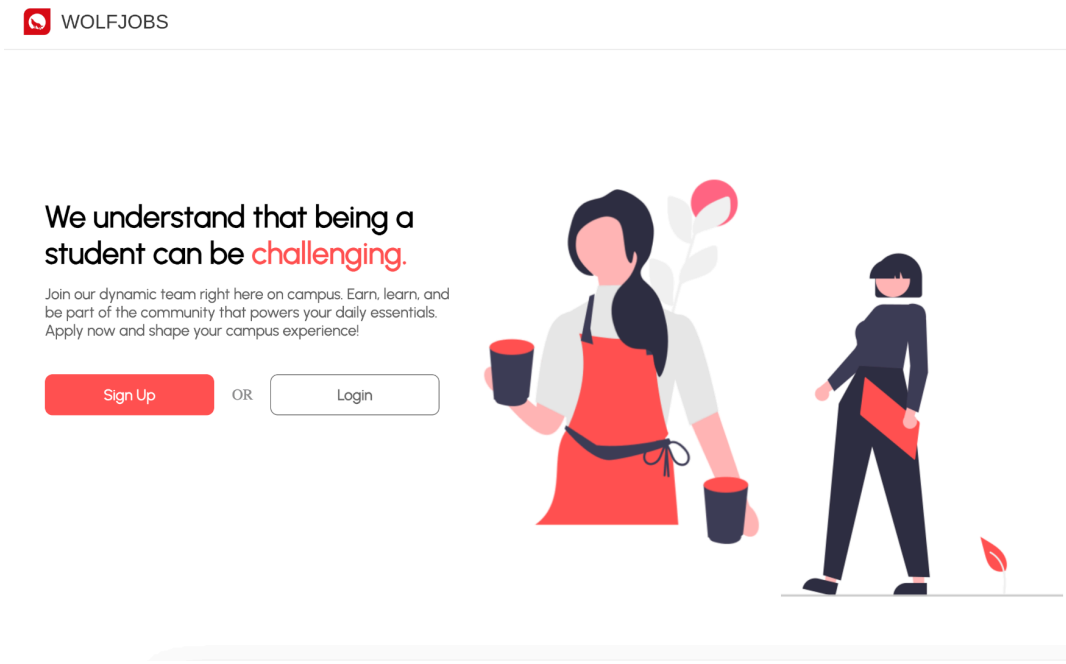
Wolf Jobs



Connecting Students to Opportunities, One Job at a Time!

Software code Documentation

Team Members

Tahreem Yasir
Amay Gada
Shazia Muckram

Introduction

WolfJobs is a comprehensive platform designed to bridge the gap between students and employers by providing a seamless way to explore, apply for, and manage job opportunities on campus. The system integrates a modern web interface with a powerful backend infrastructure to handle user management, job creation, application processes, and more, ensuring that students can find jobs easily while employers can quickly access a pool of talented candidates.

This document outlines the system architecture of the WolfJobs platform, detailing the key technologies, the structure of both the frontend and backend components, and how these elements work together to provide a cohesive and user-friendly experience. The architecture is designed to be scalable, secure, and easy to maintain, ensuring long-term growth and efficient operation of the platform.

Project Description

WolfJobs is a full-stack web application aimed at providing students with a central hub to explore and apply for on-campus job opportunities. The platform offers features such as job browsing, application management, resume uploads, and job notifications. It also includes an admin interface for employers to post job listings, review applications, and manage candidates.

On the backend, the application is built with Node.js, Express, and MongoDB, leveraging Passport.js for authentication and JWT tokens for secure login sessions. This backend ensures fast and secure processing of job data, user profiles, and applications. Additionally, the backend handles user roles, including students and employers, offering tailored access to platform features.

The frontend of WolfJobs is a modern React application, using TypeScript for type safety and Tailwind CSS for styling. It provides an interactive and responsive user interface with intuitive navigation, job exploration, and user profile management. The frontend communicates with the backend through RESTful APIs to handle actions like submitting job applications, updating profiles, and receiving notifications.

Technologies

- **Node.js & Express:** These technologies power the backend of the WolfJobs platform. Node.js provides a fast, scalable JavaScript runtime environment, while Express is used to set up and manage API routes, handle requests, and deliver responses efficiently.
- **MongoDB:** A NoSQL database used for storing user data, job listings, application records, and other dynamic content. MongoDB’s flexible schema makes it ideal for handling the varied and evolving nature of the data on WolfJobs.
- **Passport.js & JWT Authentication:** Passport.js is used for user authentication, with JWT tokens ensuring secure and stateless sessions for users. JWT enables safe login and access control without the need for session cookies.

- **React & TypeScript:** The frontend is built using React, a JavaScript library for building interactive user interfaces. TypeScript is used for adding type safety and reducing runtime errors, ensuring a more maintainable codebase.
- **Tailwind CSS:** A utility-first CSS framework that enables rapid styling of components with a focus on responsive design. Tailwind allows for highly customizable UI development with minimal code.
- **Vite:** A next-generation build tool that serves as the development server for React, enabling faster build times and smooth development experiences with hot module reloading.
- **Vitest:** A testing framework that provides a fast, comprehensive testing environment for the frontend, ensuring the platform's components and pages work as expected.

Architecture Details

Backend

The backend folder contains the server-side code responsible for handling requests, managing data, and authenticating users. It includes:

- **Controllers:** Handle the logic for various operations such as user registration, job posting, and application processing. For example, `users_controller.js` manages user-related actions, while `resume_controller.js` manages resume-related functionalities.
- **Models:** Define the structure of the data entities, such as `job.js`, `user.js`, and `application.js`, which represent jobs, users, and applications, respectively. These models interact with the MongoDB database to store and retrieve data.
- **Config:** Contains configuration files for integrating external services, such as Passport.js strategies for authentication (`passport-local-strategy.js`, `passport-jwt-strategy.js`), and database configuration (`mongoose.js`).

The backend processes all user interactions, including job management, user authentication, and resume handling, and serves data to the frontend through REST APIs.

Frontend

The frontend folder houses the React application responsible for delivering the user interface. It includes:

- **Pages:** Different components corresponding to pages like `LoginPage.tsx`, `Dashboard.tsx`, and `CreateJob.tsx`. These are the primary UI components that users interact with to explore jobs, apply, and manage their profiles.
- **Components:** Reusable UI elements, such as `NavBar.tsx`, `JobDetails.tsx`, and `ResumeViewer.tsx`, provide consistent design and functionality across the platform.

- **API:** Contains utility functions (`client.ts`, `alljobs.ts`) for interacting with the backend API, enabling features like fetching job listings, submitting applications, and updating user profiles.

The frontend ensures an engaging, responsive, and user-friendly experience, with a seamless connection to the backend for all job-related operations.

Documentation for Controller Methods in Software Architecture Document

---

Module Overview

The module implements key functionalities for managing users, job postings, applications, and related operations. It provides RESTful APIs for CRUD operations and integrates email notifications for specific status updates. The design ensures data consistency and user interaction through well-defined endpoints.

---

1. User Management

1.1 `createSession`

- **Description**: Authenticates a user using email and password, returning a JSON Web Token (JWT) upon successful login.
- **Input**: `email`, `password`
- **Output**: JWT Token, user details
- **Error Handling**: Returns a 422 error for invalid credentials or 500 for internal errors.

1.2 `signUp`

- **Description**: Registers a new user if the provided email is not already in use.
- **Input**: User details including `email`, `password`, `confirm_password`
- **Output**: JWT Token, user details
- **Error Handling**: Validates password match and checks for existing users. Handles server errors.

1.3 `getProfile`

- **Description**: Retrieves profile information of a user by ID.
- **Input**: `user ID` (via URL params)
- **Output**: User details
- **Error Handling**: Returns 500 for server errors.

1.4 `editProfile`

- **Description**: Updates the profile information for a user.
- **Input**: User ID and updated fields (e.g., `name`, `skills`, `experience`)
- **Output**: Confirmation of successful update
- **Error Handling**: Handles 500 internal server errors.

1.5 `searchUser`

- **Description**: Searches for users/jobs by name using a case-insensitive regex match.
- **Input**: `name` (via URL params)
- **Output**: List of matching users/jobs
- **Error Handling**: Handles 500 errors for server issues.

---

2. Job Management

2.1 `createJob`

- **Description**: Adds a new job posting with manager details and job specifications.
- **Input**: Job details including `name`, `description`, `pay`, `skills`
- **Output**: Job object
- **Error Handling**: Validates input and handles 500 errors for failed operations.

2.2 `index`

- **Description**: Lists all job postings sorted by creation date.
- **Output**: Array of job objects
- **Error Handling**: Handles 500 internal errors.

2.3 `editJob`

- **Description**: Updates job details for an existing job.
- **Input**: Job ID and updated fields
- **Output**: Confirmation of successful update
- **Error Handling**: Validates job existence and handles 500 errors.

2.4 `closeJob`

- **Description**: Marks a job as closed.
- **Input**: Job ID
- **Output**: Confirmation of closure
- **Error Handling**: Validates job existence and handles 500 errors.

2.5 `deleteJob`

- **Description**: Deletes a job by ID.

- **Input**: Job ID
- **Output**: Confirmation of deletion
- **Error Handling**: Validates job existence and handles 500 errors.

---

3. Application Management

3.1 `createApplication`

- **Description**: Creates a job application if one does not already exist for the applicant and job.
- **Input**: Application details including `applicantName`, `jobId`, `skills`
- **Output**: Application object
- **Error Handling**: Checks for existing applications and handles 500 errors.

3.2 `fetchApplication`

- **Description**: Retrieves all job applications sorted by creation date.
- **Output**: Array of application objects
- **Error Handling**: Handles 500 internal errors.

3.3 `acceptApplication`

- **Description**: Updates application status to accepted and sends an email notification.
- **Input**: `applicationId`
- **Output**: Confirmation of acceptance
- **Error Handling**: Validates application existence and handles email failures.

3.4 `rejectApplication`

- **Description**: Updates application status to rejected and sends an email notification.
- **Input**: `applicationId`
- **Output**: Confirmation of rejection
- **Error Handling**: Validates application existence and handles email failures.

3.5 `modifyApplication`

- **Description**: Updates application status and optional fields based on status type (e.g., grading, rating).
- **Input**: Application details including `status`, optional `answers`
- **Output**: Confirmation of update
- **Error Handling**: Validates input and handles email failures.

3.6 `modifyApplicationFinalStage`

- **Description**: Updates final application status to accepted or rejected, notifying the applicant.
- **Input**: Application details including `status`, `applicantName`, `jobName`
- **Output**: Confirmation of update
- **Error Handling**: Handles email failures and 500 errors.

---

4. History Management

4.1 `createHistory`

- **Description**: Logs user activity including calories gained or burned.
- **Input**: `date`, `total`, `burnout`, `userId`
- **Output**: History object
- **Error Handling**: Handles 500 internal errors.

4.2 `getHistory`

- **Description**: Retrieves user activity log for a specific date.
- **Input**: `userId`, `date`
- **Output**: History object
- **Error Handling**: Handles 500 internal errors.

---

5. Email Integration

- **Module**: `nodemailer`
- **Functionality**: Sends email notifications for application status updates.
- **Error Handling**: Catches and logs email sending errors, returning appropriate responses.

---

6. Error Handling and Cross-Origin Resource Sharing (CORS)

- **Error Responses**: All endpoints return structured JSON error responses with HTTP status codes.
- **CORS**: Each response sets `Access-Control-Allow-Origin` to `*`.

---

1. **Home Controller**
    - **Purpose**: Provides functionality to render the home page of the application.

- **Key Functions**:
  - `home(req, res)`: Renders the "home" page with the title "Home".

2. **Resume Management**

   - **Purpose**: Manages user resumes, including upload and retrieval functionality.
   - **Dependencies**: `Resume`, `User` models, `multer` for file uploads.
   - **Key Functions**:
     - `uploadResume(req, res)`: Uploads a new resume for a user. Replaces an existing resume if it exists.
     - `getResume(req, res)`: Fetches and serves the resume for a specific user.
     - `upload`: Multer configuration object for file uploads.

3. **Ping Endpoint**

   - **Purpose**: Provides a simple health check for the service.
   - **Key Function**:
     - `ping(req, res)`: Responds with a "Pong" message.

4. **User Controller**

   - **Purpose**: Handles user-related functionalities such as profile management, authentication, and session management.
   - **Dependencies**: `User` model.
   - **Key Functions**:
     - `profile(req, res)`: Renders the user's profile page.
     - `signUp(req, res)`: Renders the sign-up page or redirects to the profile if authenticated.
     - `signIn(req, res)`: Renders the sign-in page or redirects to the profile if authenticated.
     - `create(req, res)`: Creates a new user if credentials are valid.
     - `createSession(req, res)`: Initiates a session for the user upon sign-in.
     - `destroySession(req, res)`: Logs the user out and ends the session.

Detailed Description

1. **Home Controller**

   - **Responsibilities**:
     - Serves as the entry point to render the primary home page of the application.
   - **Implementation Notes**:

        - Uses `res.render` to dynamically render the "home" view.
2. **Resume Management**
    - **Responsibilities**:
        - Handles resume uploads with size and type validations.
        - Retrieves resumes for users by `applicantId`.
    - **Key Configurations**:
        - **Multer**: Configured to accept only `.pdf` files with a maximum size of 15MB.
    - **Error Handling**:
        - Returns appropriate error messages for invalid files, non-existing users, or database errors.
3. **Ping Endpoint**
    - **Responsibilities**:
        - Provides a lightweight API endpoint for verifying the service's availability.
4. **User Controller**
    - **Responsibilities**:
        - Manages user interactions, including registration, login, logout, and profile viewing.
    - **Error Handling**:
        - Handles validation errors such as mismatched passwords or duplicate email registrations.
        - Logs server-side issues for debugging purposes.
    - **Session Management**:
        - Utilizes authentication middleware to check user status and manage sessions.

Key Design Considerations

- **Error Handling**:
    - Comprehensive checks are implemented to ensure error handling, such as verifying user existence and handling file upload exceptions.
    - User-related actions include logging and redirection to handle both expected and unexpected failures.
- **Security**:
    - Resumes are stored with metadata (`contentType`, `fileName`) and the actual file content (`fileData`) in the database.

    - Authentication checks (e.g., `req.isAuthenticated`) ensure sensitive pages like profiles are only accessible to authenticated users.
- **Scalability**:
    - Resume upload and retrieval functionality are optimized for performance by limiting file size and validating input early in the process.

Integration Points

- **Frontend**:
    - Interacts with views like `home`, `user_profile`, `user_sign_up`, and `user_sign_in` for rendering.
    - Responses include dynamic content (e.g., titles) to enhance user experience.
- **Database**:
    - Interfaces with `Resume` and `User` models for CRUD operations.
- **Middleware**:
    - Utilizes `multer` for handling file uploads securely and efficiently.

Possible Enhancements

1. **Resume Management**:
    - Add functionality to list all resumes or filter by date.
    - Enable additional file types beyond PDFs with explicit user confirmation.
2. **User Controller**:
    - Implement email verification for sign-ups.
    - Add features for password recovery and two-factor authentication.
3. **General**:
    - Introduce logging middleware to better track system operations and debug issues.
    - Enhance error messaging with user-friendly formats for front-end consumption.

MongoDB Configuration and Connection Management

Purpose:
This module establishes a connection to a MongoDB database using the `mongoose` library and provides a reusable connection object for database operations.

Functionality:

- **Database Connection:** Uses Mongoose's `connect` method to establish a connection to the MongoDB cluster. The connection string includes cluster details, user credentials, and options like retry settings.
- **Event Management:** Implements event listeners to handle database connection events, such as errors and successful connections. Errors are logged to the console for debugging.
- **Reusability:** Exports the connection object, allowing other modules to reuse it for database operations.

Key Components:

- Connection initialization with `mongoose.connect`.
- Event listeners for handling connection errors (`db.on("error")`) and successful connections (`db.once("open")`).
- Exports the database connection object for integration with other application components.

JWT Authentication Configuration (Passport.js)

Purpose:
This module configures Passport.js to authenticate users using JSON Web Tokens (JWTs), ensuring secure access control in the application.

Functionality:

- **JWT Extraction:** Defines how JWTs are extracted from HTTP requests, specifically from the `Authorization` header as a Bearer token.
- **JWT Verification:** Validates the token using a secret key (`wolfjobs`) and extracts user information from the payload.
- **Database Integration:** Fetches user details from the database using the user ID stored in the JWT payload. If the user is found, authentication succeeds; otherwise, it fails.

Key Components:

- Configuration of JWT extraction and secret key using the `passport-jwt` strategy.

- Verification callback to validate JWTs and query user details.
- Logs for tracking errors during the database query process.
- Export of the Passport instance for use in other modules.

Local Authentication Configuration (Passport.js)

Purpose:
This module configures Passport.js for local authentication using email and password credentials.

Functionality:

- **User Verification:** Validates user credentials by querying the database for a matching email and verifying the password.
- **Error Handling:** Logs errors if user lookup fails or if the credentials are invalid.
- **Session Management:** Implements user serialization and deserialization to manage session cookies.

Key Components:

- Local strategy configuration with `passport-local`.
- Callback to validate user credentials and handle authentication logic.
- Serialization and deserialization of user sessions for efficient session management.
- Middleware to ensure authenticated access and set user data in response locals.

Middleware for Authentication

Purpose:
Provides middleware functions to check user authentication status and set authenticated user data in response objects.

Functionality:

- **Authentication Check:** Ensures that only authenticated users can access specific routes. Redirects unauthenticated users to the sign-in page.
- **User Data Handling:** Sets the current authenticated user in the response locals for easy access in views or downstream middleware.

Key Components:

- Middleware to validate if a user is authenticated (`checkAuthentication`).
- Middleware to set the authenticated user in `res.locals` (`setAuthenticatedUser`).

- Integration with Passport.js for session-based authentication management.

Integration:
This module exports the configured Passport.js instance, enabling seamless use across the application for authentication and session management.

Application Schema (`application.js`)

Purpose:
Defines the structure for storing job application data in a MongoDB collection.

Functionality:

- **Schema Design: Includes fields for applicant details (e.g., name, email, skills, phone number, etc.), job details (e.g., job ID, job name, deadline), and application-specific fields (e.g., answers to questions, status, rating).**
- **Relationships: Establishes references to other collections, such as `User` and `Job`, enabling integration with related data.**
- **Validation and Defaults: Enforces required fields like `jobname` and `applicantname` and provides default values for optional fields.**
- **Model Creation: Exports the `Application` model for performing CRUD operations on job application data.**

Email Configuration (`nodemailer.js`)

Purpose:
Configures and manages email sending functionality using Gmail's SMTP service.

Functionality:

- **Transporter Setup: Establishes a secure connection to Gmail's SMTP server with predefined credentials and configuration options.**
- **Email Sending: Provides a reusable function (`sendMail`) for sending emails, including recipient details, subject, and HTML content.**
- **Reusability: Exports the email sending function for integration with other application components, enabling notifications and communication features.**

User Schema (`user.js`)

Purpose:
Defines the structure for storing user information in a MongoDB collection.

Functionality:

- **Schema Design: Includes fields for user credentials (e.g., email, password), personal details (e.g., name, address, phone number, date of birth, gender), and professional information (e.g., skills, experience, projects).**
- **Role Management: Contains a `role` field to differentiate user roles, such as admin or standard user.**
- **Relationships: Supports referencing related documents, such as resumes, through the `resumeId` field.**
- **Validation and Defaults: Enforces required fields like `email`, `password`, and `name` while providing default values for optional fields.**
- **Timestamps: Automatically tracks creation and modification dates with the `timestamps` option.**
- **Model Creation: Exports the `User` model for use in authentication, authorization, and user management workflows.**

authOTP.js

The `authOTP.js` file defines the schema and model for handling one-time passwords (OTPs) in the application. This schema is used to store and manage OTP-related information in the database. The key features of this schema include:

- **Purpose: Manages OTPs for user authentication processes.**
- **Fields:**
  - **`userId`: Identifies the user requesting the OTP; this field is required.**
  - **`otp`: The one-time password issued to the user; this field is required.**
  - **`createdAt`: A timestamp indicating when the OTP was created. Defaults to the current date and time.**
- **Configuration: The schema includes automatic timestamping to track `createdAt` and `updatedAt`.**
- **Model: The `autoOtp` model provides methods to interact with the `autoOtp` collection in MongoDB.**

Job.js

The `Job.js` file defines the schema and model for job postings. This schema ensures the consistent structure and validation of job-related data in the database. The key features include:

- **Purpose: Represents job listings and their associated metadata.**
- **Fields:**

  - **Basic Details: `name`, `managerid`, `managerAffilication`, `location`, `description`, `pay`, `type`.**
  - **Job Status: `status`, with a default value of "open".**
  - **Required Skills: A string listing necessary skills.**
  - **Applicant Questions: Four fields (`question1` to `question4`) for capturing questions for job applicants.**
  - **Saved Status: A boolean field indicating whether the job has been saved by a user (default: `false`).**
  - **Deadline: The `jobDeadline` field specifies the deadline for applications.**
- **Model: The `Job` model provides an interface for interacting with job documents in the database.**

---

resume.js

The `resume.js` file defines the schema and model for storing applicant resumes. This schema ensures the safe and efficient management of resume files in the database. Key features include:

- **Purpose: Stores resumes uploaded by job applicants.**
- **Fields:**
  - **`applicantId`: References the applicant's user ID from the `User` model.**
  - **`fileName`: Stores the name of the uploaded file.**
  - **`fileData`: Contains the binary data of the resume file.**
  - **`contentType`: Specifies the MIME type of the file (default: `application/pdf`).**
  - **`uploadedAt`: Timestamp indicating when the resume was uploaded (default: current date and time).**
- **Model: The `Resume` model facilitates interaction with resume data in the database.**

SavedApplications.js

The `SavedApplications.js` file defines the schema and model for saving job postings that users are interested in. This schema allows users to track jobs they wish to revisit. Key features include:

- **Purpose: Tracks jobs saved by users for future reference.**
- **Fields:**
  - **`jobId`: References the specific job saved, linked to the `Job` model.**

    - **`userId`: References the user who saved the job, linked to the `User` model.**
    - **`createdAt`: A timestamp indicating when the job was saved (default: current date and time).**
- **Model: The `SavedJob` model provides methods to manage saved job data in the database.**

Module: `index.js`

Description

This module sets up the primary routing mechanism for the application. It defines an Express router to handle specific paths and delegates the handling of these paths to other modules. Specifically, all routes prefixed with `/users` are forwarded to the `users` router module. This separation of concerns allows for modular and maintainable code.

Module: `users.js`

Description

The `users.js` module serves as the main router for handling API endpoints related to user and job management. It uses Express to define various routes, each corresponding to a specific functionality or action. These routes are implemented using controller methods from the `usersApi` module, ensuring a clear separation of routing and business logic.

Key Features

- **User Management:**
    - **Session Management: API for creating user sessions.**
    - **User Signup: Endpoint for registering new users.**
    - **Profile Management: APIs for retrieving, editing, and searching user profiles.**
- **Job Management:**
    - **Job Creation and Editing: Endpoints to create, edit, and delete job listings.**
    - **Job History: APIs to manage and fetch job history for users.**
- **Application Management:**

    - **Application Handling: APIs to create, modify, accept, reject, and close applications.**
    - **Final Stage Handling: Special API to handle the final stage modifications of applications.**
- **One-Time Password (OTP) Services:**
    - **OTP Generation and Verification: Secure endpoints for handling OTP-based user verification.**
- **Saved Jobs:**
    - **Save and Fetch Jobs: APIs for users to save and retrieve saved jobs.**

Middleware Usage

- **Body Parsing: Utilizes `body-parser` middleware for handling JSON payloads in request bodies.**
- **Controller Delegation: All route actions are delegated to methods defined in the `usersApi` controller, adhering to the MVC architecture.**

API Overview

- **HTTP Methods: The module supports a variety of HTTP methods (`GET`, `POST`, `PUT`, `DELETE`) to cover CRUD operations.**
- **Dynamic Path Parameters: Routes like `/getprofile/:id` and `/search/:name` leverage dynamic path parameters for targeted data retrieval.**

Routing

The following routes are defined in the `users.js` module:

1. **User Session and Profile Management**
    - `/create-session`, `/signup`, `/edit`, `/getprofile/:id`, `/search/:name`
2. **Job Management**
    - `/createjob`, `/editjob`, `/deletejob`, `/closejob`, `/gethistory`
3. **Application Management**
    - `/createapplication`, `/modifyApplication`, `/modifyApplicationFinalStage`, `/acceptapplication`, `/rejectapplication`
4. **OTP Services**

    - **`/generateOTP`, `/verifyOTP`**
5. **Saved Jobs**
    - **`/saveJob`, `/saveJobList/:id`**
6. **Default Route**
    - **`/`: Provides index or status of available routes.**

Export

The router is exported as a module to be integrated seamlessly with the main application.

JobDetailView.tsx

Function/Component: `JobDetailView`

- **Description:**
  The `JobDetailView` component is responsible for displaying the details of a selected job. It retrieves the job's information using the `jobId` from the query parameters and displays either a `NoJobSelected` message if no job is selected, or a detailed view of the job using the `JobDetail` component if a job is selected.

- **Key Points:**
    - **Retrieves `jobId` from URL parameters using `useSearchParams`.**
    - **Uses `useState` to manage the state of the job data (`jobData`).**
    - **Retrieves the list of jobs from the `JobStore`.**
    - **Updates `jobData` based on the selected `jobId`.**
    - **Displays a `NoJobSelected` component if no job is selected, or `JobDetail` if a job is found.**

JobDetails.tsx

Function/Component: `JobDetail`

- **Description:**
  The `JobDetail` component is responsible for rendering detailed information about a specific job, including its status, description, and required skills. It also provides functionality for job applicants to apply for the job or answer a questionnaire, depending on the job's status and the user's role. It handles both

the submission of job applications and responses to job-related questionnaires.

- **Key Points:**
    - **Displays detailed information about the job (name, status, pay, description, etc.).**
    - **Conditionally shows the application button or questionnaire based on the user's role (`Applicant`) and job status.**
    - **Manages form state using `react-hook-form` to handle the questionnaire.**
    - **Allows applicants to apply for the job or submit answers to a job-specific questionnaire.**
    - **Handles API requests for creating an application or submitting questionnaire responses.**

JobFinalReview.tsx

Function/Component: `JobFinalReview`

- **Description:**
  The `JobFinalReview` component is used by managers to review the list of accepted and rejected candidates for a specific job. It filters the applications based on their status (accepted/rejected) and displays them in separate sections. The component provides a link to view the resumes of the applicants.

- **Key Points:**
    - **Retrieves the list of applications for a specific job from the `ApplicationStore`.**
    - **Filters the applications based on their status (`accepted` or `rejected`).**
    - **Displays the list of accepted and rejected candidates with details such as name, phone number, email, and skills.**
    - **Provides a link to view the applicant's resume.**
    - **Updates the lists dynamically based on the job data.**

JobGrading.tsx

Description: The `JobGrading` component is responsible for displaying a list of job applicants who are in the "grading" status. It fetches the list of applicants from the application store, filters them based on the job ID and their status, and presents their details including answers to job-related questions. Users can grade the applicants by entering a score and submitting it. This component interacts with the backend API to

update the grading status of applicants and provides feedback to the user via toast notifications.

Functions/Features:

- **State Management: Manages local state for the display list of applicants (`displayList`), which is updated based on the job's `grading` status.**
- **Grading: Allows users to input grades for applicants' answers and submit the grades to the backend API.**
- **API Integration: Sends a `POST` request to update the applicant's status and grade.**
- **Toast Notifications: Displays success or error messages based on the outcome of the API request.**
- **Dynamic List Rendering: Renders the list of applicants with relevant job information, including their answers to job-related questions.**

JobListTile.tsx

Description: The `JobListTile` component displays an individual job listing in a tile format. It shows key job details, including the role, job status, and pay rate. It also displays an affiliation tag, match status (based on the user's skills), and allows users to bookmark jobs. The component is interactive, enabling users to toggle bookmarks and view more details about the job. Additionally, it dynamically adjusts based on whether the user is an applicant or a job manager.

Functions/Features:

- **Dynamic Job Display: Displays job details such as role, job status, and pay, along with an affiliation tag.**
- **Match Status: Calculates and displays whether the job matches the applicant's skills.**
- **Bookmarking: Allows users to bookmark jobs by toggling the bookmark state, which is reflected both in the UI and backend.**
- **Interactive Links: Displays different action options based on the user's role (e.g., view application, fill questionnaire).**
- **Conditional Styling: Adjusts the display of job status and affiliation tags based on dynamic criteria.**

JobsListView.tsx

Description: The `JobsListView` component serves as a container for displaying a list of job tiles. It receives a list of jobs and a title as props, and renders each job in the

`JobListTile` component. The view is designed to be scrollable, with a fixed width and responsive layout. This component provides a unified view of multiple job listings and is ideal for listing jobs in a dashboard or job portal view.

Functions/Features:

- **Jobs List Display: Renders a list of jobs dynamically using the `JobListTile` component.**
- **Responsive Layout: Ensures that the job listings are displayed in a scrollable container with appropriate styling.**
- **Title Display: Optionally displays a title for the jobs list, defaulting to "All jobs".**
- **Side-by-Side Layout: Ensures a clean layout for job listings, with space allocated for job details and interactivity.**

Here is the description for the provided components to be included in the software architecture document:

JobManagerView

Description: The `JobManagerView` component is a central view for managing job-related tasks for users with managerial roles. It facilitates job management by providing functionality for closing, deleting, and editing jobs. The component conditionally renders different child components based on the user's role and job status. It also allows switching between various views such as job screening, grading, rating, and final review using a tab-style interface.

Key Responsibilities:

- **Display job management options (close, delete, edit) based on the manager's role and job status.**
- **Dynamically load different views for job screening, grading, rating, and final review.**
- **Handle job closure, deletion, and editing through API calls.**
- **Update the view dynamically using React Router's search parameters and manage state for form data.**

JobRating

Description: The `JobRating` component is responsible for displaying and managing the job candidates in the rating stage. It allows the manager to review applications and either

accept or reject candidates. The component interacts with the backend via API calls to update candidate status and provides real-time feedback using toast notifications.

Key Responsibilities:

- **Filter and display job applicants who are in the rating stage.**
- **Allow the manager to accept or reject candidates, updating their status through API calls.**
- **Provide real-time feedback to the user through toast notifications upon successful or failed operations.**

JobScreening

Description: The `JobScreening` component displays job applicants who are in the "applied" stage. It allows the manager to screen applicants by accepting or rejecting their applications, progressing them to the next stage of the hiring process. The component interacts with the backend to update application statuses and provides real-time notifications on success or failure.

Key Responsibilities:

- **Display job applicants in the "applied" status for screening.**
- **Allow the manager to accept or reject applicants, updating their status accordingly via API calls.**
- **Provide visual feedback and real-time updates to the user using toast notifications.**

NoJobSelected

Description: The `NoJobSelected` component is a fallback view that appears when no job has been selected by the user. It provides a message indicating that no job is currently available for viewing or interaction. This component is used to guide users to select a job for more detailed information or actions.

Key Responsibilities:

- **Display a message prompting the user to select a job for further actions.**
- **Provide a visual placeholder with a simple design to indicate that no job is currently selected.**

ResumeDropZone.tsx

Component Overview

`ResumeDropZone` is a React functional component that provides a drag-and-drop file upload interface for PDF files. The component utilizes the `react-dropzone` library to manage the file upload process. It supports only PDF file types and imposes a maximum file size limit of 15 MB. The component also provides feedback to users in the form of error logging for rejected files (e.g., due to incorrect file type or size) and passes the accepted files to a parent component through the `onFileUpload` callback.

Properties

- **`onFileUpload (Function)`: A callback function that receives an array of accepted files (PDFs) when the user uploads files.**

Key Functionality

- **File Upload Handling: Accepts files through drag-and-drop or file selection. Only PDF files up to 15 MB are accepted.**
- **Error Handling: Logs errors for rejected files (e.g., file type or size issues) to the console.**
- **Callback to Parent: Once valid files are uploaded, they are passed to the parent component using the `onFileUpload` function.**

ResumeViewer.tsx

Component Overview

`ResumeViewer` is a React functional component responsible for displaying a resume in PDF format. The component retrieves the PDF from a backend API using the applicant's ID from the URL parameters, and renders the document page by page using the `react-pdf` library. It includes navigation controls for users to move between pages of the resume.

Key Properties

- **No props: The component does not receive any external props but instead relies on URL parameters to fetch the relevant resume.**

Key Functionality

- **Resume Fetching: Fetches the applicant's resume in PDF format from the backend API using the applicant's ID.**

- **PDF Rendering: Renders the resume one page at a time, allowing navigation between pages.**
- **Page Navigation: Users can move to the next or previous page of the PDF resume using "Previous" and "Next" buttons.**
- **Cleanup: Ensures that the generated Blob URL for the resume is properly cleaned up when the component unmounts or the URL changes to prevent memory leaks.**

Dependencies

- **`react-pdf`: For rendering the PDF document.**
- **`axios`: For making HTTP requests to fetch the resume.**
- **`react-router-dom`: For accessing the URL parameters.**

LoginPage.tsx

Class/Function: `LoginPage`

- **Description: The `LoginPage` component is responsible for rendering a login form, managing form validation, and handling the form submission process for user authentication. It utilizes `react-hook-form` for form state management and `yup` for validation. Upon form submission, the user's email and a hashed password (using `CryptoJS`) are sent to the backend for authentication. The page also includes error handling for invalid input and uses Material-UI components for layout and styling.**
- **Point Description:**
  - **`useNavigate`: A React Router hook used to navigate between different routes, specifically used for redirecting users after successful login.**
  - **`useForm`: A hook from `react-hook-form` that helps in managing the form's state and handles input validation.**
  - **`yupResolver`: Used to integrate `yup` validation schema with `react-hook-form`.**
  - **`onSubmit`: Handles form submission, hashes the password, and calls the `login` function to authenticate the user.**
  - **`login` function: Sends user credentials to the backend (deprecated) for authentication.**

LogoutPage.tsx

Class/Function: `LogoutPage`

- **Description: The `LogoutPage` component handles the user logout process by clearing stored user data from both local storage and the application state. It uses the `useEffect` hook to ensure this process occurs as soon as the component mounts, redirecting the user to the login page. After the data is cleared, the user is logged out, and the state is reset, ensuring a clean state for future sessions.**
- **Point Description:**
  - **`useEffect`: A hook that runs the logout logic immediately after the component is mounted, ensuring a smooth transition to the login page.**
  - **`useUserStore`: A custom store hook that updates the user-related state values (e.g., name, address, role) to default values upon logout.**
  - **`localStorage.clear()`: Clears all user data stored in the browser's local storage.**
  - **`navigate`: Redirects the user to the login page after logging out.**

RegistrationPage.tsx

Class/Function: `RegistrationPage`

- **Description: The `RegistrationPage` component manages the user registration process, including form validation, role management, and data submission. It uses `react-hook-form` for form handling, `CryptoJS` for password security, and Material-UI for layout. Users can create an account by entering their personal details, selecting a role (Manager or Applicant), and submitting the form. Upon successful submission, the `signup` function is called to register the user, and the user is redirected to another page.**
- **Point Description:**
  - **`useForm`: Initializes and manages form data, validation rules, and error handling for the registration form.**
  - **`role` and `affiliation`: State variables used for storing the user's selected role (Applicant/Manager) and affiliation (if the role is Manager).**
  - **`onSubmit`: Handles form submission, hashes passwords, and calls the `signup` function to register the user.**
  - **`signup` function: Handles the logic for user registration, including sending the form data (email, password, skills, etc.) to the backend for processing.**

LandingPage.tsx

Class/Function: `LandingPage`

- **Description: The `LandingPage` component serves as the introductory page for the application. It presents a welcoming message to the users, introducing the platform and offering options to either sign up or log in. The page uses styled**

elements and buttons to guide users to the registration or login pages. It provides an interactive UI experience for users to get started with the application.

- **Point Description:**
  - **`useNavigate`: React Router hook used to navigate to the login or registration pages when users interact with the corresponding buttons.**
  - **UI Styling: The page uses inline styles to position and style various elements like text, buttons, and links. The text is customized to emphasize key information, such as the challenges of being a student and the benefits of joining the platform.**
  - **`navigate`: Redirects the user to the login or registration pages based on button clicks.**

Here are the point descriptions for the `CreateJob.tsx` and `JobPreview.tsx` components to be included in the software architecture document:

CreateJob Component

Point Description:

- **Purpose: The `CreateJob` component is responsible for allowing users to create a new job listing by filling in relevant details such as job role, type, location, pay, required skills, description, and deadline. Upon submission, it navigates to the next page to fill out additional job-specific questions.**

- **Core Features:**

  - **State Management: Utilizes React `useState` hooks to manage local state for job type (`jobType`) and required skills (`requiredSkills`).**
  - **Form Handling: Uses React Hook Form for managing the form state, validating inputs, and handling submission.**
  - **Form Validation: Ensures that fields like job role, pay, and deadline are required, with appropriate error handling.**
  - **Navigation: Leverages `useNavigate` from `react-router` to navigate to the job questionnaire page after form submission, passing along the job details as route state.**
  - **Material-UI: Utilizes Material-UI components (e.g., `TextField`, `FormControl`, `Button`) for creating the form interface and managing form layout.**

JobPreview Component

Point Description:

- **Purpose: The `JobPreview` component displays a preview of the job listing that the user is about to submit. It shows the job details filled out in the previous step and allows users to confirm and submit the job listing. The component also integrates with backend services to store the job listing.**

- **Core Features:**

    - **State Management: Retrieves job details and questionnaire responses passed from the previous page using `useLocation` from `react-router`.**
    - **Form Submission: Provides an `onSubmit` function that posts the job data to a server endpoint using `axios`. It handles both success and error responses and displays appropriate toast notifications.**
    - **User Interaction: Displays a summary of the job details, including role, type, location, pay, description, and required skills. It also shows the questions submitted by the user in the previous step.**
    - **Navigation: After successful job creation, navigates the user to the dashboard page, updating their job listing.**
    - **Material-UI: Uses Material-UI components like `Button` for form submission and `AiFillCheckCircle` for progress visualization.**

JobQuestionnaire Component

Description:

The `JobQuestionnaire` component is responsible for rendering a form that allows the user to input answers to four job-related questions as part of a job listing creation process. It uses `react-hook-form` for managing form state, validation, and submission. The component includes error handling and a form with text fields for each question. Upon successful form submission, the data is collected and the user is navigated to a job preview page where they can view the job listing.

Point Description:

- **Location Hook: Retrieves the location object from the React Router to access any state passed from the previous page.**
- **Navigate Hook: Allows navigation between different routes of the application, used here to redirect users to the job preview page after submission.**
- **useForm Hook: Initializes the form with default values and provides methods to handle form state, validation, and submission using `react-hook-form`.**

- **Form Fields: Includes four text input fields (question1, question2, question3, question4), each with validation for required input. Errors are displayed if the input is invalid.**
- **onSubmit Function: Handles form submission, constructs a body with the form data and state from the previous page, and uses `navigate` to redirect to the job preview page with the form data as part of the state.**
- **Styling: Custom styling is applied to form elements, including text fields and buttons, ensuring a consistent and user-friendly UI.**

This component encapsulates the functionality for collecting and validating job listing details, managing the navigation flow, and integrating with form handling libraries to enhance user experience.

Dashboard.tsx - Class/Function Documentation

Dashboard Component

Point Description:

- **The `Dashboard` component is the main view for the user, displaying different information based on the user's role (either "Manager" or "Applicant").**
- **It handles the rendering of job listings for managers, along with the option to view applications. For applicants, it displays jobs they have applied for, along with their application statuses.**
- **It interacts with multiple state management stores (`UserStore`, `JobStore`, `ApplicationStore`) to fetch and manage user data, job listings, and application statuses.**
- **The component ensures that the user is authenticated, fetching and updating the user data and job applications when the component mounts.**
- **The component dynamically renders job listings or applications based on the user role and filters the job data accordingly.**

Key Functions/Features:

1. **`useEffect` (1): Fetches user data from the local storage token on mount, and updates the user-related information in the store (e.g., name, email, role, etc.).**
2. **`useEffect` (2): Fetches job listings and applications from the backend API, updating the respective stores with the fetched data.**
3. **`useEffect` (3): Filters and updates the displayed job listings based on the user's role:**
    - **For Managers: Displays jobs managed by the logged-in user.**
    - **For Applicants: Displays jobs the applicant has applied for.**
4. **Role-based Rendering: The job list is displayed with different actions depending on whether the user is a Manager or an Applicant.**

5. **Job Creation Button (Manager Role): Managers are presented with a button to create a new job listing. The button navigates to the "Create Job" page.**
6. **Job Listings and Application Details: Displays job listings and application details, and allows the user to interact with them based on their role.**

---

updateExperience Function

Point Description:

- **This is a placeholder function that is currently not implemented.**
- **It throws an error if invoked, indicating that the function's functionality has not yet been defined.**
- **This function is expected to be used to update the `experience` attribute in the user state, but as of now, it is unimplemented and serves as a stub for future functionality.**

Key Features:

- **The function signature suggests that it would receive an argument (`experience`), which represents the user's experience data.**
- **Intended to interact with the `UserStore` to update the user's experience details.**

Class: `Explore`

Description:
The `Explore` component allows users to explore job listings through search, filter, and sort functionalities. It interacts with the backend API to fetch job and application data, updates global job and application lists, and manages the user's session and profile information. The component provides a user-friendly interface to search for jobs, filter by criteria such as location, salary, employment type, and toggle between viewing open or closed job listings. Additionally, users can sort job listings by salary, city, or employment type.

Key Features:

- **Displays job listings with filters for job type, location, salary, and employment status.**
- **Allows sorting job listings by highest pay, city, or employment type.**
- **Provides a search bar for filtering jobs by name.**
- **Displays job details and list side by side.**
- **Fetches and manages user and application data using global state from `UserStore` and `ApplicationStore`.**

Function: `handleSearchChange`

Description:
This function handles the change event for the search input field. It updates the `searchTerm` state whenever the user types into the search bar.

Key Points:

- **Accepts an input change event and updates the `searchTerm` state with the new value.**
- **Used to filter job listings by name.**

Function: `handleSortChange`

Description:
This function toggles the sorting order of job listings by salary. When called, it flips the `sortHighestPay` state between `true` and `false`, determining whether job listings should be sorted by the highest pay.

Key Points:

- **Toggles the sorting of job listings based on salary.**
- **Affects the display order of jobs based on pay scale.**

Function: `handleSortCityChange`

Description:
This function toggles the sorting order of job listings alphabetically by city. It changes the `sortAlphabeticallyByCity` state to either true or false, which determines whether jobs should be sorted by location.

Key Points:

- **Sorts job listings alphabetically by location (city).**
- **Toggles between sorted and unsorted states for job locations.**

Function: `handleSortEmploymenyTypeChange`

Description:
This function toggles the sorting order of job listings by employment type. It updates the `sortByEmploymentType` state, controlling whether job listings should be sorted based on employment type.

Key Points:

- **Toggles sorting of job listings by employment type (e.g., full-time, part-time).**
- **Changes the order of job listings according to their employment type.**

Function: `toggleJobStatus`

Description:
This function toggles the visibility of job listings based on their status (open or closed). It updates the `showOpenJobs` state to show either open or closed jobs, depending on the current value.

Key Points:

- **Switches between viewing open or closed job listings.**
- **Changes the visibility of jobs based on their application status.**

Function: `useEffect` (User Data Fetching)

Description:
This `useEffect` hook fetches user-related data from a token stored in the browser's localStorage. It decodes the token, extracts user information, and updates the global user state (using `UserStore`) with the retrieved data. If no valid token is found, the user is redirected to the login page.

Key Points:

- **Retrieves user data from the localStorage token and decodes it.**
- **Updates global user state with fetched information (e.g., name, email, role, etc.).**
- **Redirects the user to the login page if no token is found.**

Function: `useEffect` (Job and Application Data Fetching)

Description:
This `useEffect` hook fetches job and application data from the backend API when the component mounts. It then updates the global job list and application list in the state, sorting the data by job deadlines.

Key Points:

- **Fetches job and application data via API calls.**
- **Updates the `jobList` and `applicationList` global states.**
- **Sorts job and application data by job deadlines for display.**

Function: `useEffect` (Filtering and Sorting Jobs)

Description:
This `useEffect` hook manages filtering and sorting of job listings. It updates the `filteredJobList` based on search terms, sorting options, and various filter criteria (e.g., location, salary range, job type). The list is updated each time any filter or sorting criterion changes.

Key Points:

- **Filters job listings by search term, salary, location, and employment type.**
- **Sorts job listings based on salary, city, or employment type.**
- **Updates the displayed list of filtered jobs.**

This structure outlines the primary functions and classes within the `Explore` component, suitable for inclusion in a software architecture document.

Class/Function Documentation for `Notifications.tsx`

Class: `Notifications`

- **Description:**
  The `Notifications` component is responsible for managing and displaying job notifications based on their acceptance or rejection status. It interacts with the application's global state to fetch, categorize, and render job data. Users can toggle visibility for accepted and rejected job notifications. Additionally, it allows navigation to a detailed view of a job when clicked.

- **Point Description:**
  - **State Management:**
    - **`acceptedJobs`: Stores the list of jobs that have been accepted.**
    - **`rejectedJobs`: Stores the list of jobs that have been rejected.**
    - **`isAcceptedVisible`: A boolean flag to manage the visibility of the accepted job notifications.**
    - **`isRejectedVisible`: A boolean flag to manage the visibility of the rejected job notifications.**
  - **Hooks:**
    - **`useEffect` (for fetching data):**
      Fetches application and job data from APIs when the component mounts. The data is used to update the global state using store actions and to handle errors via toast notifications.

        - **`useEffect` (for filtering jobs):**
          Filters the fetched job data based on the acceptance or rejection status from the application list and updates the state for accepted and rejected jobs accordingly.
        - **`useNavigate`:**
          A navigation hook used to navigate users to the job dashboard with the selected job ID when a job is clicked.
    - **Functions:**

        - **`handleJobClick`:**
          Navigates to the job dashboard page with the ID of the selected job, allowing users to view job details.
        - **`toggleAcceptedVisibility`:**
          Toggles the visibility of the accepted jobs list.
        - **`toggleRejectedVisibility`:**
          Toggles the visibility of the rejected jobs list.
    - **User Interaction:**

        - **Users can toggle the visibility of job notifications for accepted and rejected jobs.**
        - **Clicking on a job notification navigates to the job details page, providing further insights into the selected job.**

Profile.tsx

Function Description:

- **Profile Component: This component is responsible for displaying the user profile details such as name, email, role, skills, etc. It fetches the profile data from a global store using the `useUserStore` hook and presents it in a non-editable view. It includes a toggle mechanism for enabling an edit mode, allowing the user to modify their profile. When in edit mode, the `ProfileEdit` component is rendered to facilitate updating the user profile.**

Point Description:

- **User Data Retrieval: Uses `useUserStore` to pull user information such as name, email, and role from a global state.**
- **Edit Mode Toggle: Includes icons (pencil and close) to toggle between view and edit modes.**
- **Edit Mode Rendering: When in edit mode, the `ProfileEdit` component is rendered to allow the user to edit their profile.**
- **Non-editable View: Displays the profile details in a non-editable format by default.**

- **Profile Card Layout: The profile is displayed in a card layout with defined width and scrollable content.**

ProfileEdit.tsx

Function Description:

- **ProfileEdit Component: This component provides a form for users to update their profile information. It supports editing multiple fields such as name, email, skills, and others. The form leverages React Hook Form for managing the form state and validation. Upon successful submission, it sends the updated data to the backend API and redirects the user to the login page after saving the changes.**

Point Description:

- **Form Initialization: Uses React Hook Form with default values populated from the current profile data passed via props.**
- **Form Fields: Provides text input fields for various profile attributes like name, email, skills, and role, with validation rules.**
- **Availability Drop-down: Features a drop-down to select the user's availability, with options like 4, 8, 12, 16, and 20 hours.**
- **Profile Save Handling: Submits updated profile information via a POST request to the backend API and handles success or error responses.**
- **Toast Notifications: Displays toast messages upon successful or failed profile update.**
- **Login Redirection: Upon successful profile update, the component triggers a login action to refresh the user session and navigates them to the login page.**

ResumeViewer Component

Description:
The `ResumeViewer` component is responsible for fetching and displaying a resume PDF for a specific applicant. It retrieves the PDF from the server based on the applicant's ID, which is extracted from the URL parameters. Once the PDF is loaded, the user can navigate through the pages of the resume.

- **Purpose: Displays the resume PDF of a specific applicant.**
- **Main Functionality:**
  - **Fetches the resume PDF using the applicant's ID from the URL.**
  - **Renders the PDF document using the `react-pdf` library.**
  - **Allows users to navigate through the pages of the resume with previous and next buttons.**
- **Key Properties:**

    - **`numPages`: Tracks the total number of pages in the PDF document.**
    - **`pageNumber`: Tracks the current page number displayed in the PDF.**
    - **`resumeUrl`: Stores the URL for the fetched PDF document.**
- **Functions:**
    - **`onDocumentLoadSuccess`: Handles the successful loading of the document and sets the number of pages.**
    - **`goToPreviousPage`: Moves to the previous page in the resume.**
    - **`goToNextPage`: Moves to the next page in the resume.**
- **Lifecycle:**
    - **Uses `useEffect` to fetch the resume from the server based on `applicantId`.**
    - **Cleans up resources by revoking the object URL when the component unmounts.**

ResumeDropzone Component

Description:
The `ResumeDropzone` component provides a drag-and-drop area for uploading resume files. It leverages the `react-dropzone` library to handle file uploads, validating file types and size constraints. This component allows users to upload a resume, which is then passed to the parent component for further handling.

- **Purpose: Enables users to upload resume files through a drag-and-drop interface.**
- **Main Functionality:**
    - **Accepts PDF files with a maximum size of 15 MB.**
    - **Handles both drag-and-drop file upload and file selection via a file picker.**
    - **Passes accepted files to the parent component for further processing.**
    - **Logs and handles rejected files (e.g., invalid file types or sizes).**
- **Key Properties:**
    - **`onFileUpload`: A callback function that handles the uploaded files passed from the dropzone.**
- **Functions:**
    - **`onDrop`: Handles the file drop event, processes accepted files, and logs rejected files.**
    - **`dropzoneOptions`: Defines configuration for the dropzone, including file type and size validation.**
- **Lifecycle:**
    - **Uses the `useDropzone` hook to initialize the dropzone with the provided configuration.**
    - **Dynamically updates the display based on the drag state (active or inactive).**

Saved Component

Description

The `Saved` component is responsible for managing and displaying a list of jobs that the user has saved. It interacts with an API to fetch saved job data, sorts the jobs by their deadlines, and filters the list to include only those marked as saved. The component utilizes two child components, `JobsListView` to display the list of jobs and `JobDetailView` to show detailed information about a selected job.

Point Description

- **Fetches Saved Jobs: Retrieves a list of jobs saved by the user from the backend API using the `axios` HTTP client.**
- **Sorts Jobs by Deadline: The fetched jobs are sorted by their job deadline in ascending order to display them in a chronological sequence.**
- **Filters Saved Jobs: Filters out jobs that are marked as saved, ensuring only the relevant jobs are displayed.**
- **Displays Jobs List: The filtered job list is passed to the `JobsListView` component for rendering.**
- **Displays Job Details: The `JobDetailView` component is included to display detailed information about a job when selected by the user.**
- **Error Handling: In case of an API failure, an error message is displayed using the `toast` notification library to inform the user.**

Saved Component Function

Description

This function component fetches the list of saved jobs for the current user, sorts them by deadline, and displays them using the `JobsListView` component. It also handles the display of detailed job information through the `JobDetailView` component. The component relies on the `useEffect` hook to make an API call when the component is mounted or when the `userId` changes.

Point Description

- **User Data Access: Uses the `useUserStore` custom hook to access the `userId` of the logged-in user.**
- **State Management: Manages the state of the filtered job list using `useState`, which holds the jobs that are saved and sorted.**

- **Effect Hook: The `useEffect` hook triggers the fetching of saved jobs from the API on component mount or when the `userId` changes.**
- **API Request: Makes an asynchronous `GET` request to the backend to fetch the saved job list for the user.**
- **Error Handling: Displays an error toast if the API request fails or if the status code is not 200.**
- **Data Sorting: Sorts the jobs by their deadline in ascending order to ensure the list is displayed in the correct order.**
- **Data Filtering: Filters the jobs to include only those marked as saved, and updates the state with the filtered list.**