Name: Satwik Kanhere

Roll No: 2110991287

Group: G-1

JobHunt. – Online Job Portal

The job portal I'm developing serves as a dynamic hub catering to the needs of both employers and job seekers. Employers gain privileged access, leveraging their accounts to craft comprehensive job listings. They have the autonomy to fine-tune, update, and modify these listings as needed to accurately represent their company's requirements. On the flip side, the portal offers an intuitive interface for job seekers. They create personalized profiles and navigate through a myriad of job opportunities. Tailored filters and search functionalities empower them to pinpoint roles that match their skills, experience, and aspirations. Crucially, the platform incorporates robust communication features. Employers and employees seamlessly share job postings via email, fostering a collaborative ecosystem where information dissemination is efficient and swift. To maintain the integrity of the job listings, only authorized employers possess editing privileges. This ensures that job details remain accurate and up to date, elevating the credibility of the platform for both parties involved. In essence, this job portal aims to revolutionize the job search experience, offering a user-centric, secure, and interactive platform that bridges the gap between employers and potential candidates. Its user-friendly design and emphasis on fostering meaningful connections aim to streamline the hiring process while enhancing opportunities for both employers and job seekers.

Client Folder (in a React context):

Purpose: Contains code and resources that are bundled and run in the user's browser.

Contents: Components, views, styles, and assets needed for the user interface.

Functionality: Manages the user interface and handles user interactions.

Tasks: Rendering components, managing state, handling user events, and making API requests to the server.

Example Uses: Components rendering UI elements, managing front-end logic, and initiating requests to the server for data.

Server Folder (in a React context):

Purpose: May hold code related to the backend or server-side logic (Node.js or other backend technology).

Contents: Backend logic, API routes, database connections, and server configurations.

Functionality: Handles server-side processing, database interactions, and communication with the client-side.

Tasks: Serving API endpoints, handling requests from the client, interacting with databases, and performing server-side operations.

Example Uses: Implementing server-side routes, processing client requests, and managing server resources.

In a React project, the "client" folder typically contains the React components, views, styles, and other client-side resources necessary for rendering the UI and managing the front-end logic. The "server" folder, if present in the same React project, might hold the backend logic and configurations if your project includes a server-side component, like a backend API built with Node.js or another server technology, or any server-related functionality.

Public Folder:

Purpose: Contains files that are publicly accessible and not processed by the build tools.

Contents:

index.html: The main HTML file that serves as the entry point for your React application. It typically contains a element where your React app will be rendered.

Static Assets: Any static files like images, icons, or other assets that need to be directly accessed by the browser.

Source (Src) Folder:

Purpose: Contains the source code of your React application.

Contents:

index.js: The main JavaScript file where your React app is initialized. It typically renders your app into the index.html file.

components Folder: Holds reusable React components that form the building blocks of your UI.

App.js or similar: The main component that acts as the root of your React application. It renders other components and manages the app's structure.

Other JavaScript/JSX files: Additional files containing components, utilities, or logic used throughout the application.

styles or CSS Folder: CSS files or stylesheets used to style your components.

Brief Explanation:

index.html: This file is the entry point of your React application and is the skeleton structure where your React components will be injected.

index.js: This file is responsible for initializing the React application by rendering the root component (App.js or another component) into the index.html file.

components Folder: Contains individual reusable React components. These components can be structured hierarchically to create complex UI structures.

Workflow:

The index.html file loads in the browser, typically with a element where the React app will be mounted.

The index.js file initializes the React app by rendering the root component (App.js) into this designated .

Components from the components folder are imported into the App.js file and nested to create the UI structure.

Each component handles its logic and rendering, composing the overall UI.

This structure provides a modular approach, allowing developers to organize components, styles, and assets efficiently while building React applications.

1. Employee connecting:

This directory streamlines the employee onboarding journey. It houses components responsible for the employee signup process, encompassing authentication mechanisms, profile creation, and personalized permission allocation. It ensures a user-friendly interface for employees, granting access based on roles or departments. Additionally, it manages security measures like password encryption and email verification, ensuring a secure and tailored experience for each employee joining the platform.

2. Employer connecting:

Focused on employer functionalities, this section centralizes components vital for employer actions. It oversees employer authentication, providing access to job posting tools, applicant management interfaces, and employer-specific settings. Security features, such as two-factor authentication and account verification, safeguard employer access, allowing them to efficiently manage job listings, applicant data, and company profiles within the platform.

3. landing displayed:

At the forefront of user engagement, this directory curate’s components shaping the platform's landing page. It amalgamates design elements, interactive modules, and compelling content to create an inviting and informative entry point. Engaging visuals, concise yet informative texts, and intuitive navigation elements work synergistically, captivating visitors. Its dynamic nature facilitates user interaction through sliders, call-to-action buttons, and concise highlights, presenting the platform's unique selling points and fostering exploration.

4. posted content:

This folder orchestrates job posting management components. It empowers employers to create, modify, and present job listings with precision. Components within manage form inputs for job details, enable editing functionalities, and ensure accurate job representation. Robust validation mechanisms and preview features guarantee error-free and compelling job postings, elevating the platform's credibility and attractiveness to potential job seekers. Moreover, it enables communication channels, like applicant tracking systems, fostering seamless interactions between employers and applicants.

Each folder functions as a dedicated hub, encapsulating components and functionalities tailored to specific user roles or crucial aspects of the platform, contributing cohesively to an efficient, engaging, and secure job portal experience.

Global.css

The Global.css file contains style declarations utilizing the Rubik font from Google Fonts. It assigns specific font families to different classes within the React application's UI elements, ensuring consistency and a cohesive visual appearance. The file sets different font weights for various classes to provide hierarchy and readability across the application's components.

.LogoFinished: Applies the "Rubik" font family to the logo component or element.

`.linksStyled, .AuthlinksStyled, .HeadingStyled, .ParaStyled:` Sets the "Rubik" font family and a lighter font weight to various styled components within the application, ensuring uniform typography and readability.

```
useEffect(() => {
  const callAboutUsPage = async () => {
    try {
      const res = await axios.get("/about", {
        headers: {
          Accept: 'application/json',
          'Content-Type': 'application/json',
        },
        withCredentials: true,
      });

      if (res.status === 200) {
        setUserData(res.data); // Set user data to state
        console.log(res.data);
      } else {
        throw new Error(res.statusText);
      }
    } catch (err) {
      console.log(err);
    }
  };
```

Absolutely! Let's delve deeper into the complex code within the `About.js` file:

`About.js`

The `About.js` file is a React functional component responsible for displaying user data fetched from an API endpoint. It utilizes React hooks (`useState` and `useEffect`) and the `axios` library for asynchronous data fetching.

Explanation:

`useEffect:` This hook is used to perform side effects in functional components. In this case, it's utilized to make an API call when the component mounts.

callAboutUsPage: This asynchronous function is responsible for fetching data from the /about endpoint using axios.

axios.get: Initiates a GET request to the /about endpoint with specified headers and configurations.

headers: Includes headers for accepting and sending JSON content.

withCredentials: Indicates that cross-site requests should include credentials such as cookies.

Handling Response:

If the response status is 200 (OK), the user data fetched from the server (res.data) is set to the userData state using setUserData.

If the response status is not 200, an error is thrown.

Error Handling: Catches any potential errors during the API call and logs them to the console for debugging purposes.

return Statement: Renders the UI component displaying fetched user data.

Structured UI:

Utilizes Tailwind CSS classes for styling.

Displays user details (name, email, and ID) fetched from the API within a structured and styled container.

Utilizes userData state to dynamically render user-specific information within <p> elements.

This component fetches user data asynchronously and presents it in a visually appealing and organized manner on the UI, providing users with relevant information on the "About" page of the application.

Contact.js Component Overview:

State Management:

State Initialization: It sets up initial state using useState for userData, a state object containing name, email, phone, and message.

Asynchronous Data Fetching:

Fetching User Data: Utilizes axios to perform an asynchronous GET request to an API endpoint (/getData).

Data Handling: Upon successful API response (status 200), updates the userData state with fetched user details.

Side Effects with useEffect:

useEffect Hook: Executes an effect (in this case, fetching user data) when the component mounts.

Empty Dependency Array: Ensures the effect runs only once when the component loads.

UI Rendering:

Form Structure: Renders a form component containing input fields for name, email, phone, and a textarea for a message.

Dynamically Populated Fields: Utilizes userData state to dynamically populate the form fields with fetched user data.

Functionality Summary:

The Contact.js component serves as a contact form page within the application. It manages the state for user data and fetches user information from an API endpoint using axios. When the component mounts, it initiates an API request to retrieve user details and populates the contact form fields with this data. Users can interact with the form, potentially updating the displayed user information or submitting additional contact details or messages.

This component showcases the integration of React hooks (useState and useEffect) for state management and side effects, respectively, enabling the dynamic rendering of user data within a contact form interface.

Login.js Component Overview:

Libraries Used:

React Libraries: Utilizes React's useState hook for managing component state and useNavigate for programmatic navigation within the app.

react-toastify: Integrates toast notifications for displaying success or error messages to the user upon login attempts.

State Management:

State Initialization: Initializes state using useState for formData, which stores email and password for user login.

Functions:

handleChange: Handles input change events and updates the formData state accordingly.

handleSuccess: Triggers a success toast notification upon successful login and navigates to the homepage.

postData: Performs an asynchronous POST request to the /signin endpoint using axios.

handleSubmit: Handles form submission, triggers the postData function, and logs the form data.

UI Rendering:

Toast Notifications: Configures and renders toast notifications using react-toastify for displaying login success or error messages.

Login Form: Renders a form component with input fields for email and password.

Utilizes onChange event handlers to update the formData state when inputs change.

Submits form data to the server upon button click.

Functionality Summary:

The Login.js component represents the login page of the application. It manages the state for user login credentials and handles form submission events. When the form is submitted, it triggers an asynchronous POST request to the /signin endpoint using axios to authenticate the user. Upon successful login, a success toast notification is displayed, and the user is redirected to the homepage. Error handling is implemented to notify users of incorrect credentials or other login-related errors using toast notifications.

This component exemplifies the use of React hooks, axios for HTTP requests, and toast notifications to enhance the user experience during the login process.

Home.js Component Overview:

Component Structure:

Component Import: Imports necessary components like Navbar, UserDash, DummyPost, and UserDummyDash.

Styling Import: Imports a global CSS file (Global.css) to apply consistent styles across the application.

UI Rendering:

Navbar Component: Renders a navigation bar component (Navbar) at the top of the page.

Content Layout:

Flexbox Layout (md:flex): Implements a flexbox layout for responsive design.

Title and Description: Displays a title (h1) and a description (p) introducing the purpose of the page.

Classes (HeadingStyled and ParaStyled): Applies styling classes to the title and description to maintain consistent typography and layout.

Additional Components:

UserDash, DummyPost, UserDummyDash: Includes components like UserDash, DummyPost, and UserDummyDash within the Home component, suggesting the possible rendering of content related to user dashboards or posts on the homepage.

Functionality Summary:

The Home.js component represents the homepage of the application. It includes a navigation bar at the top and a content section showcasing a title and a description inviting users to explore job opportunities. The layout uses flexbox for responsiveness, and classes (HeadingStyled and ParaStyled) apply consistent styling to the title and description. Additionally, it hints at the inclusion of further components (UserDash, DummyPost, UserDummyDash) that may display user-specific content or posts on the homepage.

This component sets the stage for the homepage's layout and content, providing a starting point for users to navigate and explore job-related information or user-specific dashboards.

Navbar.js Component Overview:

Libraries Used:

React Libraries: Imports necessary React functionalities like useState, useEffect, and components from React Router (Link, useNavigate).

External Libraries: Includes components from react-icons for the hamburger menu icon and react-modal for modal functionality.

State Management:

State Initialization: Uses useState to manage various states such as isOpen, isMobile, showModal, registrationType, and toggle.

Window Width Check: Determines the isMobile state based on the window width for responsive behavior.

Functions:

Event Handlers:

handleLogout: Performs a logout action by making an API call to /logout.

openModal and closeModal: Functions to control the visibility of the modal.

handleRegistrationType: Handles the selection of registration type (employee or employer).

UI Rendering:

Navbar Layout: Defines a navigation bar layout containing the logo, navigation links, and a logout button.

Responsive Design:

Hamburger Menu: Renders a hamburger menu icon for mobile view (isMobile) to toggle the menu.

Conditional Rendering: Shows/hides elements based on the isMobile state to maintain responsiveness.

Modal and Links:

Modal Component: Utilizes react-modal to handle registration-related modals.

Navigation Links (Link): Provides navigation links for Home, About, and Careers within the navbar.

Functionality Summary:

The Navbar.js component represents the navigation bar of the application. It handles responsive behavior using the isMobile state, displaying different elements based on screen width. It includes navigation links, a logout button, and a hamburger menu icon for mobile view to toggle the menu. Additionally, it incorporates modals for registration and logout functionalities, offering users a seamless navigation experience and access to important features like logout and registration.

SignUp.js Component Overview:

Libraries Used:

React Libraries: Utilizes useState for managing component state and useNavigate from React Router for programmatic navigation.

External Libraries: Includes react-toastify for displaying toast notifications and axios for handling HTTP requests.

State Management:

State Initialization: Uses useState to manage the form data (name, email, phone, work, password, cpassword).

Functions:

handleChange: Handles input changes and updates the formData state accordingly.

handleToast: Utilizes react-toastify to display toast notifications for different registration outcomes (success or error).

Registration Handling:

postData: Performs an asynchronous POST request to /register using axios to register a user.

Form Submission (handleSubmit): Handles form submission, triggers the postData function, and displays relevant toast notifications based on the registration outcome.

UI Rendering:

Form Layout: Renders a registration form with input fields for name, email, phone, work, password, and cpassword.

Button (Register): Includes a "Register" button that triggers the form submission upon click.

Links (Link): Provides a link to redirect users to the login page.

Functionality Summary:

The SignUp.js component represents the user registration page of the application. It manages form data using React's state (useState) and handles user input changes (handleChange). Upon form submission, it sends a POST request to the /register endpoint using axios for user registration. Toast notifications using react-toastify provide feedback to users about successful or failed registration attempts. The component offers a registration form with various input fields and a "Register" button, along with a link to redirect users to the login page after successful registration.

AboutEmployer.js Component Overview:

Libraries Used:

React Libraries: Utilizes useEffect and useState for managing component lifecycle and state.

External Libraries: Imports axios for handling HTTP requests.

State Management:

State Initialization: Initializes the userData state using useState to store employer-related data.

Data Fetching:

useEffect Hook: Fetches employer data using an asynchronous GET request to the /aboutemployer endpoint using axios.

Dependency Array: Ensures that the effect runs only once (on component mount) due to the empty dependency array ([]).

UI Rendering:

Display User Data: Renders user details (name, email, ID) obtained from the fetched userData.

Conditional Rendering: Conditionally displays fetched user details within a styled container.

Additional Components:

EditDummy Component: Includes the EditDummy component within the AboutEmployer component. Presumably, this component enables editing or configuring employer-related information.

Functionality Summary:

The AboutEmployer.js component serves as a page to display employer-specific information. Upon component mount, it triggers an HTTP GET request to fetch employer data from the /aboutemployer endpoint using axios. It renders the retrieved data, such as the employer's name, email, and ID, within a styled container. Additionally, it includes an EditDummy component that potentially allows the employer to edit or configure certain details within the employer profile or settings.

This component provides a snapshot view of employer information and possibly incorporates functionality for employers to manage or edit their profile details through the EditDummy component.

AuthenticationEmployer.js Component Overview:

Libraries Used:

React Libraries: Utilizes useState for managing component state and useNavigate from React Router for programmatic navigation.

External Libraries: Imports react-toastify for displaying toast notifications and axios for handling HTTP requests.

State Management:

State Initialization: Initializes the formData state using useState to store employer login credentials (email and password).

Functions:

handleChange: Handles input changes and updates the formData state accordingly.

handleSuccess: Displays a success toast notification upon successful login and redirects to the employer main page.

postData: Performs an asynchronous POST request to /signinemployer using axios for employer authentication.

handleSubmit: Handles form submission, triggers the postData function, and displays relevant toast notifications based on the login outcome.

UI Rendering:

Form Layout: Renders a login form with input fields for email and password.

Button (Login): Includes a "Login" button that triggers the form submission upon click.

Toast Container (ToastContainer): Sets up the toast notifications' configuration using react-toastify.

Functionality Summary:

The AuthenticationEmployer.js component represents the employer login page of the application. It manages login credentials using React's state (useState) and handles user input changes (handleChange). Upon form submission, it sends a POST request to the /signinemployer endpoint using axios for employer authentication. Toast notifications using react-toastify provide feedback to employers about successful or failed login attempts. The component offers a login form with input fields for email and password, along with a "Login" button that triggers the login process and redirects to the employer's main page upon successful authentication.

Top of Form

MainHeaderEmployer.js Component Overview:

Libraries Used:

React Libraries: Utilizes useState and useEffect for managing component state and lifecycle.

External Libraries: Imports axios for handling HTTP requests.

State Management:

State Initialization: Initializes the userData state using useState to store employer-related data.

Data Fetching:

useEffect Hook: Fetches employer data using an asynchronous GET request to the /aboutemployer endpoint using axios.

Dependency Array: Ensures that the effect runs only once (on component mount) due to the empty dependency array ([]).

UI Rendering:

Navbar Component: Includes the NavbarEmployer component that presumably contains the navigation bar specific to the employer section.

Display User Data: Renders a welcome message using the employer's name obtained from the fetched userData.

Additional Components: Includes other components (UserDash, etc.) relevant to the employer section. These might contain features or sections for employer-specific functionalities.

Functionality Summary:

The MainHeaderEmployer.js component represents the main header section for employers. Upon component mount, it triggers an HTTP GET request to fetch employer data from the /aboutemployer endpoint using axios. It displays a welcome message containing the employer's name obtained from the fetched userData. Additionally, it includes the NavbarEmployer component for navigation and possibly incorporates other components (UserDash, etc.) that serve specific functionalities or sections dedicated to employers within the application.

NavbarEmployer.js Component Overview:

Libraries Used:

React Libraries: Utilizes useState and useEffect for managing component state and lifecycle.

External Libraries: Imports axios for handling HTTP requests and react-icons for icons.

State Management:

State Initialization: Initializes various states using useState to manage the Navbar's behavior and rendering.

Event Handling:

useEffect Hook: Monitors the window size and updates the isMobile state accordingly for responsive behavior.

Handle Logout: Performs a logout action via an asynchronous GET request to the /logoutemployer endpoint when the logout button is clicked.

UI Rendering:

Navbar Structure: Generates a navbar structure with branding, navigation links, and logout functionality.

Conditional Rendering: Handles the conditional rendering of navbar elements based on the window size for responsiveness.

Hamburger Menu: Provides a toggle functionality for mobile view using the GiHamburgerMenu icon.

Navigation Links: Includes links to different sections like Home, About, and Careers using Link from react-router-dom.

Functionality Summary:

The NavbarEmployer.js component represents the navigation bar for employer-specific sections within the application. It manages the display of navigation links, such as Home, About, and Careers, with conditional rendering based on window size for responsiveness. Additionally, it provides a logout feature that triggers a logout action via an HTTP GET request to the /logoutemployer endpoint when the logout button is clicked.

The RegistrationEmployer component appears to be a form that facilitates the registration of employers. Here's a breakdown of its structure and functionality:

Libraries Used:

React Hooks: Utilizes the useState hook for managing component state.

React Router: Imports the useNavigate hook from react-router-dom for navigation.

State Management:

useState Hook: Manages form data by storing it within the component's state.

Form Handling:

Form Structure: Contains a form with various input fields to collect employer-related information.

Form Fields: Includes fields such as Name, Email, Phone, Work, Password, Company Information (Name, Industry, Company Size), Contact Information (Name, Title, Email, Phone), and Company Address (Street Address, City, State, Postal Code, Country).

Handle Change: Utilizes the handleChange function to update the form state dynamically based on user input.

Handle Submit: Invokes the handleSubmit function on form submission. It prevents the default form behavior, validates the form data, performs a POST request to the /employerregister endpoint with the form data using fetch, and handles responses accordingly (showing success or error messages).

UI Rendering:

Input Fields: Renders input fields for various data categories, wrapped within appropriate labels.

Register Button: Provides a submit button to initiate the registration process.

Navigation:

useNavigate Hook: Utilizes the navigate function to redirect users to the employer login page ("/emplogin") upon successful form submission.

The form structure is designed to collect comprehensive information related to employer registration.

Validation logic for ensuring proper data entry might be added within the handleSubmit function to enhance user data integrity.

This component aims to gather detailed information required for employer registration and handle the form submission process.

This HeroSection component seems to represent a section of a landing page, focusing on displaying introductory content and user-related information. Here's an overview of its structure and functionality:

Component Features:

Libraries Used:

React Hooks: Employs useState and useEffect hooks for managing component state and performing side effects respectively.

Axios: Utilizes Axios for making HTTP requests.

State Management:

`useState Hook:` Manages the `userData` state, initially set as an empty object.

Fetching User Data:

`useEffect Hook:` Fetches user data from the `/about` endpoint upon component mount.

`axios.get:` Makes a GET request to `/about` to retrieve user information.

`State Update:` Sets the retrieved user data into the `userData` state using `setUserData`.

UI Rendering:

`Navbar:` Imports and renders a `LandingNavbar` component.

`Content Display:` Renders introductory content, including a title (`h1`) and a description (`p`), showcasing the platform's purpose and inviting users to explore job opportunities.

Additional Component:

`UserDummyDash:` Includes a component called `UserDummyDash` within this section. Its functionality or content is not explicitly defined in the provided code snippet.

Notes:

The component seems focused on presenting an inviting and explanatory section to attract users to explore job opportunities.

It fetches user data from the `/about` endpoint, presumably to personalize or customize the landing page based on the user's information.

The `UserDummyDash` component is included but its functionality is not evident from the provided code snippet.

Overall, this `HeroSection` component is intended to create a visually appealing and informative section on a landing page, promoting job discovery and user engagement.

This `LandingNavbar` component seems to be the header/navigation bar section of a landing page. It offers various functionalities like user authentication, signup options, and responsive navigation.

Component Features:

Libraries Used:

React Hooks: Utilizes useState and useEffect for managing component state and side effects.

React Router DOM: Utilizes Link and useNavigate for navigation.

State Management:

Manages the visibility of modals, selected registration and login types, and the hamburger menu toggle state.

Responsive Design:

Adjusts navigation items and layout based on screen size using isMobile state and the hamburger menu icon for mobile view.

Authentication and Registration:

Modal Popups: Displays modals for user login and registration on click of respective buttons (Login and Sign up).

Allows users to choose between employee and employer registration/login types.

Event Handling:

Handles events like login, signup, and logout through various button click functions (openLoginModal, openModal, handleLogout, etc.).

Conditional Rendering:

Dynamically renders different content based on selected registration and login types.

Notes:

The component appears to be well-structured, managing different authentication flows and rendering the appropriate forms based on user choices.

Modals are used to provide a cleaner interface for authentication and registration actions.

Responsive design considerations are included for both mobile and larger screens.

This LandingNavbar component seems to efficiently manage user authentication and registration flows, offering a user-friendly interface for accessing different functionalities within the application.

Component Overview:

State Management:

isCreatePostOpen: Manages the visibility of a modal or post creation form.

submittedData: Holds an array of submitted job data fetched from an API.

Functions:

updateSubmittedData: Deletes a job entry by ID from the database, updates the local state, and refetches the data.

fetchSubmittedData: Fetches the submitted job data from an API endpoint and updates the local state.

UseEffect:

Uses useEffect to fetch the submitted data from the API when the component mounts.

Rendering:

Renders a component called JobDataFront and passes the fetched submittedData and the updateSubmittedData function as props.

Key Points:

DummyPost primarily manages job-related data, including fetching and deleting job entries.

The updateSubmittedData function handles deletion of job entries by ID.

fetchSubmittedData is responsible for retrieving job data from an API endpoint and updating the local state.

It utilizes useEffect to ensure data fetching occurs when the component mounts.

This DummyPost component appears to manage job-related data through interactions with an API. Here's an overview of its functionality:

Component Overview:

State Management:

isCreatePostOpen: Manages the visibility of a modal or post creation form.

submittedData: Holds an array of submitted job data fetched from an API.

Functions:

updateSubmittedData: Deletes a job entry by ID from the database, updates the local state, and refetches the data.

fetchSubmittedData: Fetches the submitted job data from an API endpoint and updates the local state.

UseEffect:

Uses useEffect to fetch the submitted data from the API when the component mounts.

Rendering:

Renders a component called JobData and passes the fetched submittedData and the updateSubmittedData function as props.

Key Points:

DummyPost primarily manages job-related data, including fetching and deleting job entries.

The updateSubmittedData function handles deletion of job entries by ID.

fetchSubmittedData is responsible for retrieving job data from an API endpoint and updating the local state.

It utilizes useEffect to ensure data fetching occurs when the component mounts.

The EditConfiguration component appears to manage the editing and deletion of job data. It renders a list of jobs fetched from an API and provides the functionality to edit, update, or delete each job entry.

Key Features:

State Management:

editMode: Manages whether the component is in edit mode or not.

editedJob: Stores the job that is being edited.

tempFormData: Holds temporary edited values of the job.

Functions:

handleEdit: Sets the component to edit mode and sets the job to be edited.

handleCancelEdit: Cancels the edit mode and clears edited data.

handleChange: Updates temporary form data as the user edits.

handleUpdate: Sends a PUT request to update the job data.

handleDelete: Sends a DELETE request to delete a job.

Rendering:

Lists all submitted jobs.

When in edit mode, displays input fields to edit job data.

Provides options to update, cancel edit, or delete a job.

Suggestions:

Input Fields: Ensure validation for input fields and proper error handling for user inputs.

Clearer Structure: Consider breaking down the edit mode section into a separate component for better readability and maintainability.

Error Handling: Enhance error handling and feedback to the user upon successful or failed API requests.

Overall, the component seems well-built to handle job editing and deletion functionalities based on the fetched data. Further enhancements could focus on refining user interactions and ensuring a smoother editing experience.

The EditDummy component seems to manage the process of editing job data by rendering the EditConfiguration component and handling the state and functions related to job data editing and deletion.

Key Features:

State Management:

isCreatePostOpen: Manages the state to control the opening and closing of the create post section.

submittedData: Holds the list of submitted job data fetched from an API.

Functions:

openCreatePost: Sets the state to open the create post section.

closeCreatePost: Sets the state to close the create post section.

updateSubmittedData: Deletes a job using an API call, updates the local state of submitted jobs, and navigates to the home page after deletion.

fetchSubmittedData: Fetches the submitted job data from the API and updates the local state.

Rendering:

Renders the EditConfiguration component, which handles the editing and deletion of job data.

Suggestions:

Clear Separation of Concerns: The component focuses on managing the job data editing functionality effectively.

Error Handling: Enhance error handling for API requests, providing proper feedback to users in case of errors.

Consistency in Naming: Ensure consistency in naming conventions to maintain code readability and understanding.

.job-container

max-width: Limits the container's maximum width to 400 pixels.

margin: Centers the container horizontally on the page.

padding: Adds 20 pixels of padding inside the container.

background: Applies a linear gradient background from light grey #dcdcdc to darker grey #a0a0a0.

border-radius: Rounds the corners of the container with a border radius of 16 pixels.

box-shadow: Adds a shadow effect to the container for a raised appearance.

h1

text-align: Centers the text horizontally within the <h1> element.

color: Sets the text color to a dark shade of grey #333.

form

display: Sets the form elements to be displayed in a column layout.

label

margin-top: Adds space on top of labels.

font-weight: Sets the font weight to bold.

color: Defines the color of labels as a darker grey #555.

.submitted-data

margin-top: Adds space on top of the submitted data container.

padding: Adds 15 pixels of padding inside the container.

background: Sets the background color of the container to white #fff.

border-radius: Rounds the corners of the container with a border radius of 8 pixels.

box-shadow: Adds a subtle shadow effect to the container.

.submitted-data h2

color: Sets the text color of <h2> elements within the submitted data container to a dark shade of grey #333.

input, textarea

width: Sets the width of input fields and textareas to 100% of their container.

padding: Adds 8 pixels of padding inside input fields and textareas.

margin-top, margin-bottom: Defines the top and bottom margin space around input fields.

const mongoose = require("mongoose");: Imports the Mongoose library, which helps in managing interactions with MongoDB from Node.js.

const db = process.env.DATABASE;: Retrieves the MongoDB connection URI from the DATABASE environment variable. This URI usually contains information like the database server, port, username, password, and database name.

mongoose.connect(db): Attempts to connect to the MongoDB database using the retrieved URI.

.then(() => { console.log(Connection successful); }): If the connection is successful, it logs a success message to the console.

.catch((err) => { console.log(No connection. Error: ${err}); }): If there's an error during the connection process, it logs an error message to the console.

It looks like the code is meant to establish a connection to the MongoDB database using the mongoose.connect() method. However, it seems incomplete as the actual URI for the database connection is expected to be stored in the DATABASE environment variable, which might need to be properly set in your environment.

var nodemailer = require("nodemailer");: Imports the Nodemailer module, allowing your application to utilize its email-sending capabilities.

var transporter = nodemailer.createTransport({ /* ... */ });: Creates a transporter object using Nodemailer's createTransport() method. This object will be responsible for sending emails. In this case, it's configured to use Gmail's SMTP service.

`service: "gmail"`: Specifies that the email service being used is Gmail.

`auth: { user: process.env.EMAIL, pass: process.env.PASSWORD }`: Configures the transporter with authentication details (`user` as the email address and `pass` as the password) retrieved from environment variables (`process.env.EMAIL` and `process.env.PASSWORD`). It's a good practice to keep sensitive information like email credentials in environment variables for security reasons.

`module.exports = transporter;`: Exports the `transporter` object, making it available for other parts of your application to send emails using this configured transporter.

This code appears to be middleware designed to authenticate users based on JWT (JSON Web Token) authorization in a Node.js application using Express.js.

Let's break down the functionality:

`const jwt = require("jsonwebtoken");`: Imports the `jsonwebtoken` library, which helps in generating and verifying JSON Web Tokens.

`const User = require("../model/userSchema");`: Imports the User model or schema required for database operations related to user authentication.

`const Authenticate = async (req, res, next) => { ... }`: Defines the `Authenticate` middleware function that takes in the `req`, `res`, and `next` parameters. This middleware will run authentication logic before handling requests.

Inside the `Authenticate` middleware:

`Retrieval and Verification of Token`:

It attempts to extract the JWT from the `req.cookies.jwtoken` property.

If no token is present, it throws an error indicating that no token was provided.

Uses `jwt.verify()` to validate the extracted token with the `process.env.SECRET_KEY`. If the verification fails, it throws an error.

`Validation of User`:

Attempts to find a user based on the decoded token's ID in the database.

If the user associated with the token is not found, it throws an error indicating that the user was not found.

`Setting Request Properties`:

If the token is valid and the user is found, it sets `req.token`, `req.rootUser`, and `req.userID` properties to be used in subsequent middleware or route handlers.

Error Handling:

If any error occurs during the authentication process, it logs the error and sends a 401 status response with an "Unauthorized" message along with the error message.

Exporting the Middleware:

Finally, the middleware function Authenticate is exported to be used within the application.

This middleware is designed to protect routes or API endpoints that require authenticated users. It checks for a valid JWT in the request cookies, verifies it, and authorizes the user based on the token's validity and the user's existence in the database.

Imports: It imports the necessary modules, jsonwebtoken and the User model/schema related to employers.

AuthenticateEmp Middleware Function: Similar to the previous middleware, this function is responsible for authenticating employer-related requests.

Inside the AuthenticateEmp middleware:

Token Verification:

Retrieves the JWT token from the req.cookies.jwtoken property.

If no token is present, it throws an error indicating that no token was provided.

Verifies the extracted token using jwt.verify() with the process.env.SECRET_KEY. If the verification fails, it throws an error.

User Validation:

Attempts to find an employer user based on the decoded token's ID in the database.

If the employer associated with the token is not found, it throws an error indicating that the user was not found.

Setting Request Properties:

If the token is valid and the employer user is found, it sets req.token, req.rootUser, and req.userID properties to be used in subsequent middleware or route handlers.

Error Handling:

If any error occurs during the authentication process, it logs the error and sends a 401 status response with an "Unauthorized" message along with the error message.

Exporting the Middleware:

Finally, the middleware function AuthenticateEmp is exported to be used within the employer-related functionalities, ensuring that routes or endpoints requiring employer authentication are protected.

This middleware serves a similar purpose to the previous one but specifically caters to employer-related authentication, verifying employer JWT tokens and authorizing employer users based on their validity and presence in the database.

The code provided defines a Mongoose schema for a job posting (jobSchema). This schema represents the structure of documents that will be stored in the MongoDB collection for job postings.

Here's a breakdown of the schema fields:

title: Title of the job.

image: URL or reference to the job image.

description: Description or details about the job.

Experience: Experience required or preferred for the job.

applicationInstructions: Instructions for applying to the job.

companyName: Name of the company offering the job.

location: Location of the job.

salaryRange: Range of salary for the job.

employmentType: Type of employment (e.g., full-time, part-time, contract).

industry: Industry related to the job.

companyDescription: Description of the company.

contactEmail: Contact email for the job application.

contactPhone: Contact phone number for the job application.

applicationDeadline: Deadline for applying to the job.

This schema is then used to create a Mongoose model named Job with the schema. The model is exported to be used elsewhere in the application. This

model encapsulates all the operations and interactions related to job postings in the MongoDB database.

Developers can use this `Job` model to perform CRUD (Create, Read, Update, Delete) operations on job postings, such as creating new job postings, retrieving job postings, updating existing postings, and deleting postings from the database.

This code defines a Mongoose schema named `userSchema` representing the structure of user documents that will be stored in the MongoDB collection for users. Here's a breakdown of the schema fields:

`name`: Name of the user.

`email`: Email address of the user.

`phone`: Phone number of the user.

`work`: Work-related information or occupation of the user.

`password`: Password for user authentication.

`cpassword`: Confirmation password for user authentication.

`tokens`: An array of tokens, each containing a token string used for user authentication.

Schema Methods and Hooks

`pre middleware`: This middleware runs before saving a user instance. It hashes the `password` and `cpassword` fields using bcrypt before saving the document.

`generateAuthToken method`: This method generates an authentication token for the user using the `jsonwebtoken` package. It signs a token with the user's ID and saves it in the `tokens` array. The generated token is then returned.

The schema is used to create a Mongoose model named `User` and is exported. This `User` model will handle operations related to user authentication, including registering new users, generating authentication tokens, and verifying user credentials during login.

This code defines another Mongoose schema named `userSchemaEmployer`. It represents the structure of user documents intended specifically for employers. Let's break down the fields in this schema:

`Personal Information`

`name`: Name of the employer.

email: Email address of the employer.

phone: Phone number of the employer.

work: Work-related information or occupation of the employer.

password: Password for employer authentication.

cpassword: Confirmation password for employer authentication.

Company Information

companyInformation: Object containing details related to the employer's company.

companyName: Name of the company.

industry: Industry in which the company operates.

companySize: Size of the company (possibly number of employees).

Contact Information

contactInformation: Object containing contact details.

contactName: Name of the contact person.

contactTitle: Title or position of the contact person.

contactEmail: Email address of the contact person.

contactPhone: Phone number of the contact person.

Company Address

companyAddress: Object containing the address details of the company.

streetAddress: Street address of the company.

city: City where the company is located.

state: State where the company is located.

postalCode: Postal code of the company's location.

country: Country where the company is located.

Tokens: An array containing tokens used for employer authentication.

Schema Methods and Hooks

pre middleware: Similar to the previous schema, this middleware hashes the password and cpassword fields using bcrypt before saving the document.

generateAuthToken method: This method generates an authentication token for the employer using the jsonwebtoken package. It signs a token with the

employer's ID and saves it in the `tokens` array. The generated token is then returned.

Finally, the schema is used to create a Mongoose model named `UserEmployer` and is exported. This `UserEmployer` model handles operations related to employer authentication and manages employer-specific data.

This Express router file seems to handle various authentication and job-related routes for both regular users and employers. Let's break down the main functionalities:

Regular User Routes

`/register POST route`: Registers a new user, checks for existing emails, validates the input, and saves the user's data. Generates a JWT token upon successful registration.

`/signin POST route`: Handles user sign-in by comparing passwords, sending login confirmation emails, and generating a JWT token for authenticated users.

`/about, /getData GET routes`: Retrieve user data after authentication using the JWT token.

`/logout GET route`: Clears the JWT token cookie upon user logout.

Employer Routes

`/employerregister POST route`: Registers a new employer, similar to user registration, but includes additional company-related information. Generates a JWT token upon successful registration.

`/signinemployer POST route`: Handles employer sign-in, authenticating based on provided credentials, and generating a JWT token for authenticated employers.

`/aboutemployer GET route`: Retrieves employer data after authentication using the JWT token.

`/logoutemployer GET route`: Clears the JWT token cookie upon employer logout.

Job Routes

`/jobs POST route`: Creates a new job listing.

`/jobs GET route`: Fetches all available job listings.

`/jobs/:id DELETE route`: Deletes a job listing by ID.

`/jobs/:id PUT route`: Updates a job listing by ID.

The code handles user and employer authentication, registration, sign-in, and respective data retrieval routes. Additionally, it includes job-related endpoints for creation, retrieval, update, and deletion.

It looks like your Express server setup is in place. Here's what's happening in your code:

`Middleware Setup:` You've initialized middleware for parsing JSON and cookies using `express.json()` and `cookie-parser`.

`Routes:` You've included routes using `require("./router/auth")`. These routes handle authentication and other functionalities like user registration, sign-in, and more based on the commented parts of the code.

`Basic Endpoints:` There are basic endpoints like `/`, `/contact`, and `/about` to demonstrate functionality and interactions. Some of them involve setting cookies (`res.cookie`) and sending simple responses.

`Listening to Port:` The server listens on the specified `PORT`, which is loaded from your environment variables through `dotenv`.

`Database Connection:` The `require("./db/conn")` indicates the connection setup to your database.

The commented sections suggest that you have additional routes for user authentication, registration, sign-in, and more. These endpoints might be included in the imported `auth` router.