String, required): Address of the applicant.
- **resume** (Object): Resume of the applicant.
- **applicantID** (Object): Id of the applicant.
- **employerID (Object)**: Id of the employer

Indexes:

- Index on **applicantID** for chronological sorting.
- Index on **employerID** for chronological sorting.

Relations:

- Each job application is associated with an author (user).

Notes:

- The **author** field establishes a relationship between job applications and user

---

Job Model (JobSchema.js)

Fields:

- **title** (String, required): Title of the Job.
- **description** (String, required): Description of the Job.
- **category** (String, required):Category of the job.
- **country** (String, required): Country of the job.
- **city** (String, required): City of the job.
- **location** (String, required): Location of the job.
- **fixedSalary** (Number): Fixed Salary if not range.
- **salaryFrom** (Number): Salary Lower Range.
- **salaryTo** (Number): Salary Upper Range
- **expired** (Boolean, required): Is the job expired?
- **jobPostedOn** (Date, default=Date.now): Date of job posting
- **postedBy** (Object, ref="user", required): Job Poster

Indexes:

- Index on **Title** for chronological sorting.

Relations:

- Each job is associated with an author (user).

Middlewares

Authentication Middleware (auth.js)

- Validates JWT tokens sent in the `Authorization` header.
- Attaches the authenticated user's information to the request object.
- Protects routes that require authentication.

AsyncErrorHandler Middleware (catchAsyncError.js)

- Handles asynchronous errors in route handlers.
- Wraps functions in a promise catcher.
- Forwards caught errors to the next middleware.

ErrorHandle Middleware (error.js)

- Defines a custom error handler class.
- Handles invalid or expired JWT tokens.
- Manages duplicate key and cast errors.
- Sends structured error responses to clients.

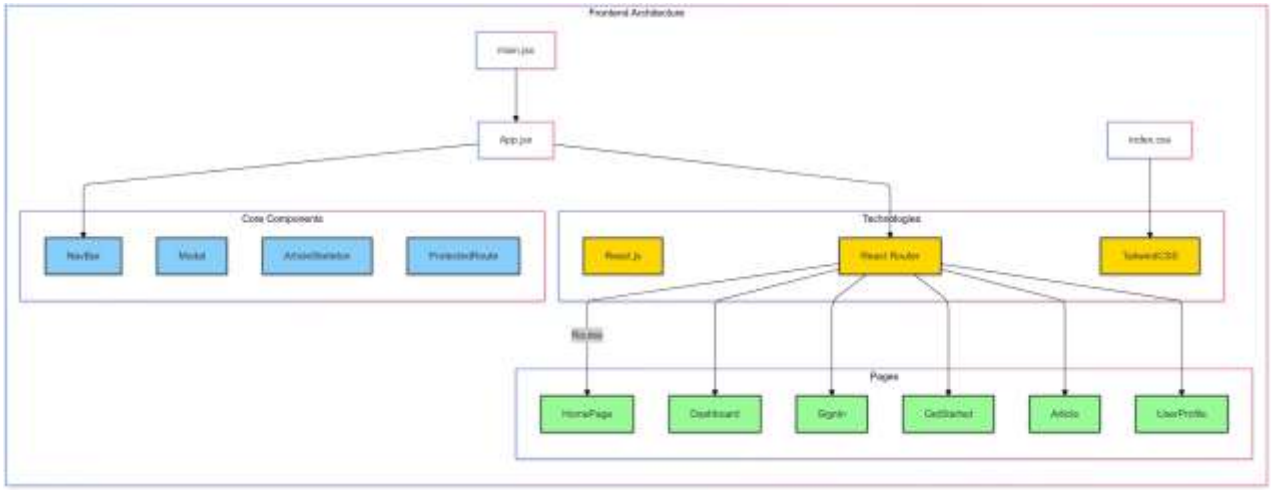
Frontend Design

Technologies Used

- **React.js**: JavaScript library for building user interfaces.
- **React Router DOM**: Handling client-side routing.
- **Tailwind CSS**: Utility-first CSS framework for styling.
- **Vite**: Build tool for faster development.

Project Structure

- **main.jsx**: Entry point of the React application.
- **App.jsx**: Main application component.
- **components/**: Reusable UI components.
- **pages/**: Page components corresponding to routes.
- **routes/**: Defines client-side routing.
- **lib/**: Utility functions.
- **index.css**: Global CSS and Tailwind directives.
- **public/**: Static assets.



Frontend Design - Routing, State Management and Key Components

---

Routing

Implemented using React Router:

- `/`: Home page.
- `/register`: User registration page.
- `/login`: User login page.
- `/job/:id`: View a specific Job details
- `/job/getall`: Get all jobs.
- `/job/post`: Post a Job.
- `/job/me`: View all my Jobs.
- `/application/:id`: View a specific Application details
- `/applications/me`: View all my applications

State Management

- **Local State**: Managed using `useState` and `useEffect` hooks.
- **Authentication State**:
  - Stored in `localStorage`.
  - Accessed via custom hooks or utility functions.
- **Data Fetching**:
  - Utilizes `fetch` API.
  - Handles loading and error states.

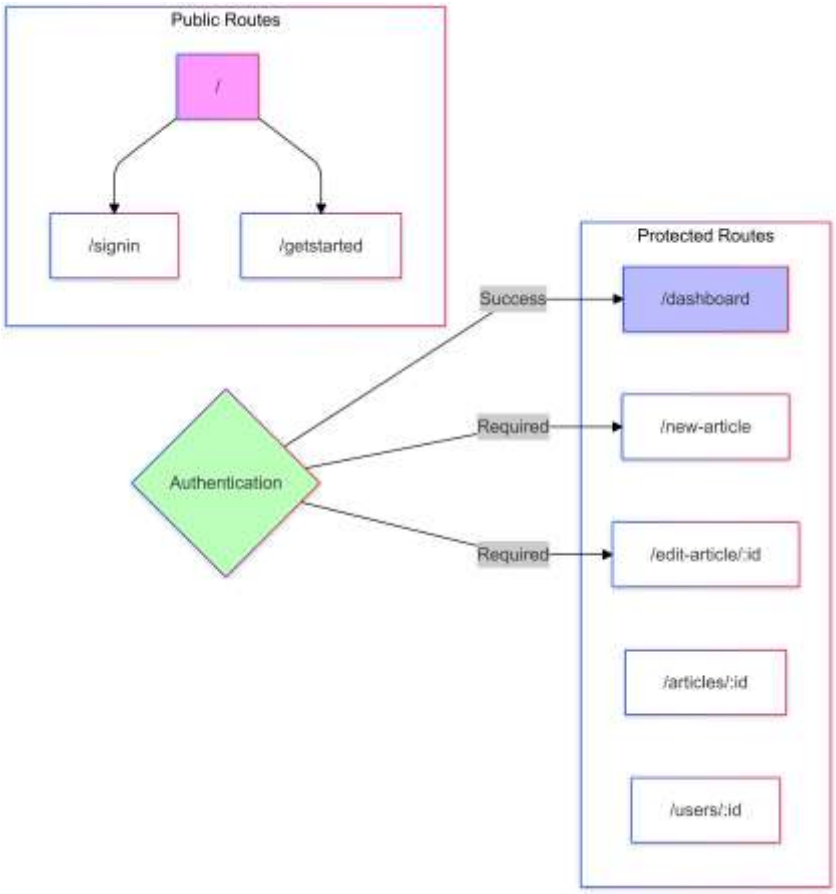
Key Components

NavBar

- Displays navigation links.
- Shows different options based on authentication state.
- Includes a logout mechanism.

Modal

- Generic modal component for displaying messages and actions.
- Used for confirmations, alerts, and information display.


Public Routes
/
/signin
/getstarted
Protected Routes
Success
/dashboard
Required
/new-article
Authentication
Required
/edit-article/:id
/articles/:id
/users/:id

Security Considerations

Authentication

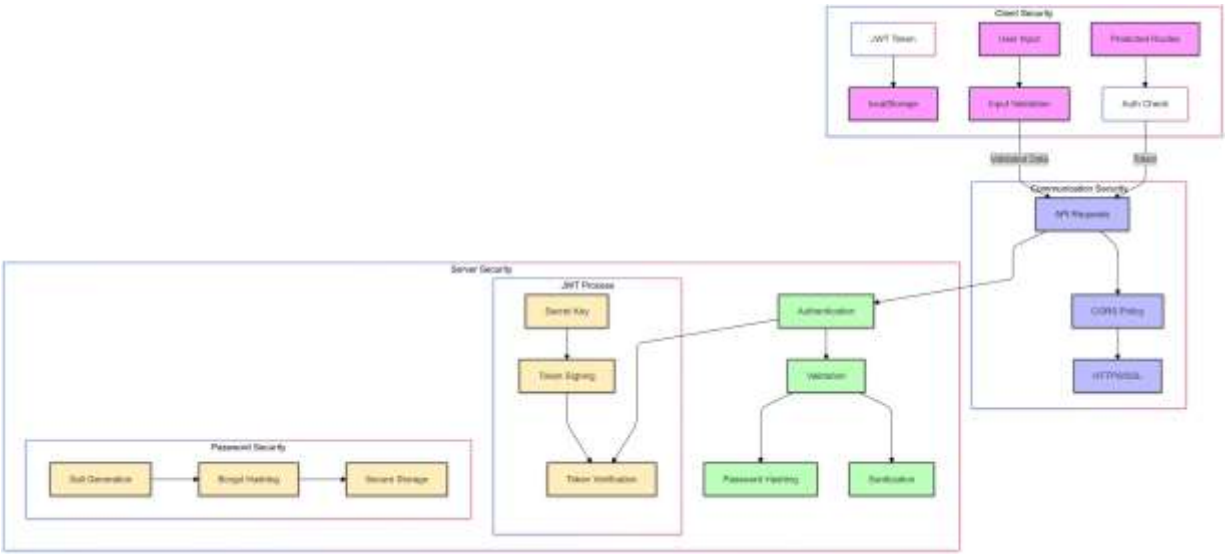
- **JWT Tokens**:
  - Securely generated and signed with a secret key.
  - Stored in the client's `localStorage`.
- **Password Security**:
  - Passwords hashed using `bcrypt` before storing in the database.
  - Plain passwords are never stored or logged.

Authorization

- **Protected Routes**:
  - Backend routes require valid JWT tokens.
  - Frontend routes use higher-order components to restrict access.
- **Input Validation**:
  - Sanitization of inputs to prevent injection attacks.
  - Validation rules applied on both client and server sides.

CORS Configuration

- **Access-Control Policies**:
  - Configured to allow requests from trusted origins.
  - Proper headers set for `Access-Control-Allow-Origin`, `Methods`, and `Headers`.



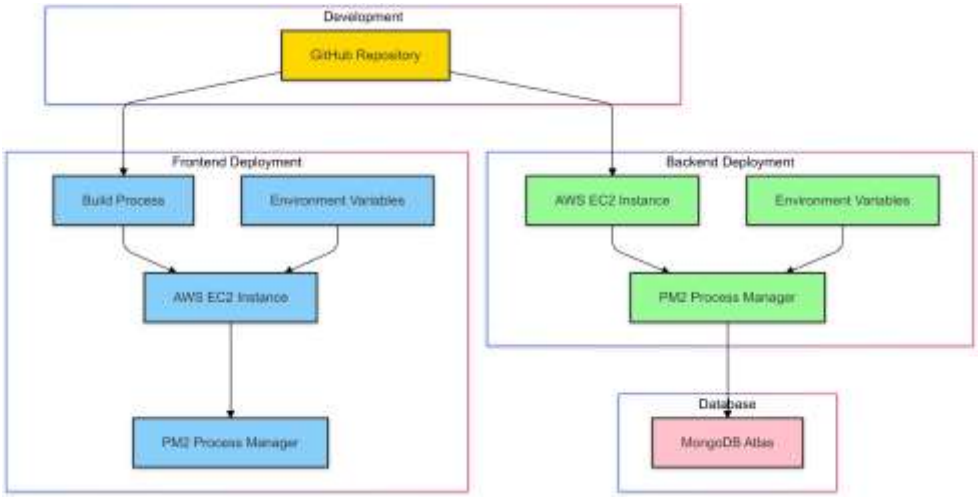
Deployment Plans

Environment Setup

- **Backend Environment Variables**:
  - `PORT`: Port number for the server.
  - `MONGO_URI`: MongoDB connection URI.
  - `JWT_SECRET`: Secret key for signing JWTs.
  - `JWT_EXPIRE`: Time of JWTs expiry.
  - `COOKIE_EXPIRE`: Time for Cookie Expiry.
  - `CLOUDINARY_CLOUD_NAME`: Cloud Name of Cloudinary.
  - `CLOUDINARY_CLOUD_NAME`: User Name of Cloudinary.
  - `CLOUDINARY_CLOUD_NAME`: Secret key for signing Cloudinary.
- **Frontend Environment Variables**:
  - `VITE_API_URL`: Base URL for the backend API.

Deployment Steps

1. **Backend Deployment**:
   - Host on platforms like Heroku, AWS EC2, or DigitalOcean.
   - Ensure environment variables are securely set.
   - Use process managers like PM2 for process management.
2. **Frontend Deployment**:
   - Build the React application using `npm run build`.
   - Host static files on services like Vercel or AWS.
3. **Domain and SSL**:
   - Configure a custom domain.
   - Set up SSL certificates for secure HTTPS communication.
4. **Database Hosting**:
   - Use managed MongoDB services like MongoDB Atlas.
   - Configure IP whitelisting and security measures.
5. **Continuous Deployment**:
   - Set up CD pipelines to automatically deploy on code changes.
   - Use GitHub Actions or other CI/CD tools.


Development
GitHub Repository
Frontend Deployment
Build Process
Environment Variables
AWS EC2 Instance
PM2 Process Manager
Backend Deployment
AWS EC2 Instance
Environment Variables
PM2 Process Manager
Database
MongoDB Atlas

Future Enhancements

Technical Improvements

- **Switch to TypeScript**:
  - Introduce TypeScript for type safety and better maintainability.
- **State Management Library**:
  - Implement Redux or Context API for more complex state needs.
- **WebSockets**:
  - Use Socket.IO for real-time features like live comments or notifications.

Feature Enhancements

- **Comment System**:
  - Allow users to comment on job postings.
- **Employer Following**:
  - Implement a social feature where users can follow employers.
- **Notifications**:
  - Real-time notifications for interactions.
- **Search Functionality**:
  - Implement search to find jobs by title, content.
- **Analytics Dashboard**:
  - Provide employers with insights on job views and interactions.

Conclusion

The CareeerSync project is a full-stack web application that We have developed using modern technologies to ensure scalability, security, and a great user experience. By using React.js for the frontend, Express.js and Node.js for the backend, and MongoDB as our database, We have created a seamless platform for knowledge sharing.

During development, there was an emphasis on clear architecture with well-organized frontend and backend directories, modular components and well-defined routes.

Working on CareerSync has taught me a lot about web development and the importance of user-centered design. It is truly a fun experience.