JOBBER

Detailed Technical Document

Table of contents-

Introduction

The purpose of this document is to outline the **design and architecture** of **Jobber**, a MERN-based web application that facilitates the posting and application process for gig opportunities. This document provides a comprehensive overview of the system's structure, components, and functionalities, serving as a blueprint for developers and stakeholders involved in the project.

Jobber is designed to address the growing demand for a centralized platform to connect gig providers with job seekers. The system enables:

- **Gig Providers**: To create and manage gig postings.
- **Job Seekers**: To browse available gigs and apply using a seamless, user-friendly interface.

This document focuses on the **technical design**, detailing the architecture, system components, data flow, and implementation strategies. It aims to establish a shared understanding among the development team to ensure consistency and alignment throughout the development lifecycle.

Technology Stack

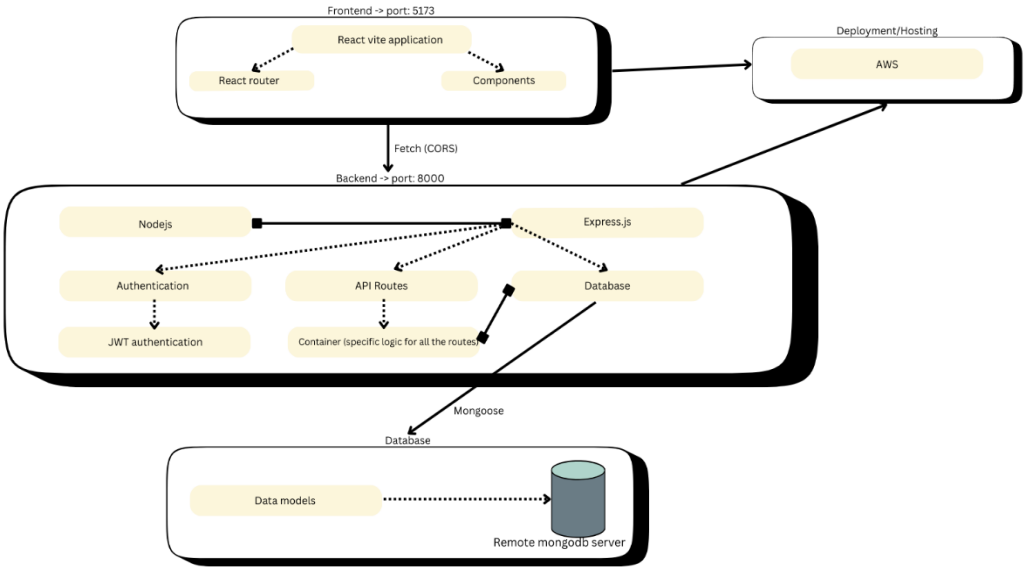
Jobber is built using the **MERN stack**, a popular combination of technologies for developing modern web applications. The MERN stack consists of:

- **MongoDB**: A NoSQL database used to store user profiles, gig postings, and application data. Its flexible schema supports dynamic data structures, enabling efficient storage and retrieval.
- **Express.js**: A lightweight and flexible backend framework that powers the RESTful APIs, handling server-side logic, routing, and middleware functionalities.
- **React**: A JavaScript library used to build the application's dynamic and responsive user interface. It ensures seamless interaction between users and the system.
- **Node.js**: A runtime environment that enables server-side execution of JavaScript, providing the foundation for scalability and real-time capabilities.

Architecture

The **Jobber** system follows a **three-tier architecture** with the following layers:

1. **Frontend (Client-Side)**:
    - Built with React.
    - Provides a responsive and interactive user interface.
    - Handles gig postings, applications, and profile management.
    - Communicates with the backend through RESTful API calls.
2. **Backend (Server-Side)**:
    - Powered by Node.js and Express.js.
    - Implements business logic and processes user requests.
    - Ensures secure data handling and session management using JWT.
    - Acts as the intermediary between the frontend and database.
3. **Database (Data Layer)**:
    - Uses MongoDB to store and manage data.
    - Maintains collections for users, gigs, and applications.
    - Handles efficient data queries and updates.



Frontend Design

Technologies Used-

- **React**: A JavaScript library for building dynamic and interactive user interfaces through reusable components. Utilized for building the user interface, allowing you to create reusable components for gig postings, applications, and user profiles. It ensures a smooth, dynamic, and responsive user experience as users interact with the system.
- **React Router DOM**: A routing library for React that enables navigation between different views and components based on the URL. Used to implement navigation between different pages (e.g., gig listings, individual gig details, user profiles). It helps manage URL paths and renders the appropriate components based on the user's navigation, enabling single-page application (SPA) functionality.
- **Tailwind CSS**: A utility-first CSS framework used for rapidly building custom designs with a responsive, mobile-first approach. Used to style the components with a utility-first approach, providing flexibility and speed in creating custom, responsive designs without writing custom CSS. It allows you to focus on structure while easily managing layout, spacing, and responsiveness.
- **Vite**: A modern build tool that provides fast and optimized development and production builds, leveraging native ES modules. Employed as the build tool for fast development and optimized production builds. It ensures quick hot-reloading during development, enabling you to see changes instantly, and provides efficient bundling for production, improving the overall performance of your application.

Directory Structure

- /src ->
  - ❖ Main.jsx – Entry point of the react app
  - ❖ App.jsx – Component structure starts
  - ❖ Index.css – Tailwind CSS
- /src/components
  - ❖ Header.jsx – Header
  - ❖ Routes.jsx – public and protected routes
  - ❖ Dashboard.jsx – Routing for all the protected routes
  - ❖ ProtectedRoute.jsx – Logic for protected routes
- /src/components/profile
  - ❖ Profile.jsx, BasicProfile.jsx
  - ❖ Card.jsx, CardSection.jsx
  - ❖ PostedSection.jsx, AppliedSection.jsx
- /src/components/postJob
  - ❖ PostJob.js
- /src/components/login_signup
  - ❖ Login.jsx
  - ❖ Signup.jsx
- /src/components/landing
  - ❖ Landing.jsx
- /src/components/jobPage
  - ❖ BasicInfo.jsx, DetailedInfo.jsx, JobPage.jsx
- /src/components/findJob
  - ❖ FindJob.jsx
  - ❖ JobCard.jsx, JobSection.jsx
  - ❖ SearchSection.jsx

Routing

Public routes

- / - Homepage or Landing page
- /login – Login page
- /signup – Signup page
- /about – About page of the site
- /contact – Contact page with owner's contact info and form

Protected routes

- /dashboard/findjob – All the available jobs displayed here
- /dashboard/postjob – Any job can be posted from here
- /dashboard/jobpage – Individual job page with detailed job information
- /dashboard/profile – User profile with track of all the applied and posted jobs and their status

Routing is implemented using React Router DOM.

State management

- State management done using useState and useEffect hook.
- Current logged in user state stored in localStorage.

Backend Design

Technologies Used

- **Node.js:** A JavaScript runtime environment for building scalable and high-performance server-side applications.
- **Express.js:** A lightweight and flexible web application framework for creating robust RESTful APIs.
- **MongoDB:** A NoSQL database used for efficient and flexible data storage and retrieval.
- **Mongoose:** An Object Data Modeling (ODM) library that provides a seamless way to interact with MongoDB.
- **JWT (JSON Web Tokens):** A secure method for implementing user authentication and session management.
- **bcrypt:** A password hashing library used to enhance security by encrypting user credentials.

Directory structure

- **container/**
  - appliedjob.js - Handles operations related to jobs applied by users.
  - applyhandler.js - Contains logic for handling job applications.
  - joblist.js - Manages job listings and their operations.
  - profilehandler.js - Handles user profile-related logic.
- **middleware/**
  - authmiddleware.js - Authentication middleware for verifying user credentials.
- **models/**
  - applicationschema.js - Schema for storing job application data.
  - joblist.js - Schema for job listings.
  - jobmapping.js - Schema for mapping jobs to users.
  - user.js - Schema for user data.
- **routes/**
  - auth.js - Routes for authentication (login, register).
  - joblist.js - Routes for job listings (CRUD operations).

- server.js - Entry point of the backend application.

Api Design

The backend exposes the following API endpoints for the job portal:

Authentication:

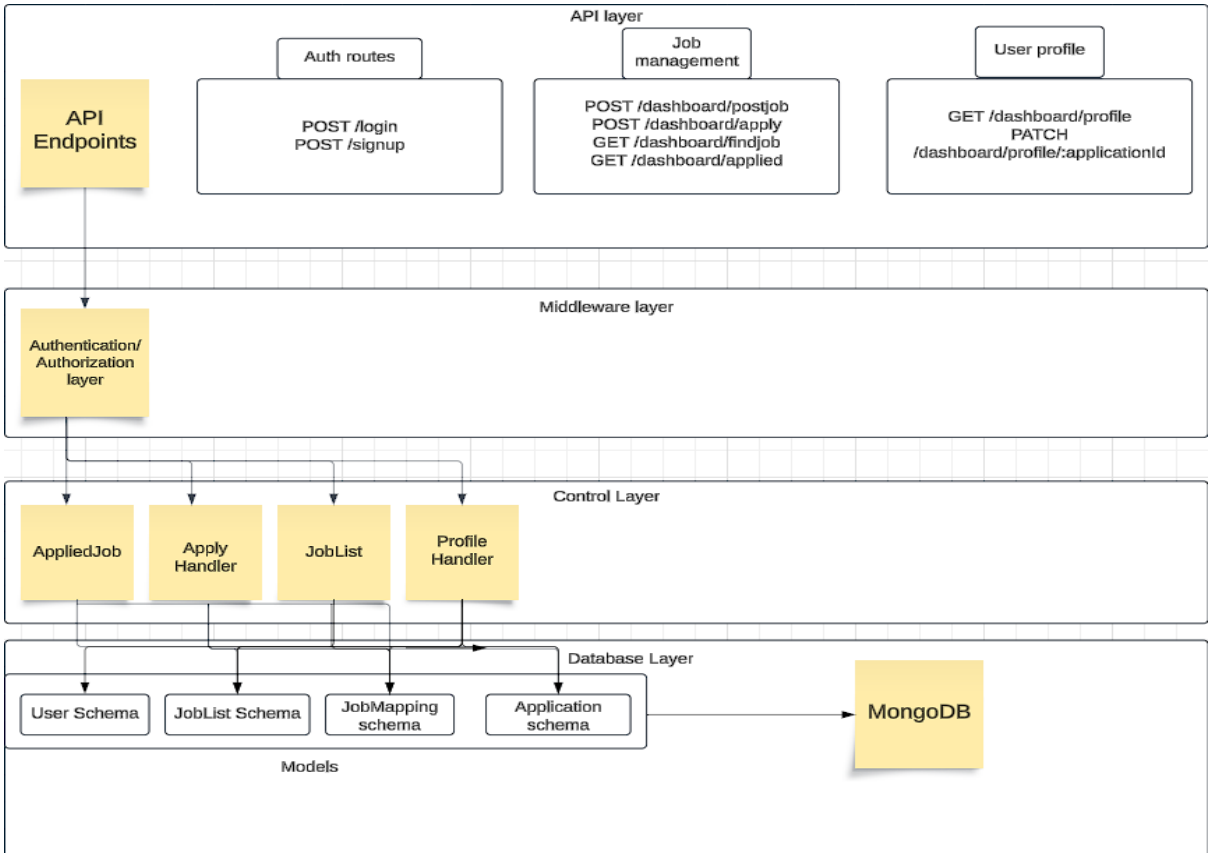
- **POST** /login
  Authenticates the user and generates a JWT token.
- **POST** /Signup
  Registers a new user and redirects to login on successful register.

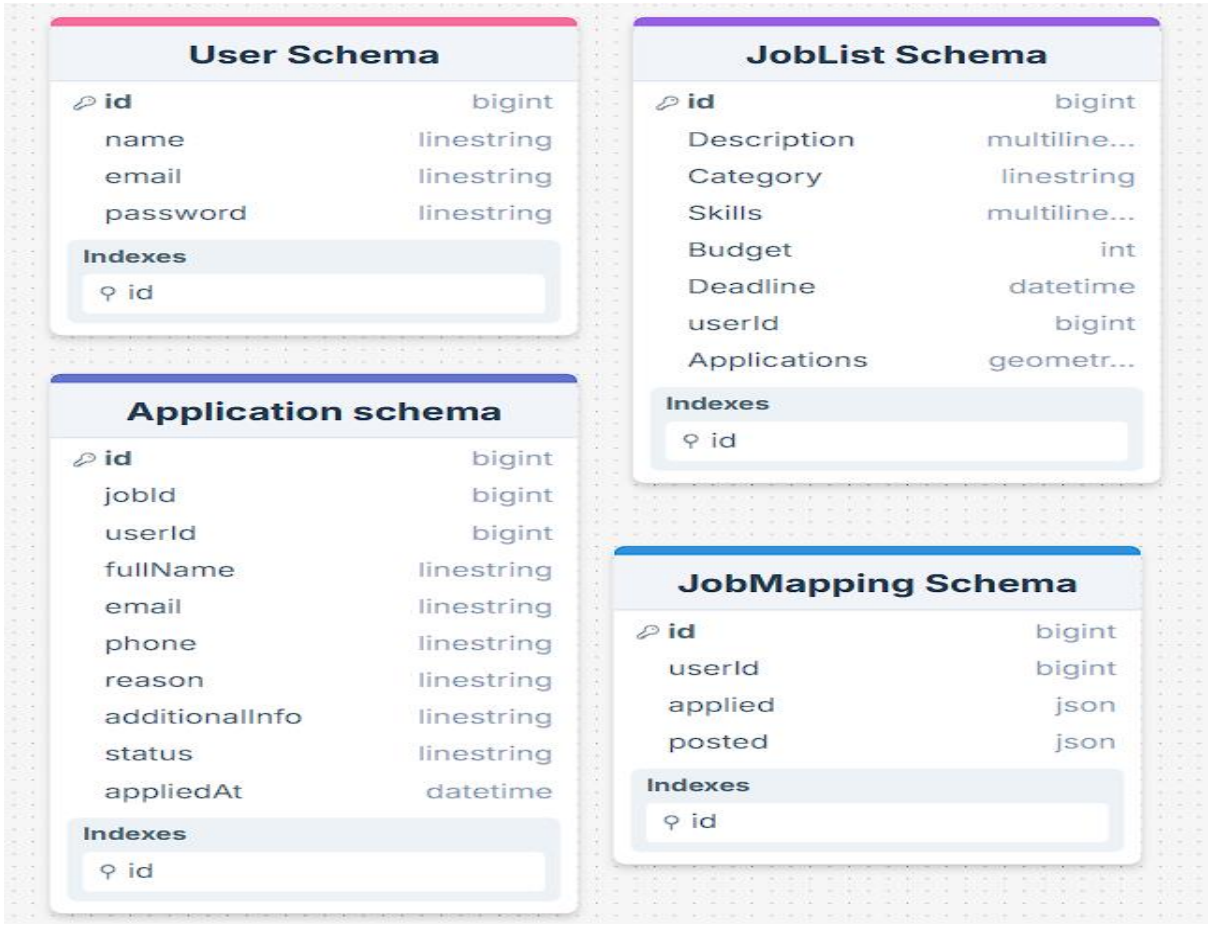
JOB MANAGEMENT:

- **POST** /dashboard/postjob
  creates a new job posting
- **GET** /dashboard/findjob
  Fetches the list of available jobs.
- **POST** /dashboard/Apply
  Allows a user to apply for a job.
- **GET** /dashboard/Applied
  Retrieves the jobs a user has applied for

User Profile:

- **GET** /dashboard/profile
  Fetches the profile details of the logged-in user, including job applications and postings.
- **PATCH** /dashboard/profile/:applicationId
  Updates the status of a job application.


API layer
Auth routes
Job management
User profile
API Endpoints
POST /login
POST /signup
POST /dashboard/postjob
POST /dashboard/apply
GET /dashboard/findjob
GET /dashboard/applied
GET /dashboard/profile
PATCH
/dashboard/profile/:applicationId
Middleware layer
Authentication/ Authorization layer
Control Layer
AppliedJob
Apply Handler
JobList
Profile Handler
Database Layer
User Schema
JobList Schema
JobMapping schema
Application schema
MongoDB
Models


User Schema
id bigint
name linestring
email linestring
password linestring
Indexes
id
JobList Schema
id bigint
Description multiline...
Category linestring
Skills multiline...
Budget int
Deadline datetime
userId bigint
Applications geometr...
Indexes
id
Application schema
id bigint
jobId bigint
userId bigint
fullName linestring
email linestring
phone linestring
reason linestring
additionalInfo linestring
status linestring
appliedAt datetime
Indexes
id
JobMapping Schema
id bigint
userId bigint
applied json
posted json
Indexes
id

Database Schema

User Schema

- **Collection Name**: Users
  This schema is stored in the user's collection in MongoDB.
- Fields:
    - name (String, *required*, *unique*)

Constraints: Must be unique, Required for user registration.

    - email (String, *required*, *unique*, *trimmed*)

Constraints: Must be unique, Required for user registration.

    - password (String, *required*)
        - The hashed password of the user.
        - Constraints:
            - Required for user authentication.
- **Indexes**: Unique indexes are applied to name and email to ensure no duplicate entry.
- **Validation**: Ensures that all required fields are provided when creating or updating a user.

JobList Schema

The JobListSchema represents job postings in the application.

- **Collection Name**: joblists
- **Fields**:
    - **name** (String, *required*, *unique*): The title of the job.
    - **description** (String, *required*, *unique*, *trimmed*): A detailed description of the job.
    - **category** (String, *required*): The category of the job.
    - **skills** ([String], *required*): A list of skills required for the job.
    - **budget** (Number, *required*, *min: 0*): The budget for the job.
    - **deadline** (Date, *required*): The deadline to complete the job.
    - **userId** (ObjectId, *required*, *ref: 'User'*): References the user who posted the job.

    - **applications** ([ObjectId], *ref: 'Application'*): References the applications submitted for the job.

- **Additional Features**:
    - **Timestamps**: Automatically tracks created At and updated At.

This schema is connected to User and Application through references.

Application Schema

The ApplicationSchema represents job applications submitted by users.

- **Collection Name**: applications
- **Fields**:
    - **jobId** (ObjectId, *required*, *ref: 'JobList'*): References the job being applied for.
    - **userId** (ObjectId, *required*, *ref: 'User'*): References the user who applied for the job.
    - **fullName** (String, *required*): The full name of the applicant.
    - **email** (String, *required*): The email address of the applicant.
    - **phone** (String, *required*): The phone number of the applicant.
    - **reason** (String, *required*): The reason or cover letter provided by the applicant.
    - **additionalInfo** (String): Any additional information provided by the applicant.
    - **status** (String, *default: 'Pending'*): The current status of the application (e.g., Pending, Accepted, Rejected).
    - **appliedAt** (Date, *default: Date.now*): The timestamp when the application was submitted.
- **Additional Features**:
    - The schema connects to JobList and User through references, linking
    - applications to jobs and applicants.

JobMapping Schema

The JobMappingSchema tracks the jobs a user has posted and applied for.

- **Collection Name**: jobmappings
- **Fields**:

- **userId** (ObjectId, *required*, *ref: 'User'*): References the user associated with the mapping.
- **applied** ([ObjectId], *ref: 'Application'*): An array of application IDs representing jobs the user has applied for.
- **posted** ([ObjectId], *ref: 'JobList'*): An array of job IDs representing jobs the user has posted.

- **Features**:
  - Links a user to their job postings and applications using references to JobList and Application.
  - Facilitates easy access to all jobs a user has applied for and posted.

Deployment Plans

Environment Setup

Clearly define the environment variables for both backend and frontend:

- **Backend Environment Variables**:
    - PORT: Port number for the server.
    - DB_CONNECTION_STRING: MongoDB connection URI.
    - JWT_SECRET: Secret key for signing JWTs.
- **Frontend Environment Variables**:
    - VITE_API_URL: Base URL for the backend API.

Deployment Steps

Outline the specific steps to deploy both backend and frontend:

Backend Deployment:

- Host the backend on AWS EC2

Frontend Deployment:

- Host the static files on services such as **Vercel**, **Netlify**, or **AWS S3** with CloudFront for efficient delivery.

Database Hosting:

- Use managed database services like **MongoDB Atlas** for seamless scaling and monitoring.

Future Scope

- **Payment Integration**: Integrate payment gateways to allow users to settle their payments seamlessly.
- **Socket.io**: Use Socket.io to integrate chatting functionality for improved workflow.
- **WebRTC**: Implement video chat functionality to facilitate virtual interviews.
- **Mobile Application**: Develop a mobile version of the platform for better accessibility and user engagement.
- **AI-Assisted Features**: Introduce AI-based recommendations or assistance, such as candidate matching or automated interview scheduling.

Conclusion

Jobber is a full-stack web application built using the MERN stack, leveraging its versatility, ease of deployment, and robust features. The project has been developed with the latest syntax and industry best practices, ensuring a modern, scalable, and maintainable solution.

The architecture has been carefully designed to ensure a clear separation between the frontend and backend, allowing for seamless debugging and efficient feature development in future iterations. This modular approach not only simplifies maintenance but also ensures the platform is well-suited for future scalability and enhancements, making it a reliable and adaptable solution for long-term use.